Project Report
on

Solving Orr- Sommerfeld Equation Using the Compound Matrix Method

Submitted by

Group 1

Garima Goel (2023PMA5166)

Ganpat (2023PMA5201)

Mrityunjai (2023PMA5202)

Aarti Kapoor (2023PMA5183)

Anjali Dhull (2023PMA5173)

Manish (2023PMA5160)

submitted to

Dr. Geetanjali Chattopadhyay , Assistant Professor

DEPARTMENT OF MATHEMATICS

Malaviya National Institute of Technology Jaipur

Contents

Chapter 1

The Compound Matrix Method

Imagine working with a high-order ODE (like fourth-order) and trying to find solutions. These equations can be stiff, i.e. some parts of the solution grow or shrink really fast, causing instability during calculations. Traditional methods fails in those cases.
Instead of working directly with the individual solutions of the ODE, this method works with minors—mathematical pieces of the solution matrix that describe how groups of solutions behave together.
The Compound Matrix Method is a numerical technique designed to solve eigenvalue problems for stiff ordinary differential equations (ODEs). This method focuses on avoiding numerical instability issues by shifting the focus from individual solutions to their combined behavior..

1.1 How It Works

1. **Rewrite the ODE:** Start with a high-order ODE and rewrite it as a system of first-order equations.

2. **Construct the Compound Matrix:** Compute the compound matrix, where each element corresponds to the evolution of a minor from the original solution matrix.

3. **Solve the Transformed System:** Solve the compound system numerically for the required eigenvalue. The eigenvalue condition is imposed at one boundary by requiring certain minors to vanish.

4. **Reconstruct the Solution:** Once the eigenvalue is determined, use the minors to reconstruct the corresponding eigenfunction.

1.2 Compound Matrix Method for a Fourth-Order ODE

We solve the fourth-order differential equation:

$$y^{(4)} + a_3(x)y^{(3)} + a_2(x)y^{(2)} + a_1(x)y' + a_0(x)y = 0,$$

by the following steps:

1. Rewrite as a First-Order System

Define:

$$\mathbf{u}(x) = \begin{bmatrix} y \\ y' \\ y'' \\ y''' \end{bmatrix},$$

so that the equation becomes:

$$\mathbf{u}'(x) = \mathbf{A}(x)\mathbf{u}(x),$$

where:

$$\mathbf{A}(x) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -a_0(x) & -a_1(x) & -a_2(x) & -a_3(x) \end{bmatrix}.$$

2. Transition to the Compound Matrix

Define the second compound matrix system:

$$\Psi'(x) = \mathbf{B}(x)\Psi(x),$$

where:

- $\Psi(x)$ is the vector of all 2×2 minors.
- The entries of $\mathbf{B}(x)$ are given by:

$$B_{(i,j),(k,l)} = A_{i,k}\delta_{j,l} - A_{i,l}\delta_{j,k} + A_{j,l}\delta_{i,k} - A_{j,k}\delta_{i,l}.$$

3. Solve the System and Apply Boundary Conditions

[2] Numerically integrate $\Psi'(x) = \mathbf{B}(x)\Psi(x)$ with initial conditions at $x = 0$, and enforce the boundary conditions at $x = 1$.

4. Reconstruct the Solution

Combine the solution $\Psi(x)$ with the boundary conditions to reconstruct the original function $y(x)$.

5. Eigenvalue Search and the Discriminant Function

[4] In many problems one is interested in finding values of a parameter (or eigenvalue) c for which the differential equation with boundary conditions has nontrivial solutions. After the compound system (**??**) is integrated from a starting point $x = x_{\text{start}}$ to an ending point $x = x_{\text{end}}$, a *discriminant* function is defined by

$$D(c) = \Psi_1(x_{\text{end}}),$$

where Ψ_1 is one chosen component of the compound solution (often corresponding to one particular minor). The appropriate eigenvalue is expected to satisfy

$$D(c) \approx 0.$$

A practical numerical procedure to determine c is as follows:

(a) **Scan:** Evaluate the residual $D(c)$ over a range on the real axis (or in the complex plane) and identify sign changes.

(b) **Refine:** Apply a root-finding method (e.g., Brent's method) to accurately locate the zeros of $D(c)$.

