UNIVERSIDADE ESTADUAL DE CAMPINAS

Faculdade de Engenharia Civil, Arquitetura e Urbanismo

IC639: Métodos Numéricos para Engenharia Civil

List 5

Numerical Solutions of Nonlinear Equations

Student:

Carlos Henrique Chama Puga - 195416

Advisors:

Porf. Dr. Philippe Devloo

Dr. Giovane Avancini

Campinas

2024

Contents

1 Introduction

The task of solving a nonlinear system of equations is not trivial. In general, what is done is to linearize the system and solve it. However, there are times when it is not possible or, the linearization does not yield a satisfactory result. In these cases, iterative methods are applied to solve the nonlinear problem numerically.

In this work, two methods are presented to solve nonlinear systems of equations: Newton's Method and the Quasi-Newton Method. The first method is a generalization of the Newton-Raphson method for one variable problem. It consists, in one dimension, of finding a function ϕ such that

$$g(x) = x - \phi(x)f(x) \tag{1.1}$$

gives a quadratic convergence rate to the solution. In Eq. (1.1), $g(x)$ is the approximated solution and $f(x)$ is the function that we want to find the root evaluated at x. In this context, function ϕ is chosen to be the inverse of the derivative of $f(x)$, assuming that $f'(x)$ is not zero.

Newton's method can be extended to multidimensional problems. In this case, Eq. (1.1) becomes

$$\boldsymbol{G}(\boldsymbol{x}) = \boldsymbol{x} - A(\boldsymbol{x})^{-1}\boldsymbol{F}(\boldsymbol{x}), \tag{1.2}$$

in which the matrix $A(\boldsymbol{x})$ is given by the derivatives of the functions that com-

pose the system of equations

$$A(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \frac{\partial F_n}{\partial x_2} & \cdots & \frac{\partial F_n}{\partial x_n} \end{bmatrix}. \tag{1.3}$$

Matrix $A(x)$ is also known as the Jacobian matrix. In this context, the Eq. (1.2) is rewritten and Newton's method is given by

$$\boldsymbol{x}^k = \boldsymbol{x}^{k-1} - J(\boldsymbol{x}^{k-1})^{-1}\boldsymbol{F}(\boldsymbol{x}^{k-1}), \tag{1.4}$$

where $J(\boldsymbol{x})$ is the Jacobian matrix evaluated at $\boldsymbol{x}$. Attention to the fact that the linear system of equations $J(\boldsymbol{x}^{k-1})^{-1}\boldsymbol{F}(\boldsymbol{x}^{k-1})$ must be solved in each iteration of Newton's method.

Broyden's method comes as an alternative to Newton's method in which the Jacobian matrix is not calculated, being replaced by an approximation matrix that is updated in each iteration. This method belongs to a family of methods called Quasi-Newton methods.

Broyden's method idea lies in the fact a first guess $\boldsymbol{x}^0$ is given as the system's solution. Then, the next approximation $\boldsymbol{x}^1$ is evaluated by the Newton's method

$$\boldsymbol{x}^1 = \boldsymbol{x}^0 - J(\boldsymbol{x}^0)^{-1}\boldsymbol{F}(\boldsymbol{x}^0),$$

from $\boldsymbol{x}^2$ on, Newton's method is not applied, instead, the Secant method is generalized to multidimensional problems. The Secant method approximates

the derivative of $f(x^1)$ by

$$f'(x^1) \approx \frac{f(x^1) - f(x^0)}{x^1 - x^0}. \tag{1.5}$$

For multidimensional problems, however, $\boldsymbol{x}^k$ and $\boldsymbol{x}^{k-1}$ are vectors. Therefore, the derivative of $\boldsymbol{F}(\boldsymbol{x})$ is evaluated by Eq. (1.3), which leads to

