Broyden updates; Quasi-Newton methods for nonlinear systems of equations

Abdulrahman Alabdulkareem*

Abstract. In this work, we give a brief history of the root-finding problem and the use of Newton's method. Then, we discuss Quisi-Newton methods and derive both types of Broyden Updates along with an efficient implementation of the algorithm. We run our implementation of both types of Broyden Updates along with Newton's method and an implementation of Broyden Updates provided by the Scipy library on two realistic real world problems. We finally discuss the application of Broyden Updates in many different areas of science and the different variants of the algorithm that were later proposed.

Key words. Broyden Updates, Quasi-Newton methods

AMS subject classifications. 68Q25, 68R10, 68U05

1. Introduction.

1.1. The root-finding problem.

The root-finding problem is the problem of finding a solution to a non-linear system of equations. Root-finding problems arise in many areas of science whether it's engineering, statistics, social sciences, and many others. The history of such algorithms date back decades ago where Newton developed an algorithm to find roots of polynomials.

The general root-finding problem is defined as finding a point $x_* \in \mathbb{R}^n$ such that it satisfies

$$f(x_*) = 0 \tag{1.1}$$

Where $f : \mathbb{R}^n \to \mathbb{R}^n$.

As mentioned in [31], the history of solutions and algorithms for the root-finding problem and the analysis of such algorithms depends on properties on f. If we take f to be a linear n-dimensional system of equations then we can find extensive study in the works of Golub [9], Stewart [34], and Young [41].

For the case or f being a 1 dimensional system of non-linear equations then we can refer to Heitzinger [11], Householder [12], and Traub [35].

However, we are interested in the intersection of the above two cases where f in Equation (1.1) is both n-dimensional and a non-linear system of equations. When an analytic inverse of f is not trivially available, then we can use iterative methods to solve such a problem. Iterative methods are a class of algorithms that find a better approximate to the solution with each step of the algorithm.

1.2. Newton's Method.

The first iterative root finding method was proposed by Newton more than 300 years ago which is referred to as "Newton's Method" [39], or sometimes referred

*Department of Computational Science and Engineering, Massachusetts Institute of Technology, Cambridge, MA. (arkareem@mit.edu).

to as the "Newton-Raphson Method" [42, 39] referring to Joseph Raphson who later proposed the same method after Newton [39, 3].

The motivation begins with a taylor expansion of f and a random initial guess x_0 for x_* to get

$$f(x_0 + \Delta x) \approx f(x_0) + J(x_0)\Delta x \tag{1.2}$$

Our goal is to find a new guess $x_1 = x_0 + \Delta x$ that fulfills

$$f(x_*) = 0 \Rightarrow f(x_0 + \Delta x) = 0 \Rightarrow f(x_0) + J(x_0)\Delta x = 0 \Rightarrow \Delta x = J(x_0)^{-1} f(x_0) \tag{1.3}$$

Where x_0 is some initial guess, and

$$where \;\; x_n = \begin{bmatrix} x_n^{(1)} \\ \vdots \\ x_n^{(N)} \end{bmatrix} \;\; and \;\; F(x_n) = \begin{bmatrix} f_1(x_n) \\ \vdots \\ f_M(x_n) \end{bmatrix} \tag{1.4}$$

and $J(x_n)$ is the Jacobian of F at x_n

$$J(x_n) = J(x)|_{x=x_n}, \;\; J(x) = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x^{(1)}} & \cdots & \frac{\partial f_1(x)}{\partial x^{(N)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M(x)}{\partial x^{(1)}} & \cdots & \frac{\partial f_M(x)}{\partial x^{(N)}} \end{bmatrix} \tag{1.5}$$

Thus we get the update step that

$$\Delta x_n = J(x_n)^{-1} F(x_n) \tag{1.6}$$

And thus we get the n-dimensional version of Newton's method which iteratively solves Equation (1.1) written as

$$Newtons\ method : x_{n+1} = x_n - J(x_n)^{-1} F(x_n), \tag{1.7}$$

A problem that might seem apparent from the above update step is that we need to calculate the inverse of the Jacobian in each iteration, which would be very expensive for an $(N \times M)$ matrix. However, this issue can be trivially avoided by solving the linear system $J_{(x_n)} * v = F(x_n)$ instead of actually computing the inverse of the Jacobian to get $v = J^{-1}_{(x_n)} * F_{(x_n)}$.

1.3. Computational Complexity of Newton's Method. The real major problems with Newton's method is that, in most real-world problems, it is incredibly computationally expensive to calculate the Jacobian. If all the functions have partial derivatives with analytical solutions which means the Jacobian is directly available, then we still need to evaluate $(N * M)$

complicated functions which is expensive. On the other hand, if we don't have analytical functions for the partial derivatives, then we would need to evaluate each function f_i $(n+1)$-times to obtain an approximation of the Jacobian using finite-difference approximation (technically this would be using the secant method and not Newton's method if we were to use finite-differences to calculate the Jacobian).

In either case calculating the Jacobian is expensive, and this needs to be done every single iteration of Newton's method [43].

1.4. Quasi-Newton methods. To overcome the computational burden with Newton's method, Quasi-Newton methods were developed. The class of Algorithms called Quasi-Newton methods is referred to root-finding algorithms that don't directly compute the Jacobian. The simplest Quasi-Newton method is simply calculating the Jacobian once $\mathrm{J}(x_0)$ and then use that for all iterations, which is clearly not very good as it will diverge in all cases where $\mathrm{J}(x_0)$ doesn't directly point to the optimal solution. Another very simple Quasi-Newton method is to calculate the Jacobian once every k-steps.

