CS 3430 S26: Scientific Computing
Assignment 2
Ice Cream Cones, Bypass Surgery, and Delta Cephei

Vladimir Kulyukin
SoC – CoE – USU

January 17, 2026

Learning Objectives

After completing this assignment, you will know how to:

1. Use **SymPy** to build symbolic mathematical models (expressions), simplify them, and compute limits and derivatives when appropriate.
2. Use **SymPy's** `lambdify` to convert symbolic expressions into **callable Python functions** that can be evaluated numerically.
3. Use **NumPy** to perform numerical computations efficiently, including evaluating mathematical models over arrays of input values.
4. Design and interpret **structural unit tests** that verify whether a SymPy expression contains the expected mathematical structure (symbols, powers, trigonometric functions, π, square roots, etc.).
5. Design and interpret **numerical unit tests** that validate symbolic results by comparing them against reference numerical computations.
6. Generate and save scientific plots using **matplotlib** to visualize the behavior of mathematical models over time or across parameter ranges.
7. Develop a deeper understanding of the practical tradeoffs between **exact symbolic computation** (SymPy) and **floating-point numerical computation** (NumPy) in scientific computing.

Introduction

We will dive into some of the topics of Lectures 3 and 4. You will save your coding solutions in the the following three files included in the zip:

1. `cs3430_s26_hw_2_prob_1.py`;
2. `cs3430_s26_hw_2_prob_2.py`;
3. `cs3430_s26_hw_2_prob_3.py`.

As usual, all problems have the corresponding unit tests files:

1. `cs3430_s26_hw_2_prob_1_uts.py`;
2. `cs3430_s26_hw_2_prob_2_uts.py`;
3. `cs3430_s26_hw_2_prob_3_uts.py`.

The assignment structure is used to maximize the modularity of the assignment. You can work on this assignment one problem at a time and, within each problem, one function at a time.

My software and hardware details:

- **Operating system:** Ubuntu 22.04.5 LTS (Jammy Jellyfish);
- **Kernel:** Linux 6.8.0-90-generic;
- **Architecture:** x86_64 (64-bit);
- **Processor:** Intel(R) Core(TM) i7-3770 CPU @ 3.40 GHz; (4 physical cores, 8 hardware threads);
- **Python:** 3.10.12;
- **Emacs:** GNU Emacs 27.1 modified by Debian.

Reading Assignment

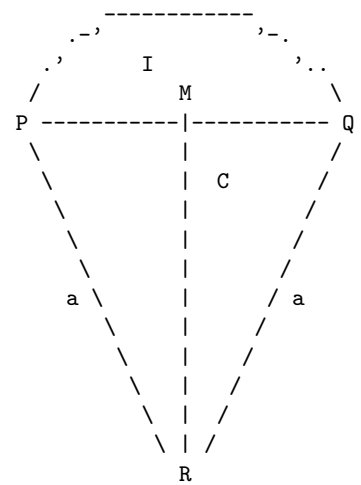
Review the CS 3430 S26 PDFs of Lectures 3 and 4 in Canvas and/or your notes.

Problem 1: Ice Cream Cones (1 point)

Recall that in Lectures 3 and 4, we analyzed the ice cream cone limit problem. Let us quickly review it.

The Ice Cream Cone Limit Problem

We view the waffle cone as an isosceles triangle $\triangle PQR$ and the visible ice cream as the half circle on top of it.



In the above text art picture, we have

- Base: $|PQ| = b$ (b is the diameter of the half circle);
- Equal sides: $|PR| = |QR| = a$;
- M is the mid point of PQ;
- RM is perpendicular to PQ;
- $PM = b/2$;
- Apex angle: $\angle PRQ = \theta$;

- I is the area of the ice cream; and C is the area of the waffle cone.

Let $I(\theta)$ denote the area of the visible ice cream as a function of θ and $C(\theta)$ be the area of the cone as a function of the apex angle θ. Our problem is to evaluate this limit.

$$\boxed{\lim_{\theta \to 0^+} \frac{I(\theta)}{C(\theta)}}$$

Our aim is to verify the following hypothesis: as the cone becomes pointy, the visible ice cream area becomes negligible compared to the cone area in this 2D model. Hence, the above limit must evaluate to 0.

What You Need to Implement

Your task in this problem is to implement several SymPy functions that build symbolic expressions to compute the above limit. You will write your solutions in

`cs3430_s26_hw_2_prob_1.py`.

The file already contains documentation strings for each function. **Read those documentation strings carefully**, because they describe the mathematical expressions you must construct and return.

Your task is to replace the `### YOUR CODE HERE` lines with your code blocks.

Unit Tests

For this problem, the unit tests are in

`cs3430_s26_hw_2_prob_1_uts.py`.

I write these unit tests not only for grading helpers (of course, they help Reagan and me test your code), but also as **learning tools** to help you learn best SciComp practices:

- how to verify the **structural consistency** of SymPy expressions;
- how to **lambdify** SymPy expressions into NumPy-aware Python functions;
- how to run **numerical experiments** in NumPy and compare them to symbolic expressions produced by SymPy;
- how to test both **raw** and **simplified** symbolic formulas (e.g., by applying `sp.simplify`).