$$A_1(\boldsymbol{x}^1 - \boldsymbol{x}^0) = \boldsymbol{F}(\boldsymbol{x}^1) - \boldsymbol{F}(\boldsymbol{x}^0), \tag{1.6}$$

where matrix A_1 can be defined as

$$A_1 = J(\boldsymbol{x}^0) + \frac{\left[\boldsymbol{F}(\boldsymbol{x}^1) - \boldsymbol{F}(\boldsymbol{x}^0) - J(\boldsymbol{x}^0)\left(\boldsymbol{x}^1 - \boldsymbol{x}^0\right)\right]\left(\boldsymbol{x}^1 - \boldsymbol{x}^0\right)^t}{||\boldsymbol{x}^1 - \boldsymbol{x}^0||^2},$$

and $\boldsymbol{x}^2$ can be obtained by

$$\boldsymbol{x}^2 = \boldsymbol{x}^1 - A_1^{-1}\boldsymbol{F}(\boldsymbol{x}^1).$$

Broyden's method is then repeated to determine the solution until n iterations are reached or the norm of the residual is smaller than a given tolerance. Generalizing the method, the update of the approximation matrix A is given by

$$A_{k+1} = A_k + \frac{\boldsymbol{y}_k - A_k\boldsymbol{s}_k}{||\boldsymbol{s}_k||^2}\boldsymbol{s}_k^t, \tag{1.7}$$

where $\boldsymbol{y}_k = \boldsymbol{F}(\boldsymbol{x}^{k+1}) - \boldsymbol{F}(\boldsymbol{x}^k)$ and $\boldsymbol{s}_k = \left(\boldsymbol{x}^{k+1} - \boldsymbol{x}^k\right)$. The approximated solution $\boldsymbol{x}^{k+1}$ is obtained by

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - A_{k+1}^{-1}\boldsymbol{F}(\boldsymbol{x}^k). \tag{1.8}$$

List 5 aims to implement Newton's and Broyden's methods to solve two

systems of equations. The first system is given by

$$1. \begin{cases} e^{xy} + x^2 + y = -1.2 \\ x^2 + y^2 + x = 0.55 \end{cases}, \tag{1.9}$$

and the second system is given by

$$2. \begin{cases} -x \cos y = 1 \\ xy + z = 2 \\ e^{-z} \sin x + y + x^2 + y^2 = 1 \end{cases}. \tag{1.10}$$

In the next sections, the methods are implemented and the results are discussed. The following bibliography is referred to in this work: (1, 2).

2 NonLinearSolver Implementation

The implementation of the NonLinearSolver class is the result of the theoretical background presented in the previous section. The fields of the class are presented in Code 1.

```
@dataclass
class NonLinearSolver:
    equations: list[callable] = field(default_factory=list)
    gradients: list[callable] = field(default_factory=list)
    x0: list[float] = field(default_factory=list)

    max_iter: int = 50
    tolerance: float = 1e-15

    x_list: list[list] = field(init=False, default_factory=list)
    diff: list[float] = field(init=False, default_factory=list)
    diff_log: list[float] = field(init=False, default_factory=list)
    residual: list[float] = field(init=False, default_factory=list)

```

```
    exact_solution: list[float] = field(default_factory=list)
```

Code Listing 1: NonLinearSolver class fields

In Code 1, equations are the nonlinear system of equations to be solved, gradients are the gradients of the equations, x0 is the initial guess for the solution, max_iter is the maximum number of iterations, and tolerance is the convergence criterion. The fields x_list, diff, and diff_log, residual are used to store the results of the solver. The exact_solution field stores the exact solution for the problem, in case of having one.

