Θέματα Αριθμητικής Ανάλυσης

Μέθοδος Newton-Raphson για επίλυση μη-γραμμικών εξισώσεων.

Γιώργος Ματλής

2 Ιουνίου 2022

1 Εισαγωγή

Η μέθοδος Newton-Raphson είναι μια αριθμητική επαναληπτική μέθοδος που μπορεί να χρησιμοποιηθεί για την επίλυση μη-γραμμικών εξισώσεων. Η διαδικασία εύρεσης της ρίζας ξεκινάει έχοντας μια προσεγγιστική λύση της πραγματικής λύσης και, σε κάθε επανάληψη, ανανεώνει την προσέγγιση μέχρι το τοπικό σφάλμα της επανάληψης να είναι μικρότερο απο μια τιμή ανοχής. Παρακάτω περιγράφεται η μέθοδος για την επίλυση n μη-γραμμικών εξισώσεων με n αγνώστους.

2 Περιγραφή Μεθόδου

Η μέθοδος Newton-Raphson στην γενική μορφή της έχει n εξισώσεις με n αγνώστους

$$\begin{cases} f_1(x_1, x_2, ..., x_n) = 0, \\ f_2(x_1, x_2, ..., x_n) = 0 \\ . \\ . \\ . \\ f_n(x_1, x_2, ..., x_n) = 0 \end{cases} \tag{1}$$

όπου $f_i(x_1, x_2, ..., x_n) : R^n \longrightarrow R, i = 1, ..., n$ είναι μη-γραμμικές συναρτήσεις. Το (1) μπορεί να γραφεί ως

$$\mathbf{f(x) = 0} \tag{2}$$

Έχοντας μία αρχική προσέγγιση $\mathbf{x}^0$ της ακρριβής λύσης, θέλουμε να βρούμε ένα $\Delta\mathbf{x}$ ώστε $x^0 + \Delta x = x^*$, η ρίζα της (2), δηλαδή να ισχύει $\mathbf{f}(\mathbf{x}^0 + \Delta\mathbf{x}) = \mathbf{f}(\mathbf{x}^*) = 0$. Η εύρεση του Δx γίνεται διαμέσου της επαναληπτικής διαδικασίας, όπου σε κάθε επανάληψη $k = 1, 2, 3, ..., n$ υπολογίζουμε διαδοχικές προσεγγίσεις Δx και βελτιώνουμε τη λύση $x^{k+1} = x^k + \Delta x^k$.Χρησιμοποιώντας τις σειρές Taylor πρώτου βαθμού για τη λύση x^* και μικρό Δx έχουμε.

$$0 = f(\mathbf{x}^*) = f(\mathbf{x}) + (\mathbf{x}^* - \mathbf{x})\mathbf{J}(x) + O(\|\mathbf{x}^* - \mathbf{x}\|^2) \tag{3}$$

όπου $\mathbf{J}$ είναι ο Ιακωβιανός πίνακας.

$$\begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \dots & \frac{\partial f_1(\mathbf{x})}{\partial x_i} \\ . & . & . \\ . & . & . \\ . & . & . \\ \frac{\partial f_n(\mathbf{x})}{\partial x_1} & \dots & \frac{\partial f_n(\mathbf{x})}{\partial x_i} \end{bmatrix} \tag{4}$$

Παραλείποντας τους τετραγωνικούς όρους μπορούμε να καταλήξουμε στο $0 = f(\mathbf{x}) + \Delta\mathbf{x}\mathbf{J}(\mathbf{x})$ όπου $\Delta\mathbf{x} = \mathbf{x}^* - \mathbf{x}$, απο όπου προκύπτει ότι

$$\Delta\mathbf{x} \approx -[\mathbf{J}]^{-1}\mathbf{f}(\mathbf{x}). \tag{5}$$

Η (5) μας βοηθάει να ορίσουμε την γενική επαναληπτική μέθοδο

$$\mathbf{x}^{k+1} = \mathbf{x}^k - [\mathbf{J}^k]^{-1} f(\mathbf{x}^k) \tag{6}$$

όπου

- k είναι ο δείκτης επανάληψης.
- $\mathbf{x}^{k+1}$ είναι η προσεγγιστική λύση της k επανάληψης.
- $\mathbf{J}$ ο Ιακωβιανός πίνακας.
- $\mathbf{f}(\mathbf{x})$ η συνάρτηση της (1).

