02_BVP_problems

December 10, 2025

[43]:
```
from __future__ import print_function

%matplotlib inline

import numpy
import matplotlib.pyplot as plt
```

1 Boundary Value Problems: Discretization

1.0.1 Overview

These lectures cover the fundamentals of finite difference methods for boundary value problems (BVPs) in one dimension, including **Dirichlet** and **Neumann** problems, error analysis, stability, higher-order methods, and nonlinear BVPs.

Main references:
- Randall LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations* (SIAM, 2007).
- Z. Li, Z. Qiao, T. Tang, *Numerical Solution of Differential Equations.*

1.1 Model Problems

The simplest boundary value problem (BVP) we will run into is the one-dimensional version of Poisson's equation

$$u''(x) = f(x).$$

Usually we solve this equation on a finite interval with either Dirichlet or Neumann boundary conditions. Because there are two derivatives in the equation we need two boundary conditions to solve the PDE (really and ODE in this case) uniquely. To start let us consider the following basic problem

$$\begin{aligned} u''(x) = f(x) \quad & \Omega = [a, b] \\ u(a) = \alpha \quad & u(b) = \beta. \end{aligned}$$

BVPs of this sort are often the result of looking at the steady-state form of a time dependent PDE. For instance, if we were considering the steady-state solution to the heat equation

$$u_t(x,t) = \kappa u_{xx}(x,t) + \Psi(x,t) \quad \Omega = [0,T] \times [a,b] u(x,0) = u^0(x) \quad u(a,t) = \alpha(t) \quad u(b,t) = \beta(t)$$

we would solve the equation where $u_t = 0$ and arrive at

$$u''(x) = -\frac{\Psi}{\kappa},$$

a version of Poisson's equation above.

In higher spatial dimensions the second derivative turns into a Laplacian. Notation varies for this but all these are equivalent statements:

$$\begin{aligned} \nabla^2 u(\vec{x}\,) &= f(\vec{x}\,) \\ \Delta u(\vec{x}\,) &= f(\vec{x}\,) \\ \sum_{i=1}^{N} u_{x_i x_i} &= f(\vec{x}\,). \end{aligned}$$

1.2 Finite Difference Methods for 1D Boundary Value Problems

We consider the Dirichlet two-point boundary value problem in one dimension:

$$u''(x) = f(x), \quad x \in (a,b),$$

with boundary conditions

$$u(a) = \alpha, \quad u(b) = \beta.$$

This is a **two-point boundary value problem** because data are specified at both ends.

- **Dirichlet:** values of $u(x)$ at the endpoints.
- **Neumann:** values of $u'(x)$ at the endpoints.
- **Robin (mixed):** linear combination of $u(x)$ and $u'(x)$ at each endpoint.

1.2.1 A Finite Difference Method

Let $x \in [0,1]$. We want to solve

$$u''(x) = f(x), \quad x \in (0,1),$$

with boundary conditions

$$u(0) = \alpha, \quad u(1) = \beta.$$

We prescribe a grid of $m+2$ equally spaced points:

$$x_j = jh, \quad j = 0, 1, \dots, m+1, \quad h = \frac{1}{m+1}.$$

Grid function:

$$U_j \approx u(x_j), \quad j = 0, \dots, m+1.$$

- $U_0 = \alpha$,
- $U_{m+1} = \beta$,
- unknowns: $U_1, \dots, U_m$.

At interior grid points:

$$u''(x_j) = f(x_j), \quad j = 1, \dots, m.$$

Using the centered finite difference:

$$u''(x_j) \approx \frac{1}{h^2}\left(u(x_{j-1}) - 2u(x_j) + u(x_{j+1})\right) + O(h^2),$$

we obtain

$$\frac{1}{h^2}(U_{j-1} - 2U_j + U_{j+1}) = f(x_j), \quad j = 1, \dots, m.$$

Matrix form:

$$AU = F$$

with

$$A = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \ddots & \vdots \\ 0 & 1 & -2 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{bmatrix}.$$

Exercise: Solving a Boundary Value Problem with Finite Differences We want to solve the BVP

$$u_{xx} = e^x, \quad x \in [0, 1], \quad u(0) = 0, \quad u(1) = 3$$

using the finite difference method.

$$
\begin{aligned}
u_{xx} &= e^x \\
u_x &= A + e^x \\
u &= Ax + B + e^x \\
u(0) &= B + 1 = 0 \Rightarrow B = -1 \\
u(1) &= A - 1 + e^1 = 3 \Rightarrow A = 4 - e
\end{aligned}
$$

$$
u(x) = (4 - e)x - 1 + e^x
$$

Homework 1: Copy this into your script

```
import numpy
import matplotlib.pyplot as plt

# Problem setup
a = 0.0
b = 1.0
u_a = 0.0
u_b = 3.0
f = lambda x: numpy.exp(x)   # right-hand side function
u_true = lambda x: (4.0 - numpy.exp(1.0)) * x - 1.0 + numpy.exp(x)  # exact solution
```

Question 1 – Implement the solver Write a Python function that constructs and solves the finite difference approximation of the BVP:

```
def solve_bvp(a, b, u_a, u_b, f, m):
    """
    Solve the boundary value problem:
        u''(x) = f(x),   x   [a, b],
        u(a) = u_a,  u(b) = u_b

    Parameters
    ----------
    a : float
        Left boundary point
    b : float
        Right boundary point
    u_a : float
        Boundary condition at x = a
    u_b : float
        Boundary condition at x = b
    f : callable
        Function f(x) on the right-hand side
    m : int
        Number of interior grid points
```

```
    Returns
    -------
    x_bc : ndarray
        Array of all grid points including boundaries
    U : ndarray
        Approximate solution at all grid points
    """

    # Step 1: Discretize domain
    # x_bc = numpy.linspace(...)
    # x = interior points
    # h = step size

    # Step 2: Build matrix A (size m x m) for finite difference operator

    # Step 3: Build RHS vector, including boundary adjustments

    # Step 4: Solve linear system for interior values

    # Step 5: Reconstruct full solution with boundary values

    return x_bc, U
```

After writing the function, test it with:

```
m = 10
x_bc, U = solve_bvp(a, b, u_a, u_b, f, m)

plt.plot(x_bc, U, 'o-', label="Numerical")
plt.plot(x_bc, u_true(x_bc), 'k', label="Exact")
plt.xlabel("x")
plt.ylabel("u(x)")
plt.title("Solution to $u_{xx} = e^x$")
plt.legend()
plt.show()
```

Question 2 – Convergence analysis Use your function to test convergence.

1. Solve the problem for different values of `m`, for example:

   `m_values = [10, 20, 40, 80, 160]`

2. For each case, compute the error:

   `error = numpy.max(numpy.abs(U - u_true(x_bc)))`

3. Store the pair `(h, error)` where `h = (b - a)/(m+1)`.

4. Plot error versus `h` on a log-log scale.

5. Estimate the slope of the error curve → this corresponds to the **order of accuracy**.
6. Construct a table with:

- grid size `m`,
- step size `h`,
- error,
- ratio of convergence `p` between successive refinements.

Hint: The formula is:

$$p \approx \frac{\log\left(\frac{E_k}{E_{k+1}}\right)}{\log\left(\frac{h_k}{h_{k+1}}\right)}$$

where * E_k is the error at mesh size h_k, * E_{k+1} is the error at the finer mesh h_{k+1}.

If the method is second-order, they should observe $p \approx 2$.

1.3 Error Analysis

A natural question to ask given our approximation U_i is how close this is to the true solution $u(x)$ at the grid points x_i. To address this we will define the error E as

$$E_j = U_j - u(x_j), \quad j = 0, \ldots, m+1.$$

This leaves E as a vector still so often we ask the question how does the norm of E behave given a particular h. For the ∞-norm we would have

$$||E||_\infty = \max_{1 \le i \le m} |E_i| = \max_{1 \le i \le m} |U_i - u(x_i)|$$

If we can show that $||E||_\infty$ goes to zero as $h \to 0$ we can then claim that the approximate solution U_i at any of the grid points $E_i \to 0$. If we would like to use other norms we often define slightly modified versions of the norms that also contain the grid width h where

$$||E||_1 = h \sum_{i=1}^{m} |E_i|$$

$$||E||_2 = \left(h \sum_{i=1}^{m} |E_i|^2 \right)^{1/2}$$

These are referred to as *grid function norms*.

