Ch. 7 - Initial and Boundary Value Problems of Differential Equations

Section 7.1 - Initial Value Problems: Euler, Runge-Kutta and Adams Methods

Start with a basic/essential problem:

Compute approx. sol'n of ordinary differential equation (**ODE**) with given initial value:

$$\frac{dy}{dt} = f(t, y) \ , \qquad y(0) = y_0$$

You may have seen the simple, classice approach: **Euler's method**

- "stepping" or "marching" method
- computes sequence of values at discrete "future" times based on existing "history":

$$y_n = y(t_n); \ t_n = t_0 + n\Delta t = t_0 + nh$$

- Independent vs dependent variables; discretize which kind?

Discretize **independent** variable

- Convert continuous domain to discrete domain
- "Step" along discrete values of independent variable
- Compute approximate values of dependent variable
- Basic plan:

1. Approximate the derivative with a simple forward difference estimator

$$\frac{dy}{dt} = f(t, y) \ , \qquad y(0) = y_0 \rightarrow \frac{y_{n+1} - y_n}{h} \approx f(t_n, y_n)$$

2. Knowing y_n and t_n, compute y_{n+1}:

$$y_{n+1} = y_n + hf(t_n, y_n)$$

Euler's method:

- Computes the rate of change at the beginning of a time step (from t_n to t_{n+1})
- Ignores the change in that rate of change over short time steps.
- Makes assumption that is, of course, not completely valid so truncation error is incurred.

$\exists$ numerous generalizations of Euler's method of the form

$$y_{n+1} = y_n + h\phi$$

- ϕ is some function of known values
- ϕ is chosen to reduce the error incurred during a timestep.
- Example
    - Euler formula can be used to estimate future values (beyond start of the step)

- Estimated derivatives at those times can be used to account for (and adjust for) how the rate of change varies across the timestep.

Heun's method:

1. Compute the right-hand side (RHS) at initial time (left end of the timestep).
2. Compute Euler estimate of y_{t+h}
3. Use Euler formula to estimate derivative at final time (right end of the timestep)
4. Use the average of the derivative estimates from left and right ends

Euler: $y_{n+1} = y_n + hf(t_n, y_n)$

Apply Euler formula to estimate value at end of step: $y_{euler} = y(t) + hf(t, y(t))$

Improve rate estimate by averaging estimated rates at ends of interval:

$$y(t+h) = y(t) + \frac{h}{2}\big(f(t, y(t)) + f(t+h, y_{euler})\big); \qquad \text{Eq.}(7.1.13\text{a})$$

Modified Euler-Cauchy:

1. Compute derivate (RHS) at initial time (left side of interval)
2. Use that derivative value to compute "Euler" estimate of derivative at mid-interval.
3. Use mid-interval value to estimate rate of change over the interval.

$$rate_{left} = f(t, y(t))$$

$$y_{mid} = y(t) + \frac{h}{2} rate_{left}$$

$$rate_{mid} = f(t + \frac{h}{2}, y_{mid})$$

$$y_{RK2}(t+h) = y(t) + h(rate_{mid})$$

Put the pieces together:

$$y_{RK2}(t+h) = y(t) + hf\big(t + \frac{h}{2}, y(t) + \frac{h}{2}\ f(t, y(t))\big); \qquad \text{Eq.}7.1.13\text{b}$$

Why is modified Euler-Cauchy given subscript "RK2"?

Runge-Kutta (RK) Methods

- Family of numerical ODE solvers that aim to reduce truncation error.
- Aim to reduce truncation error by including rate estimates at additional points in the timestep.
- Increase the order of truncation error by making the neglected terms involve higher powers of h
- Euler's method and modified Euler-Cauchy method are first 2 methods in the R-K family.

A very commonly used member of the family is **Fourth-Order Runge-Kutta (RK4)**:

1. Compute initial rate estimate: $f_1 = f(t_n, y_n)$

2. Use initial rate estimate to estimate midstep values: $f_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2} f_1)$
3. Use the midstep estimate to compute improved midstep estimate

$$f_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2} f_2)$$

4. Use improved midstep estimate to compute right-side estimate

$$f_4 = f(t_n + h, y_n + h f_3)$$

5. Compute weighted sum of contributions with coefficients chosen to cancel terms in the Taylor series. Achieve local truncation error $\sim O(h^5) \implies$ global error $\sim O(h^4)$:

$$y_{n+1} = y_n + \frac{h}{6}[f_1 + 2f_2 + 2f_3 + f_4]$$

We will return for a closer look at error analysis, but for now pause to consider what is nice about Runge-Kutta methods:

Do you need previous values of y (before y_n) to compute y_{n+1}?

