

```

1  # Groundwater Modeling Coding Assignment #2
2  # Jim Finnegan
3  # 1D Transport Equation
4  # functions for FE, FD, and analytical solution
5
6  import numpy as np
7  from scipy.sparse import diags
8  from math import exp, sqrt
9  from scipy.special import erfc
10
11
12 def finite_element(d, r):
13     """
14     Computes matrix of C/C0, 0<x<200, 0<t<400 using Galerkin FEM
15     @param d: float, hydrodynamic dispersion coefficient (m^2/d)
16     @param r: float, retardation coefficient (unitless)
17     @return C: array, matrix where rows are time steps and columns are C/C0(x
18 )
19     """
20     # PARAMETERS
21     # user inputs
22     d = float(d)
23     r = float(r)
24     # other parameters
25     v, L, dx, t, dt = 0.5, 200, 2, 400, 10
26     # matrix dimensions
27     rows = int(t / dt) + 1
28     n_el = int(L / dx)
29     cols = n_el + 1
30     # initial conditions
31     C = np.zeros((rows, cols))
32     C[:, 0] = 1 # boundary condition: C/C0 = 1 at x=0
33
34     # CONSTRUCT STIFFNESS AND STORAGE MATRICES
35     # element matrices
36     alpha = (r * dx) / 6
37     lam_1 = d / dx
38     lam_2 = v / 2
39     Ae = [[lam_1 - lam_2, -lam_1 + lam_2], [-lam_1 - lam_2, lam_1 + lam_2]]
40     # element stiffness matrix
41     Be = [[2 * alpha, alpha], [alpha, 2 * alpha]]
42     # element storage matrix
43
44     # global matrices
45     A = np.zeros((cols, cols))
46     B = np.zeros((cols, cols))
47     for i in range(1, cols):
48         A[i, i] += Ae[1][1] # assemble Ae elements
49         A[i, i - 1] += Ae[1][0]
50         A[i - 1, i] += Ae[0][1]
51         A[i - 1, i - 1] += Ae[0][0]
52         B[i, i] += Be[1][1] # assemble Be elements
53         B[i, i - 1] += Be[1][0]
54         B[i - 1, i] += Be[0][1]
55         B[i - 1, i - 1] += Be[0][0]
56
57     LH = (A / 2 + B / dt)
58     RH = (-A / 2 + B / dt)
59
60     # TIME STEPPING
61     for k in range(1, rows):
62         b_f = np.dot(RH, C[k - 1, :]) # solve RHS

```

```

58     b_f[0] = LH[0][0] + LH[0][1] * C[k - 1][1]      # boundary condition
59     C[k, :] = np.linalg.solve(LH, b_f)              # solve LHS
60     return C
61
62
63 def finite_difference(d, r):
64     """
65     Computes matrix of C/C0, 0<x<200, 0<t<400 using Crank-Nicholson FDM
66     @param d: float, hydrodynamic dispersion coefficient (m^2/d)
67     @param r: float, retardation coefficient (unitless)
68     @return C: array, matrix where rows are time steps and columns are C/C0(
x)
69     """
70     d = float(d)
71     r = float(r)
72
73     # other parameters
74     v, L, dx, t, dt = 0.5, 200, 2, 400, 10
75     # matrix dimensions
76     rows = int(t / dt) + 1
77     cols = int(L / dx) + 1
78
79     # initial conditions
80     C = np.zeros((rows, cols))
81     C[:, 0] = 1 # boundary condition: C/C0 = 1 at x=0
82
83     # simplified variables from central difference derivation
84     G = (d * dt) / (2 * r * dx ** 2)
85     H = (v * dt) / (4 * r * dx)
86     lam_1, lam_2, lam_3, lam_4 = G + H, 2 * G + 1, G - H, 2 * G - 1
87
88     # CENTERED DIFFERENCE SCHEME
89     # Left hand side - k+1
90     A_diagonals = [np.ones(cols - 1) * lam_1, np.ones(cols) * -lam_2, np.
ones(cols - 1) * lam_3]
91     A = diags(A_diagonals, offsets=[-1, 0, 1], shape=(cols, cols)).toarray()
92     # Right hand side - k
93     B_diagonals = [np.ones(cols - 1) * -lam_1, np.ones(cols) * lam_4, np.
ones(cols - 1) * -lam_3]
94     B = diags(B_diagonals, offsets=[-1, 0, 1], shape=(cols, cols)).toarray()
95
96     for k in range(1, rows):
97         b = np.dot(B, C[k - 1, :])      # solve RHS
98         b[0] = -(1 + lam_1)             # boundary condition
99         C[k, :] = np.linalg.solve(A, b)  # solve LHS
100
101     return C
102
103
104 def analytical(d):
105     """
106     @param d: float, hydrodynamic dispersion coefficient (m^2/d)
107     @return C: array, matrix where rows are time steps and columns are C/C0(
x)
108     """
109     # initial conditions
110     # for R = 1
111     v = 0.1
112     d = float(d)
113     L, dx = 200, 2

```

```

114     dist = np.linspace(2, L, num=int(L / dx))
115     dist = [int(x) for x in dist]
116     t, dt = 400, 10
117     time = np.linspace(10, t, num=int(t / dt))
118     time = [int(t) for t in time]
119
120     # initialize grid for C
121     # x is distance (one column is 2 ft), y is time (one row is 10 days)
122     C = np.zeros((len(time) + 1, len(dist) + 1))
123     C[:, 0] = 1 # boundary condition: C/C0 = 1 at x=0
124
125     # calculate C using analytical solution for all x>0
126     for x in range(len(dist)):
127         for t in range(len(time)):
128             C[t + 1][x + 1] = (1 / 2) * (exp(v * dist[x] / d) * erfc((dist[x]
129 ] + v * time[t]) / (2 * sqrt(d * time[t])))
129                                     + erfc((dist[x] - v * time[t]) / (2
130 * sqrt(d * time[t]))))
131     return C
132
133
134 def analytical_vfive(d):
135     """
136     @param d: float, hydrodynamic dispersion coefficient (m^2/d)
137     @return C: array, matrix where rows are time steps and columns are C/C0(
138 x)
139     """
140     v = 0.5
141     D = float(d)
142     L, dx = 200, 2
143     dist = np.linspace(2, L, num=int(L / dx))
144     dist = [int(x) for x in dist]
145     t = 200
146
147     # calculate C using analytical solution for all x>0
148     C = np.zeros(len(dist) + 1)
149     for x in range(len(dist)):
150         try:
151             C[x] = (1 / 2) * (exp(v * dist[x] / D) * erfc((dist[x] + v * t
152 ) / (2 * sqrt(D * t))) + erfc(
153 (dist[x] - v * t) / (2 * sqrt(D * t))))
154         except OverflowError:
155             C[x] = 0 # set C = 0 if math overflow error
156
157     return C

```