The E defined above is known as the *global error*. One of our main goals throughout this course is to understand how E behaves given other factors as we defined later.

1.3.1 Local Truncation Error

The *local truncation error* (LTE) can be defined by replacing the approximate solution U_i by the approximate solution $u(x_i)$. Since the algebraic equations are an approximation to the original

BVP, we do not expect that the true solution will exactly satisfy these equations, this resulting difference is the LTE.

For our one-dimensional finite difference approximation from above we have

$$\frac{1}{h^2}(U_{i+1} - 2U_i + U_{i-1}) = f(x_i).$$

Replacing U_i with $u(x_i)$ in this equation leads to

$$\tau_i = \frac{1}{h^2}(u(x_{i+1}) - 2u(x_i) + u(x_{i-1})) - f(x_i).$$

In this form the LTE is not as useful but if we assume $u(x)$ is smooth we can repalce the $u(x_i)$ with their Taylor series counterparts, similar to what we did for finite differences. The relevant Taylor series are

$$u(x_{i\pm1}) = u(x_i) \pm u'(x_i)h + \frac{1}{2}u''(x_i)h^2 \pm \frac{1}{6}u'''(x_i)h^3 + \frac{1}{24}u^{(4)}(x_i)h^4 + \mathcal{O}(h^5)$$

This leads to an expression for τ_i of

$$\begin{aligned}
\tau_i &= \frac{1}{h^2}\left[u''(x_i)h^2 + \frac{1}{12}u^{(4)}(x_i)h^4 + \mathcal{O}(h^5)\right] - f(x_i) \\
&= u''(x_i) + \frac{1}{12}u^{(4)}(x_i)h^2 + \mathcal{O}(h^4) - f(x_i) \\
&= \frac{1}{12}u^{(4)}(x_i)h^2 + \mathcal{O}(h^4)
\end{aligned}$$

where we note that the true solution would satisfy $u''(x) = f(x)$.

As long as $ u^{(4)}(x_i) $ remains finite (smooth) we know that $\tau_i \to 0$ as $h \to 0$

Let $\hat{U}$ be the $m \times 1$ column vector whose components are given by the exact function values

$$\hat{U} = \begin{bmatrix} u(x_1) \\ u(x_2) \\ \vdots \\ u(x_m) \end{bmatrix}.$$

Then, the column vector E of errors is re-defined as

$$E = U - \hat{U}.$$

We can also write the vector of LTEs as

$$\tau = A\widehat{U} - F$$

which implies

$$A\widehat{U} = F + \tau.$$

Note that here it is the exact solution evaluated stencil, not the approximated function.

1.3.2 Global Error

What we really want to bound is the global error E. To relate the global error and LTE we can substitute $E = U - \widehat{U}$ into our expression for the LTE to find

$$AE = -\tau.$$

This means that the global error is the solution to the system of equations we defined for the approximation except with τ as the forcing function rather than F!

This also implies that the global error E can be thought of as an approximation to similar BVP as we started with where

$$e''(x) = -\tau(x) \quad \Omega = [0,1] e(0) = 0 \quad e(1) = 0.$$

We can solve this ODE directly by integrating twice since to find to leading order

$$\begin{aligned} e(x) &\approx -\frac{1}{12}h^2 u''(x) + \frac{1}{12}h^2(u''(0) + x(u''(1) - u''(0))) \\ &= \mathcal{O}(h^2) \to 0 \quad \text{as} \quad h \to 0. \end{aligned}$$

1.3.3 Stability

We showed that the continuous analog to E, $e(x)$, does in fact go to zero as $h \to 0$ but what about E? Instead of showing something based on $e(x)$ let's look back at the original system of equations for the global error

$$A_h E_h = -\tau_h$$

where we now denote a particular realization of the system by the corresponding grid spacing h.

If we could invert A_h we could compute E_h directly. Assuming that we can and taking an appropriate norm we find

$$\begin{aligned} E_h &= (A_h)^{-1}\tau_h \\ ||E_h|| &= ||(A_h)^{-1}\tau_h|| \\ &\leq ||(A_h)^{-1}||\ ||\tau_h|| \end{aligned}$$

We know that $\tau_h \to 0$ as $h \to 0$ already for our example so if we can bound the norm of the matrix $(A_h)^{-1}$ by some constant C for sufficiently small h we can then write a bound on the global error of

$$||E_h|| \leq C||\tau_h||$$

demonstrating that $E_{h} \to 0$ at least as fast as $\tau_h \to 0$.

We can generalize this observation to all linear BVP problems by supposing that we have a finite difference approximation to a linear BVP of the form

$$A_h U_h = F_h,$$

where h is the grid spacing.

We say the approximation is *stable* if $(A_h)^{-1}$ exists $\forall h < h_0$ and there is a constant C such that

$$||(A_h)^{-1}|| \leq C \quad \forall h < h_0.$$

1.3.4 Consistency

A related and important idea for the discretization of any PDE is that it be consistent with the equation we are approximating. If

$$||\tau_h|| \to 0 \quad \text{as} \quad h \to 0$$

then we say an approximation is *consistent* with the differential equation.

1.3.5 Convergence

We now have all the pieces to say something about the global error E. A method is said to be *convergent* if

$$||E_h|| \to 0 \quad \text{as} \quad h \to 0.$$

If an approximation is both consistent ($||\tau_h|| \to 0$ as $h \to 0$) and stable ($||E_h|| \leq C||\tau_h||$) then the approximation is convergent.

We have only derived this in the case of linear BVPs but in fact these criteria for convergence are often found to be true for any finite difference approximation (and beyond for that matter). This statement of convergence can also often be strengthened to say

$$\mathcal{O}(h^p) \text{ LTE } + \text{ stability } \Rightarrow \mathcal{O}(h^p) \text{ global error.}$$

It turns out the most difficult part of this process is usually the statement regarding stability. In the next section we will see for our simple example how we can prove stability in the 2-norm.

1.3.6 Stability in the 2-Norm

Recalling our definition of stability, we need to show that for our previously defined A that

$$(A_h)^{-1}$$

exists and

$$||(A_h)^{-1}|| \leq C \quad \forall h < h_0$$

for some C.

How do we know that $(A_h)^{-1}$ exists?

We can assume now that A is in fact invertible but can we bound the norm of the inverse? Recall that the 2-norm of a symmetric matrix is equal to its spectral radius,

$$||A||_2 = \rho(A) = \max_{1 \leq p \leq m} |\lambda_p|.$$

Since the inverse of A is also symmetric the eigenvalues of A^{-1} are the inverses of the eigenvalues of A implying that

$$||A^{-1}||_2 = \rho(A^{-1}) = \max_{1 \leq p \leq m} \left|\frac{1}{\lambda_p}\right| = \frac{1}{\min_{1 \leq p \leq m} |\lambda_p|}.$$

If none of the λ_p of A are zero for sufficiently small h and the rest are finite as $h \to 0$ we have shown the stability of the approximation.

The eigenvalues of the matrix A from above can be written as

$$\lambda_p = \frac{2}{h^2}(\cos(p\pi h) - 1)$$

with the corresponding eigenvectors v^p

$$v_j^p = \sin(p\pi jh)$$

as the j-th component with $j = 1, \dots, m$.