No, so given y_0, $f(t, y)$ you have what you need to compute $y_{n+1} \iff$ **self-starting**

Computing the RK4 estimate involves computing everything we need for the lower order RK2 estimate. What could we do with such information?

Estimate local error $\implies$ possible to implement automated adaptive stepsize control.

Adams method: Multi-stepping techniques

What we were really doing above is approximating the fundamental theorem of calculus (that a function is the integral of its derivative):

$$y_{n+1} = y_n + \int_{t_n}^{t_n + h} f(t, y) dt$$

For Runge-Kutta, we estimated the interval based on information at t_n.

Adams methods take an alternative approach:

Use previous values to construct a locally valid polynomial approximant $p(t, y)$ and integrate the polynomial:

$$y_{n+1} = y_n + \int_{t_n}^{t_n + h} p(t, y) dt$$

Adams-Bashforth methods Construct the polynomial based on the current point and n previous points.

The first member of the family with $n = 0$ uses an order 0 polynomial:

$$p_1(t) = \text{constant} = f(t_n, y_n)$$

leading to

$$y_{n+1} = y_n + \int_{t_n}^{t_n+h} f(t_n, y_n)\, dt = y_n + hf(t_n, y_n)$$

which should look familiar (because it is Euler's method).

Aside on notation: The convention followed in the book is that the subscript on the polynomial indicates the number of points used, so the subscript is one greater than the degree of the polynomial.

Start to get something new when using 2 points, y_{n-1}, y_n, and degree 1 polynomial:

$$p_2 = f_{n-1} + \frac{f_n - f_{n-1}}{h}(t - t_{n-1})$$

$$y_{n+1} = y_n + \int_{t_n}^{t_n+h} p(t, y) dt$$

$$y_{n+1} = y_n + \int_{t_n}^{t_n+h} \left(f_{n-1} + \frac{f_n - f_{n-1}}{h}(t - t_{n-1})\right) dt$$

$$y_{n+1} = y_n + \frac{h}{2}\left[3f(t_n, y_n) - f(t_{n-1}, y_{n-1})\right]; \qquad \text{Eq.}(7.1.23)$$

This is a two-step method: Need y_{n-1} and y_n to compute y_{n+1}, so NOT self-starting.
Consider initial state: given y_0 **NOT** enough info to compute y_1 Need **bootstrapping** technique to get started.

Let's look at implementations of these basic methods:

Euler's method

Let's start with a familiar initial value problem:

$$\frac{dy}{dt} = -y\ ; \qquad y(0) = 1$$

```
import numpy as np
# define "right-hand side" function giving rate of change
def rhs(y,t):
    """
    compute right-hand side function specifying rate of change
    Args:
        t: float value of independent variable
        y: float value of dependent variable
    Returns:
        rate: float value for rate of change of y
    """
    rate = -y
    return rate
```

Now implement and test a function to take a single step:

```
def euler_step(f,y,t0,t1):
    """
```

```
    compute next value for Euler's method ODE solver
    Args:
        f: name of right-hand side function that gives rate of change of y
        y: float value of dependent variable
        t0: float initial value of independent variable
        t1: float final value of independent variable

    Returns:
        y_new: float estimated value of y(t1)
    """
    f0 = f(y,t0)
    h = (t1-t0)
    y_new = y + h*f0
    return y_new

euler_step(rhs,1,0,.1)
```

0.9

Construct and test a multistep Euler solver that can be used for longer intervals:

```
def euler_solve(f,y0,t):
    """
    Euler ODE solver
    Args:
        f: name of right-hand side function that gives rate of change of y
        y0: float initial value of dependent variable
        t: numpy array of float values of independent variable
    Returns:
        y: numpy array of float values of dependent variable
    """
    n = t.size
    y = [y0] #create solution as a list (of 1D numpy arrays) to which additional arra
    for i in range(n-1):
        y.append(euler_step(f,y[i],t[i],t[i+1])) #append new value to the list
    return np.array(y) #return the solution converted to a numpy array
steps = 4
t = np.linspace(0,1,steps+1)
y = euler_solve(rhs,1,t)
print("t values: ", t)
print("y values: ", y)
```

t values: [0. 0.25 0.5 0.75 1.] y values: [1. 0.75 0.5625 0.421875 0.31640625]