This is the context where the algorithm for Broyden's updates was developed in 1965 [5], The idea at a high level is that the Jacobian $\mathrm{J}(x_n)$ can be approximated using the Jacobian at the previous step $\mathrm{J}(x_{(n-1)})$ along with other components that were already calculated in this current iteration.

2. Brief Derivation of Broyden Updates. The update step of the Jacobian must fulfil the following the secant equation to be a valid new Jacobian

$$J_{n+1}(x_{n+1} - x_n) = F(x_{n+1}) - F(x_n) \tag{2.1}$$

However the above system is underdetermined, so we assume that the Jacobian only updated in the direction of $(x_{n+1} - x_n)$ which means

$$\forall v \;\; s.t. \;\; v^T((x_{n+1} - x_n)) = 0, \;\; \texttt{we have} \;\; J_{n+1}v = J_n v \tag{2.2}$$

$$J_{n+1}v = J_n v \Rightarrow (J_{n+1} - J_n)v = (\Delta J_n)v = 0 \tag{2.3}$$

Which means that the Jacobian does not change for any direction that is perpendicular to the change in x.

We still have many choices for ΔJ_n that fullfil both Equation (2.1) and Equation (2.2) so we employ another assumption that ΔJ_n is a rank 1 matrix. These two assumptions give us a unique value of ΔJ_n, here's how we calculate that. First a rank 1 matrix fulfilling Equation (2.2) must be written $\Delta J_n = w(x_{n+1} - x_n)^T$ for some free choice of w. Then we plug this into Equation (2.1) to get the value of w

$$(J_n + w(x_{n+1} - x_n)^T)(x_{n+1} - x_n) = F(x_{n+1}) - F(x_n) \tag{2.4a}$$

$$w = ((F(x_{n+1}) - F(x_n)) - J_n(x_{n+1} - x_n))/((x_{n+1} - x_n)^T(x_{n+1} - x_n)) \tag{2.4b}$$

Thus we get the update equation for the Jacobian

$$J_{n+1} = J_n + (\Delta J_n) = J_n + w(x_{n+1} - x_n)^T \tag{2.5a}$$

$$J_{n+1} = J_n + ((F(x_{n+1}) - F(x_n)) - J_n(x_{n+1} - x_n))\frac{(x_{n+1} - x_n)^T}{(x_{n+1} - x_n)^T(x_{n+1} - x_n)} \tag{2.5b}$$

Using $\Delta x_n = (x_{n+1} - x_n)$ and $\Delta F_n = F(x_{n+1}) - F(x_n)$ we get the simple expression

$$J_{n+1} = J_n + (\Delta F_n - J_n \Delta x_n)\frac{\Delta x_n^T}{\Delta x_n^T \Delta x_n} \tag{2.6}$$

The above two assumptions that Broyden makes gave us a unique update step ΔJ_n which just so happens to minizimize the Frobenius norm for all matrices that fulfil the secant condition Equation (2.1)

$$\Delta J_n = \arg\min_{\Delta J_n \in \mathcal{S}} \|\Delta J_n\|_F \;\; where \; \forall S \in \mathcal{S}, \; (2.1) \; is \; true \tag{2.7}$$

However, we can work directly on the inverse of the Jacobian by taking advantage of the Sherman–Morrison formula [2] where the inverse of a rank one updated matrix $A_{n+1}^{-1} = (A_n + uv^T)^{-1}$ can be written as

$$A_{n+1}^{-1} = A_n^{-1} - \frac{A_n^{-1} u v^T A_n^{-1}}{1 + v^T A_n^{-1} u} \tag{2.8}$$

Plugging in $J_n = A_n$ and $v = \Delta x_n$ and $u = w$ we get

$$J_{n+1}^{-1} = J_n^{-1} - \frac{J_n^{-1}\left(\frac{\Delta F_n - J_n \Delta x_n}{\Delta x_n^T \Delta x_n}\right)(\Delta x_n)^T J_n^{-1}}{1 + (\Delta x_n)^T J_n^{-1}\left(\frac{\Delta F_n - J_n \Delta x_n}{\Delta x_n^T \Delta x_n}\right)} = J_n^{-1} + \frac{\left(\Delta x_n - J_n^{-1}\Delta F_n\right)\Delta x_n^T J_n^{-1}}{(\Delta x_n^T J_n^{-1} \Delta F_n)} \tag{2.9a}$$

With that, Equation (2.9a) gives us what is called the Type 1 Broyden Update for the Jacobian. Now, just like Newton's update we utilize the updated inverse of the Jacobian to get our new value of x using Equation (1.6)

2.1. Broyden Updates Type 2. Another set of assumptions (specifically Equation (2.2) is changed to be a change only in the direction of Δf) end up minimizing the Frobinius norm of the change in the inverse of Jacobian [15]

$$\Delta J_n^{-1} = \arg\min_{\Delta J_n^{-1} \in \mathcal{S}} \|\Delta J_n^{-1}\|_F \;\; where \; \forall S \in \mathcal{S}, \; (2.1) \; is \; true \tag{2.10}$$

And we end up with the update equation

$$J_{n+1}^{-1} = J_n^{-1} + \frac{\left(\Delta x_n - J_n^{-1}\Delta F_n\right)\Delta F_n^T}{(\Delta F_n^T \Delta F_n)} \tag{2.11}$$

Which gives us what is known as Type 2 Broyden Updates.

