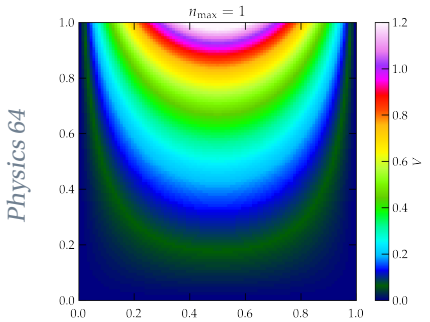
Problem Set 5

due: Monday, 23 February 2026



Problem 1 – Rectifier (10 points) A half-wave rectifier takes as an input a sine wave, $\sin\omega t$, and creates the output

$$f(t) = \begin{cases} \sin\omega t & \sin\omega t \geq 0 \\ 0 & \sin\omega t < 0 \end{cases}$$

You can make such a rectifier using a diode, which allows current to flow readily in the forward direction, but blocks it in the reverse. This is often the first step in converting an alternating voltage into a dc voltage.

(a) The original wave has period $\tau = 2\pi/\omega$. Develop a Fourier series for $f(t)$ using periodic functions with the same period τ.

(b) Since the function is continuous, but has a discontinuous first derivative, the nonzero terms in the series should have coefficients that fall off like $1/n^2$. Do they?

(c) Using Python and numpy, prepare a plot of the Fourier series through the first n terms, showing separate traces for $n = 1$, $n = 2$, and one or two others of your own choosing. Are you impressed with the convergence?

Problem 2 – No Integration by Parts! (10 points)

While it is important to know how to integrate by parts and to evaluate the coefficients in a Fourier expansion of $f(x)$ "from scratch," it can be useful to have a routine that does a numerical version of the expansion automatically. When we study solutions to partial differential equations in a few weeks, we will see that each "Fourier mode" has a different (but simple) time evolution and that if we know the initial condition of the quantity we model (e.g., the temperature distribution in a rod), we can compute the time-dependent solution once we know the amplitudes of the initial distribution in the various Fourier modes.

The first step you need to take is to determine the basis functions, which will be either sines or cosines of multiples of a variable like $\xi = \pi(x-a)/(b-a)$. To evaluate the coefficients of your expansion, you will need to perform integrals of the product of the basis functions with the function $f(x)$ that you seek to expand. Fortunately, scipy has you covered with the function scipy.integrate.quad. To integrate a function $g(x)$ from a to b, you would call

```
integral, error = quad(lambda x: g(x), a, b)
```

Of course, if you already have a function $g(x)$ you don't need to use a lambda function, but you will probably want to multiply a pair of functions in your routine, so I provide here the lambda template. Note that quad returns a 2-tuple containing both the value of the integral and an estimate of the error (which you'll probably ignore).

Since we contemplate various outputs from that Fourier expansion, we will implement a class, FourierExpansion, that will compute the coefficients in the Fourier expansion using basis functions consistent with the given boundary conditions. It will know how to plot the (decaying) amplitude of the coefficients as a function of n, to evaluate the series for an input array of x values, and evaluate the error in the series representation for a given value of $n_{\max}$, the total number of terms in the expansion. That is, given a function $f(x)$ defined on the interval $a \le x \le b$, and subject to boundary conditions at a and b that either $f = 0$ or $f' = 0$, your routine should derive the coefficients of a Fourier expansion and be able to evaluate the expansion function both inside the specified interval and its extension outside the interval. Furthermore, the code should be able to plot the rate at which the coefficients decrease with n on a semilog plot.

Test your class/object on the following functions:

- $y = e^{-6x^2}$ on the interval $-1 \le x \le 1$ using boundary conditions $f'(-1) = f'(1) = 0$ and compare the rate of convergence for the same function and the same interval, but using the boundary conditions $f(-1) = f(1) = 0$. You'll probably want to use at least 50 terms in the sum.
- $y = x$ on $0 \le x \le 1$ using boundary conditions of your choosing.

A scaffold for your class might be:

```
class FourierExpansion:
    def __init__(self, f, leftBC, rightBC, n_max=20):
        """
        Develop the Fourier series expansion of function f(x) on the interval
        specified by BCs. Each BC is describes a position (x) and a boolean
        describing whether the value of the function (True) or its derivative (False)
        vanishes at x. That is, if leftBC is (0, True) and rightBC is (1, False), then
        the basis functions must vanish at x = 0 and have vanishing derivative at
        x = 1.
```

```
    Internally, we will represent the interval as from 0 to pi, so the mapping
    between real x and internal y is y = pi * (x - a) / (b - a).
    """
    self.function = f
    self.a = leftBC[0]
    self.b = rightBC[0]
    self.L = self.b - self.a
    self.basis = np.sin if leftBC[1]] else np.cos
    # Your expansion will involve multiples (either integral or
    # half-integral) of x/L.
    # self.n is a list of those multiples
    self.n = <something> # a numpy array of the values of n you use in your expansion
    self._amps = None    # _amps will hold the amplitudes
    self.set_amps()      # actually compute the coefficients in the expansion
                         # and store in _amps

def set_amps(self):
    "Set the amplitudes in self._amps"
    # stuff

def __call__(self, x):
     "Evaluate the series approximation to f(x)"
     return <something>

def show(self, xvals):
    "Plot the honest version of f(x) and the Fourier series version"
    # stuff
    # You could add a residual panel, too, if you wish

def plot_amps(self):
    fig, ax = plt.subplots()
    ax.semilogy(self.n, np.abs(self._amps), 'o')
    ax.set_xlabel('$n$')
    ax.set_ylabel('$|a_n|$')
    ax.grid()

@property
def amps(self):
    "Generate a pandas DataFrame showing the coefficients "
    import pandas
    return pandas.DataFrame(dict(n=self.n, an=self._amps))

@property
def power(self):
    """Augment the DataFrame created by self.amps with a column showing the
        power in each Fourier mode, and another showing the normalized
        percentage of power in each mode.
    """
    # stuff

def rms_error(self):
    "Compute the root-mean-square error of the Fourier representation"
    x = np.linspace(self.a, self.b, 101)
    dy = self(x) - self.function(x)
    return np.std(dy)
```