If we can show that the eigenvalues are away from the origin then we know $||A||_2$ will be bounded. In this case the eigenvalues are negative so we need to show that they are always strictly less than zero.

$$\lambda_p = \frac{2}{h^2}(\cos(p\pi h) - 1)$$

Use a Taylor series to get an idea of how this behaves with respect to h

---

Exercise Check that (λ_p, v_j^p) are in fact the eigenpairs of the matrix A:

$$\begin{aligned}
(Av^p)_j &= \frac{1}{h^2}(v_{j-1}^p - 2v_j^p + v_{j+1}^p) \\
&= \frac{1}{h^2}(\sin(p\pi(j-1)h) - 2\sin(p\pi jh) + \sin(p\pi(j+1)h)) \\
&= \frac{1}{h^2}(\sin(p\pi jh)\cos(p\pi h) - 2\sin(p\pi jh) + \sin(p\pi jh)\cos(p\pi h) \\
&= \lambda_p v_j^p.
\end{aligned}$$

Exercise Compute $\min_{1 \le p \le m} |\lambda_p|$:

$$\begin{aligned}
\lambda_1 &= \frac{2}{h^2}(\cos(p\pi h) - 1) \\
&= \frac{2}{h^2}\left(-\frac{1}{2}p^2\pi^2h^2 + \frac{1}{24}p^4\pi^4h^4 + \mathcal{O}(\Delta^6)\right) \\
&= -p^2\pi^2 + \mathcal{O}(h^2).
\end{aligned}$$

Note that this also gives us an error bound as this eigenvalue also will also lead to the largest eigenvalue of the inverse matrix. We can therefore say

$$||E^h||_2 \le ||(A^h)^{-1}||_2 ||\tau^h||_2 \approx \frac{1}{\pi^2}||\tau^h||_2.$$

1.3.7 Stability in the ∞-Norm

The straight forward approach to show that $||E||_\infty \to 0$ as $h \to 0$ would be to use the matrix bound

$$||E||_\infty \leq \frac{1}{\sqrt{h}}||E||_2.$$

For our example problem we showed that $||E||_2 = \mathcal{O}(h^2)$ so this implies that we at least know that $||E||_\infty = \mathcal{O}(h^{3/2})$. This is unfortunate as we expect $||E||_\infty = \mathcal{O}(h^2)$ due to the discretization. In order to alleviate this problem let's go back and consider our definition of stability but this time consider the ∞-norm.

Our matrix A can be seen as a number of discrete approximations to *Green's functions* in each column. This is more broadly applicable later on so we will spend some time reviewing the theory of Green's functions and apply them to our simple example problem.

[ ]:

1.3.8 Green's Functions

Consider the BVP with Dirichlet boundary conditions

$$u''(x) = f(x) \quad \Omega = [0,1] u(0) = \alpha \quad u(1) = \beta.$$

Pick a fixed point $\bar{x} \in \Omega$, the Green's function $G(x;\bar{x})$ solves the BVP above with

$$f(x) = \delta(x - \bar{x})$$

and $\alpha = \beta = 0$. You could think of this as the result of a steady-state problem of the heat equation with a point-loss of heat somewhere in the domain.

To find the Green's function for our particular problem we can integrate just around the point $\bar{x}$ near the δ function source to find

$$\int_{\bar{x}-\epsilon}^{\bar{x}+\epsilon} u''(x)dx = \int_{\bar{x}-\epsilon}^{\bar{x}+\epsilon} \delta(x-\bar{x})dx$$
$$u'(\bar{x}+\epsilon) - u'(\bar{x}-\epsilon) = 1$$

recalling that by definition the integral of the δ function must be 1 if the interval of integration includes $\bar{x}$. We see that the jump in the derivative at $\bar{x}$ from the left and right should be 1.

After a bit of algebra we can solve for the Green's function for our model BVP as

$$G(x;\bar{x}) = \begin{cases} (\bar{x}-1)x & 0 \leq x \leq \bar{x} \\ \bar{x}(x-1) & \bar{x} \leq x \leq 1 \end{cases}.$$

One imporant property of linear PDEs (or ODEs) in general is that they exhibit the principle of superposition. The reason we care about this with Green's functions is that if we have a $f(x)$ composed of two δ functions, it turns out the solution is the sum of the corresponding two Green's functions. For instance if

$$f(x) = \delta(x - 0.25) + 2\delta(x - 0.5)$$

then

$$u(x) = G(x; 0.25) + 2G(x; 0.5).$$

This of course can be extended to an infinite number of δ functions so that

$$f(x) = \int_0^1 f(\bar{x})\delta(x - \bar{x})d\bar{x}$$

and therefore

$$u(x) = \int_0^1 f(\bar{x})G(x; \bar{x})d\bar{x}.$$

To incorporate the effects of boundary conditions we can continue to add Green's functions to the solution to find the general solution of our original BVP as

$$u(x) = \alpha(1 - x) + \beta x + \int_0^1 f(\bar{x})G(x; \bar{x})d\bar{x}.$$

So why did we do all this? Well the Green's function solution representation above can be thought of as a linear operator on the function $f(x)$. Written in perhaps more familiar terms we have

$$\mathcal{A}u = f \quad u = \mathcal{A}^{-1}f.$$

We see now that our linear operator $\mathcal{A}$ may be the continuous analog to our discrete matrix A.

To proceed we will modify our original matrix A into a slightly different version based on the same discretization. Instead of moving the boundary terms to the right-hand-side of the equation instead we will introduce two new "unknowns", called *ghost cells*, that will be placed at the edges of the grid. We will label these U_0 and U_{m+1}. In reality we know the value of these points, they are the boundary conditions!

The modified system then looks like

$$A = \frac{1}{h^2}\begin{bmatrix} h^2 & 0 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 & 1 \\ & & & & & 0 & h^2 \end{bmatrix} \qquad U = \begin{bmatrix} U_0 \\ U_1 \\ \vdots \\ U_m \\ U_{m+1} \end{bmatrix} \qquad F = \begin{bmatrix} \alpha \\ f(x_1) \\ \vdots \\ f(x_m) \\ \beta \end{bmatrix}$$

This has the advantage later that we can implement more general boundary conditions and it isolates the algebraic dependence on the boundary conditions. The drawbacks are that the matrix no longer has as simple of a form as before.

Let's finally turn to the form of the matrix A^{-1}. Introducing a bit more notation, let A_j denote the jth column and A_{ij} denote the ith jth element of the matrix A.

We know that

$$AA_j^{-1} = e_j$$

where e_j is the unit vector with 1 in the jth row (jth column of the identity matrix).

Note that the above system has some similarities to a discretized version of the Green's function problem. Here e_j represents the δ function, A the original operator, and A_j^{-1} the effect that the jth δ function (corresponding to the $\bar{x}$) has on the full solution.

It turns out that we can write down the inverse matrix directly using Green's functions (see LeVeque for the details) but we end up with

$$A_{ij}^{-1} = hG(x_i; x_j) = \begin{cases} h(x_j - 1)x_i, & i = 1, 2, \dots j \\ h(x_i - 1)x_j, & i = j, j+1, \dots, m \end{cases}.$$

We can also write the effective right-hand side of our system as

$$F = \alpha e_0 + \beta e_{m+1} + \sum_{j=1}^{m} f_j e_j$$

and finally the solution as

$$U = \alpha A_0^{-1} + \beta A_{m+1}^{-1} + \sum_{j=1}^{m} f_j A_j^{-1}$$

whose elements are

$$U_i = \alpha(1 - x_i) + \beta x_i + h \sum_{j=1}^{m} f_j G(x_i; x_j).$$

Alright, where has all this gotten us? Well, since we now know what the form of A^{-1} is we may be able to get at the ∞-norm of this matrix. Recall that the ∞-norm of a matrix (induced from the ∞-norm) for a vector is

$$||C||_\infty = \max_{0 \le i \le m+1} \sum_{j=0}^{m+1} |C_{ij}|$$

Note that due to the form of the matrix A^{-1} the first row's sum is

$$\sum_{j=0}^{m+1} A_{0j}^{-1} = 1$$

as is the last rows A_{m+1}^{-1}. We also know that for the other rows $A_{i,0}^{-1} < 1$ and $A_{i,m+1}^{-1} < 1$.