3. Algorithm. Initially, we start with a guess x_0 and evaluate $F(x_0)$ and for the only time in the algorithm, explicitly calculate the Jacobian at the initial guess $J(x_0)$ (we discuss initializing the Jacobian further in subsection 3.1 where we never actually calcualte the Jacobian). After that, we start the loop until convergence (The convergence condition is discussed in subsection 3.2). As with regular Newton's method, the update rule for x is

$$x_n = x_{n-1} - J(x_{n-1})^{-1}F(x_{n-1}) \tag{3.1}$$

(In subsection 3.3 we discuss a modification to the update rule using a line search update)
And the update rule for the inverse of the Jacobian is

$$J_{n+1}^{-1} = J_n^{-1} + (\Delta x_n - J_n^{-1}\Delta f_n)\frac{\Delta x_k^T J_n^{-1}}{\Delta x_k^T J_n^{-1}\Delta f_n} \tag{3.2}$$

We can take out from the right side the matrix J_n^{-1}

$$J_{n+1}^{-1} = \left(I + (\Delta x_n - J_n^{-1}\Delta f_n)\frac{\Delta x_k^T}{\Delta x_k^T J_n^{-1}\Delta f_n}\right) * J_n^{-1} \tag{3.3}$$

We can see the recursive pattern. To fully take advantage of that we can define two new variable $a_n = (\Delta x_n - J_n^{-1}\Delta f_n)$ and $b_n^T = \frac{\Delta x_k^T}{\Delta x_k^T J_n^{-1}\Delta f_n}$ to get

$$J_{n+1}^{-1} = \left(I + a_n b_n^T\right) * J_n^{-1} \tag{3.4}$$

Thus we only need to store $a_i \in \mathbb{R}^n$ and $b_i \in \mathbb{R}^n$ for $i = 1, \ldots, k$ to calculate J_n^{-1} which requires $\mathcal{O}(kn)$ space instead of $\mathcal{O}(n^2)$. This also means that once the number of iterations exceeds a threshold ($k > n$), it is more efficient to store the entire matrix J_n^{-1} instead of the individual vectors that construct it. This condition (referred to as collapse) can be seen in the official implementation of Broyden's method in the scipy module[38] for python in the exact line 671 in the file "_nonlin.py" [1]

We have opted not to implement a collapse since we have noticed that all our experiments terminated with the number of iterations less than n which means the algorithm converged before a collapse would have happened.

A similar process is implemented for Type 2 Broyden Updates where Equation (2.11) give us that $a_n = \left(\Delta x_n - J_n^{-1}\Delta F_n\right)$ and $b_n = \frac{\Delta F_n^T}{(\Delta F_n^T \Delta F_n)}$

$$J_{n+1}^{-1} = J_n^{-1} + \left(I + a_n b_n^T\right) \tag{3.5}$$

(Notice how Equation (3.4) we have a recursive multiplication while in Equation (3.5) we have a recursive summation which makes a slight difference in the implementation)

[1] https://github.com/scipy/scipy/blob/main/scipy/optimize/_nonlin.py#L671

3.1. Initial Jacobian Estimate. For the initial Jacobian estimate, we could explicitly approximate the Jacobian using a finite difference approximation, however, that tends to still take a significantly long time especially when the problem is high dimensional. So instead, we use a technique from [14] and set the initial jacobian to the Identity matrix times a constant as follows

$$J_0 = \frac{1}{\alpha} * I \quad and \quad J_0^{-1} = \alpha * I \tag{3.6}$$

For the choice of alpha it should depend on the problem and a bad choice of alpha can make the algorithm diverge or converge extremely slowly. The decision by [14] for all their calculations is to use $\alpha = 0.3$. However, we have experimented and used the reciprocal of the initial norm of F, this seems to be stable

$$\alpha = \frac{-1}{\|F(0)\|_2} \tag{3.7}$$

3.2. Convergence Condition. For the convergence of our algorithm we simply check that the L2 norm of the function is less than a small value ϵ which we pick to be $\epsilon = 6e - 6$ thus we terminate once

$$\|F(x_n)\|_2 \leq 6e - 6 \tag{3.8}$$

3.3. Line Search Update. We have found that the convergence of Broyden updates is much more guaranteed if we utilize some form of Iterative procedure (particularly a Line Search update) in each iteration of the Broyden update. As [26] describes in section 3.5 "For general nonlinear functions, it is necessary to use an iterative procedure. The line search procedure deserves particular attention because it has a major impact on the robustness and efficiency of all nonlinear optimization methods."

Concretely, instead of updating our guess x_n as follows

$$x_n = x_{n-1} - J(x_{n-1})^{-1} F(x_{n-1}) \tag{3.9}$$

We instead partially take a step towards the update direction as follows

$$x_n = x_{n-1} - \alpha_n J(x_{n-1})^{-1} F(x_{n-1}) \;\; where \; \alpha_n \in (0, 1] \tag{3.10}$$

The particular choice of α depends on the line search algorithm used. Frist let's define $\phi(\alpha)$ to be the squared L2 norm of the function evaluated when α is our step length:

$$\phi(\alpha) = \|F\left(x_n - \alpha J(x_n)^{-1} F(x_n)\right)\|_2^2 \tag{3.11}$$

We start the algorithm with $\alpha_k = 1$ then 'backtrack' by slightly decreasing α until the following update condition is fulfilled

$$\frac{\phi(\alpha_k) - \phi(0)}{\alpha_k - 0} \leq -\epsilon * \phi(0) \tag{3.12}$$

The update step for α is then a cubic interpolation of $\phi(0)$ $-\phi(0)$ and the most recent value $\phi(\alpha)$ which is provided in section 3.5 of [26].

