CS 3430 S26: Scientific Computing
Assignment 1
Floats, Error, Differentiation, Integration

Vladimir Kulyukin
SoC – CoE – USU

January 10, 2026

Learning Objectives

After completing this assignment, you should be able to:

1. Explain how floating-point numbers are represented and why floating-point arithmetic is finite, discrete, and unevenly spaced.
2. Identify and analyze sources of numerical error, including roundoff, truncation error, and catastrophic cancellation.
3. Implement and evaluate numerically stable algorithms for summation, differentiation, and integration.
4. Apply central divided differences and Richardson extrapolation to improve numerical differentiation and understand their limitations.
5. Implement Romberg integration and explain how extrapolation accelerates convergence compared to the trapezoidal rule.
6. Distinguish between symbolic and numerical computation using NumPy and SymPy, and understand when each is appropriate.

Introduction

In this assignment, we will investigate some of the topics of Lectures 1 and 2. You will save your coding solutions in the five files included in the zip with code prompts.

1. `cs3430_s26_hw_1_prob_1.py`;
2. `cs3430_s26_hw_1_prob_2.py`;
3. `cs3430_s26_hw_1_prob_3.py`;
4. `cs3430_s26_hw_1_prob_4.py`;
5. `cs3430_s26_hw_1_prob_5.py`.

Each problem has a corresponding unit tests file, where you will save your interpreation comments (more on this below).

1. `cs3430_s26_hw_1_prob_1_uts.py`;
2. `cs3430_s26_hw_1_prob_2_uts.py`;
3. `cs3430_s26_hw_1_prob_3_uts.py`;

4. `cs3430_s26_hw_1_prob_4_uts.py`;
5. `cs3430_s26_hw_1_prob_5_uts.py`.

I use the above assignment structure 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. When I say below **in your comments, write ...**, I mean the comments in a specific `_uts.py` file. E.g., your comments on problem 1 should be saved and submitted in `cs3430_s26_hw_1_prob_1_uts.py`, etc.

When you read the text of each assignment and want to copy Python code segments into your Python IDE/interpreter, make sure that all the commas and quotes paste properly. I use Linux, LaTeX , and GNU Emacs to typeset my documents and some characters may not paste properly into your Python editors. Be cognizant of this source of potential error. All sample outputs, numerical results, and unit test runs shown in this assignment were generated on the following system. Minor numerical differences across machines and operating systems are expected and acceptable.

- **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

All computations were performed on a standard desktop workstation using double-precision floating-point arithmetic. The examples in this assignment are designed to illustrate general numerical behavior rather than machine-specific quirks.

Reading Assignment

Review the CS 3430 S26 PDFs of Lectures 1 and 2 in Canvas and/or your notes. Read these PDFs included in the HW 1 zip.

1. CS3430_S26_CargoCultProgramming.pdf;
2. CS3430_S26_TypeAnnotations.pdf.

Problem 1: Floating-Point Gaps and Machine Epsilon (1 point)

In Lectures 1 and 2, we discussed the fact that floating-point numbers are *finite, discrete, and unevenly spaced.* In particular:

- Floating-point numbers are more densely packed near 1.0 than near very large values;
- The smallest distinguishable difference near 1.0 is called *machine epsilon*;
- Machine epsilon is not an abstract mathematical constant — it is a concrete property of your hardware and floating-point representation.

Part A: Floating-Point Gaps

Implement the function

`float_gap(x: float) -> float`

that returns the distance between a floating-point number `x` and the *next representable floating-point number greater than* `x`.

- You must use `math.nextafter(x, math.inf)`.
- The function should return the numerical gap

$$\text{nextafter}(x, +\infty) - x.$$

Then, experimentally compare the gap:

- near $x = 1.0$,
- and near $x = 10^{10}$.

In your comments, briefly explain:

- why the gap near 10^{10} is much larger than the gap near 1.0,
- and how this relates to the idea of *relative vs absolute precision*.

Part B: Machine Epsilon

Implement the function

`machine_epsilon() -> float`

that experimentally computes machine epsilon, defined as the smallest positive number ε such that

$$1.0 + \varepsilon > 1.0$$

in floating-point arithmetic.

- Your implementation must use a loop that repeatedly halves a candidate value.
- You may not use `numpy.finfo` or any hard-coded constants.

In your comments, briefly explain:

- why the loop eventually stops,
- why $1.0 + \varepsilon/2 = 1.0$ but $1.0 + \varepsilon > 1.0$,
- and how machine epsilon relates to floating-point spacing near 1.0.

Part C: Interpretation

Compare the value returned by `machine_epsilon()` to the gap returned by `float_gap(1.0)`.
In your comments, briefly explain why these two values are close but not necessarily bit-for-bit identical.