Ένας αλγόριθμος με τα βήματα υλοποίησης της επαναληπτικής μεθόδου μπορεί να γραφτεί ως

1. Θέσε τον δείκτη επανάληψης $k = 0$, θεώρησε αρχική προσέγγιση $\mathbf{x}^0$ και την τιμή ανοχής ϵ.
2. Για κάθε k υπολόγισε το $\mathbf{x}^{\kappa+1}$ χρησιμοποιώντας
   (a) Τον Ιακωβιανό πίνακα $\mathbf{J}^k(\mathbf{x}^k)$.
   (b) Λύσε το διάνυσμα $\mathbf{f}(\mathbf{x}^k)$.
3. Υπολόγησε το τοπικό σφάλμα $x^{k+1} - x^k$ και αν ισχύει $\|x^{k+1} - x^k\|_2 < \epsilon$ τότε η λύση είναι η x^{k+1} και η μέθοδος έχει συγκλίνει. Διαφορετικά προχώρα στο βήμα 4.
4. Αύξησε τον δείκτη επανάληψης $k = k + 1$ και συνέχισε απο το βήμα 2.

3 Εκτέλεση

Έχουμε το σύστημα δύο εξισώσεων με δύο αγνώστους

$$\begin{cases} f_1(x, y) = x + x * y - 4, \\ f_2(x, y) = x + y - 3 \end{cases} \tag{7}$$

Η λύση του συστήματος είναι $(x^*, y^*) = (2, 1)$, οπότε ισχύει

$$\begin{cases} f_1(x^*, y^*) = 0, \\ f_2(x^*, y^*) = 0 \end{cases} \tag{8}$$

Θέλουμε να βρούμε προσεγγιστικές λύσεις x, y των πραγματικών λύσεων x^* και y^* ώστε $f_1(x, y) = 0$ και $f_2(x, y) = 0$. Ξεκινώντας με αρχική προσέγγιση $[x^0 = 1.98,\ y^0 = 1.02]$, τιμή ανοχής $\epsilon = 1e-8$ και μέγιστη επανάληψη $k = 200$, εκτελώντας τον προηγούμενο αλγόριθμο έχουμε τα παρακάτω αποτελέσματα

Table 1: Αποτελέσματα Εκτέλεσης μεθόδου

Επανάληψη k	x_k	y_k
0	1.9800000000000000	1.02000000000000
1	1.9899999999999989	1.0100000000000011
2	1.995000000000009	1.004999999999991
3	1.9975000000000076	1.0024999999999924
4	1.998749999999964	1.001250000000036
5	1.9993750000001445	1.0006249999998555
6	1.9996875000001382	1.0003124999998618
7	1.9998437500005233	1.0001562499994767
8	1.9999218750002323	1.0000781249997677
9	1.9999609374993617	1.0000390625006383
10	1.99998046874997	1.00001953125003
11	1.9999902343811033	1.0000097656188967
12	1.9999951171883603	1.0000048828116397
13	1.999997558636368	1.000002441363632
14	1.9999987793814664	1.0000012206185336
15	1.9999993895127928	1.0000006104872072
16	1.999999695035113	1.000000304964887
17	1.9999998479358863	1.0000001520641137
18	1.9999999238664867	1.0000000761335133
19	1.9999999617811939	1.0000000382188061
20	1.999999985020497	1.000000014979503
21	1.9999999998437261	1.0000000001562739
22	1.9999999998437261	1.0000000001562739

Από τον παραπάνω πίνακα συμπεραίνουμε ότι η προσεγγιστική λύση του μη-γραμμικου συστήματος στην 22η επανάληψη είναι $x = 1.1.9999999998437261$ και $y = 1.0000000001562739$. Οι επαναλήψεις σταματάνε στην 22η επανάληψη τα αποτελέσματα της οποίας είναι ίδια με την 21η. Έτσι το σφάλμα γίνεται μηδενικό. Αυτό προκύπτει από την έλλειψη ακρίβειας της γλώσσας προγραμματισμού Python. Τα τοπικά προσεγγιστικά σφάλματα για κάθε επανάληψη υπολογίζονται με βάση τον τύπο

$$e_k = [(x^{k+1} - x^k)^2 + (y^{k+1} - y^k)^2]^{1/2} = \|x^{k+1} - x^k\|_2 \tag{9}$$