Notice how this process involves calling the underlying function $F(\cdot)$ while attempting to find a good value for α which is a deviation from the standard broyden update which only calls the underlying function once per iteration. However, this extra cost comes with the reward that the algorithm converges much quicker, converges with less total calls to the underlying function, and is less likely to diverge.

Algorithm 3.1 Broyden Updates

procedure BROYDEN($x0$, α, ϵ, f, $istype1$)
$xn \leftarrow x0$
$fn \leftarrow f(xn)$
$fn\_norm \leftarrow \|fn\|_2$
$an \leftarrow []$
$bn \leftarrow []$
while $fn\_norm > \epsilon$ **do**
  $jfn \leftarrow$ **JACOBIAN_SOLVE**$(fn, \alpha, an, bn, istype1)$
  $\Delta x \leftarrow -1 * jfn$
  $xn \leftarrow xn + \Delta x$
  $\Delta fn \leftarrow f(xn) - fn$
  $fn \leftarrow fn + \Delta fn$
  $fn\_norm \leftarrow \|fn\|_2$
  $jfn\_new \leftarrow$ **JACOBIAN_SOLVE**$(fn, \alpha, an, bn, istype1)$
  $\Delta jfn \leftarrow jfn\_new - jfn$
  if $istype1$ **then**
    $const \leftarrow (\Delta x).T * \Delta jfn$
    $b \leftarrow \Delta x$
  else
    $const \leftarrow (\Delta fn).T * \Delta fn$
    $b \leftarrow \Delta fn$
  end if
  $an.append((\Delta x - \Delta jfn)/const)$
  $bn.append(b)$
end while
return xn

3.4. Full Algorithm. The full algorithm is implemented in Algorithm 3.1, type 1 and type 2 Broyden updates are similar enough that both are implemented in a single algorithm.

Algorithm 3.2 Solving Jacobian, returns $J^{-1} * f$

```
procedure JACOBIAN_SOLVE(fn, α, an, bn, istype1)
r ← α * fn
for a, b ← an[i], bn[i] do
  if istype1 then
    v ← r
  else
    v ← fn
  end if
  r ← r + a * (b.T * v)
end for
return r
```

Algorithm 3.2 solves and returns a vector r that fulfills the equation $J * r = fn$ by utilizing the fast implementation of the inverse Jacobian shown in Equation (3.4) for type 1 and Equation (3.5) for type 2.

4. Validation.

4.1. N-Dimensional Linear System. The first validation problem is a set of n linear equations parameterized by n slopes and y-intercepts:

$$f_i(x) = \alpha_i * x_i + \beta_i \tag{4.1a}$$

$$F(x) = Ax + B \tag{4.1b}$$

Where $A \in \mathbb{R}^{n \times n}$ is a diagonal matrix with diagonal elements sampled from a uniform $\alpha_i \in \mathcal{U}(-10, 10)$ and $B \in \mathbb{R}^n$ are also elements sampled from a uniform $\beta_i \in \mathcal{U}(-10, 10)$

4.1.1. Results. Just as expected both Newton's method and Broyden updates converge in one or two steps (assuming A is not ill-conditioned).

The previous example was very trivial and is only meant to show that the algorithm is functioning and not behaving wildly. Next we will look at an actual real world application involving integrals.

4.2. Solving a non-linear integral equation. This problem was proposed in [24] and was also used as an example problem in [44].

The goal is to find a solution for the function $u(t)$ where u is a scalar function $u : \mathbb{R} \to \mathbb{R}$ and $t \in [0, 1]$ Such that:

$$G(t) = u(t) + \int_0^1 H(s,t)(u(s) + s + 1)^3 ds = 0 \tag{4.2}$$

where

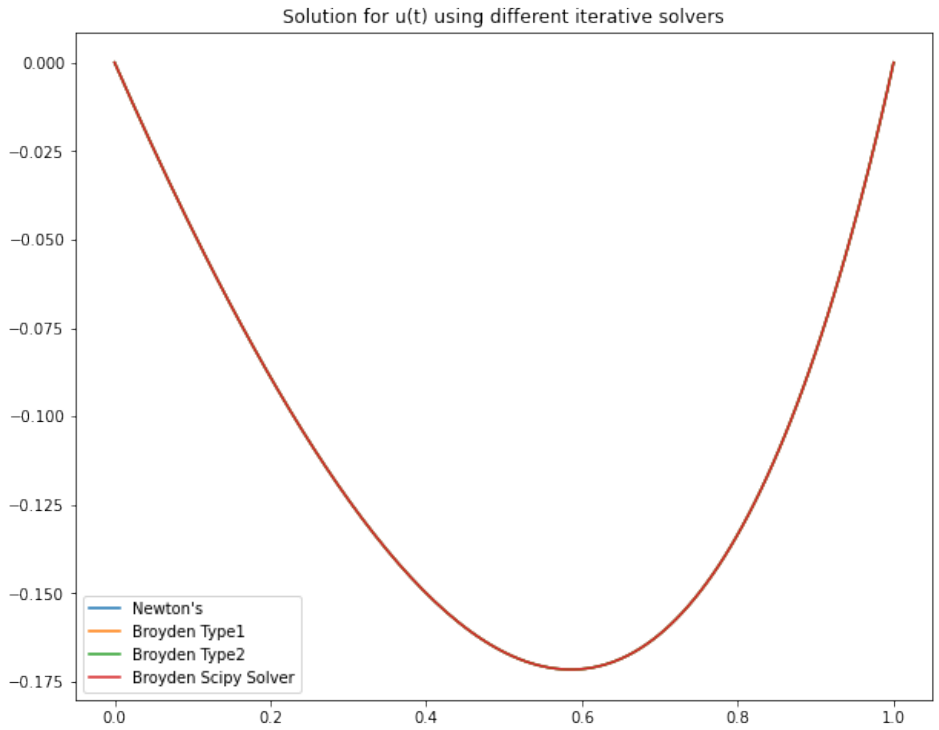

Figure 1. *Solving $u(t)$ fulfilling Equation (4.2) using different solvers. They all achieve extremely similar approximate solutions.*

$$H(s,t) = \begin{cases} s(1-t) & s \leq t \\ t(1-s) & t \leq s \end{cases} \tag{4.3}$$

and $u(0) = u(1) = 0$.