The intermediate rows are also all bounded as

$$\sum_{j=0}^{m+1} |A_{ij}^{-1}| \le 1 + 1 + mh < 3$$

using the fact we know that

$$h = \frac{1}{m+1}.$$

This completes our stability wanderings as we can now say definitively that

$$||A^{-1}||_\infty < 3 \quad \forall h.$$

1.4 Neumann Boundary Conditions

So far, we studied Dirichlet problems (values of $u(x)$ at boundaries). Now consider a **mixed problem**:

$$u''(x) = f(x) \quad \Omega = [a, b] u(a) = \alpha \quad u'(b) = \sigma.$$

We can incorporate other types of boundary conditions into our discretization using the modified version of our matrix. Let's try to do this for our original problem but with one side having Neumann boundary conditions:

Steps:
1. Grid: $x_j = a + jh$, $j = 0, \ldots, m+1$, $h = (b-a)/(m+1)$.
2. Grid function: $U_j \approx u(x_j)$.
3. PDE at interior: $u''(x_j) = f(x_j)$, $j = 1, \ldots, m$.
4. Replace u'' with centered difference:

$$\frac{1}{h^2}(U_{j-1} - 2U_j + U_{j+1}) = f(x_j).$$

Boundary conditions:
- Dirichlet at $x = a$: $U_0 = \alpha$.
- Neumann at $x = b$: handled in several ways.

Exercise: Implementing Neumann Boundary Conditions in Finite Differences We want to solve the BVP

$$u''(x) = f(x), \quad x \in [-1, 1],$$

with **mixed boundary conditions**

$$u(-1) = \alpha, \qquad u'(1) = \sigma.$$

You are given the **exact solution**

$$u(x) = -(5 + e)x - (2 + e + e^{-1}) + e^x.$$

From this expression:

- the forcing term is $f(x) = u''(x) = e^x$,
- the left boundary value is $\alpha = u(-1) = 3$,
- the right boundary derivative is $\sigma = u'(1) = -5$.

Homework 2: Copy this into your script

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

# Exact solution and RHS
u_exact = lambda x: -(5.0 + np.e) * x - (2.0 + np.e + np.exp(-1.0)) + np.exp(x)
f = lambda x: np.exp(x)

# Domain and boundary data
a, b = -1.0, 1.0
alpha = u_exact(a)    # Dirichlet at x=-1
sigma = -5.0          # Neumann at x=1
```

Task 1 — Implement solver functions Write **three Python functions**, one for each way of enforcing the Neumann boundary condition:

1. **First-order one-sided** difference

$$u'(1) \approx \frac{u_N - u_{N-1}}{h} = \sigma$$

2. **Second-order one-sided** difference

$$u'(1) \approx \frac{-3u_N + 4u_{N-1} - u_{N-2}}{2h} = \sigma$$

3. **Second-order centered (ghost point)** Introduce ghost point u_{N+1} and use

$$u'(1) \approx \frac{u_{N+1} - u_{N-1}}{2h} = \sigma \quad \Rightarrow \quad u_{N+1} = u_{N-1} + 2h\sigma$$

Function template:

```
def solve_bvp_neumann(a, b, alpha, sigma, f, m, method="first_order"):
    """
    Solve u''(x) = f(x), x in [a,b],
    with u(a)=alpha and u'(b)=sigma.

    Parameters
    ----------
    a, b : floats
        Interval boundaries
    alpha : float
        Dirichlet boundary value at x=a
    sigma : float
        Neumann boundary value at x=b
    f : callable
```

```
        Function f(x) on RHS
    m : int
        Number of interior points
    method : str
        "first_order", "one_sided_2nd", or "centered_2nd"

    Returns
    -------
    x_bc : ndarray
        Grid points including boundaries
    U : ndarray
        Approximate solution at all grid points
    """
    # Step 1: build grid
    # x_bc = np.linspace(a, b, m+2)
    # x = x_bc[1:-1]
    # h = (b-a)/(m+1)

    # Step 2: build matrix A (m x m) for finite differences

    # Step 3: build RHS vector b_vec = f(x)

    # Step 4: enforce u(a)=alpha (Dirichlet)

    # Step 5: enforce u'(b)=sigma depending on `method`

    # Step 6: solve system, reconstruct full solution

    return x_bc, U
```

Each method modifies the **last interior equation** differently.

Task 2 — Test your solver Use:

```
m = 10
for method in ["first_order", "one_sided_2nd", "centered_2nd"]:
    x_bc, U = solve_bvp_neumann(a, b, alpha, sigma, f, m, method)
    plt.plot(x_bc, U, 'o-', label=f"{method}")
plt.plot(x_bc, u_exact(x_bc), 'k--', label="Exact")
plt.legend()
plt.title("Neumann BVP with different boundary treatments")
plt.show()
```

You should see that all three methods produce a solution close to the exact one.

Task 3 — Convergence study

1. Choose a sequence of grid sizes:

   `m_values = [10, 20, 40, 80, 160]`

2. For each method and each `m`:

   - compute solution with `solve_bvp_neumann`,
   - compute step size $h = (b-a)/(m+1)$,
   - compute error:

     `error = np.max(np.abs(U - u_exact(x_bc)))`

3. Compute the **observed order of accuracy** between successive grid sizes:

$$p = \frac{\log(E_k/E_{k+1})}{\log(h_k/h_{k+1})}.$$

4. Make a **table** with columns: `m, h, error, p`.

5. Plot **error vs h** in log-log scale for each method.

1.5 Existance and Uniqueness

One question that should be asked before embarking upon a numerical solution to any equation is whether the original is *well-posed*. Well-posedness is defined as a problem that has a unique solution and depends continuously on the input data (inital condition and boundary conditions are examples).

Consider the BVP we have been exploring but now add strictly Neumann boundary conditions

$$u''(x) = f(x) \quad \Omega = [0,1] u'(0) = \sigma_0 \quad u'(1) = \sigma_1.$$

We can easily discretize this using one of our methods developed above but we run into problems.

[44]:
```
# Problem setup
a = -1.0
b = 1.0
alpha = 3.0
sigma = -5.0
f = lambda x: numpy.exp(x)

# Descretization
m = 50
x_bc = numpy.linspace(a, b, m + 2)
x = x_bc[1:-1]
h = (b - a) / (m + 1)

# Construct matrix A
A = numpy.zeros((m + 2, m + 2))
diagonal = numpy.ones(m + 2) / h**2
```

```
A += numpy.diag(diagonal * -2.0, 0)
A += numpy.diag(diagonal[:-1], 1)
A += numpy.diag(diagonal[:-1], -1)

# Construct RHS
b = f(x_bc)

# Boundary conditions
A[0, 0] = -1.0 / h
A[0, 1] = 1.0 / h
A[-1, -1] = -1.0 / (h)
A[-1, -2] =  1.0 / (h)

b[0] = h / 2.0 * f(x_bc[0]) - alpha
b[-1] = h / 2.0 * f(x_bc[-1]) - sigma

# Solve system
try:
    U = numpy.linalg.solve(A, b)
except numpy.linalg.LinAlgError as e:
    print(e)
    import traceback
    traceback.print_exc()
```

```
Singular matrix
```

```
Traceback (most recent call last):
  File "/tmp/ipykernel_7276/1496036973.py", line 35, in <module>
    U = numpy.linalg.solve(A, b)
        ^^^^^^^^^^^^^^^^^^^^^^^^