Compute values of exact solution $y(t) = exp(-t)$ for comparison:

```
n = t.size
y_exact = np.zeros(n)
for i in range(n):
    y_exact[i] = np.e**(-t[i])
```

```
error = (y_exact - y)
rel_error = (y_exact - y)/y_exact
print("e_abs: ", error)
print("e_rel: ", rel_error)
```

e_abs: [0. 0.02880078 0.04403066 0.05049155 0.05147319] e_rel: [0. 0.03698094 0.07259429 0.10689062 0.13991864]

Now that we are ready to look at plots, let's shift over to the notebook version...

and return for Section 7.2...

Section 7.2 - Error Analysis for Time-Stepping Routines

Explicit methods vs. Implicit methods

Earlier considered solving first order ODE

$$\frac{d\boldsymbol{y}}{dt} = f(\boldsymbol{y}, t)$$

using Euler's method

$$y_{n+1} = y_n + hf(t_n, y_n)$$

and observed relevant properties...

Observed properties of Euler's method:

- Discretization method: converts **differential equation** (with a **continuously valued independent variable**) to a **difference equation** (where the independent variable is replaced by an index value that takes on **discrete values**)
  - Equation for computing dependent value(s) $\boldsymbol{y}_{n+1}$ at next discrete value of idependent variable t_{n+1} is an **iteration formula** or **difference equation**.
- **Explicit method:** iteration formula specifies $\boldsymbol{y}_{n+1}$ as explicit function of values at t_n
  - $\boldsymbol{y}_{n+1}$ is isolated on the left side of the formula and does not appear on RHS.
  - More specific language: **explicit Euler method** or **forward Euler method**.
- Explicit Euler exhibits 1^{st}-order global truncation error (2^{nd}-order local error).
- Explicit Euler shows **instability** if stepsize is too large (compared to rate of change)
- Explicit Euler obtained using a 2^{nd}-order forward difference to estimate the derivative at the beginning (left edge) of the step.

Now consider alternative approach: estimate derivative at the end (right edge) of the step using the corresponding **backward difference** estimate:

Instead of $y'(t_n) = \frac{1}{h}(y_{n+1} - y_n)$, use $y'(t_{n+1}) = \frac{1}{h}(y_{n+1} - y_n)$ to obtain:

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}) \text{ instead of } y_{n+1} = y_n + hf(t_n, y_n)$$

New iteration formula specifies y_{n+1} **implicitly** y_{n+1} no longer appears as an isolated term, but appears in multiple terms in the iteration formula). This is called the **implicit Euler method** or **backward Euler method**.

In general, this would involve using a solver or rootfinding technique at each step (which would be expensive and, therefore, undesirable). Before coming back to a general fix to address this issue, let's return to our example problem:

$$y' = f(y,t) = \lambda y \; ; \; y(0) = y_0$$

Due to the simple rate specification, $f(u,t) = -\lambda y$, we can obtain an explicit iteration formula describing the approximate solution obtained using the implicit Euler method:

$$y_{n+1} = y_n + hf(t_n, y_{n+1}) = y_n + h\lambda y_{n+1} \implies (1-\lambda h)y_{n+1} = y_n$$

or

$$y_{n+1} = \frac{1}{1-\lambda h} y_n$$

We can solve this difference equation to obtain the implicit euler solution:

$$y_N = y_0 \big(\frac{1}{1-\lambda h}\big)^N$$

Compare this with the explicit Euler method:

$$y_{n+1} = (1+\lambda h)y_n \implies y_N = y_0(1+\lambda h)^N$$

Solution are geometric sequences.

Implications of geometric sequence solutions:

- Each iterate is a multiple of the previous iterate.
- Solutions grow or decay depending on magnitude of the multiplicative factor.
- The solution (and the error in the solution) decays when factor has magnitude < 1.
- Solution (and error) grow when factor has magnitude $> 1 \iff$ **instability**