Notice how the problem above does not fit our paradigm of root-finding. This is because the goal of root-finding when applied above is to find a value t that fulfils Equation (4.2). However, our goal is to find a function $u(\cdot)$ that is a continuous mapping from $[0,1]$ to $\mathbb{R}$ that fulfills Equation (4.2). This goal, as it is, is not achievable using our current root-finding algorithms as they are not designed to find continuous functions such as $u : [0,1] \to \mathbb{R}$. However, what we can do is discretize u into n equidistant points at locations $t = t_i = \frac{i}{n+1} = ih$ for $i = 1, \ldots, n$ thus we only approximate u at the points t_i, which means we can't evaluate the integral except at those points as well.

To solve this problem we also need to discretize the integral using the same equidistant mesh for $n-$points at the locations $s = s_i = t_i = \frac{i}{n+1} = ih$ for $i = 1, \ldots, n$ and use a riemann sum. Thus, we can finally define $F : \mathbb{R}^n \to \mathbb{R}^n$ to be a function that takes as input an n-dimensional vector x which represents the values of $u(t_i)$ and outputs an n-dimensional vector which is the approximated values of $G(t_i)$. Concretely

Comparison of Algorithms in different metrics							
Dimension(n)	2^3	2^4	2^5	2^6	2^7	2^8	2^9
	Number of calls to the Function $F(\cdot)$						
Newtons	31	55	103	199	391	775	N/A
Broyden T1	32	38	42	50	48	49	48
Broyden T2	39	39	40	43	43	43	45
Scipy	32	38	42	50	48	49	48
	L2 norm of the function evaluated at the output $F(x_{out})$						
Newtons	1.4e-11	1.9e-11	2.6e-11	3.7e-11	5.2e-11	7.3e-11	N/A
Broyden T1	1.8e-6	1.1e-6	1.7e-6	2.5e-7	3.1e-7	2.0e-7	1.2e-7
Broyden T2	1.1e-6	6.0e-6	2.9e-6	2.3e-7	1.7e-7	2.2e-6	4.0e-7
Scipy	1.8e-6	1.1e-6	1.7e-6	2.5e-7	3.1e-7	2.0e-7	1.2e-7
	CPU clock time in seconds						
Newtons	0.04	0.25	2	14	123	1001	N/A
Broyden T1	0.04	0.18	0.75	3	17	71	348
Broyden T2	0.05	0.18	0.70	3	15	70	324
Scipy	0.04	0.18	0.75	3.9	17	77	336

Table 1

Comparison of results for different algorithms. N/A means the algorithm was terminated due to an execution time exceeding 1 hour.

$$F(x) = \begin{bmatrix} F_1(x) \\ F_2(x) \\ \vdots \\ F_n(x) \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \tilde{u}(t_1) \\ \tilde{u}(t_2) \\ \vdots \\ \tilde{u}(t_n) \end{bmatrix} = \begin{bmatrix} \tilde{u}(h) \\ \tilde{u}(2h) \\ \vdots \\ \tilde{u}(nh) \end{bmatrix}, \quad h = \frac{1}{n+1} \tag{4.4}$$

Where $\tilde{u}(\cdot)$ is an approximate estimate of $u(\cdot)$ on the mesh points, and F_i is the value of $G(t_i)$ evaluated on the mesh points using $\tilde{u}$ and using an integral approximation, $F_i = \tilde{G}(t_i; \tilde{u}) = \tilde{G}(ih; x)$. Concretely

$$F_i(x) = x_i + h * (\sum_{j=1}^{i} t_j(1-t_i)(x_j+t_j+1)^3 \tag{4.5a}$$

$$+ \sum_{j=i+1}^{n} t_i(1-t_j)(x_j+t_j+1)^3) \tag{4.5b}$$

(note that $u(0) = u(1) = 0$ which is why the summation skips the endpoints 0 and n+1)

Thus we have framed the problem of finding a problem $u(\cdot)$ that fulfils $G(t) = 0$ for $t \in [0, 1]$ as a root-finding problem where we require finding roots of F ($F(x) = 0$), that gives n-point approximations $\tilde{u}(t) \approx u(t)$ where higher values of n gives better approximations of u.

4.2.1. Results. We compare the results of Broyden Updates (both Type 1 and Type 2) with the results of using Newton's method as well as an implementation of Broyden Updates provided by Scipy [38].

We can see the solutions of the four different methods for $n = 2^8$ in Figure 1 and all output solutions are extremely similar.

In Table 1 we compare the four algorithms when run on 7 different scales of the problem $n = 8, 16, 32, 64, 128, 256, 512, 1024$ and we compare 3 metrics which are the number of calls to the function F, the $L2$ norm of solution and the CPU clock time. We terminate the simulation when an algorithm takes more than an hour to terminate which is why some experiments have "N/A" in them.