Τα αποτελέσματα των σφαλμάτων μπορείται να τα βρείτε παρακάτω

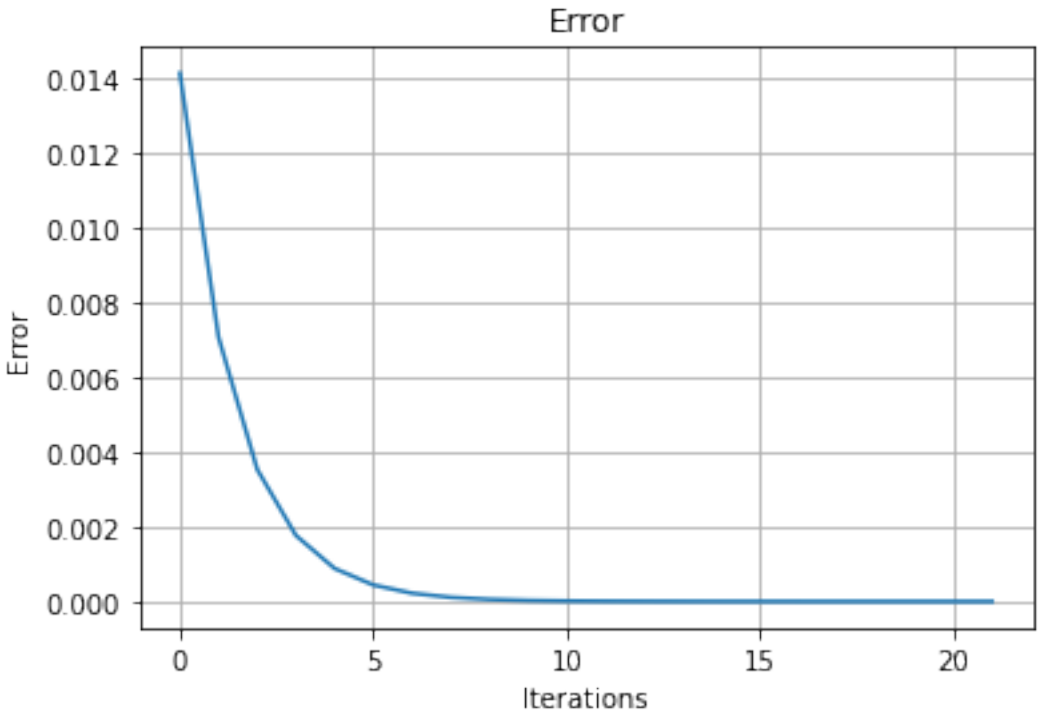
Table 2: Σφάλματα Εκτέλεσης μεθόδου

Επανάληψη k	$\|x^{k+1} - x^k\|_2$
0	0.014142135623729393
1	0.0070710678118797695
2	0.0035355339059307783
3	0.0017677669529047836
4	0.0008838834767383055
5	0.0004419417382327117
6	0.0002209708696654172
7	0.00011048543414893334
8	5.5242716048882274e-05
9	2.7621359500485057e-05
10	1.3810687993855974e-05
11	6.905332245256962e-06
12	3.4527288844053596e-06
13	1.7263942742734132e-06
14	8.628559965538201e-07
15	4.3207380902103435e-07
16	2.1623434709140006e-07
17	1.0738208498923628e-07
18	5.361949306725743e-08
19	3.2865337692729e-08
20	2.096321161530971e-08
21	0.0

Το ακριβές σφάλμα δίνεται απο τον τύπο

$$E_k = [(x^{k+1} - x^*)^2 + (y^{k+1} - y^*)^2]^{1/2} = \|x^{k+1} - x^*\|_2 \tag{10}$$

οπότε μετά απο 22 επαναλήψεις το ακριβές σφάλμα θα είναι $E_k = 2.210046443918695e - 10$



