Θέματα Αριθμητικής Ανάλυσης

Μέθοδος Πεπερασμένων Διαφορών για την επίλυση διαφορικών εξισώσεων 2ης τάξης με συνοριακές συνθήκες.

Γιώργος Ματλής

17 Ιουνίου 2022

1 Εισαγωγή

Η μαθηματική μοντελοποίηση των περισσότερων προβλημάτων της Φυσικής, των Τεχνολογικών και άλλων εφαρμοσμένων επιστημών οδηγεί σε Μερικές Διαφορικές Εξισώσεις (Μ.Δ.Ε). Οι εξισώσεις αυτές συμπληρώνονται με κατάλληλες συνοριακές συνθήκες, έτσι ώστε να περιγράφουν μονοσήμαντα τη λύση των φυσικών προβλημάτων που μοντελοποιούν. Τέτοια προβήματα μόνο σε σπάνιες και απλές περιπτώσεις είναι δυνατόν να λυθούν αναλυτικά. Έτσι προσφεύγουμε στην προσεγγιστική επίλυση τους με αριθμητικές επαναληπτικές μεθόδους. Λόγω της μεγάλης σημασίας που έχουν οι Μ.Δ.Ε, η αριθμητική τους επίλυση είναι το κεντρικό θέμα στην Αριθμητική Ανάλυση.
Οι κυριότερες κατηγορίες μεθόδων Αριθμητικής Ανάλυσης για την αριθμητική επίλυση Μ.Δ.Ε αλλά και των Σ.Δ.Ε είναι δύο: Οι μέθοδοι των πεπερασμένων στοιχείων και οι μέδοδοι των πεπερασμένων διαφορών.Η μέθοδος των πεπερασμένων στοιχείων εφαρμόζεται σε περισσότερα προβλήματα, η ανάλυση της είναι πιο συστηματοποιημένη και γίνεται σε πολύπλοκούς χώρους. Η ανάλυση των πεπερασμένων διαφορών γίνεται με στοιχειώδη μέσα. Παρακάτω θα εξετάσουμε την μέθοδο των πεπερασμένων διαφορών για την επίλυση Σ.Δ.Ε.

2 Μέθοδος Πεπερασμένων Διαφορών

Οι μέθοδοι πεπερασμένων διαφορων για την επίλυση συνοριακών προβλημάτων κατασκευάζονται, αν αντικαταστήσουμε τις παραγώγους που υπαρχουν στη συνήθη διαφορική εξίσωση με κατάλληλες προσεγγίσεις τους. Θεωρούμε το πρόβλημα συνοριακών τιμών της μορφής

$$\begin{cases} -p(x)u"(x) + q(x)u = f(x) \\ x \in [a,b], u(a) = c1, u(b) = c2 \end{cases} \tag{1}$$

όπου $p(x)$, $q(x)$, $f(x)$ είναι συνεχής στο διάστημα [a,b]. Υποθέτουμε επίσης ότι $q(x) \geq 0$ και ότι το πρόβλημα έχει μοναδική λύση στο [a,b]. Διαιρούμε το διάστημα [a,b] σε $n+1$ υποδιαστήματα μήκους h. Επιλέγουμε το βήμα $h = \frac{b-a}{n+1}$, έτσι ώστε $x_{n+1} = b$. Ζητάμε προσεγγίσεις U_i της λύσης $u"(x_i)$ δηλαδή προσεγγίζουμε το διάνυσμα $(u(x_0), u(x_1), ..., u(x_{n+1}))^T$ με ένα διάνυσμα $U = (U_0, U_1, ..., U_{n+1})^T \in R_0^{n+2}$στα σημεία $x_i = a + ih$, $i = 1, 2, 3, ..., n$, για το οποίο ισχύει

$$
\begin{aligned}
&-p(x_i)\left(\frac{U_{i+1}-2U_i+U_{i-1}}{h^2}\right)+q(x_i)U_i=f(x_i)\\
&i=1,2,3,...,n, U_0=c1, U_{n+1}=c2
\end{aligned}
\tag{2}
$$