(c) **Determine Temporal Growth (if applicable):** When the eigenvalue is complex, $c = c_r + ic_i$, and if the physical problem involves a time-dependent exponential factor $\exp(-i\omega t)$ with $\omega = k\,(c)$, then the temporal growth (or decay) rate is given by

$$\gamma = k\ \mathrm{Im}(c),$$

where k is a wavenumber.

Chapter 2

Orr Summerfeld equation for Plane Poiseuille Flow

2.1 The Orr-Sommerfeld equation

The Orr-Sommerfeld equation governs the stability of disturbances in parallel shear flows[1]. For Plane Poiseuille flow, it is given by:

$$\left(\frac{D^2-\alpha^2}{Re}\right)(D^2-\alpha^2)\phi - i\alpha(U(D^2-\alpha^2)\phi - U''\phi) = 0$$

where:

- D denotes differentiation with respect to the wall-normal coordinate.
- α is the streamwise wave number.
- Re represents the Reynolds number.
- U is the base flow velocity profile.
- ϕ is the perturbation stream function.

This equation helps analyze flow stability by determining whether disturbances grow or decay over time.

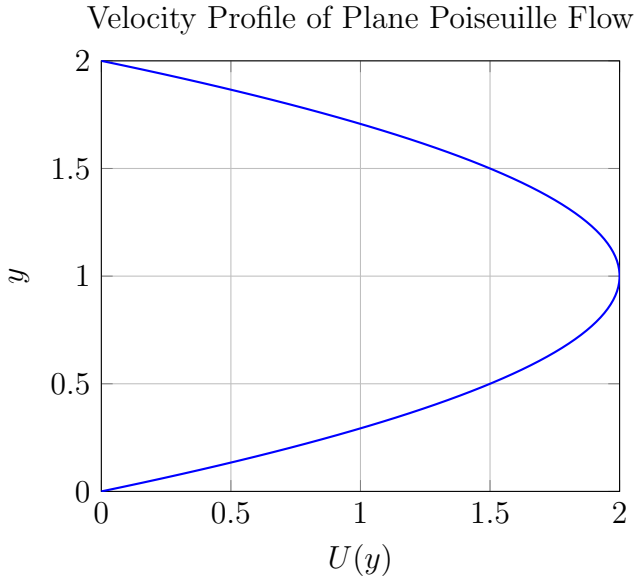
2.2 Plane Poiseuille Flow

Plane Poiseuille flow describes the movement of a viscous fluid between two parallel plates under the influence of a pressure gradient.This flow plays a crucial role in hydrodynamics and has widespread applications, including lubrication theory and industrial fluid transport.[3]

- Plane Poiseuille flow describes the steady, viscous flow of a fluid between two stationary, parallel plates.
- When the flow is driven by a constant pressure gradient, the velocity distribution in the wall-normal direction becomes parabolic.
- In a non-dimensional form (with the channel walls located at $y = 0$ and $y = 2$), the velocity profile is commonly written as:

$$U(y) = y(2 - y)$$

- This profile indicates that the maximum velocity occurs at the centerline ($y = 1$), and the fluid adheres to a no-slip condition at the walls.



Chapter 3

Numerical code and Output

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy.optimize import root_scalar, root

###################################
# 1. Orr-Sommerfeld operator for plane Poiseuille flow
###################################
def A_matrix_plane(y, k, R, c):
    """
    Construct the 4 4   O r r Sommerfeld  matrix for plane
   Poiseuille flow.

    The base flow is:
         U(y) = 1 - y  ,       U''(y) = -2,
    with y in [-1, 1].

    Starting from the  O r r Sommerfeld  equation
           ''''  - 2 k     '' +  k      = i k R [ (U-c)(  ''-
   k    ) - U''    ],
    we obtain a  f i r s t order  system by letting:
          x   =   ,   x   =   ',   x   =   '',   x   =   '''.
    Then, writing:
          x  ' =   A        +   A       '',
    the coefficients are defined as:
           A     = -  k   - i k R [ k   (U-c) + U'' ],
           A     = 2  k   + i k R (U-c).
    """
    U = 1 - y**2
    U_dd = -2.0
    ikR = 1j * k * R

    A = np.zeros((4, 4), dtype=np.complex128)
```

```
    A[0, 1] = 1.0
    A[1, 2] = 1.0
    A[2, 3] = 1.0
    A[3, 0] = - k**4 - ikR * ( k**2 * (U - c) + U_dd )
    A[3, 1] = 0.0
    A[3, 2] = 2 * k**2 + ikR * (U - c)
    A[3, 3] = 0.0
    return A