The class methods are SetMaxIteration, to set the max number of iterations, SetMethod, to set whether the solver will use the Newton or the Broyden method, SetExactSolution, to set the exact solution for the problem, and GetError to get the list of differences, logarithm differences, and residuals. The implementation of the Newton method is presented in Code 2.

```
def Newton(self) -> None:
    xval = self.x0

    if any(self.exact_solution):
            self.diff.append(NORM(xval - self.exact_solution))

    self.SaveResidual(xval)

    for _ in range(self.max_iter):
        G = ARRAY(list(map(lambda eq: eq(*xval), self.equations)))
        Grad_G = ARRAY(list(map(lambda grad: grad(*xval), self.gradients)))

        x_next = xval - LINSOLVE(Grad_G, G)

        self.x_list.append(x_next)
        self.SaveResidual(x_next)

        if any(self.exact_solution):
            self.diff.append(NORM(x_next - self.exact_solution))
```

```
20
21          if NORM(self.residual[-1]) < self.tolerance:
22              break
23
24          xval = x_next
```

Code Listing 2: Newton method implementation

In line 2, xval is set to the initial guess. In case of having an exact solution, line 5 evaluates the difference between it and the initial guess. Line 7 saves the residual, i.e. the system evaluated at $\boldsymbol{x}^k$ to plot the convergence. Lines 10 and 11 evaluate the equations and their gradients at xval. Line 13 solves the linear system of equations using the LINSOLVE function, from numpy module, updating the solution. If the tolerance, 10^{-15} by default, is reached the loop is broken. If the exact solution is set, the difference between the current solution and the exact solution is stored in the diff list.

The Broyden method is implemented in Code 3.

```
1  def Broyden(self) -> None:
2      v0 = self.x0.copy()
3
4      G0 = ARRAY(list(map(lambda eq: eq(*v0), self.equations)))
5      Grad_G0 = ARRAY(list(map(lambda grad: grad(*v0), self.gradients)))
6
7      v1 = v0 - LINSOLVE(Grad_G0, G0)
8
9      self.x_list.append(v1)
10     self.SaveResidual(v1)
11
12     G1 = ARRAY(list(map(lambda eq: eq(*v1), self.equations)))
13
14     del_x = v1 - v0
15
16     if any(self.exact_solution):
17         self.diff.append(NORM(v1 - self.exact_solution))
18
```

```
19      del_G = G1 - G0
20
21      for _ in range(self.max_iter):
22          grad_G1 = Grad_G0 + OUTER(del_G - Grad_G0 @ del_x, del_x) / (del_x
    @ del_x)
23
24          Grad_G0 = grad_G1
25
26          xnext = v1 - LINSOLVE(grad_G1, G1)
27
28          v0 = v1
29          v1 = xnext
30
31          self.SaveResidual(v1)
32
33          G0 = G1
34          G1 = ARRAY(list(map(lambda eq: eq(*v1), self.equations)))
35
36          del_x = v1 - v0
37          del_G = G1 - G0
38
39          self.x_list.append(v1)
40
41          if any(self.exact_solution):
42              self.diff.append(NORM(v1 - self.exact_solution))
43
44          if NORM(self.residual[-1]) < self.tolerance:
45              break
```

Code Listing 3: Broyden method implementation

Similarly to Newton's method, in lines 4 and 5 the equations and gradients are evaluated at the initial guess. The second guess is calculated in lines 7 and 8, following Newton's method procedure. Line 12 calculates the system of equations at the second guess and line 14 calculates the difference between the two guesses.

The Broyden method loop starts at line 21, with the evaluation of the gradi-

ent of the system of equations at the second guess, but using the initial guess gradient. This is the main difference between the Newton and Broyden methods. Line 26 solves the linear system of equations and updates the solution. The loop continues until the tolerance or the maximum number of iterations is reached.