Στην εξίσωση (2), για προσέγγιση της τιμής της δεύτερης παραγώγου $u"(x_i)$ χρησιμοποιούμε το ανάπτυγμα Taylor με κεντρικές διαφορές.
Η εξίσωση (2) ορίζει ένα γραμμικό σύστημα με n εξισώσεις και n αγνώστους $U_1, U_2, ..., U_n$ της μορφής:

$$\mathbf{AU}=\mathbf{F} \tag{3}$$

όπου

$$U=[U_1, U_2, ..., U_n]^T, \tag{4}$$

$$F=[h^2F(x_1)+U_0, h^2F(x_2), h^2F(x_3), ..., h^2F(x_n)+U_{n+1}], \tag{5}$$

και A είναι ένας $n \times n$ τριδιαγώνιος πίνακας. Επειδή ο πίνακας A είναι τριδιαγώνιος, το γραμμικό σύστημα (4) μπορεί να επιλυθεί χρησιμοποιώντας επαναληπτικές αριθμητικές μεθόδους όπως η μέθοδος Gauss-Seidel, Jacobi και SOR. Ο A έχει την μορφή:

$$
\begin{bmatrix}
2+h^2q(x_1) & -1 & 0 & . & . & . & 0\\
-1 & 2+h^2q(x_2) & -1 & . & . & . & .\\
0 & -1 & 2+h^2q(x_3) & -1 & . & . & .\\
. & . & . & . & . & . & .\\
. & . & . & . & . & . & .\\
. & . & . & . & . & . & .\\
. & . & . & . & -1 & 2+h^2q(x_{n-1}) & -1\\
0 & . & . & . & . & -1 & 2+h^2q(x_n)
\end{bmatrix}
\tag{6}
$$

Επιπλέον ο πίνακας A είναι συμμετρικός, δηλαδή ισχύει $a_{ij}=a_{ji}$ και μπορεί να αποδειχθεί ότι είναι αντιστρέψιμος, γεγονός που συνεπάγεται ύπαρξη και μοναδικότητα της προσεγγιστικής λύσης U. Μάλιστα ο A είναι θετικά ορισμένος, στην περίπτωση που η συνάρτηση q λαμβάνει μόνο θετικές τιμές και ο A είναι γραμμικά υπέρτερος κατά γραμμές.
Ένας αλγόριθμος με τα βήματα υλοποίησης της επαναληπτικής μεθόδου μπορεί να γραφτεί ως

1. Αρχικοποίησε τις τιμές των a και b του διαστήματος [a,b].
2. Θέσε τον αριθμό n των υποδιαστημάτων του διαστήματος [a,b] και υπολόγισε το βήμα h ως $h=\frac{b-a}{n+1}$. Στη συνέχεια κατασκεύασε μία διαμέριση $\Delta(I=[a,b])=a=x_0<x_1<...<x_{n+1}=b$ βάση του βήματος h.
3. Θέσε τις συναρτήσεις p, q και f, καθώς και τις συνοριακές συνθήκες $u(a)$ και $u(b)$.
4. Δημιούργησε τον πίνακα **A** με βάση το (6).
5. Δημιούργησε το διάνυσμα **F** της συνάτησης f με βάση το (5).
6. Βρές τη λύση του γραμμικού συστήματος χρησιμοποιώντας μια αριθμητική επαναληπτική μέθοδο όπως η Gauss-Seidel, Jacobi και SOR. Παρακάτω χρησιμοποιείται η επαναληπτική μέθοδος Gauss-Seidel.

7. Υπολόγισε το σφάλμα μεταξύ της λύσης U και της δεύτερης παραγώγου $u"$ για κάθε σημείο x_i της διαμέρισης με τον τύπο

$$\left(\sum_{i=1}^{n} h|U_i - u(x_i)|^2\right)^{1/2} \tag{7}$$

3 Εκτέλεση Μεθόδου

Για τα δεδομένα, $u(x) = cos(\phi x)$, $q = 2\phi^2$, $f(x) = 3\phi^2 cos(\phi x)$, $\phi = 10\pi$, $b = cos(2\phi)$ και για τα βήματα διαμέρισης $h = 0.1$, $h = 0.05$, $h = 0.025$, $h = 0.01$, $h = 0.005$ ο πίνακας των σφαλμάτων θα είναι