########################
# 2. Compound matrix construction
########################
def compound_matrix(A):
    """
    Constructs the second compound (6 6 ) matrix from the 4 4  matrix A
    by forming all 2 2  minors with respect to the basis:
         {(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)}.
    """
    basis = [(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
    B = np.zeros((6, 6), dtype=np.complex128)
    for row, (p, q) in enumerate(basis):
        for col, (r, s) in enumerate(basis):
            term1 = A[p, r] * (1 if q == s else 0)
            term2 = -A[p, s] * (1 if q == r else 0)
            term3 = A[q, s] * (1 if p == r else 0)
            term4 = -A[q, r] * (1 if p == s else 0)
            B[row, col] = term1 + term2 + term3 + term4
    return B

def compound_rhs_plane(y, psi, k, R, c):
    """
    Right-hand side of the compound system for plane Poiseuille flow:
         psi' = B(y) psi,
    where B(y) is constructed from the matrix A(y, k, R, c).
    """
    A = A_matrix_plane(y, k, R, c)
    B = compound_matrix(A)
    return B @ psi

########################
# 3. Eigenvalue solvers for plane Poiseuille flow
########################
def solve_eigenvalue_plane(k, R, c_guess, y_span=(-1, 1), num_points=50, plot_flag=False):
    """
    Search for a real eigenvalue (phase speed c) for plane Poiseuille flow.
```

```
77
78      An IVP for the compound system is solved from y=-1 to y
        =1.
79      The boundary residual is defined as the real part of
        p s i  at y=1.
80      Brent's method (via root_scalar) is used to find c such
        that the residual vanishes.
81      """
82      psi0 = np.zeros(6, dtype=np.complex128)
83      psi0[-1] = 1.0  # normalization
84
85      def boundary_condition_residual(c):
86          sol = solve_ivp(
87              fun=lambda y, psi: compound_rhs_plane(y, psi, k,
        R, c),
88              t_span=y_span,
89              y0=psi0,
90              method='DOP853',
91              t_eval=np.linspace(y_span[0], y_span[1],
        num_points),
92              rtol=1e-7,
93              atol=1e-9
94          )
95          return np.real(sol.y[0, -1])
96
97      if plot_flag:
98          c_values = np.linspace(0.1, 1.5, 200)
99          residuals = [boundary_condition_residual(c) for c in
        c_values]
100         plt.figure(figsize=(8, 6))
101         plt.plot(c_values, residuals, label='Residual at y =
        {}'.format(y_span[1]))
102         plt.axhline(0, color='black', linestyle='--')
103         plt.xlabel('Eigenvalue candidate c')
104         plt.ylabel('Residual (Re)')
105         plt.title('Residual vs. c for Plane Poiseuille Flow')
106         plt.legend()
107         plt.grid(True)
108         plt.show()
109
110     result = root_scalar(boundary_condition_residual, bracket
        =[0.1, 1.5], method='brentq')
111     if result.converged:
112         print(f"Computed eigenvalue c = {result.root}")
113         return result.root
114     else:
115         print("Eigenvalue computation did not converge.")
116         return None
117
```