Here we notice the major difference between Newton's method and Broyden's. The number of calls to the underlying function seems to barely grow for Broydens method while for Newton's method it grows linearly with the number of dimensions n. Which is why when we reached $n = 2^9$, while Broyden's algorithm called the function less than 50 times, Newton's was terminated forcefully after taking more than an hour and has called the underlying function more than 3000 times yet still has not converged.

From these experiments we make the suggestion that when considering solving a root-finding problem, we recommend scaling down the problem and using Newton's method. If newton's method successfully find a solution for the scaled down problem then slightly increase the number of dimensions until you reach the desired number of dimensions. However, if newton's method starts to take an extremely long time to converge before reaching the desired number of iterations then that is where Broyden Updates shine and show their advantage.

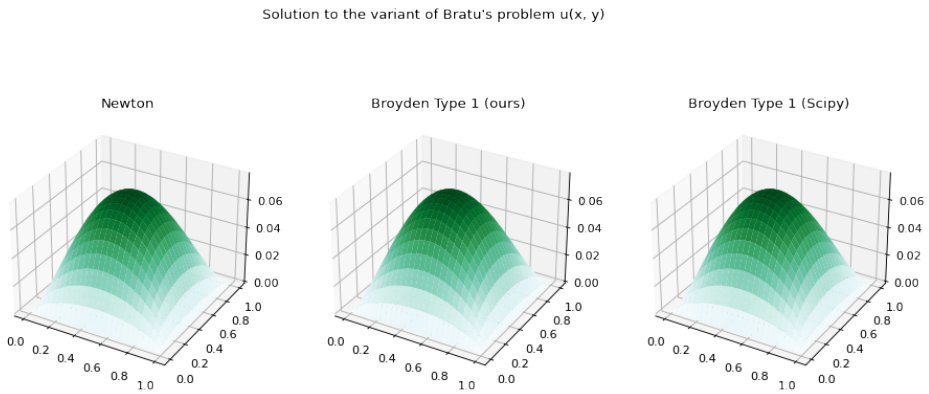

Figure 2. *Solving $u(x, y)$ fulfilling Equation (4.6) using different solvers for $n = 50^2$. They all achieve extremely similar approximate solutions. Type 2 Broyden updates are not shown as the algorithm did not converge.*

4.3. A variant of Bratu's problem. This problems comes from [8] where we attempt to solve the partial differential equation

$$u_{xx} + u_{yy} + u_x + e^u = 0 \tag{4.6}$$

Comparison of Algorithms in different metrics							
Dimension(n)	40^2	50^2	60^2	70^2	80^2	90^2	100^2
	Number of calls to the Function $F(\cdot)$						
Newtons	3205	5005	7205	9805	12805	16205	20005
Broyden T1	858	1054	1278	1578	2000	2552	3243
Broyden T2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Scipy	860	1054	1276	1578	1998	2552	3232
	L2 norm of the function evaluated at the output $F(x_{out})$						
Newtons	1.4e-7	1.8e-7	2.1e-7	2.5e-7	2.8e-7	3.2e-7	3.6e-7
Broyden T1	6.0e-6	6.0e-6	6.0e-6	6.0e-6	6.0e-6	6.0e-6	6.0e-6
Broyden T2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Scipy	5.9e-6	5.9e-6	6.0e-6	6.0e-6	6.0e-6	6.0e-6	6.0e-6
	CPU clock time in seconds						
Newtons	5.4	18	41	113	211	530	812
Broyden T1	1.2	2.4	4.5	18	38	352	564
Broyden T2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Scipy	0.8	1.9	3.9	10	19	57	78

Table 2

Comparison of results for different algorithms. N/A means the algorithm was terminated due to an execution time exceeding 20 minutes or diverging.

Where $u : [0,1]^2 \to \mathbb{R}^2$ and equal to 0 on the boundaries $u(0,y) = u(1,y) = u(x,0) = u(x,1) = 0$

This function is a variant of Bratu's problem [23] with the difference being the u_x term which breaks the standard symmetry of Bratu's problem.

To convert this to a root-finding problem, we use a standard second-order finite difference approximation with an $n \times n$ grid to approximate Equation (4.6).

4.3.1. Results. We compare the results of Broyden Updates (both Type 1 and Type 2) with the results of using Newton's method as well as an implementation of Broyden Updates provided by Scipy [38]. To be exact, the implementation provided by Scipy is type 1 broyden updates which is an important difference for this problem.

We can see the solutions of the four different methods for $n = 50^2$ in Figure 2 and all output solutions are extremely similar. The output for Type 2 is not shown as the algorithm did not converge.

The first thing we notice is that type 2 broyden updates always diverge which is a significant change from Table 2 where the two types were not significantly different in that experiment. We have tried experimenting different values of alpha as discussed in Subsection 3.1 but we could never get the algorithm to converge. To verify that this is not a mistake in our implementation, we have attempted to run the experiments using a Type 2 broyden updates implemented by Scipy (not shown in Table 2) and it has also failed to converge.

In terms of calls to the underlying function, we notice the same pattern as usual where Newtons method needs significantly more calls as we increase the dimensionality of the prob-

lem while Broyden updates increase at a more reasonable pace and this is the major improvement provided by Broyden updates over Newtons method.

In terms of CPU clock time we notice a similar pattern where Newtons method is slower. However, in this problem we notice that Scipy's implementation of the algorithm is an order of magnitude faster than our implementation (despite calling the underlying function the same number of times). Further investigation on the major difference in timing revealed that scipy uses an optimized dot product implementation to calculate the same value we calculate in Algorithm 3.2 and this difference in implementation only shows significantly once the number of iterations of Broyden updates is high enough which is why Table 1 does not show a major difference in execution speed.