Table 1: Αποτελέσματα Εκτέλεσης μεθόδου

h	$E = \left(\sum_{i=1}^{n} h\|U_i - u(x_i)\|^2\right)^{1/2}$
0.1	0.0586294275707552
0.05	0.002172459351334059
0.025	0.00014020109528850688
0.01	3.6130952303345687e-06
0.005	2.260069196399484e-07

Απο την θεωρία γνωρίζουμε οτι όσο πιο μικρό βήμα h έχουμε, η τιμή της δεύτερης παραγώγου $u"$ σε ένα σημείο x_i της διαμέρισης προσεγγίζεται με μεγαλύτερη ακριβεία απο την πεπερασμένη διαφορά $\frac{1}{h^2}[v(x-h)-2v(x)+v(x+h)]$. Εδώ πρέπει να σημειώσουμε πως καθώς μειώνουμε το βήμα h, θα πρέπει να μειωνούμε ανάλογα την τιμή ανοχής (Tolerance) και να αυξάνουμε την μέγιστη τιμή επαναλήψεων k_{max} εφόσον χρησιμοποιούμε μια απο τις παραπάνω αριθμητικές μεθόδους επίλυσης γραμμικών συστημάτων. Σε περίπτωση που χρησιμοποιούμε άμεσες μεθόδους(π.χ Gauss Elimination) για την λύση του γραμμικού συστήματος, δεν χρειάζετε να μεταβάλουμε την τιμή ανοχής και το μέγιστο αριθμό επαναλήψεων.

4 Κώδικας Πεπερασμένων Διαφορών

```
import sympy as sp
import numpy as np
import math
from scipy.sparse import csr_matrix
import GaussSeidel

class FiniteDifferences:

  def __init__(self, x, f, h):
    self.x = x     # X nodal points
    self.f = f     # function f
    self.M = len(x)
    self.h = h     # Step

  def create_stiffness_matrix(self, p, q, boundary_conditions):
```

```
    A = csr_matrix((self.M-2, self.M-2), dtype=np.float64).toarray()
# Triagonal matrix
    F = np.zeros(self.M-2, dtype=np.float64)    # F vector for function f
    U = np.zeros(np.size(F), dtype=np.float64)    # Solution of the linear system
    h = self.h ** 2.    # Step

    # Create triagonal matrix
    for i in range(0, self.M-2):
      for j in range(0, self.M-2):
        if i == j:
          A[i][j] = (2. * p(_)) + h*q(_)
        elif i == j-1 or i == j+1:
          A[i][j] = -1. * p(_)
        else:
          A[i][j] = 0.

    # Set boundary conditions to vector F
    F[0] = h * self.f(self.x[1]) + boundary_conditions[0]
    F[self.M-3] = h * self.f(self.x[self.M-2]) + boundary_conditions[1]

    for i in range(1, self.M-3):
      F[i] = h * self.f(self.x[i+1])

    return A, F, U

  def gauss_elimination(self,A, F):
    n = len(F) #n is matrix size

    #Elimination phase
    for k in range(0,n-1): #k is matrix row
        for i in range(k+1,n): #i is matrix col
                if A[i,k] != 0.0:
                  factor = A[i,k]/A[k,k]
                  A[i,k+1:n] = A[i,k+1:n] - np.multiply(factor,A[k,k+1:n])
                  F[i] = F[i] - np.multiply(factor,F[k])

    #Back substitution
    for k in range(n-1,-1,-1):
          F[k] = (F[k] - np.dot(A[k,k+1:n],F[k+1:n]))/A[k,k]

    return F


  def solve(self, p, q, boundary_conditions):
    A, F, initial_U = self.create_stiffness_matrix(p, q, boundary_conditions)

    # Solve linear system
    F = F[:, np.newaxis]
```

```
    initial_U = initial_U[:, np.newaxis]

    GS = GaussSeidel.GaussSeidel(initial_U, A, F, 3e-7, 1000)
    Solutions, iter, Errors, U = GS.gaussSeidel()

    return U, A, F

  # Method to calculate the error
  def error(self, err_arr):
    error_sum = 0
    for i in range(1, len(self.x)-1):
      error_sum += self.h * (abs(err_arr[i-1]) ** 2.)

    return error_sum ** 1./2.

  def compare_solutions(self, U, u):
    u_exact = []    # Array to store exact solutions at nodal points 'xps'
    u_fd = []       # Array to store approximation solutions at nodal points 'xps'
    error_arr = []    # Array to store local absolute error
    U = np.asarray(U)

    for i in range(1, len(self.x)-1):
      u_exact.append(u(self.x[i]))    # Calculate the exact solution u
      u_fd.append(U[i-1][0])          # Get the approximated solution U
      error_arr.append(abs(u_fd[i-1] - u_exact[i-1]))   # Calculate the absolute lo

    return u_exact, u_fd, self.error(error_arr), error_arr


if __name__ == "__main__":
    # Interval
    interval = [0,2]

    # Subintervals
    J = 100

    # h is step size
    h = 0.005
    #h = (interval[1] - interval[0])/J   # Step size. Reducing the step we get bett

    # functions
    phi = 10. * math.pi
    u = lambda x: math.cos(phi * x)
    q = lambda x: 2. * (phi ** 2.)
    p = lambda x: 1.
    b = math.cos(2. * phi)
    f = lambda x: 3. * (phi**2.) * math.cos(phi * x)
```

```
    # Initial x nodal points
    x_points = np.arange(interval[0], interval[1] + h, h)

    # Boundary conditions
    boundary_conditions = np.array([1., b], dtype=np.float64)


    model = FiniteDifferences(x_points, f, h)

    # U is the solution to the linear system
    # A is the triagonal matrix
    # F is the vector of the function f
    U, A, F = model.solve(p, q, boundary_conditions)


    # Evaluate U solution with the exact solution
    u_exact, fd, err, err_arr = model.compare_solutions(U, u)


    # Uncomment to compare the exact solution with the approximated
    """
    for i in range(len(err_arr)):
      print(f'{u_exact[i]}     ==>     {fd[i]}     {err_arr[i]}\n')
    """
    print(f'\nError: {err[0]}')
```