Takeaway: This problem is not about memorizing constants. It is about understanding how floating-point numbers behave *on real machines* (specifically on your machine!), with real precision limits. Remember:

$$\texttt{Math} \neq \texttt{SciComp}.$$

Below is my writeup for Problem 1. You can use it as an example for your comments in cs3430_s26_hw_1_prob_1_uts.py. It gives you the level of detail that your comments should have and how they should be structured. structured. Your submission must include both the unit test output and your interpretation at this level of specificity. Passing tests alone is not sufficient: you must explain what the numbers mean. Do not use any fancy math notation in your comments. Write your interpretation in plain text, as shown below.

```
Test 1: Gap Near 1e10 vs Gap Near 1.0

My Output:

[Test] Gap near 1e10 vs gap near 1.0
Gap near 1.0  = 2.220446049250313081e-16
Gap near 1e10 = 1.907348632812500000e-06
Ratio (1e10 / 1.0) = 8.590e+09

My Interpretation:

The floating-point gap near x = 1.0 is approximately 2.22e-16,
which is extremely small.

In contrast, the floating-point gap near x = 1e10 is approximately
1.91e-06, which is many orders of magnitude larger.

The reported ratio, gap(1e10) / gap(1.0) = 8.59e+09, confirms that
floating-point numbers are not evenly spaced.

This demonstrates that floating-point arithmetic preserves roughly
constant relative precision, not constant absolute precision.
As the magnitude of numbers increases, the absolute spacing between
representable floats increases dramatically.

Test 2: Gap Near 1.0

My Output:

[Test] Gap near 1.0
Computed gap at 1.0      = 2.220446049250313081e-16
Reference gap (nextafter)= 2.220446049250313081e-16
Relative error           = 0.000e+00

My Interpretation:

The computed gap at x = 1.0 exactly matches the reference gap
obtained using math.nextafter.

The relative error is reported as 0.000e+00, indicating bit-level
agreement between the implementation and the IEEE-754 definition
of the next representable floating-point number.

This confirms that float_gap(1.0) correctly measures the spacing
between adjacent floating-point numbers near 1.0.

Test 3: Machine Epsilon vs Gap Near 1.0

My Output:
```

```
[Test] Machine epsilon vs gap near 1.0
Machine epsilon = 2.220446049250313081e-16
Gap near 1.0    = 2.220446049250313081e-16
Relative difference = 0.000e+00

My Interpretation:

The experimentally computed machine epsilon is exactly equal to
the floating-point gap near 1.0 on this machine.

The reported relative difference is 0.000e+00, showing that the two
quantities are numerically identical here.

This is expected because machine epsilon measures the smallest value
that can change 1.0 in floating-point arithmetic, while float_gap(1.0)
directly measures the spacing between adjacent representable floats
at 1.0. Both describe the same precision limit of double-precision
floating-point arithmetic.

Test 4: Machine Epsilon Properties

My Output:

[Test] Machine epsilon properties
Computed machine epsilon = 2.220446049250313081e-16
1.0 + eps     = 1.0000000000000000222e+00
1.0 + eps/2.0 = 1.0000000000000000000e+00

My Interpretation:

Adding machine epsilon to 1.0 produces a value slightly larger than 1.0,
confirming that eps is large enough to affect the stored floating-point
number.

However, adding eps/2 to 1.0 produces exactly 1.0, because eps/2 is
smaller than the spacing between representable floating-point numbers
near 1.0 and is therefore lost due to roundoff.

This demonstrates that floating-point arithmetic has a finite resolution
and that numbers smaller than this resolution cannot influence computed
results.
```

Problem 2: Cancellation and Numerical Stability (1 point)

In Lecture 1, we discussed *loss of significance* (also called *catastrophic cancellation*) and why mathematically equivalent expressions can behave very differently in floating-point arithmetic. In this problem, we investigate numerical stability using the function

$$g(x) = \frac{1 - \cos(x)}{x^2},$$

which is mathematically well-defined, but numerically fragile for very small values of `x`.

Part A: Naive vs Stable Formulas

Implement the following functions in `cs3430_s26_hw_1_prob_2.py`: `g_naive(x)`, `g_stable(x)`, `g_ref(x)`.

- `g_naive(x)` directly evaluates
$$g(x) = \frac{1 - \cos(x)}{x^2}.$$
- `g_stable(x)` uses the trigonometric identity
$$1 - \cos(x) = 2\sin^2(x/2)$$

to avoid subtracting nearly equal floating-point numbers.

- `g_ref(x)` is a reference approximation based on a Taylor expansion of `cos(x)` near zero.

$$\frac{1}{2} - \frac{x^2}{24} + \frac{x^4}{720}.$$

All three functions compute the same mathematical quantity, but they do not behave the same numerically.