```
118 def solve_eigenvalue_problem_plane(R, alpha, y_start, y_end,
    c):
119     """
120     For a given (possibly complex) eigenvalue candidate c,
    integrate the compound
121     system on y in [y_start, y_end] for plane Poiseuille flow
     and evaluate the
122     discriminant D = psi(y_end). The true eigenvalue
    satisfies D = 0.
123 
124     Here, alpha is the wavenumber (alias for k).
125     """
126     psi0 = np.zeros(6, dtype=np.complex128)
127     psi0[-1] = 1.0
128     sol = solve_ivp(
129         fun=lambda y, psi: compound_rhs_plane(y, psi, alpha,
    R, c),
130         t_span=(y_start, y_end),
131         y0=psi0,
132         method='DOP853',
133         t_eval=np.linspace(y_start, y_end, 50),
134         rtol=1e-7,
135         atol=1e-9
136     )
137     Y_final = sol.y[:, -1]
138     D = Y_final[0]
139     return D, sol
140 
141 ####################
142 # 4. Complex Eigenvalue Search and Classification
143 ####################
144 def solve_complex_eigenvalue_plane(k, R, c_initial, y_span
    =(-1,1), num_points=50):
145     """
146     Use a 2D root finder to search for a complex eigenvalue c
     for plane Poiseuille flow.
147     We define the residual as F(c) = [Re(D(c)), Im(D(c))],
148     where D(c) is the discriminant from the compound system.
149     """
150     def F(v):
151         c_candidate = v[0] + 1j*v[1]
152         D, _ = solve_eigenvalue_problem_plane(R, k, y_span
    [0], y_span[1], c_candidate)
153         return [np.real(D), np.imag(D)]
154 
155     v0 = [np.real(c_initial), np.imag(c_initial)]
156     sol = root(F, v0, method='hybr', tol=1e-8)
157     if sol.success:
158         return sol.x[0] + 1j*sol.x[1]
```

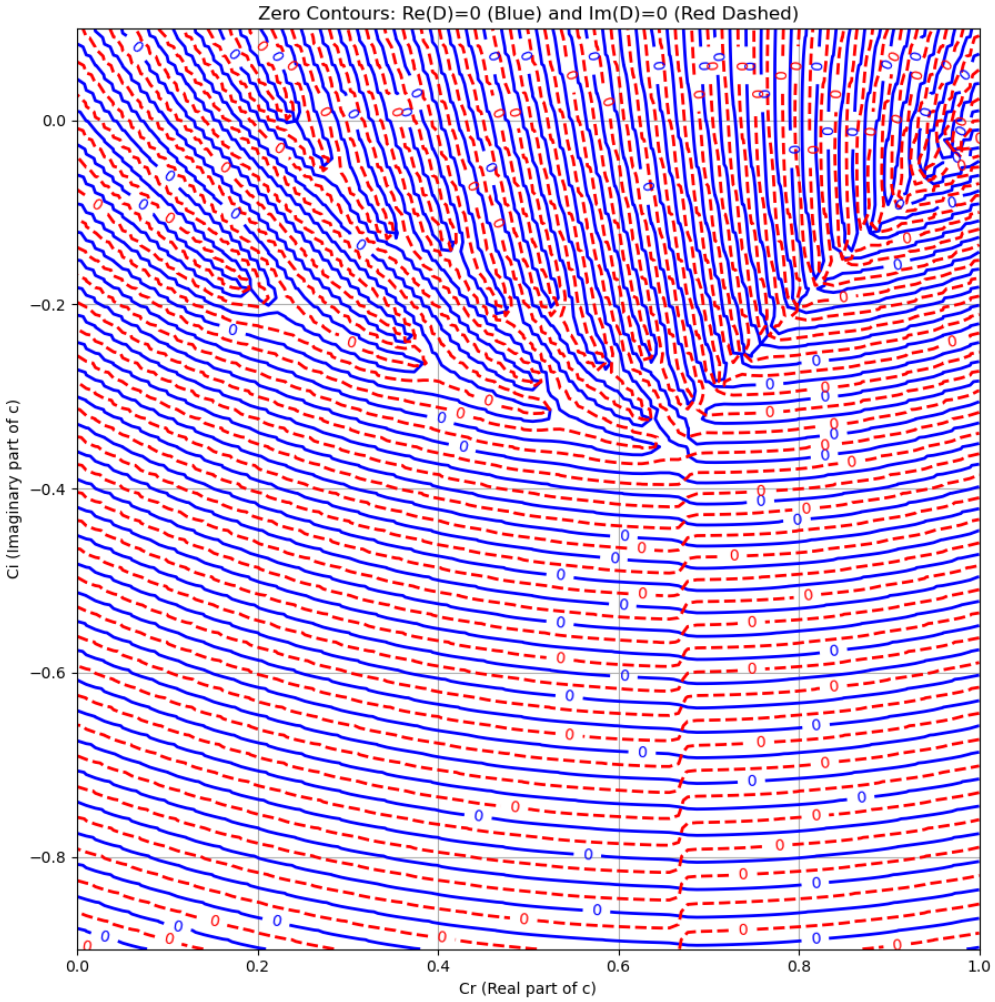
```
    else:
        raise RuntimeError("Complex eigenvalue search did not converge")

def find_all_eigenvalues_complex(k, R, y_span=(-1,1), num_points=50,
                                 N_real=5, N_imag=6, tol=1e-3, max_eigen=30):
    """
    Scan a grid of initial guesses over the complex-c plane and use solve_complex_eigenvalue_plane
    to collect up to max_eigen distinct eigenvalues.