5. Broyden Updates Real World Applications. Broyden Updates are used in many different areas of science. Below we briefly go over several examples of it's uses.

An example is calculations in quantum chemistry electron-structures where broyden Updates are used to solve nonlinear equations such as the nuclear coupled-cluster theory and other equations in quantum chemistry [1]. Another example is in power system problems such as reactive power planning [18] or in distribution systems [40]. Another area is the analysis of vertical boiling channel which is of concern to the the safety of nuclear power reactors [21]. It's also used for image deblurring techniques [16], in radiation transfer problems in Astrophysics [25], in determining pressure distribution in gas pipeline networks [33], in nonlinear eigenproblems [13], in the analysis of in-compressible fluid flow in fluid mechanics [7], and in so many other areas of science.

For more examples of uses of Broyden Updates, Pérez and Lopes persent a survey of recent applications [29].

6. Variants of Broyden Updates. Many different variants of Broyden Updates have been developed. Many variants focus on a more memory-efficient implementations such as the Broyden rank reduction method [37], the autoadaptative limited memory method [44], and many others [36, 30, 17, 10].

Other works focus on developing variants of Broyden Updates that decrease the number of iterations and making the algorithm converge faster mostly by using some form of a hybrid model [28, 27, 4, 22, 20].

Some other works focus on modifying the algorithm to handle special conditions such as handling linear constraints using the Lagrange-multiplier technique [32], or extending the algorithm to a non-archimedean framework with the added difficulty of no inner products [6], or guaranteeing the positive definiteness and symmetrical property for the Hessian [19].

Appendix A. My Contribution to Scipy's Implementation of Broyden Updates. During the process of implementing and refining my implementation of Broyden's algorithm, I was trying to optimize the execution speed of my algorithm as much as possible to make it comparable with production level implementations of the algorithm. Interestingly, while looking at the source code for the implementation of Broyden's algorithm in Scipy (The defacto python scientific computing library) to see what sort of tricks they utilized for efficiency, I discovered that in every iteration of both types of Broyden updates, they make an extra unnecessary call to matvec. Which made my implementation actually slightly faster in certain experiments.

After discovering this, I made an attempt to fix the tiny piece of un-optimized code and submitted it to the official Scipy team (pull #16099). As of May 3rd, all the tests have passed and my contribution was officially added to the Scipy master branch.

REFERENCES

[1] A. Baran, A. Bulgac, M. M. Forbes, G. Hagen, W. Nazarewicz, N. Schunck, and M. V. Stoitsov, *Broyden's method in nuclear structure calculations*, Physical Review C, 78 (2008), p. 014318.

[2] M. S. Bartlett, *An inverse matrix adjustment arising in discriminant analysis*, The Annals of Mathematical Statistics, 22 (1951), pp. 107–111.

[3] N. Bićanić and K. H. Johnson, *Who was '–raphson'?*, International Journal for Numerical Methods in Engineering, 14 (1979), pp. 148–152, https://doi.org/https://doi.org/10.1002/nme.1620140112, https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.1620140112, https://arxiv.org/abs/https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.1620140112.

[4] M. Bonković, A. Hace, and M. Cecić, *Modified broyden method for noise visual servoing*, in Proc. Of EUROSIM, 2007.

[5] C. G. Broyden, *A class of methods for solving nonlinear simultaneous equations*, Mathematics of computation, 19 (1965), pp. 577–593.

[6] X. Dahan and T. Vaccon, *On a non-archimedean broyden method*, in Proceedings of the 45th International Symposium on Symbolic and Algebraic Computation, 2020, pp. 114–121.

[7] M. Engelman, G. Strang, and K.-J. Bathe, *The application of quasi-newton methods in fluid mechanics*, International Journal for Numerical Methods in Engineering, 17 (1981), pp. 707–718.

[8] H.-r. Fang and Y. Saad, *Two classes of multisecant methods for nonlinear acceleration*, Numerical linear algebra with applications, 16 (2009), pp. 197–221.

[9] G. H. Golub and C. F. Van Loan, *Matrix computations. johns hopkins studies in the mathematical sciences*, 1989.

[10] R. Haelterman, I. Lahouli, M. Shimoni, and J. Degroote, *Limited memory switched broyden method for faster image deblurring*, in 2017 Fifteenth IAPR International Conference on Machine Vision Applications (MVA), IEEE, 2017, pp. 366–369.

[11] W. Heitzinger, I. Troch, and G. Valentin, *Praxis nichtlinearer Gleichungen: mit zahlreichen Anwendungsbeispielen für Ingenieure, Mathematiker und Naturwissenschaftler*, Hanser, 1985.

[12] A. S. Householder, *The numerical treatment of a single nonlinear equation*, McGraw-Hill, 1970.

[13] E. Jarlebring, *Broyden's method for nonlinear eigenproblems*, SIAM Journal on Scientific Computing, 41 (2019), pp. A989–A1012.

[14] M. Kawata, C. Cortis, and R. Friesner, *Efficient recursive implementation of the modified broyden method and the direct inversion in the iterative subspace method: Acceleration of self-consistent calculations*, The Journal of chemical physics, 108 (1998), pp. 4426–4438.

[15] E. Kvaalen, *A faster broyden method*, BIT Numerical Mathematics, 31 (1991), pp. 369–372.