  File "/home/victor/anaconda3/envs/numerics/lib/python3.11/site-
packages/numpy/linalg/linalg.py", line 409, in solve
    r = gufunc(a, b, signature=signature, extobj=extobj)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  File "/home/victor/anaconda3/envs/numerics/lib/python3.11/site-
packages/numpy/linalg/linalg.py", line 112, in _raise_linalgerror_singular
    raise LinAlgError("Singular matrix")
numpy.linalg.LinAlgError: Singular matrix
```

We can see why A is singular, the constant vector $e = [1, 1, 1, 1, 1, \ldots, 1]^T$ is in fact in the null-space of A. Our numerical method has actually demonstrated this problem is *ill-posed*! Indeed, since the boundary conditions are only on the derivatives there are an infinite number of solutions to the BVP (this could also occur if there were no solutions).

Another way to understand why this is the case is to examine this problem again as the steady-state problem originating with the heat equation. Consider the heat equation with $\sigma_0 = \sigma_1 = 0$ and $f(x) = 0$. This setup would preserve any heat in the rod as none can escape through the ends of the rod. In fact, the solution to the steady-state problem would simply to redistribute the heat in

the rod evenly across the rod based on the initial condition. We then would have a solution

$$u(x) = \int_0^1 u^0(x)dx = C.$$

The problem comes from the fact that the steady-state problem does not know about this bit of information by itself. This means that the BVP as it stands could pick out any C and it would be a solution.

The solution is similar if we had the same setup except $f(x) \neq 0$. Now we are either adding or subtracting heat in the rod. In this case there may not be a steady state at all! You can actually show that if the addition and subtraction of heat exactly cancels we may in fact have a solution if

$$\int_0^1 f(x)dx = 0$$

which leads again to an infinite number of solutions.

1.6 Higher Order Methods

So far: second-order schemes. Now: methods for **fourth-order accuracy**.

Why?
- Higher accuracy for same grid size.
- Better convergence without refining too much.

1.6.1 Fourth order finite differences

One approach is to replace the second-order finite difference approximation of the second-order derivative with a fourth-order finite difference approximation:

$$u''(x_j) \approx \frac{1}{12h^2}(-U_{j-2} + 16U_{j-1} - 30U_j + 16U_{j+1} - U_{j+2}).$$

To implement this finite difference approximation on our standard, uniform grid $x_j = jh$ for $j = 0, 1, \ldots, m, m+1$ with $h = (b-a)/(m+1)$, we find no problem for $j = 2, \cdots, m-1$. However, we find that we cannot apply this finite difference approximation for $j = 0, 1, m$ and $m+1$. For Dirichlet boundary conditions, we can set the values of U_0 and U_{m+1}, but we still do not know how to treat U_1 and U_m.

For U_1, we must use a fourth-order finite difference approximation that uses U_0, U_1 and values at grid points to the right of x_1. We can derive the following finite difference approximation:

$$u''(x_1) \approx \frac{1}{12h^2}(10U_0 - 15U_1 - 4U_2 + 14U_3 - 6U_4 + U_5).$$

Similarly, for U_m, we must use a fourth-order finite difference approximation of the form

$$u''(x_m) \approx \frac{1}{12h^2}(U_{m-4} - 6U_{m-3} + 14U_{m-2} - 4U_{m-1} - 15U_m + 10U_{m+1}).$$

Using this finite difference method, we obtain a nearly pentadiagonal matrix except for the first and last rows. The resulting linear system of equations can be solved efficiently in $O(m)$ operations.

1.6.2 Extrapolation Methods

Another method to obtain fourth-order accuracy is to use extrapolation methods. Consider our second-order finite difference method. Suppose we compute the solution on a grid with mesh spacing h which we denote by $U_j \approx u(x_j) = u(jh)$. Now, suppose we compute the solution on a finer grid with mesh spacing $h/2$ which we denote $Vj \approx u(jh/2)$. We know that because the method uses centered, second-order finite difference approximation, the odd powers of h in the error vanish identically so that

$$U_j = u(x_j) + C_2h^2 + C_4h^4 + C_6h^6 + ...$$

The coefficients C_2, C_4 , C_6 and so on are independent of h. Similarly,

$$\begin{aligned} V_{2j} &= u(x_j) + C_2\left(\frac{h}{2}\right)^2 + C_4\left(\frac{h}{2}\right)^4 + C_6\left(\frac{h}{2}\right)^6 + ... \\ &= u(x_j) + \frac{1}{4}C_2h^2 + \frac{1}{16}C_4h^4 + \frac{1}{64}C_6h^6 + ... \end{aligned}$$

Thus, if we consider the linear combination

$$\tilde{U}_j = aU_j + bV_{2j}$$

we find that

$$\begin{aligned} \tilde{U}_j = & \quad aU_j + bV_{2j} \\ = & \quad au(x_j) + aC_2h^2 + bu(x_j) + b\frac{1}{4}C_2h^2 + O(h^4) \\ = & \quad (a+b)u(xj) + (a + \frac{1}{4}b)C_2h^2 + O(h^4). \end{aligned}$$

Observe that if we set $a + b = 1$ and $a + b/4 = 0$ we find that $a = -1/3$ and $b = 4/3$ and so

$$\tilde{U}_j = u(x_j) + O(h^4).$$

Thus, if we compute the solution on the grid with mesh width h and obtain U_j and then compute the solution on the grid with mesh width $h/2$ and obtain V_j, then the quantity

$$\tilde{U}_j = \frac{1}{3}(4V_{2j} - U_j)$$

gives a fourth-order accurate solution on the grid with mesh width h.

This extrapolation method requires solving the problem numerically twice. While this may seem inefficient for the problems we have studied thus far, it is a useful approach when implementing a fourth-order finite difference approximation is difficult or impractical.