    N_real and N_imag determine the number of grid points for the initial guess.
    """
    initial_guesses = []
    # Adjust these ranges based on expected eigenvalue distribution.
    real_range = np.linspace(0.1, 1.5, N_real)
    imag_range = np.linspace(-0.9, 0.1, N_imag)
    for r in real_range:
        for i in imag_range:
            initial_guesses.append(r + 1j*i)

    found = []
    for guess in initial_guesses:
        try:
            ev = solve_complex_eigenvalue_plane(k, R, guess, y_span, num_points)
            # Check against duplicates
            if not any(np.abs(ev - candidate) < tol for candidate in found):
                found.append(ev)
                print(f"Found eigen: {ev:.6f} from initial guess {guess:.6f}")
        except Exception as e:
            continue
        if len(found) >= max_eigen:
            break
    return found

def classify_mode(ev):
    """
    Simple heuristic classification based on the imaginary part of c.
    (These thresholds are chosen arbitrarily to mimic the families observed in Mack, 1976.)
      - P family: Im(c) > -0.05  (triangles)
```

```
197       - A family: -0.3 < Im(c) <= -0.05  (circles)
198       - S family: Im(c) <= -0.3  (squares)
199     """
200     if ev.imag > -0.05:
201         return 'P'
202     elif ev.imag > -0.3:
203         return 'A'
204     else:
205         return 'S'
206 
207 ##########################
208 # 5. Compute the Eigenfunction for a Selected Eigenvalue
209 ##########################
210 def compute_eigenfunction_plane(k, R, c, y_span=(-1,1),
    num_points=200, delta=1e-5):
211     """
212     Compute the eigenfunction (phi(y)) for a given eigenvalue
     c by integrating the
213     original 4 4  system:
214          Y' = A_matrix_plane(y, k, R, c) Y,
215     where Y = [phi, phi', phi'', phi'''].
216 
217     To avoid the trivial solution imposed by the homogeneous
    boundary conditions,
218     we start slightly inside the boundary at y = y_start +
    delta.
219 
220     Here we choose initial conditions approximating a no-slip
     boundary in a channel:
221          phi(-1+delta) = 0, phi'(-1+delta)=0, phi''(-1+delta)
     = 1, phi'''(-1+delta) = 0.
222     The eigenfunction is returned as phi(y) (i.e. the first
    component).
223     """
224     y0 = y_span[0] + delta
225     Y0 = [0, 0, 1, 0]  # chosen normalization near the wall
226     sol = solve_ivp(lambda y, Y: A_matrix_plane(y, k, R, c) @
     Y,
227                     t_span=(y0, y_span[1]), y0=Y0,
228                     t_eval=np.linspace(y0, y_span[1],
    num_points),
229                     rtol=1e-7, atol=1e-9)
230     return sol.t, sol.y[0]  # return y and phi(y)
231 
232 ########################
233 # 6. Main Script: Eigenvalue Spectrum, Table, and Plots in
    the Complex c-plane
234 ########################
235 if __name__ == "__main__":
```

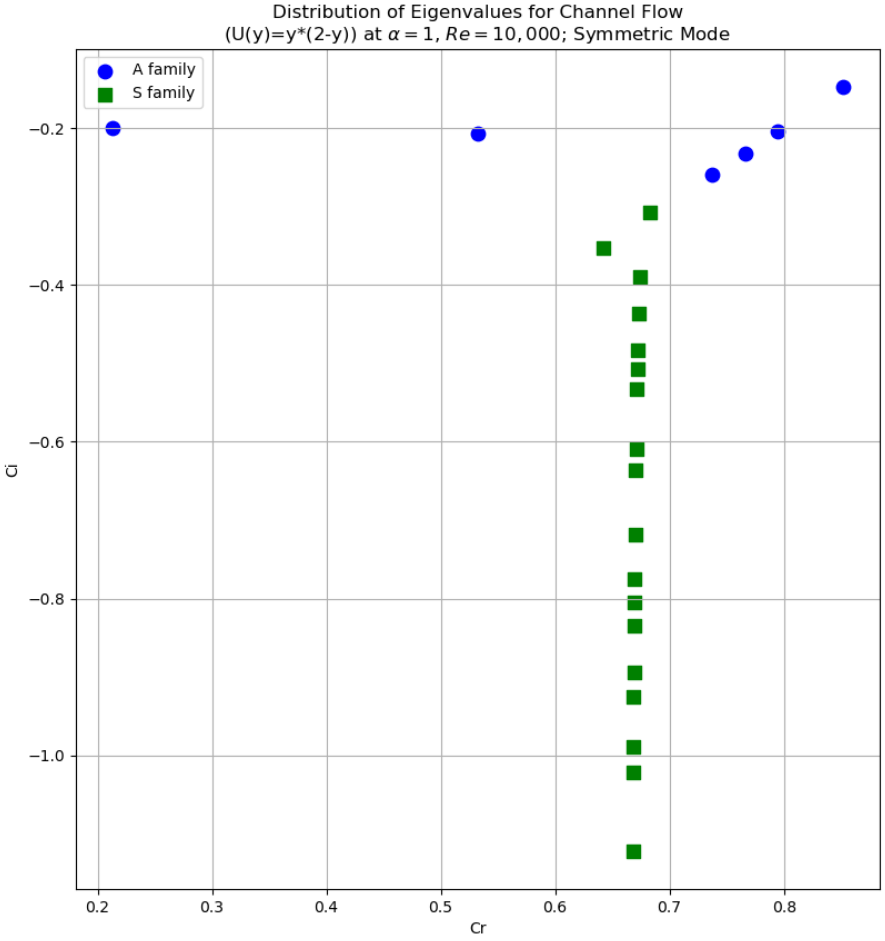
```
    # Parameters for plane Poiseuille flow
    k = 1.0              # Wavenumber (  )
    R = 10000.0          # Reynolds number
    y_domain = (-1, 1)  # Channel walls at y = -1 and y = 1