Additionally, a smaller method is implemented to coordinate the solver, as shown in Code 4.

```
def Solve(self) -> None:
    self.method()

    for i in range(1, len(self.diff)):
        self.diff_log.append(LOG(self.diff[i]) / LOG(self.diff[i-1]))
```

Code Listing 4: NonLinearSolver method

The Solve method not only calls correctly either the Newton or Broyden method but also calculates the logarithm of the differences between the solutions. An example of the usage of the NonLinearSolver class is presented in Code 5.

```
def main():
    n_iter = 10

    eq1 = lambda x, y: E ** (x * y) + x ** 2 + y - 1.2
    eq2 = lambda x, y: x ** 2 + y ** 2 + x - 0.55

    grad_eq1 = lambda x, y: [y * E ** (x * y) + 2 * x, x * E ** (x * y) +
    1]
    grad_eq2 = lambda x, y: [2 * x + 1, 2 * y]

    G = [eq1, eq2]
    grad_G = [grad_eq1, grad_eq2]

    x0 = [.1 for _ in range(2)]

    solver_newton = NonLinearSolver(G, grad_G, x0)
```

```
16    solver_newton.SetMaxIteration(n_iter)
17    solver_newton.SetMethod("Newton")
18
19    solver_newton.Solve()
20
21    solver_broyden = NonLinearSolver(G, grad_G, x0)
22    solver_broyden.SetMaxIteration(n_iter)
23    solver_broyden.SetMethod("Broyden")
24
25    solver_broyden.Solve()
```

Code Listing 5: NonLinearSolver example

Lines 4 to 11 define the system of equations and their gradients. Line 13 declares the initial guess, while line 14 sets the exact solution. Lines 16 to 21 create the NonLinerSolver object and set the parameters for the Newton method. The same is done by lines 23 - 28 for the Broyden method. The errors can be accessed and plotted using the GetError method and the matplotlib module.

3 Results

This section presents the main results for both Newton and Broyden methods. The initial guess and number of iterations vary according to the system of equations but are the same for both methods. Tolerance is set to 10^{-15} for both examples.

3.1 First Nonlinear System

The first nonlinear system given Eq. (1.9) uses $x_0 = \{0.1, 0.1\}$ as the initial guess, and has 10 iterations for both methods. Table 3.1 presents the results for the first system using Newton and Broyden methods.

Table 3.1: Results for the first nonlinear system

Newton's Method			
Iteration	x	y	residual
0	0.1	0.1	0.43736
1	0.4627600085	0.0734399488	0.18007
2	0.3969527820	0.0354011946	0.00917
3	0.3938604085	0.0321511366	2.88e-05
4	0.3938494529	0.0321427390	2.910e-10
5	0.3938494528	0.0321427389	2.220e-16
Broyden's Method			
Iteration	x	y	residual
0	0.1	0.1	0.43736
1	0.4627600085	0.0734399488	0.18007
2	0.3856395478	0.0055231374	0.04633
3	0.3902269923	0.0344443408	0.00631
4	0.3941767806	0.0314940531	0.00083
5	0.3938644022	0.0321212671	3.09e-05
6	0.3938492796	0.0321429715	3.47e-07
7	0.3938494532	0.032142738	8.49e-10
8	0.3938494528	0.0321427389	6.84e-13
9	0.3938494528	0.0321427389	1.11e-16

Figure 3.1 depicts the convergence for both methods. Newton's method converges in 5 iterations, while Broyden's takes almost the double, 9 iterations. Both methods achieve the same solution, $x = 0.3938494528$ and $y = 0.0321427389$, with a residual of the order of 10^{-16}.