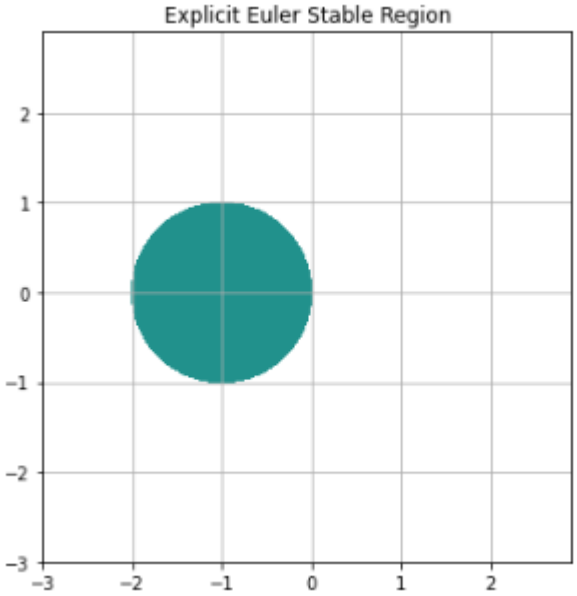
Seek **stability criterion:** for each method, when is multiplier magnitude < 1? For larger systems, multiplier arises as an eigenvalue which can be complex

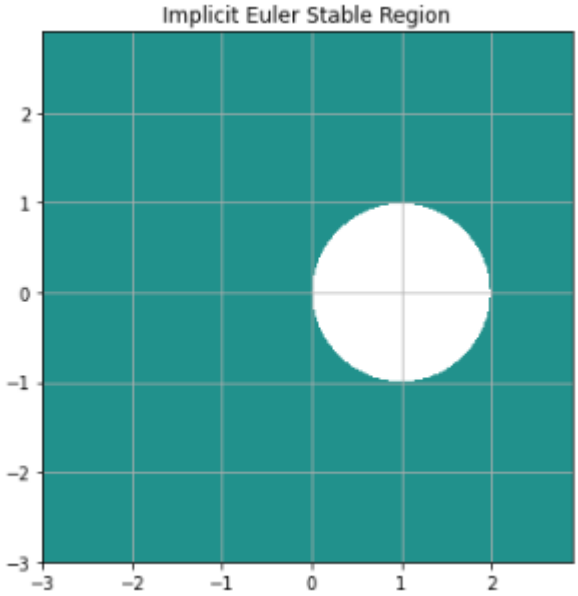
- Adopt complex notation: $\lambda h = z$
- Think of z as a complex variable (even though, in this simple case, we focus on what happens along the real axis).
- Explicit Euler:
  - multiplicative factor is $(1+z)$
  - Stable $\iff \big|1+z\big| < 1$
- Implicit Euler
  - multiplicative factor is $(1-z)^{-1}$
  - Stable $\iff \big|\frac{1}{1-z}\big| < 1$

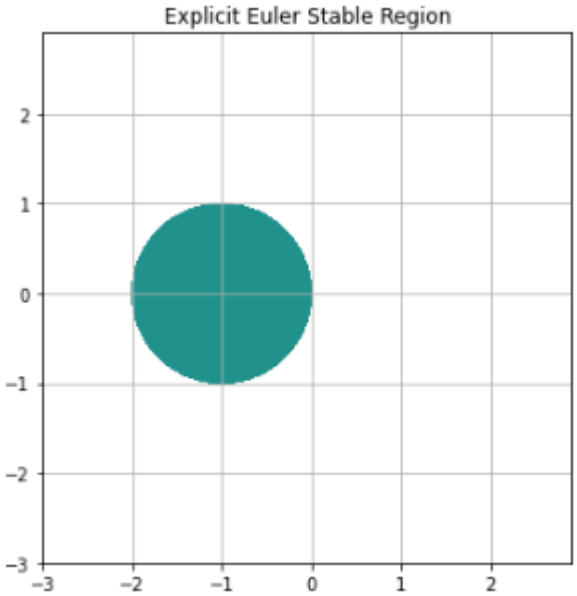
Stable regions (in complex plane and along real axis)

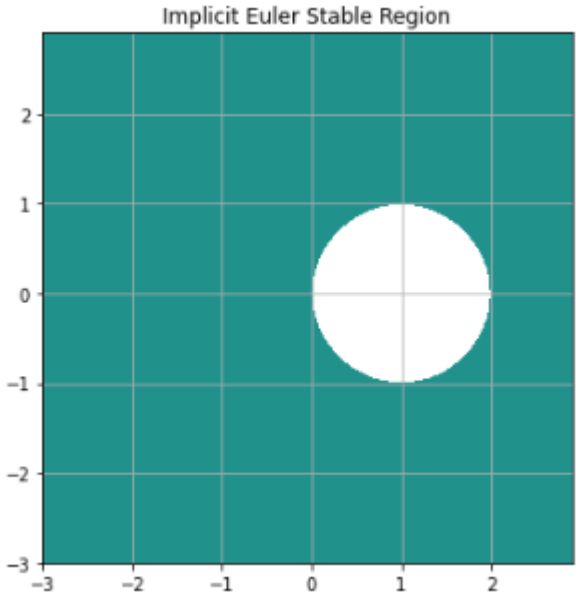
- Explicit Euler: $\big|1+z\big| < 1$
  - Interior of circle at $z = -1$ with radius 1; Project onto real axis: $-2 < \lambda h < 0$
- Implicit Euler: $\big|\frac{1}{1-z}\big| < 1 \implies |z-1| > 1$