Part B: Numerical Experiments

The function

`compare_errors(xs)`

computes absolute errors of `g_naive(x)` and `g_stable(x)` relative to `g_ref(x)` for a list of values `xs`.
Use this function with values such as:

```
x = 1e-1, 1e-3, 1e-5, 1e-7, 1e-8
```

to observe how numerical error changes as `x` becomes very small.

Part C: Interpretation

Run the unit tests in:

`cs3430_s26_hw_1_prob_2_uts.py`

and paste the **full unit test output** into comments at the end of that file.
Then, **in plain ASCII comments**, interpret the output at the same level of detail as in Problem 1. Your interpretation must briefly explain:

- why `g_naive(x)` and `g_stable(x)` agree for moderate values of `x`;
- why `g_naive(x)` becomes inaccurate for very small `x`;
- why `g_stable(x)` tracks the reference approximation much better;
- how this behavior illustrates the difference between *mathematical correctness* and *numerical stability*.

Takeaway: This problem is not about algebraic cleverness. It is about understanding that *how a formula is evaluated* matters just as much as *what the formula is*. And, often, a more stable formula computes much better than an unstable one. Bottom line: mathematically equivalent expressions can behave very differently on real machines with finite precision.

Mathematical Background (For Reference)

We briefly justify the two mathematical identities used in this problem.

Trigonometric Identity. The identity

$$1 - \cos(x) = 2\sin^2(x/2)$$

follows from the standard double-angle formula for cosine. Recall that for any angle θ,

$$\cos(2\theta) = 1 - 2\sin^2(\theta).$$

Substituting $\theta = x/2$ gives

$$\cos(x) = 1 - 2\sin^2(x/2).$$

Rearranging terms yields

$$1 - \cos(x) = 2\sin^2(x/2).$$

Although this identity is mathematically equivalent to the original expression, it is numerically more stable for small values of x, because it avoids subtracting two nearly equal floating-point numbers.

Taylor Expansion Used in `g_ref(x)`. To construct a high-accuracy reference approximation near $x = 0$, we use the Taylor expansion of $\cos(x)$ about $x = 0$:

$$\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \cdots$$

Substituting this expansion into

$$g(x) = \frac{1 - \cos(x)}{x^2}$$

gives

$$g(x) = \frac{\frac{x^2}{2} - \frac{x^4}{24} + \frac{x^6}{720} - \cdots}{x^2}.$$

Dividing each term by x^2 yields

$$g(x) = \frac{1}{2} - \frac{x^2}{24} + \frac{x^4}{720} - \cdots.$$

The function `g_ref(x)` uses the first three nonzero terms of this series as a reference approximation, which is accurate for small x and avoids floating-point cancellation.

My sample interpretation for the code behavior on my machine is below.

```
Test 1: Error Comparison Table

My Output:

[Test] Error comparison table
x         | err_naive         | err_stable
-------------------------------------------
 1.0e-01 |  2.480e-11 |  2.480e-11
 1.0e-03 |  7.831e-12 |  0.000e+00
 1.0e-05 |  4.137e-08 |  5.551e-17
 1.0e-07 |  3.996e-04 |  5.551e-17
 1.0e-08 |  5.000e-01 |  0.000e+00

My Interpretation:

For moderate values of x (e.g., 1e-1), the naive and stable formulas
have nearly identical errors. Cancellation is not yet severe, so both
methods behave similarly.

As x becomes smaller, the error of the naive formula grows rapidly,
while the stable formula remains extremely accurate.

At x = 1e-08, the naive formula has an error of 0.5, which is catastrophic,
while the stable formula has essentially zero error.

This table clearly demonstrates that numerical error depends strongly
on how a formula is evaluated, not just on the mathematics of the formula.

---

Test 2: Naive vs Stable for Moderate x

My Output:

[Test] Naive vs Stable for moderate x
x = 0.1
g_naive(x)  = 4.9958347219741783e-01
g_stable(x) = 4.9958347219742333e-01
abs diff    = 5.496e-15

My Interpretation:

For x = 0.1, the naive and stable formulas agree to within about 5e-15.
This difference is on the order of machine precision and indicates that
```

```
no serious cancellation is occurring at this scale.

This confirms that both formulas are mathematically correct and numerically
reliable for moderate values of x.

---

Test 3: Reference Approximation Near Zero

My Output:

[Test] Reference approximation near zero
x =  1.0e-04  g_ref(x) = 4.9999999958333335e-01
x =  1.0e-06  g_ref(x) = 4.9999999999995831e-01
x =  1.0e-08  g_ref(x) = 5.0000000000000000e-01

My Interpretation:

As x decreases toward zero, the reference approximation g_ref(x)
approaches 0.5, which matches the theoretical limit of g(x).

This confirms that the Taylor-series-based reference is accurate
for very small values of x and can be used as a reliable benchmark
for comparing numerical methods.

---

Test 4: Cancellation for Tiny x

My Output:

[Test] Cancellation for tiny x
x = 1e-08
g_ref(x)    = 5.0000000000000000e-01
g_naive(x)  = 0.0000000000000000e+00
g_stable(x) = 5.0000000000000000e-01
err_naive   = 5.000e-01
err_stable  = 0.000e+00

My Interpretation:

For x = 1e-08, the naive formula evaluates to exactly 0.0.
This happens because 1 - cos(x) suffers catastrophic cancellation:
the subtraction loses all meaningful digits in floating-point arithmetic.

As a result, the naive formula produces a completely incorrect value.

In contrast, the stable formula produces exactly 0.5, matching the
reference value with zero error.

This is a clear example of catastrophic cancellation and how a
numerically stable reformulation avoids it.

---

Test 5: Stable Formula Tracks Reference

My Output:

[Test] Stable formula tracks reference
x =  1.0e-06
g_stable(x) = 4.9999999999995831e-01
g_ref(x)    = 4.9999999999995831e-01
abs diff    = 0.000e+00
x =  1.0e-08
```

```
g_stable(x) = 5.0000000000000000e-01
g_ref(x)    = 5.0000000000000000e-01
abs diff    = 0.000e+00

My Interpretation:

For very small values of x, the stable formula matches the reference
approximation exactly, down to machine precision.

This confirms that the stable formulation preserves numerical accuracy
even when the naive formula completely fails.

Overall, these results demonstrate the central lesson of scientific
computing: mathematically equivalent formulas can behave very
differently on real machines due to finite precision.
```