Your goal is to implement your functions so that all unit tests pass. As always, start small: get one function working at a time and re-run the unit tests frequently. I would focus only on the UTs for the problem you are working on and comment out all the other UTs.

A Note on `sp_to_np`

In `cs3430_s26_hw_2_prob_1.py`, I provided the function `sp_to_np(expr)` for you **as is**. This function uses SymPy's `lambdify` utility to convert a symbolic SymPy expression (for example, an expression involving `sin`, `sqrt`, and Symbols) into a **callable Python function** that can be evaluated numerically using NumPy.

Such functions are important in scientific computing, because they bridge two worlds:

- **SymPy** helps us build and reason about formulas symbolically (exact mathematics),
- **NumPy** helps us evaluate formulas efficiently and run numerical experiments on many values (fast computation).

I gave `sp_to_np` as a completed function, because using `lambdify` correctly requires several details that are difficult to guess without experience (for example, choosing a consistent ordering of symbols and specifying the correct numerical backend). In this assignment, you should focus on learning how to **construct correct SymPy expressions** and how to **validate them with unit tests**, while still getting hands-on experience

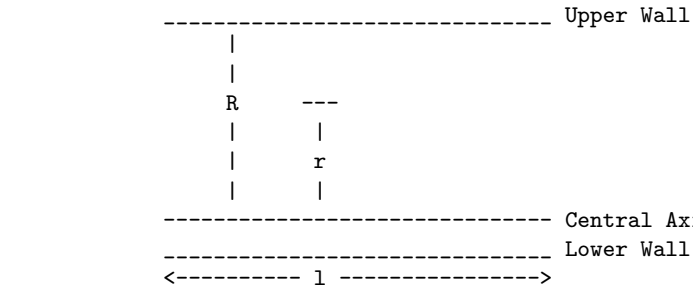
with the practical workflow of converting symbolic formulas to numerical code. As you play with SymPy and Numpy, you will be able to understand what this function does.

For each function, I wrote two unit tests (structural and numerical). The structural tests verify the structural correctness of SymPy expressions; the numerical tests lambidfy SymPy expressions and validate the obtained Python functions numerical.

My unit test output is saved in `cs3430_s26_hw_2_prob_1_uts.txt` included in the homework zip file.

Problem 2: Law of Laminar Flow (2 points)

In this problem, we model blood flow through a blood vessel (e.g., an artery) as a cylindrical tube of radius R and length l.



Due to friction at the vessel walls, the flow velocity v is greatest near the central axis and decreases as the distance r from the axis increases, until the velocity becomes 0 at the walls. A classical simplified model that describes this relationship is the **law of laminar flow**, discovered by the French physician Jean Louis Marie Poiseuille in 1840. The law states that

$$v(r) = \frac{P}{4\eta l}\left(R^2 - r^2\right), \tag{1}$$

where P is the pressure difference between the two ends of the tube, l is the tube length, R is the tube radius, r is the distance from the central axis, and η is the **viscosity** of the blood.

What is viscosity? Viscosity is a measure of how much a fluid resists flowing. For example, water has low viscosity and flows easily, while honey has high viscosity and flows slowly. Blood viscosity depends on several factors (temperature, hematocrit levels, etc.), but for this problem we treat η as a positive constant.

Average rate of change vs. velocity gradient. If we move outward from $r = r_1$ to $r = r_2$, the average rate of change of the velocity is

$$\frac{\Delta v}{\Delta r} = \frac{v(r_2) - v(r_1)}{r_2 - r_1}.$$

If we let $\Delta r \to 0$, we obtain the instantaneous rate of change of velocity with respect to r, called the **velocity gradient**:

$$\texttt{velocity gradient} = \lim_{\Delta r \to 0} \frac{\Delta v}{\Delta r} = \frac{dv}{dr}.$$

Differentiating 1 with respect to r gives

$$\frac{dv}{dr} = \frac{P}{4\eta l}\frac{d}{dr}(R^2 - r^2) = \frac{P}{4\eta l}(0 - 2r) = -\frac{Pr}{2\eta l}.$$

Bypass Surgery Motivation

In real life, cardiovascular risk increases when blood vessels narrow due to plaque, because a narrower vessel carries less blood flow to tissue. A possible clinical intervention in severe cases is a **bypass surgery**, which creates an alternate route for blood to flow around a narrowed or blocked artery. In this problem, we do **not** model real patient physiology and we do **not** make medical diagnoses. Instead, we use Poiseuille's laminar flow model to demonstrate why radius is such a critical parameter in fluid flow models. Unit test 10 for this problem shows you to implement and test a simple bypass surgery diagnosis model based on the law of laminar flow.

What You Need to Implement

Your task is to implement several SymPy functions in

`cs3430_s26_hw_2_prob_2.py`.