    print("Starting  O r r Sommerfeld  eigenvalue computation for plane Poiseuille flow...")

    # (A) Real eigenvalue search (optional, for comparison)
    c_guess_real = 0.3   # Initial guess for a real eigenvalue
    eigenvalue = solve_eigenvalue_plane(k, R, c_guess_real, y_span=y_domain, num_points=50, plot_flag=True)
    print("Computed eigenvalue (real search):", eigenvalue)

    # (B) PART 2: Evaluate discriminant and build a contour plot in the complex-c plane.
    # Here we assess the discriminant for a sample candidate.
    Re_val = R
    alpha = k   # for notational consistency, alpha = k
    y_start = y_domain[0]
    y_end = y_domain[1]
    c_guess_complex = 0.3 + 0.1j  # A sample complex candidate

    # Evaluate the discriminant for a test candidate:
    D_eval, sol_eval = solve_eigenvalue_problem_plane(Re_val, alpha, y_start, y_end, c_guess_complex)
    print("For c =", c_guess_complex, ", the discriminant D = ", D_eval)

    # Build a grid on the complex-c plane.
    Cr_vals = np.linspace(0, 1, 100)
    Ci_vals = np.linspace(-0.9, 0.1, 100)
    Cr, Ci = np.meshgrid(Cr_vals, Ci_vals)
    # Precompute the real and imaginary parts of the discriminant on the grid.
    Dr = np.zeros_like(Cr, dtype=float)
    Di = np.zeros_like(Ci, dtype=float)

    print("Computing the discriminant on the c-grid for contour plotting...")
    for i in range(Cr.shape[0]):
        for j in range(Cr.shape[1]):
            c_candidate = Cr[i, j] + 1j * Ci[i, j]
            D_val, _ = solve_eigenvalue_problem_plane(Re_val, alpha, y_start, y_end, c_candidate)
            Dr[i, j] = np.real(D_val)
            Di[i, j] = np.imag(D_val)
```