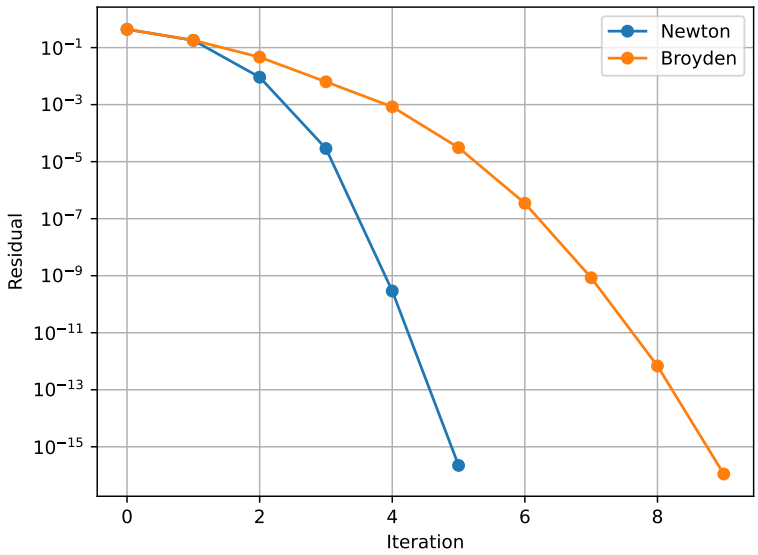

Figure 3.1: Convergence for the first nonlinear system

3.2 Second Nonlinear System

The second nonlinear system given Eq. (1.10) uses $x_0 = \{0.1, 0.1, 0.1\}$ as the initial guess, and has 30 iterations for both methods. Table 3.2 presents the results for the second system using Newton and Broyden methods.

Table 3.2: Results for the second nonlinear system.

Newton's Method				
Iteration	x	y	z	Residual
0	0.1	0.1	0.1	2.32838
1	-0.98379345166	2.2156661291	1.88681273224	5.74369
2	1.39885410699	2.0132350005	-1.29854408331	4.07938
3	3.15367430755	1.73619021977	-3.96154226685	39.75890
4	2.58232829725	1.92098936848	-3.06620946128	11.63550
5	2.20584340512	2.02126203014	-2.49633866143	2.78584
6	2.04009533921	2.07754843321	-2.2477262380	0.36281
7	2.01019345695	2.09121239898	-2.2041500598	0.00896
8	2.00936979322	2.0917047033	-2.20300865265	5.50e-06
9	2.00936923461	2.09170513415	-2.2030079444	1.81e-12
10	2.00936923461	2.0917051341	-2.20300794446	4.44e-16

Table 3.2: Results for the second nonlinear system (continued).

Broyden's Method				
Iteration	x	y	z	Residual
0	0.1	0.1	0.1	
1	-0.98379345166	2.2156661291	1.88681273224	2.32838
2	-2.86800321300	-0.736524949	5.02933927946	5.74369
3	-0.96629288250	-0.639022704	1.14171550902	9.38550
4	-1.14214386855	-0.772153622	1.19899623589	0.32993
5	-1.27546528568	-1.086899010	0.80710460716	0.64812
6	-1.06426239703	-0.520529040	1.50883577012	1.56098
7	-1.06810147286	-0.383391728	1.63192385971	0.20765
8	-1.05115463611	-0.351985295	1.63848950012	0.10285
9	-1.00884044531	-0.111511495	1.84570543100	0.04046
10	-1.04138584935	-0.224753681	1.75881524320	0.11954
11	-1.01392825338	0.3424018291	2.29684285546	0.03378
12	-1.03006301317	-0.094337434	1.89731370887	0.10673
13	-1.02203741634	0.0573997855	2.05329309938	0.07035
14	-0.99440018782	0.7406037349	2.75706745631	0.06132
15	-1.02471257131	0.1091210711	2.10797608681	0.58571
16	-1.02143379112	0.1486759010	2.15081073230	0.03928
17	-1.02190742245	0.2041423038	2.20724613288	0.02584
18	-1.01907494179	0.2027702169	2.20770475416	0.00591
19	-1.02102506714	0.1970132648	2.20050276823	0.00215
20	-1.02025912697	0.1996163781	2.20366469063	0.00143
21	-1.02025768870	0.1996113540	2.20365643426	1.09e-05
22	-1.02025684824	0.1995979744	2.20363943856	4.50e-06
23	-1.020257309945	0.1996053765	2.20364884435	5.57e-06
24	-1.02025730998	0.1996053775	2.20364884555	6.70e-10
25	-1.02025730998	0.1996053775	2.20364884553	1.61e-13
26	-1.02025730998	0.1996053775	2.20364884553	3.30e-16