The documentation strings in that file describe each function precisely. As in Problem 1, the unit tests in

`cs3430_s26_hw_2_prob_2_uts.py`

are written as learning tools. They will help you learn:

- how to check the **structural correctness** of a SymPy expression (symbols, powers, and expression structure);
- how to convert SymPy expressions into NumPy-callable functions using `lambdify`;
- how to perform **numerical validation** by comparing lambdified outputs to NumPy reference computations.

My unit test output is saved in `cs3430_s26_hw_2_prob_2_uts.txt` and is included in the homework zip file.

Problem 3: Cepheid Variable Stars (2 points)

A **Cepheid variable star** is a star whose brightness increases and decreases in a regular, periodic pattern. One of the most easily visible Cepheid variable stars is **Delta Cephei** (I absolutely love this name!), for which the time between moments of maximum brightness is approximately **5.4 days**. The average brightness of this star is **4.0** and its brightness varies by ± 0.35.

Based on the data collected by astronomers, the brightness of Delta Cephei at time t (measured in days) is modeled by

$$B(t) = 4.0 + 0.35 \sin\left(\frac{2\pi t}{5.4}\right).$$

Explicit Differentiation of $B(t)$

To compute the rate of change of the brightness, we differentiate $B(t)$ with respect to time t. Recall the standard chain rule fact:

$$\frac{d}{dt}\sin(u(t)) = \cos(u(t)) \cdot u'(t).$$

In our case,

$$B(t) = 4.0 + 0.35 \sin\left(\frac{2\pi t}{5.4}\right),$$

so we obtain

$$B'(t) = \frac{dB}{dt} = 0 + 0.35 \cos\left(\frac{2\pi t}{5.4}\right) \cdot \frac{d}{dt}\left(\frac{2\pi t}{5.4}\right).$$

Since

$$\frac{d}{dt}\left(\frac{2\pi t}{5.4}\right) = \frac{2\pi}{5.4},$$

we conclude that the rate of change of brightness is

$$\boxed{B'(t) = 0.35 \cdot \frac{2\pi}{5.4} \cos\left(\frac{2\pi t}{5.4}\right).}$$

What You Will Implement

In the file `cs3430_s26_hw_2_prob_3.py`, you will implement several functions that combine:

- **SymPy** for building symbolic expressions and differentiating them;
- **NumPy** for evaluating expressions numerically on arrays of values;
- **lambdification** (`lambdify`) to convert symbolic formulas into callable Python functions;
- **matplotlib** for plotting brightness over time.

As in Problems 1 and 2, each function includes a documentation string that explains what it computes. The docstrings describe the mathematical meaning of each function and how it fits into the problem. These functions will enable you to:

1. Build the brightness model $B(t)$ in SymPy;
2. Compute the **rate of change of brightness** by differentiating $B(t)$ with respect to t;
3. Evaluate the brightness function numerically on a grid of time values;
4. Plot $B(t)$ over a time interval and save plots as PNG files.

Unit Tests

The unit tests in `cs3430_s26_hw_2_prob_3_uts.py` show you how to:

- Verify that a SymPy object has the correct **structure** (i.e., contains the expected symbols and mathematical components such as `sin`, `cos`, and `pi`);
- Convert SymPy expressions into NumPy functions using `sp_to_np(...)`;
- Compare symbolic models to numerical reference computations in NumPy;
- Generate plots and save them to `.png` files.

Numeric Output Requirement

One of the goals of this problem is to evaluate the rate of change of brightness after **one day** and compute it to **10 decimal places**. This is done by evaluating the lambdified derivative function at $t = 1$.

Plot Output Requirement

This problem also gives you practice with scientific visualization. Your unit tests will generate and save two plots:

- a 90-day brightness plot (shorter time horizon),
- a 365-day brightness plot (one full year time horizon).

These plots allow you to visually confirm that the brightness oscillates periodically and smoothly, as expected from a sinusoidal model.

My unit test output is saved in

`cs3430_s26_hw_2_prob_3_uts.txt.`

You can compare your output with mine. After the tests run successfully, you should also see the saved plot files in your working directory:

```
cs3430_s26_hw_2_prob_3_cepheid_brightness_90_days.png
cs3430_s26_hw_2_prob_3_cepheid_brightness_365_days.png
```

Open these PNG files and inspect them visually. They should look like smooth sinusoidal oscillations. I included my PNG files in the zip. Your images should look similar.

What To Submit

Zip your 3 Python files and 2 PNG images in CS3430_S26_HW2.zip and submit it in Canvas. Your zip must contain the following 3 files.

These files must have your implementations.

1. `cs3430_s26_hw_2_prob_1.py`;
2. `cs3430_s26_hw_2_prob_2.py`;
3. `cs3430_s26_hw_2_prob_3.py`;

The zip must also contain these images:

1. cs3430_s26_hw_2_prob_3_cepheid_brightness_90_days.png
2. cs3430_s26_hw_2_prob_3_cepheid_brightness_365_days.png

All 5 files must be present. Submissions missing any required file will be considered incomplete.