4 Κώδικας Newton-Raphson σε Python

```
import torch as th  # Pytorch library
import matplotlib.pyplot as plt


class NewtonRaphson:

  def __init__(self, x, tol, functions, maxIterations):
    self.x = x  # Initial x vector
    self.tolerance = tol  # Tolerance
    self.functions = functions
    self.maxIterations = maxIterations  # Max iterations

  # Calculate the jacobian matrix
  # The Jacobian method in pytorch calculates n unknown variables for one function
  # for n functions we loop through n times and store the results in a list.
  def jacobianMatrix(self):
    jacobian = []
    for i in range(len(self.functions)):
      jacobian.append(list(map(lambda x: x.item(), list(th.autograd.functional.jacc
    return th.tensor(jacobian, dtype=th.float64)

  # Calculate inverse jacobian
  def inverseJacobian(self):
```

```
    return th.inverse(self.jacobianMatrix())

  # Calculate the functions by replacing the unknown variables with their values
  def functionCalculation(self):
    functionValues = []
    for function in self.functions:
      functionValues.append(function(self.x))
    return th.tensor([functionValues], dtype=th.float64)

  # Check if the local error for all the unknown variables is greater than the tole
  def islocalErrorGreaterThanTolerance(self, counter, local_variable_errors):
    errorsGreaterThanTolerance = 0
    for i in range(len(self.x)):
      if local_variable_errors[counter][i] >= self.tolerance:
        errorsGreaterThanTolerance += 1

    return True if errorsGreaterThanTolerance == len(self.x) else False


  # Main Newton-Raphson algorithm
  def newtonRaphson(self):
    X = [self.x] # Store the results in X
    local_variable_errors = [[self.x[i].item() for i in range(len(self.x))]]
# Store local variable erros
    local_errors = []   # Store local L^2 norm errors
    counter = 0
    while counter < self.maxIterations and self.islocalErrorGreaterThanTolerance(co

      # Update the counter
      counter += 1

      # Calculate the jacobian and find a new solution given previous solution
      X.append(th.sub(self.x, th.transpose(th.mm(self.inverseJacobian(), th.transpo

      # Calculate the local error for each unknown variables and store it in an arr
      # Calculate the local error for all unknown variables
      local_variable_errors.append([])
      local_error = 0
      for i in range(len(self.x)):
        local_variable_errors[counter].append(abs(X[counter][0][i].item() - self.x[
        local_error += (X[counter][0][i].item() - self.x[i].item())**2

      local_errors.append(local_error**(1./2.))

      # Update the solution x
      self.x = th.tensor(list(map(lambda x: x.item(), list(X[counter][0]))), dtype=

    return X, local_variable_errors, local_errors
```

```
  # Exact error
  def exact_error(self, exact_solution, approx_solution):
    exact_error = 0
    for i in range(len(self.x)):
      exact_error += (exact_solution[i] - approx_solution[-1][0][i].item())**2

    return exact_error**(1./2.)

  # Graph the error for all variables
  def graphError(self, error):
    fig = plt.figure(1)
    plt.title("Error")
    plt.xlabel("Iterations")
    plt.ylabel("Error")
    plt.plot(error)
    plt.grid()
    plt.show()

if __name__ == "__main__":
    # Initial X
    initialX = th.tensor([1.98, 1.02], dtype=th.float64)

    # Functions
    functions = []
    functions.append(lambda x: x[0]+x[0]*x[1]-4)
    functions.append(lambda x: x[0]+x[1]-3)
    tolerance = 1e-08

    # Real solutions
    exact_solution = [2., 1.]

    # Max iterations
    maxIterations = 200

    NR = NewtonRaphson(initialX, tolerance, functions, maxIterations)
    x, variable_erros, local_errors = NR.newtonRaphson()

    # Show X results
    print('X results: \n')
    for solution in range(1,len(x)):
      for variable in range(len(initialX)):
        print(x[solution][0][variable].item(), end=" ")
      print()

    # Show variable error results
    print('\nVariable error results: \n')
    for var_err in variable_erros:
```

```
    print(var_err)

# Show local L^2 norm error
print('\nL^2 local error results: \n')
for lcl_err in local_errors:
    print(lcl_err)

# Show the exact error
print(f'\nThe exact error between x* and x is: {NR.exact_error(exact_solution,

# Graph the error for one variable
NR.graphError(local_errors)
```