5 Κώδικας Gauss-Seidel

```
import numpy as np
import matplotlib.pyplot as plt
import time


class GaussSeidel:
  def __init__(self, initialX, A, b, tolerance, maxIterations):
    self.x = initialX
    self.tolerance = tolerance
    self.A = A
    self.b = b
    self.maxIterations = maxIterations

  def gaussSeidel(self):
    # Diagonal matrix
    D = np.diag(self.A, dtype=np.float64)

    # Lower diagonal
    L = np.tril(self.A, -1, dtype=np.float64)
```

```
# Upper diagonal
U = np.triu(self.A, 1, dtype=np.float64)
iter = 0

# Check whether there is a 0.0 in the diagonal of A
if 0. in D:
  # If so, the jacobi method will not converge
  return np.array([np.NAN]), iter, None, None

# (L+D)^-1
inverseLD = np.linalg.inv(L+np.diag(D), dtype=np.float64)

# -(L+D)^-1
B = np.asmatrix(-inverseLD) * np.asmatrix(U)

# (L+D)^-1b
C = np.asmatrix(inverseLD) * np.asmatrix(self.b)

# x will initially be a vector with random values
x = np.random.uniform(size=(np.size(self.b), 1), dtype=np.float64)

# The new vector x(newX) will be x0(the initial x guess)
newX = self.x

# List of local errors
# Calculate the infinite norm error value of the initial guess x(x0) and the ra
localErrors = [np.linalg.norm(newX - x, np.inf)]

# List of x solutions
xSolutions = [self.x]

# Start time
startTime = time.time()

while iter < self.maxIterations and localErrors[iter] > self.tolerance:
  x = newX
  iter += 1

  # Calculate a new x solution
  newX = np.asmatrix(B) * np.asmatrix(x) + np.asmatrix(C)

  # Add the local error to the list of local errors
  localErrors.append(np.linalg.norm(newX - x, np.inf))

  # Add the new solution x to the list of solutions
  xSolutions.append(newX)
```

```
    x = newX
    xSolutions.append(x)

    # End time
    endTime = time.time()
    print(f'Gauss-Seidel completed at {endTime - startTime} \n')

    return xSolutions, iter, localErrors, x

# Plot the local erros
def plotError(self, error):
    figure = plt.figure(1)
    plt.title("Error")
    plt.plot(error[1:])
    plt.grid()
    plt.show()
```