1.6.3 Deferred corrections

Another approach involves solving the problem twice, but on the same grid. Suppose we implement a second-order method that leads to the linear system $AU = F$. For this problem, we have found already that the error vector E satisfies

$$AE = -\tau.$$

Now, if we knew τ explicitly, we could solve this linear system of equations for the error and thus obtain the exact solution by computing $U - E$. However, we do not know the truncation error vector explicitly. We do know that

$$\tau_j = \frac{1}{12}h^2 u^{(iv)}(x_j) + O(h^4).$$

Using our computed solution, we can approximate $u^{(iv)}(x_j)$ by using a finite difference approximation. For the specific case in which \$u' '= f \$, we have $u^{(iv)} = f''$, so we can compute analytically or approximate instead the second-order derivative of f. In fact, we can combine these two problems into one and solve instead the linear system $AU = \tilde{F}$ with

$$\tilde{F}_j = f(x_j) + \frac{1}{12}h^2 f''(x_j).$$

The result will be an approximation that is fourth-order accurate. This method is called the method of *deferred corrections*.

1.7 General Linear Second Order Discretization

Let's now describe a method for solving the equation

$$a(x)u''(x) + b(x)u'(x) + c(x)u(x) = f(x) \quad \Omega = [a,b] u(a) = \alpha \quad u(b) = \beta.$$

Try discretizing this using second order finite differences and write the system for

$$a(x)u''(x) + b(x)u'(x) + c(x)u(x) = f(x) \quad \Omega = [a,b] u(a) = \alpha \quad u(b) = \beta.$$

The general, second order finite difference approximation to the above equation can be written as

$$a_i \frac{U_{i+1} - 2U_i + U_{i-1}}{h^2} + b_i \frac{U_{i+1} - U_{i-1}}{2h} + c_i U_i = f_i$$

leading to the matrix entries

$$A_{i,i} = -\frac{2a_i}{h^2} + c_i$$

on the diagonal and

$$A_{i,i\pm1} = \frac{a_i}{h^2} \pm \frac{b_i}{2h}$$

on the sub-diagonals. We can take of the boundary conditions by either using the ghost-points approach or by incorporating them into the right hand side evaluation.

1.7.1 Example

Consider the steady-state heat conduction problem with a variable $\kappa(x)$ so that

$$(\kappa(x)u'(x))' = f(x), \quad \Omega = [0,1] u(0) = \alpha \quad u(1) = \beta$$

By the chain rule we know

$$\kappa(x)u''(x) + \kappa'(x)u'(x) = f(x).$$

It turns out that in this case this approach is not really the best approach to solving the problem. In many cases it is best to discretize the original form of the physics rather than a perhaps equivalent formulation. To demonstrate this let's try to construct a system to solve the original equations

$$(\kappa(x)u'(x))' = f(x).$$

First we will approximate the expression

$$\kappa(x)u'(x)$$

but at the points half-way in between the points x_i, i.e. $x_{i+1/2}$.

We also will take this approximation effectively to be $h/2$ and find

$$\kappa(x_{i+1/2})u'(x_{i+1/2}) = \kappa_{i+1/2}\frac{U_{i+1} - U_i}{h}.$$

Now taking this approximation and differencing it with the same difference centered at $x_{i-1/2}$ leads to

$$\begin{aligned}(\kappa(x_i)u'(x_i))' &= \frac{1}{h}\left[\kappa_{i+1/2}\frac{U_{i+1} - U_i}{h} - \kappa_{i-1/2}\frac{U_i - U_{i-1}}{h}\right] \\ &= \frac{\kappa_{i+1/2}U_{i+1} - \kappa_{i+1/2}U_i - \kappa_{i-1/2}U_i + \kappa_{i-1/2}U_{i-1}}{h^2} \\ &= \frac{\kappa_{i+1/2}U_{i+1} - (\kappa_{i+1/2} - \kappa_{i-1/2})U_i + \kappa_{i-1/2}U_{i-1}}{h^2}\end{aligned}$$

Note that these formulations are actually equivalent to $\mathcal{O}(h^2)$. The matrix entries are

$$\begin{aligned}A_{i,i} &= -\frac{\kappa_{i+1/2} - \kappa_{i-1/2}}{h^2} \\ A_{i,i\pm1} &= \frac{\kappa_{i\pm1/2}}{h^2}.\end{aligned}$$

Note that this latter discretization is symmetric. This will have consequences as to how well or quickly we can solve the resulting system of linear equations.

1.8 Non-Linear Equations

Our model problem, Poisson's equation, is a linear BVP. How would we approach a non-linear problem? As a new model problem let's consider the non-linear pendulum problem. The physical system is a mass m connected to a rigid, massless rod of length L which is allowed to swing about a point. The angle $\theta(t)$ is taken with reference to the stable at-rest point with the mass hanging downwards.

This system can be described by

$$\theta''(t) = \frac{-g}{L}\sin(\theta(t)).$$

We will take $\frac{g}{L} = 1$ for convenience.

Looking at the Taylor series of sin we can approximate this equation for small θ as

$$\sin(\theta) \approx \theta - \frac{\theta^3}{6} + \mathcal{O}(\theta^5)$$

so that

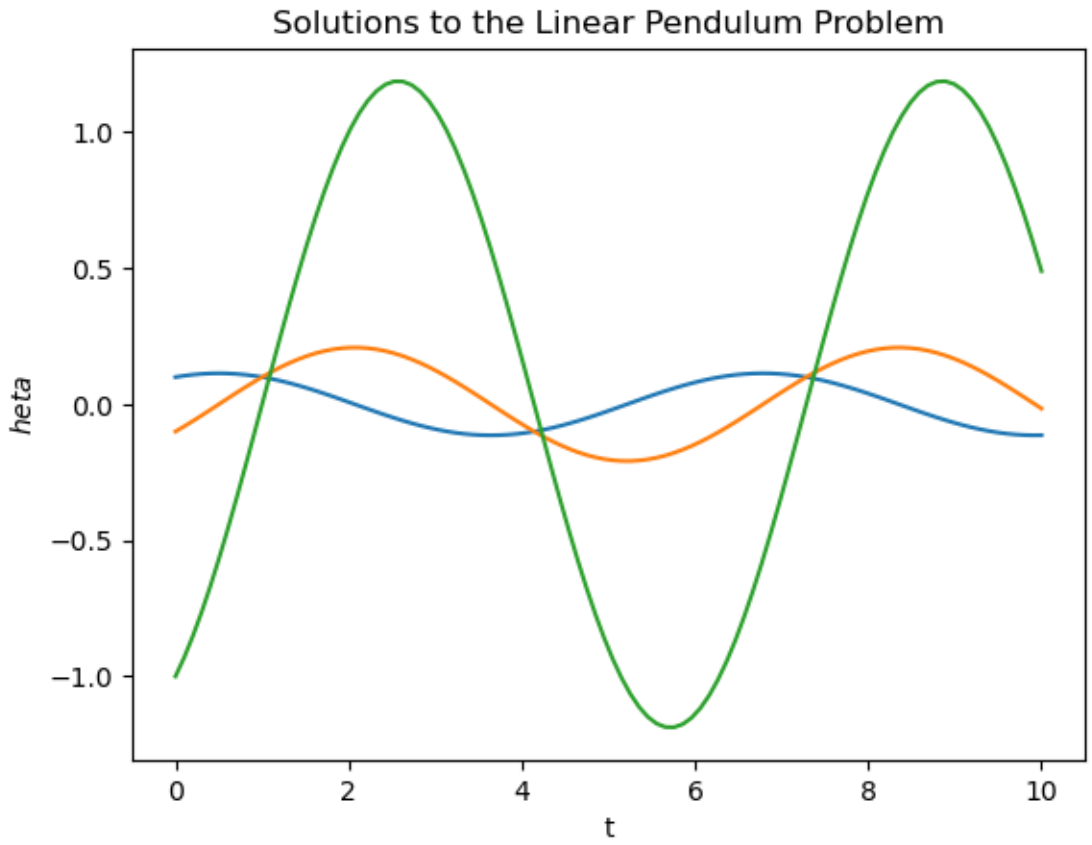
$$\theta'' = -\theta.$$

We know that this equation has solutions of the form

$$\theta(t) = C_1 \cos t + C_2 \sin t.$$

We clearly need two boundary conditions to uniquely specify the system which can be a bit awkward given that we usually specify these at two points in the spatial domain. Since we are in time we can specify the initial position of the pendulum $\theta(0) = \alpha$ however the second condition would specify where the pendulum would be sometime in the future, say $\theta(T) = \beta$. We could also specify another initial condition such as the angular velocity $\theta'(0) = \sigma$.

[45]:
```
# Simple linear pendulum solutions
def linear_pendulum(t, alpha=0.01, beta=0.01, T=1.0):
    C_1 = alpha
    C_2 = (beta - alpha * numpy.cos(T)) / numpy.sin(T)
    return C_1 * numpy.cos(t) + C_2 * numpy.sin(t)