- Exterior of circle at $z = 1$ with radius 1; Project onto real axis: $\lambda h < 0, \lambda h > 2$









See Ch. 7 notebook for code to produce these plots.

Interpretation of stability criteria

- Interesting case is when $\lambda > 0$ so the ODE solution is stable.
  - Explicit Euler: Stability requires $-2 < \lambda h < 0 \implies h < \frac{2}{|\lambda|}$
    - Example: $y' = -25y$ (i'e' $\lambda = 25$)
      - Observed stable solution for $h = 0.02$
      - Observed UNstable solution for $h = 0.1$
      - Now know that the cutoff point is $h = 2/25 = 0.08$
    - Choose stepsize too large, numerical solution goes unstable.
  - Implicit Euler is stable except for $0 < \lambda h < 2$
    - For interesting case with $\lambda < 0$, $\lambda h < 0$ (always in the stable region).
    - **Implicit Euler gives stable numerical solution for any stepsize**
    - Solution may be INACCURATE for large h, but it does not go unstable.

Enhanced stability makes the implicit approach attractive, but remember the added cost. In general, implicit approach requires rootfinding to compute the new dependent value that satisfies the iteration formula. Only for very simple cases does an implicit method leads to an iteration formula with explicit solution.

This has led to a middle ground **predictor-corrector** approach:

- Use an explicit method to compute a **predicted** value for y_{n+1}
- Then use an implicit method, plugging the predicted value in the rate equation, to compute a **corrected** value for y_{n+1}.

More specifically, the scheme above is a **predict-evaluate-correct (PEC)** approach.

Solving an ODE involves estimating a solution based on a given rate of change. Predictor-corrector approach aims to carry forward information about how the value is changing.

> Contrast with Runge-Kutta methods that compute a new value, then immediately ignore all the old values.)

In that context, it is natural to consider predictor-corrector schemes based on **multi-step methods** such as the Adams method that was mentioned above, but not yet implemented:

$$y_{n+1} = y_n + \frac{h}{2}\big[3f(t_n, y_n) - f(t_{n-1}, y_{n-1})\big]; \qquad \text{Eq.}(7.1.23)$$

Instead of rootfinding to solve for y_{n+1} in the Adams-Moulton formula, pair it with the corresponding Adams-Bashforth method:

- Use the explicit Adams-Bashforth method to compute a predicted value y^P_{n+1}
- Plug the prediction into the right-hand side of the implicit Adams-Moulton formula

$$y^P_{n+1} = y_n + \frac{h}{2}\big[3f(t_n, y_n) - f(t_{n-1}, y_{n-1})\big]; \qquad \text{Eq.}(7.1.29)$$
$$y_{n+1} = y_n + \frac{h}{2}\big[\ f(t_{n+1}, y^P_{n+1}) + f(t_n, y_n)\big]$$

Together the equations provide a way to employ an implicit method without the expense of solving nonlinear equations.

Section 7.3 - Advanced Time-Stepping Algorithms

Please read this section on your own to learn about exponential and adaptive stepping methods.

Section 7.4 - Boundary Value Problems: The Shooting Method

So far, we have dealt with **initial value problems (IVPs)**

- Given values at some initial "time"
- Step forward from there to wherever the solution takes us.

Now consider **boundary value problems (BVPs):**

- Conditions given at endpoints of an interval.
- Example: typical second order boundary value problem has the form:

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right)$$

with boundary conditions

$$\alpha_1 y(a) + \beta_1 \frac{dy(a)}{dt} = \gamma_1$$
$$\alpha_2 y(b) + \beta_2 \frac{dy(b)}{dt} = \gamma_2$$

The shooting method

One approach to such a problem is a shooting method:

- Satisfy the condition at one end of the interval and guess the other condition that should be satisfied there.
- Apply a stepping method to get a numerical solution at other end of the interval.
- Based on the "residual" (how far the solution is from satisfying the second condition), adjust the guess.
- Repeat until tolerance is satisfied.