[16] I. Lahouli, R. Haelterman, J. Degroote, M. Shimoni, G. De Cubber, and R. Attia, *Accelerating existing non-blind image deblurring techniques through a strap-on limited-memory switched broyden method*, IEICE TRANSACTIONS on Information and Systems, 101 (2018), pp. 1288–1295.

[17] I. Lahouli, R. Haelterman, J. Degroote, M. Shimoni, G. De Cubber, and R. Attia, *Accelerating existing non-blind image deblurring techniques through a strap-on limited-memory switched broyden method*, IEICE TRANSACTIONS on Information and Systems, 101 (2018), pp. 1288–1295.

[18] L. L. Lai and J. Ma, *Application of evolutionary programming to reactive power planning-comparison with nonlinear programming approach*, IEEE Transactions on power systems, 12 (1997), pp. 198–206.

[19] S. S. Mahmood and N. S. Muhanah, *Symmetric and positive definite broyden update for unconstrained optimization*, Baghdad Science Journal, 16 (2019).

[20] M. Mamat, I. Mohd, and L. W. June, *Hybrid broyden method for unconstrained optimization*, (2009).

[21] M. Mirzaee, A. Zolfaghari, A. Minuchehr, and M. Aghaie, *A drift-flux analysis of the diversely heated channel using the broyden method*, Applied Thermal Engineering, 150 (2019), pp. 464–481.

[22] H. Mohammad and M. Y. Waziri, *On broyden-like update via some quadratures for solving nonlinear systems of equations*, Turkish Journal of Mathematics, 39 (2015), pp. 335–345.
[23] A. Mohsen, *A simple solution of the bratu problem*, Computers & Mathematics with Applications, 67 (2014), pp. 26–33.
[24] J. J. Moré and M. Y. Cosnard, *Numerical solution of nonlinear equations*, ACM Transactions on Mathematical Software (TOMS), 5 (1979), pp. 64–85.
[25] S. Nicolas, L. Bigarré, and F. Paletou, *Broyden's method for the solution of the multilevel non-lte radiation transfer problem*, Astronomy & Astrophysics, 527 (2011), p. A1.
[26] J. Nocedal and S. J. Wright, *Numerical optimization*, Springer, 1999.
[27] I. Osinuga and S. Yusuff, *Construction of a broyden-like method for nonlinear systems of equations*, Annals. Computer Science Series, 15 (2017), pp. 128–135.
[28] I. A. Osinuga and S. O. Yusuff, *Quadrature based broyden-like method for systems of nonlinear equations*, Statistics, Optimization & Information Computing, 6 (2018), pp. 130–138.
[29] R. Pérez and V. L. R. Lopes, *Recent applications and numerical implementation of quasi-newton methods for solving nonlinear systems of equations*, Numerical Algorithms, 35 (2004), pp. 261–285.
[30] M. B. Reed, *L-broyden methods: a generalization of the l-bfgs method to the limited-memory broyden family*, International Journal of Computer Mathematics, 86 (2009), pp. 606–615.
[31] W. C. Rheinboldt, *Methods for solving systems of nonlinear equations*, SIAM, 1998.
[32] A. Rodríguez-Ferran and A. Huerta, *Adapting broyden method to handle linear constraints imposed via lagrange multipliers*, International Journal for Numerical Methods in Engineering, 46 (1999), pp. 2011–2026.
[33] K. A. Sidarto, A. Kania, L. Mucharam, R. A. Widhymarmanto, et al., *Determination of gas pressure distribution in a pipeline network using the broyden method.*, Journal of Engineering & Technological Sciences, 49 (2017).
[34] G. W. Stewart, *Introduction to matrix computations*, Elsevier, 1973.
[35] J. F. Traub, *Iterative methods for the solution of equations*, vol. 312, American Mathematical Soc., 1964.
[36] B. van de Rotten and S. V. Lunel, *A limited memory broyden method to solve high-dimensional systems of nonlinear equations*, in EQUADIFF 2003, World Scientific, 2005, pp. 196–201.
[37] B. van de Rotten and S. V. Lunel, *A memory-efficient broyden method to compute fixed points of non-linear maps arising in periodically forced processes*, IMA Journal of Applied Mathematics, 80 (2015), pp. 585–607.
[38] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python*, Nature Methods, 17 (2020), pp. 261–272, https://doi.org/10.1038/s41592-019-0686-2.
[39] T. Yamamoto, *Historical developments in convergence analysis for newton's and newton-like methods*, Journal of Computational and Applied Mathematics, 124 (2000), pp. 1–23, "https://reader.elsevier.com/reader/sd/pii/S0377042700004179?token=3D8949FFB231E2EEA27B25336E535706C5B60E294C4110C14072E7BE7139B388BBC00EBE078245BD948C65AEB16ABDA originRegion=us-east-1&originCreation=20220501192826".
[40] H. Yang, F. Wen, and L. Wang, *Newton-raphson on power flow algorithm and broyden method in the distribution system*, in 2008 IEEE 2nd International Power and Energy Conference, IEEE, 2008, pp. 1613–1618.
[41] D. M. Young, *Iterative solution of large linear systems*, Elsevier, 1971.
[42] T. J. Ypma, *Historical development of the newton–raphson method*, SIAM review, 37 (1995), pp. 531–551.
[43] F. J. Zeleznik, *Quasi-newton methods for nonlinear equations*, Journal of the ACM (JACM), 15 (1968), pp. 265–271.
[44] M. Ziani and F. Guyomarc'h, *An autoadaptative limited memory broyden's method to solve systems of nonlinear equations*, Applied mathematics and computation, 205 (2008), pp. 202–211.