alpha = [0.1, -0.1, -1.0]
beta = [0.1, 0.1, 0.0]
T = [1.0, 1.0, 1.0]
t = numpy.linspace(0, 10.0, 100)
fig = plt.figure()
axes = fig.add_subplot(1, 1, 1)
for i in range(len(alpha)):
    axes.plot(t, linear_pendulum(t, alpha[i], beta[i], T[i]))
axes.set_title("Solutions to the Linear Pendulum Problem")
axes.set_xlabel("t")
axes.set_ylabel("$\theta$")

plt.show()
```



But how would we go about handling the fully non-linear problem? First let's discretize using our approach to date with the second order, centered second derivative finite difference approximation to find

$$\frac{1}{\Delta t^2}(\theta_{i+1} - 2\theta_i + \theta_{i-1}) + \sin(\theta_i) = 0.$$

The most common approach to solving a non-linear BVP like this (and many non-linear PDEs for that matter) is to use Newton's method. Recall that if we have a non-linear function $G(\theta)$ and we want to find θ such that

$$G(\theta) = 0$$

we can expand $G(\theta)$ in a Taylor series to find

$$G(\theta^{[k+1]}) = G(\theta^{[k]}) + G'(\theta^{[k]})(\theta^{[k+1]} - \theta^{[k]}) + \mathcal{O}((\theta^{[k+1]} - \theta^{[k]})^2)$$

If we want $G(\theta^{[k+1]}) = 0$ we can set this in the expression above (this is also known as a fixed point iteration) and dropping the higher order terms we can solve for $\theta^{[k+1]}$ to find

$$\begin{aligned} 0 &= G(\theta^{[k]}) + G'(\theta^{[k]})(\theta^{[k+1]} - \theta^{[k]}) \\ G'(\theta^{[k]})\theta^{[k+1]} &= G'(\theta^{[k]})\theta^{[k]} - G(\theta^{[k]}) \end{aligned}$$

At this point we need to be careful, if we have a system of equations we cannot simply divide through by $G'(\theta^{[k]})$ (which is now a matrix) to find our new value $\theta^{[k+1]}$. Instead we need to invert the matrix $G'(\theta^{[k]})$. Another way to write this is as an update to the value $\theta^{[k+1]}$ where

$$\theta^{[k+1]} = \theta^{[k]} + \delta^{[k]}$$

where

$$J(\theta^{[k]})\delta^{[k]} = -G(\theta^{[k]}).$$

Here we have introduced notation for the **Jacobian matrix** whose elements are

$$J_{ij}(\theta) = \frac{\partial}{\partial \theta_j} G_i(\theta).$$

So how do we compute the Jacobian matrix? Since we know the system of equations in this case we can write down in general what the entries of J are.

$$\frac{1}{\Delta t^2}(\theta_{i+1} - 2\theta_i + \theta_{i-1}) + \sin(\theta_i) = 0.$$

$$J_{ij}(\theta) = \begin{cases} \frac{1}{\Delta t^2} & j = i-1, j = i+1 \\ -\frac{2}{\Delta t^2} + \cos(\theta_i) & j = i \\ 0 & \text{otherwise} \end{cases}$$

With the Jacobian in hand we can solve the BVP by iterating until some stopping criteria is met (we have converged to our satisfaction).

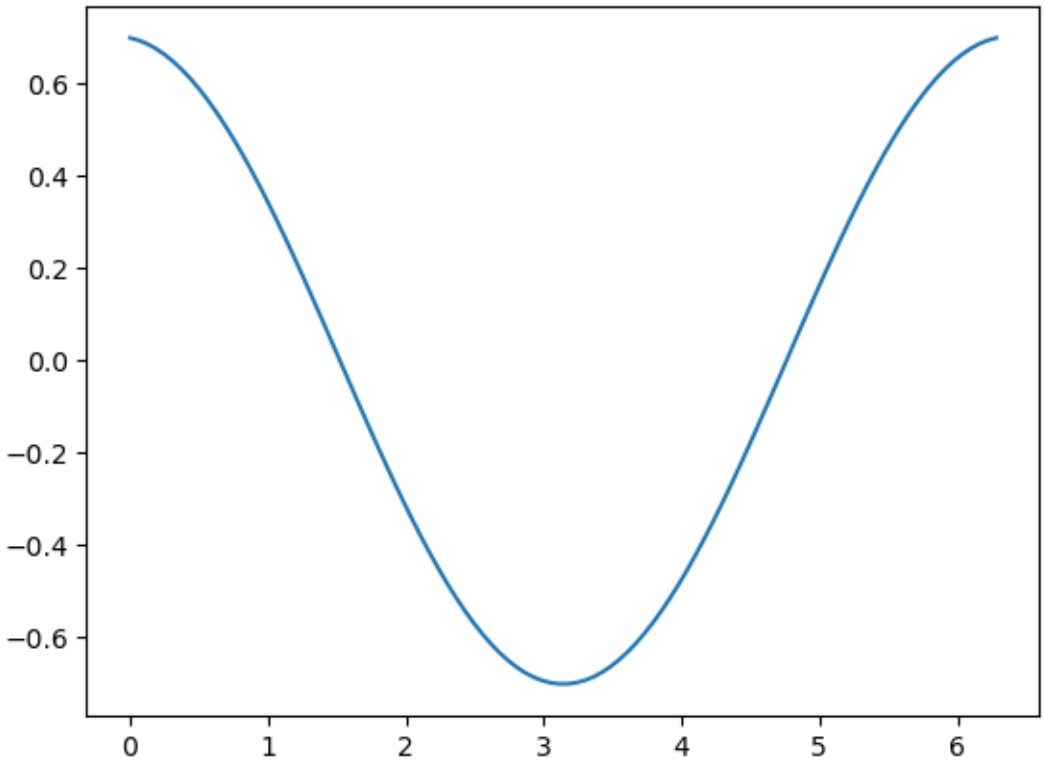
1.8.1 Example

Solve the linear and non-linear pendulum problem with $T = 2\pi$, $\alpha = \beta = 0.7$. - Does the linear equation have a unique solution - Do you expect the original problem to have a unique solution (i.e. does the non-linear problem have a unique solution)?

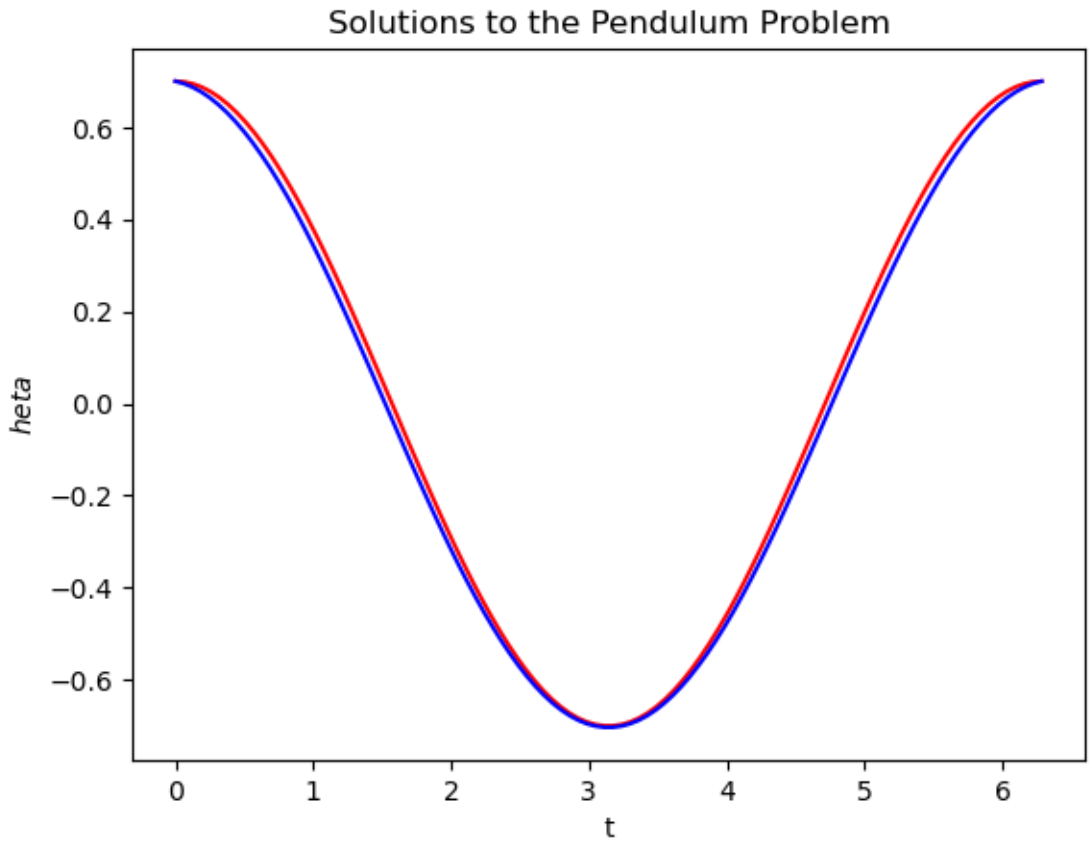
[46]:
```
def solve_nonlinear_pendulum(m, alpha, beta, T, max_iterations=100,␣
↪tolerance=1e-3, verbose=False):

    # Discretization
    t_bc = numpy.linspace(0.0, T, m + 2)
    t = t_bc[1:-1]
    delta_t = T / (m + 1)
    diagonal = numpy.ones(t.shape)
    G = numpy.empty(t_bc.shape)

    # Initial guess
    theta = 0.7 * numpy.cos(t_bc)
    theta[0] = alpha
    theta[-1] = beta
```

```
    # Main iteration loop
    success = False
    for num_step in range(1, max_iterations):