To be more concrete, consider the basic case where the boundary conditions are

$$y(a) = \alpha; y(b) = \beta$$

(This notation is not exactly self-consistent, but it consistent with the text.)

As usual, we convert the second order ODE to a first order system in term of the vector

$$\boldsymbol{y} = (y, \frac{dy}{dt})^T = (y_0, y_1)^T$$

Now have an initial value of `y[0]`$= y$ at $t = a$, and we have to determine the initial value of the velocity `y[1]` at $t = a$:

$$\frac{dy(a)}{dt} = A$$

After some initial exploring to find value A_1 that produces a value $y(b) < \beta$ and A_2 that produces a value $y(b) > \beta$, apply bisection rootfinding method to find A^* that leads to $|y(b) - \beta| <$ tolerance.

General form of BVP:

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right)$$

$$\alpha_1 y(a) + \beta_1 \frac{dy(a)}{dt} = \gamma_1$$

$$\alpha_2 y(b) + \beta_2 \frac{dy(b)}{dt} = \gamma_2$$

Example Problem

Here is an implementation of the shooting method example from pp. 156-7 in Kutz.

$$y'' + (x^2 - sin\ x)y' - (cos^2\ x)y = 5 \qquad x \in [0, 1]$$

$$y(0) = 3; \quad y'(1) = 5$$

Apply procedure to reduce higher order ODEs:

1. Rearrange ODE with highest order derivative by itself: $\frac{dy_N}{dt} = RHS$
2. Replace every solitary independent variable on RHS with y_0
3. Replace every n-th order derivative on RHS with y_n
4. Write $f(\boldsymbol{y}, t) = \frac{d\boldsymbol{y}}{dt} = \frac{d}{dt}\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_N \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ RHS \end{pmatrix}$

Application:

1. $y'' = RHS = (cos^2\ x)y - (x^2 - sin\ x)y' + 5$
2. $RHS = (cos^2\ x)y_0 - (x^2 - sin\ x)y_1 + 5$
3. $$f(\boldsymbol{y}, t) = \frac{d\boldsymbol{y}}{dt} = \frac{d}{dt}\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} y_1 \\ (cos^2\ x)y_0 - (x^2 - sin\ x)y_1 + 5 \end{pmatrix}$$

$$y_0(0) = 3, \quad y_1(1) = 5$$

> Switch back to Ch. 7 notebook for implementation...

---

§ 7.5 - Implementation of Shooting and Convergence Studies

A significant applicatin of shooting methods involve computation of eigenvalues and eigenvectors.

- The text presents a complicated nonlinear example. You are welcome to take a look at that, but do not spend too much time. There are some issues with the presentation.
- Notebook presents a cleaner linearized version that works out nicely.

Start with "constant potential" version: $n(x) = n_0$ for $|x| < 1$ and $x = 0$ elsewhere. Choose length unit so that $L = 1$ so the problem becomes:

$$\frac{d^2\psi_n}{dx^2} - \beta_n\psi_n = 0$$

with boundary conditions

$$\psi(-1) = 0,\ \psi(1) = 0$$

You should know how to do some things with this system:

1. Write the analytic solution once you know an eigenvalue.
2. Write an equation you can solve for the eigenvalues using a rootfinding technique.

That said, let's see what happens if we apply a numerical ODE solver with a shooting technique where we use the conditions $y(-1) = 0,\ y'(-1) = A$ to create an IVP so we can use a stepping method to

compute $y(1)$.

Since this problem is linear, we can choose the scale ($y \to y/A$) so $y'(-1) = 1$. Shooting to find a solution with $|y_n(1)| < tol$ then becomes a matter of finding an appropriate value $\beta = \beta_n$.

Basic plan:

- Start with the largest reasonable value $\beta = n_0$ (above which the solutions of the ODE diverge)
- Solve the ODE and check the value at $x = 1$.
- Adjust or refine as appropriate.
    - Note that parity of the eigenfunction comes into play here: - For even modes, $y(1) > 0$ implies that β needs to increase to decrease $|y(1)|$
    - The opposite is true for odd modes.

With that plan in mind, check out the notebook implementation that aims to find the first 4 modes.

See § 7.5 in Ch. 7 notebook...

and continue reading through material from § 7.6 on Direct Solve and Relaxation methods.