```
275
276     # --- Graph: Zero Contour Lines ---
277     # Create a single-axes figure for the zero-contours.
278     fig, ax = plt.subplots(figsize=(8, 6))
279     cs1 = ax.contour(Cr, Ci, Dr, levels=[0], colors='blue',
    linewidths=2)
280     cs2 = ax.contour(Cr, Ci, Di, levels=[0], colors='red',
    linestyles='dashed', linewidths=2)
281     ax.clabel(cs1, inline=True, fontsize=10)
282     ax.clabel(cs2, inline=True, fontsize=10)
283     ax.set_xlabel('Cr (Real part of c)')
284     ax.set_ylabel('Ci (Imaginary part of c)')
285     ax.set_title('Zero Contours: Re(D)=0 (Blue) and Im(D)=0 (
    Red Dashed)')
286     ax.grid(True)
287     # Mark the computed (real) eigenvalue on the real axis.
288     if eigenvalue is not None:
289         ax.plot(eigenvalue, 0, 'ko', markersize=8, label='
    Computed Eigenvalue')
290         ax.legend()
291
292     plt.tight_layout()
293     plt.show()
294
295     # (C) Complex eigenvalue search for up to 30 modes.
296     print("\nSearching for complex eigenvalues (up to 30
    modes)...")
297     eigen_complex_list = find_all_eigenvalues_complex(k, R,
    y_span=y_domain, num_points=50,
298                                                       N_real
    =5, N_imag=6, tol=1e-3, max_eigen=30)
299     # Sort eigenvalues by real part.
300     eigen_complex_list.sort(key=lambda x: x.real)
301
302     # Print table of computed eigenvalues.
303     print("\nTable of computed eigenvalues (first 30 modes):"
    )
304     print("{:<5s} {:>12s} {:>12s} {:>6s}".format("Mode", "Re(
    c)", "Im(c)", "Fam"))
305     for i, ev in enumerate(eigen_complex_list[:30], 1):
306         fam = classify_mode(ev)
307         print("{:<5d} {:12.6f} {:12.6f} {:>6s}".format(i, ev.
    real, ev.imag, fam))
308
309     # (D) Plot the eigenvalue distribution in the complex c-
    plane.
310     # Markers: P -> triangle ('^'), A -> circle ('o'), S ->
    square ('s')
311     marker_dict = {'P': '^', 'A': 'o', 'S': 's'}
```