        # Construct Jacobian matrix
        J = numpy.diag(diagonal * -2.0 / delta_t**2 + numpy.cos(theta[1:-1]), 0)
        J += numpy.diag(diagonal[:-1] / delta_t**2, -1)
        J += numpy.diag(diagonal[:-1] / delta_t**2, 1)

        # Construct vector G
        G = (theta[:-2] - 2.0 * theta[1:-1] + theta[2:]) / delta_t**2 + numpy.
 ↪sin(theta[1:-1])

        # Take care of BCs
        G[0] = (alpha - 2.0 * theta[1] + theta[2]) / delta_t**2 + numpy.
 ↪sin(theta[1])
        G[-1] = (theta[-3] - 2.0 * theta[-2] + beta) / delta_t**2 + numpy.
 ↪sin(theta[-2])

        # Solve
        delta = numpy.linalg.solve(J, -G)
        theta[1:-1] += delta

        if verbose:
            print(" (%s) Step size: %s" % (num_step, numpy.linalg.norm(delta)))

        if numpy.linalg.norm(delta) < tolerance:
            success = True
            break

    if not success:
        print(numpy.linalg.norm(delta))
        raise ValueError("Reached maximum allowed steps before convergence␣
 ↪criteria met.")

    return t_bc, theta

t, theta = solve_nonlinear_pendulum(100, 0.7, 0.7, 2.0 * numpy.pi,␣
 ↪tolerance=1e-9, verbose=True)
plt.plot(t, theta)
plt.show()
```

```
(1) Step size: 0.25127822620106616
(2) Step size: 0.0006452832859492562
(3) Step size: 1.07602510886597e-08
(4) Step size: 3.0623701047904136e-14
```



[ ]:
```
# Linear Problem
alpha = 0.7
beta = 0.7
##### if we change alpha = 0.2 and beta = 0.2, I think it converges better
T = 2.0 * numpy.pi
t = numpy.linspace(0, T, 100)
fig = plt.figure()
axes = fig.add_subplot(1, 1, 1)
axes.plot(t, linear_pendulum(t, alpha, beta, T), 'r-', label="Linear")

# Non-linear problem
t, theta = solve_nonlinear_pendulum(100, alpha, beta, T)
axes.plot(t, theta, 'b-', label="Non-Linear")

axes.set_title("Solutions to the Pendulum Problem")
axes.set_xlabel("t")
axes.set_ylabel("$\theta$")

plt.show()
```



1.8.2 Accuracy

Note that there are two different ideas of convergence going on in our non-linear solver above, one is the convergence of the finite difference approximation controlled by h and the convergence of the Newton iteration. We expect both to be second order (Newton's method converges quadratically under suitable assumptions). How do these two methods combine to affect the global error though?

First let's compute the LTE

$$\begin{aligned}
\tau_i &= \frac{1}{\Delta t^2}(\theta(t_{i+1}) - 2\theta(t_i) + \theta(t_{i-1})) + \sin\theta(t_i) \\
&= \frac{1}{\Delta t^2}\left(\theta(t_i) + \theta'(t_i)\Delta t + \frac{1}{2}\theta''(t_i)\Delta t^2 + \frac{1}{6}\theta'''(t_i)\Delta t^3 + \frac{1}{24}\theta^{(4)}(t_i)\Delta t^4 - 2\theta(t_i)\right. \\
&\qquad \left. +\theta(t_i) - \theta'(t_i)\Delta t + \frac{1}{2}\theta''(t_i)\Delta t^2 - \frac{1}{6}\theta'''(t_i)\Delta t^3 + \frac{1}{24}\theta^{(4)}(t_i)\Delta t^4 + \mathcal{O}(\Delta t^5)\right) + \sin\theta(t_i) \\
&= \frac{1}{\Delta t^2}\left(\theta''(t_i)\Delta t^2 + \frac{1}{12}\theta^{(4)}(t_i)\Delta t^4\mathcal{O}(\Delta t^6)\right) + \sin\theta(t_i) \\
&= \theta''(t_i) + \sin\theta(t_i) + \frac{1}{12}\theta^{(4)}(t_i)\Delta t^2 + \mathcal{O}(\Delta t^4).
\end{aligned}$$

For Newton's method we can consider the difference of taking a step with the true solution to the

BVP $\hat{\theta}$ vs. the approximate solution θ. We can formulate an analogous LTE where

$$G(\Theta) = 0 \quad G(\hat{\Theta}) = \tau.$$

Following our discussion from before we can use these two expressions to find

$$G(\Theta) - G(\hat{\Theta}) = -\tau$$

and from here we want to derive an expression of the global error $E = \Theta - \hat{\Theta}$.

Since $G(\theta)$ is not linear we will write the above expression as a Taylor series to find

$$G(\Theta) = G(\hat{\Theta}) + J(\hat{\Theta})E + \mathcal{O}(||E||^2).$$

Using this expression we find

$$J(\hat{\Theta})E = -\tau + \mathcal{O}(||E||^2).$$

Ignoring higher order terms then we have a linear expression for E which we can solve.

This motivates another definition of stability then involving the Jacobian of G. The nonlinear difference methods $G(\Theta) = 0$ is *stable* in some norm $||\cdot||$ if the matrices $(J_{\Delta t})^{-1}$ are uniformly bounded in the norm as $\Delta t \to 0$. In other words $\exists C$ and Δt^0 s.t.

$$||(J_{\Delta t})^{-1}|| \leq C \quad \forall \Delta t < \Delta t^0.$$

Given this sense of stability and consistency ($||\tau|| \to 0$ as $\Delta t \to 0$) then the method converges as

$$||E_{\Delta t}|| \to 0 \quad \text{as} \quad \Delta t \to 0.$$

Note that we are still not guaranteed that Newton's method will converge, say from a bad initial guess, even though we have shown convergence. It can be proven that Newton's method will converge from a sufficiently good initial guess. It also should be noted that although Newton's method may have an error that is to round-off does not imply that the error will follow suit.

1.9 Exercises

Exercise 1.
Use the centered second-order finite difference formula to solve

$$u'' = -\pi^2 \cos(\pi x), \quad x \in (0,1), \quad u(0) = 1,\ u(1) = -1.$$

Exercise 2.
Show that

$$u_p(x) = \sin(p\pi x), \quad p = 1, 2, 3, ...$$

are eigenfunctions of $\frac{\partial^2}{\partial x^2}$ with eigenvalues

$$\lambda_p = -p^2\pi^2.$$

Verify that

$$u_{pj} = \sin(p\pi jh), \quad j = 1, \ldots, m$$

are eigen-grid functions of A with eigenvalues

$$\lambda_p = \frac{2}{h^2}(\cos(p\pi h) - 1).$$

Exercise 3.
Consider the boundary value problem

$$u'' = 3u - 2u', \quad 0 < x < 1, \quad u(0) = e^3,\ u(1) = 1.$$

1. Construct a second-order finite difference method and write as $AU = F$.

2. Determine analytically the solution.

3. Verify numerically that the method is second-order accurate.

Exercise 4.
Solve $u''(x) = f(x)$, $0 < x < 1/2$, with $u(0) = 1$, $u'(1/2) = -\pi$ and $f(x) = -\pi^2\cos(\pi x)$.

Implement using:
1. One-sided first order approximation
2. Centered 2nd order approximation 3. One-sided second order approximation

Analyze convergence with log-log plot.

Exercise 5.
Solve $u''(x) = e^x$ in $(0, 1)$ with $u(0) = 3$, $u'(1) = -5$.

1. Write second-order FD code (verify 2nd order).

2. Use extrapolation (verify 4th order).

3. Use deferred correction (verify 4th order).

Exercise 6.
Solve nonlinear BVP

$$u + \sin(u) = 0, \quad 0 < x < T, \quad u(0) = \alpha,\ u(T) = \beta,$$

using centered 2nd-order FD + Newton's method.