Figure 3.2 shows the convergence for both methods. Newton's method converges in 10 iterations, while Broyden's takes 26 iterations. Differently from

the first system, the methods do not converge to the same solution. Newton's method converges to $x = 2.00936923461$, $y = 2.0917051341$, and $z = -2.20300794446$, while Broyden's method converges to $x = -1.02025730998$, $y = 0.1996053775$, and $z = 2.20364884553$. Regardless of the achieved solution, the residuals are of the order of 10^{-16} for both methods.

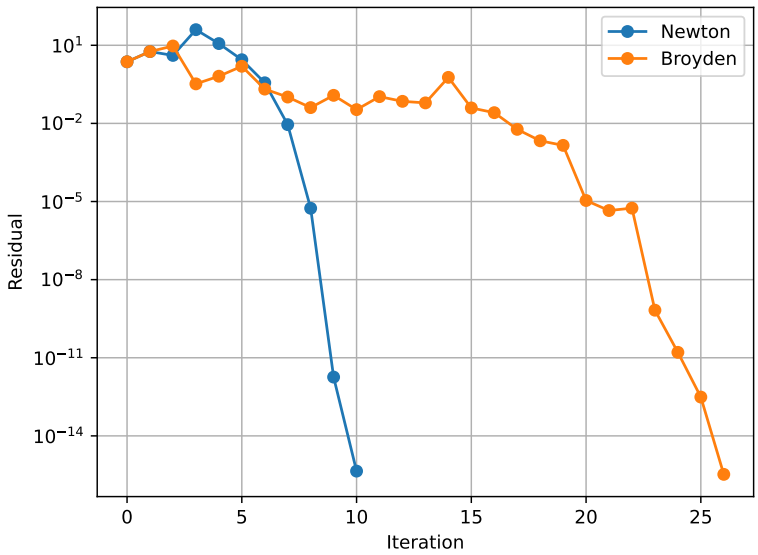

Figure 3.2: Convergence for the second nonlinear system

Both solutions are easily verified by substituting the values into the system of equations.

4 Conclusions

This work, by solving two different nonlinear systems of equations, presented the Newton and Broyden methods. For the same initial guess, the first system yielded the same solution for both methods, while the second system showed a different solution for each method. This fact evidences the importance and dependence of the initial guess for the solution to converge to one answer or another.

Newton's method is also influenced by the Jacobian matrix, which takes into account the derivatives of the functions that compose the system of equations. Broyden's method, on the other hand, does not calculate the Jacobian matrix, but approximates it by a matrix that is updated in each iteration, which makes it more flexible and less computationally expensive. The difference in terms of computational effort for a single iteration is n^2+n scalar evaluations plus $O(n^3)$ arithmetic operations for Newton's method, while Broyden's method requires n scalar evaluations plus $O(n^2)$ arithmetic operations.

Despite all the computational effort, the results show that the Newton method converges faster than Broyden's method. Because an approximation is made for the Jacobian matrix, Broyden's method loses the quadratic convergence, instead having the so-called superlinear convergence.

In conclusion, both methods are efficient for solving nonlinear systems of equations. Each one has its advantages and disadvantages, and the choice of which one to use needs to be made according to the problem to be solved and the resources available. In the given cases, Newton's method allegedly performed better, but this might not be the case for other equations.

References

1 CUNHA, M. C. de C. *Métodos numéricos*. [S.l.]: Editora da UNICAMP, 2000.

2 BURDEN, R. L.; FAIRES, J. D. *Numerical analysis, brooks*. [S.l.]: Cole publishing company Pacific Grove, CA:, 1997.

A GitHub Repository

The source code for this report and every code mentioned can be found in the following GitHub repository: CarlosPuga14/MetodosNumericos_2024S1.