```
312     colors = {'P': 'red', 'A': 'blue', 'S': 'green'}
313
314     plt.figure(figsize=(8, 6))
315     for ev in eigen_complex_list:
316         fam = classify_mode(ev)
317         plt.scatter(ev.real, ev.imag, marker=marker_dict[fam
    ], color=colors[fam], s=80,
318                     label=f"{fam} family" if fam not in plt.
    gca().get_legend_handles_labels()[1] else "")
319
320     plt.xlabel("Re(c)")
321     plt.ylabel("Im(c)")
322     plt.title("Distribution of Eigenvalues of Plane
    Poiseuille Flow\n"
323               "at    = 1 and Re = 10,000; Symmetric Mode (
    After Mack, 1976)")
324     plt.gca().invert_yaxis()  # In many OS studies, Im(c) is
    plotted inverted.
325     plt.grid(True)
326     plt.legend()
327     plt.show()
328
329     # (E) Compute the eigenfunction for the unstable mode.
330     # Here we select the eigenvalue with the maximum
    imaginary part as the unstable mode.
331     if eigen_complex_list:
332         unstable_mode = max(eigen_complex_list, key=lambda x:
     x.imag)
333         print("Unstable eigenvalue (maximum Im):",
    unstable_mode)
334         y_vals, phi_vals = compute_eigenfunction_plane(k, R,
    unstable_mode, y_span=y_domain, num_points=200)
335
336         plt.figure(figsize=(8, 6))
337         plt.plot(y_vals, phi_vals, 'b-', linewidth=2)
338         plt.xlabel("y")
339         plt.ylabel("Eigenfunction   (y)")
340         plt.title(f"Eigenfunction for Unstable Mode c = {
    unstable_mode:.6f}")
341         plt.grid(True)
342         plt.show()
343
344     input("Press Enter to exit...")
```



Computed eigenvalue c = 0.8040318832313728. For c = (0.3+0.1j), the discriminant D = (1.0300490383740409e+38-2.0774231916183491e+37j)


Distribution of Eigenvalues for Channel Flow
(U(y)=y*(2-y)) at $\alpha = 1, Re = 10,000$; Symmetric Mode
A family
S family
−0.2
−0.4
−0.6
−0.8
−1.0
Ci
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Cr

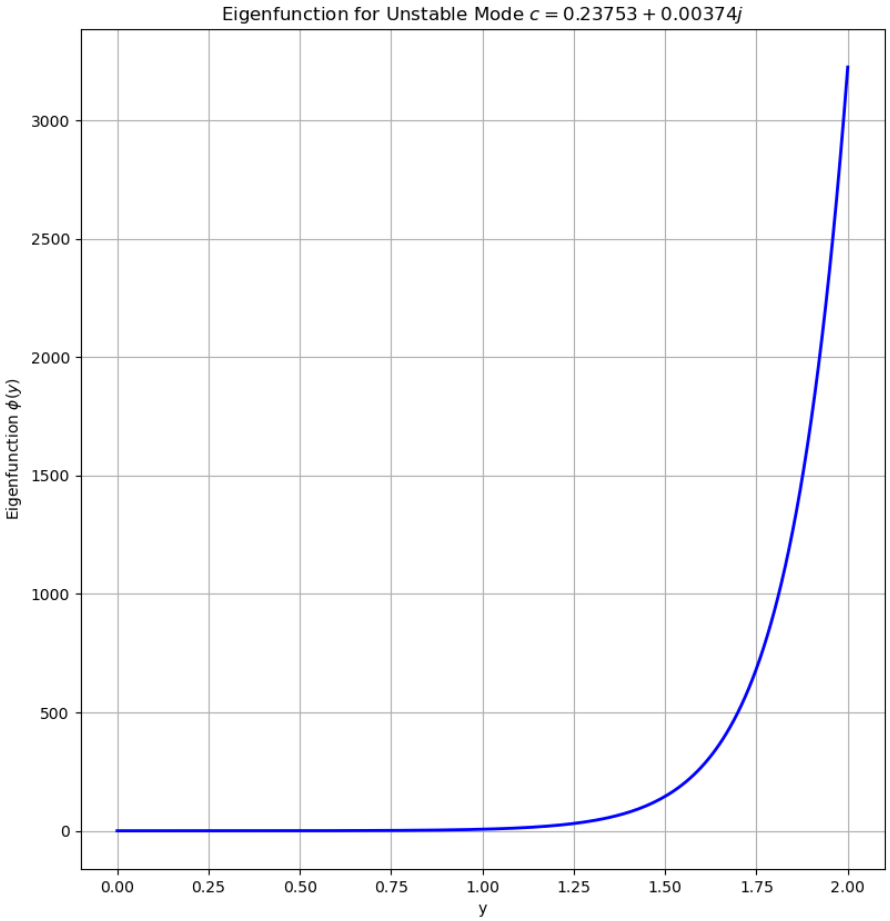

Eigenfunction for Unstable Mode $c = 0.23753 + 0.00374j$
Eigenfunction $\phi(y)$
y
3000
2500
2000
1500
1000
500
0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

Table of computed eigenvalues (first 25 modes):

Mode	Re(c)	Im(c)	Fam
1	0.212726	-0.199361	A
2	0.532045	-0.206465	A
3	0.642418	-0.352544	S
4	0.668096	-1.121914	S
5	0.668453	-1.021734	S
6	0.668458	-0.989315	S
7	0.668680	-0.925822	S
8	0.668898	-0.894760	S
9	0.669172	-0.834052	S
10	0.669230	-0.804385	S
11	0.669488	-0.775178	S
12	0.669853	-0.718122	S
13	0.670429	-0.635876	S
14	0.670766	-0.609388	S
15	0.671590	-0.532406	S
16	0.672007	-0.507672	S
17	0.672323	-0.483260	S
18	0.673208	-0.435795	S
19	0.674512	-0.389826	S
20	0.678426	-0.364217	S
21	0.682860	-0.307612	S
22	0.737416	-0.258717	A
23	0.766494	-0.231585	A
24	0.794818	-0.203529	A
25	0.851449	-0.147426	A

Bibliography

[1] Criminale, W. O., Jackson, T. L., & Joslin, R. D. (2018). *Theory and computation in hydrodynamic stability*. Cambridge University Press.

[2] Mack, L. M. (1976). A numerical study of the temporal eigenvalue spectrum of the blasius boundary layer. *Journal of Fluid Mechanics*, *73*(3), 497–520.

[3] Ng, B., & Reid, W. (1979a). An initial value method for eigenvalue problems using compound matrices. *Journal of Computational Physics*, *30*(1), 125–136.

[4] Ng, B., & Reid, W. (1979b). A numerical method for linear two-point boundary-value problems using compound matrices. *Journal of Computational Physics*, *33*(1), 70–85.