Problem 3: Floating-Point Summation and Order Effects (1 point)

In Lecture 1, we discussed how floating-point arithmetic is not associative: the result of a sequence of additions can depend on the order in which the additions are performed.

In this problem, we investigate numerical error in summation and compare three different summation strategies.

Part A: Summation Algorithms

You are given three functions to implement in `cs3430_s26_hw_1_prob_3.py`:

```
naive_sum(xs)
sorted_sum(xs)
kahan_sum(xs)
```

- `naive_sum(xs)` adds the numbers in the order they appear.
- `sorted_sum(xs)` adds numbers in increasing order of magnitude.
- `kahan_sum(xs)` uses Kahan's compensated summation algorithm to reduce floating-point error.

All three functions compute the same mathematical quantity, but they can produce very different numerical results. Kahan summation is a numerically stable algorithm for adding a sequence of floating-point numbers. Its goal is to reduce the loss of low-order bits that occurs when very small numbers are added to a large running sum.

The key idea is to maintain a separate *compensation variable* that tracks rounding error lost during each addition and reinjects that error into the next step. A high-level pseudocode description of Kahan summation is shown below.

```
initialize sum = 0.0
initialize compensation = 0.0

for each value x in the input sequence:
    y = x - compensation
    t = sum + y
    compensation = (t - sum) - y
    sum = t

return sum
```

In this algorithm, the variable `compensation` stores the part of each addition that was lost due to floating-point rounding. By subtracting this compensation from the next input value, the algorithm recovers information that would otherwise be permanently lost.

Kahan summation does not make floating-point arithmetic exact, but it can dramatically reduce accumulated error in long sums or in sums that mix very large and very small values.

Part B: Order Sensitivity and Error

The unit tests construct sequences that mix very large and very small numbers, such as:

```
[1e16] + [1.0, 1.0, ..., 1.0] + [-1e16]
```

Mathematically, the large terms cancel, and the sum should be the number of small terms. However, floating-point arithmetic may lose the small terms entirely depending on the order of addition.

Part C: Interpretation

Run the unit tests in:

`cs3430_s26_hw_1_prob_3_uts.py`

Paste the full unit test output into comments at the end of that file. Then, **in plain ASCII comments**, interpret the results. Your interpretation must explain:

- why naive summation is sensitive to the order of inputs;
- why sorting by magnitude often improves accuracy;
- what Kahan summation does and why it reduces numerical error;
- how these results demonstrate that floating-point addition is not associative.

Takeaway: This problem shows that numerical error is not just about formulas. Even simple operations like summation require careful algorithmic design to obtain reliable results on real machines.

Below is my interpretation of what running the unit tests on my machine.

```
Test 1: Basic Correctness on Small Input

My Output:

[Test] Basic correctness on small input
Input values: [0.1, 0.2, 0.3, -0.6, 1.0]
math.fsum   = 1.0000000000000000e+00
naive_sum   = 1.0000000000000000e+00
sorted_sum  = 1.0000000000000000e+00
kahan_sum   = 1.0000000000000000e+00

My Interpretation:

For this small input, all summation methods produce exactly the same
result, which is 1.0.

The reference value computed by math.fsum is also exactly 1.0, confirming
that the true mathematical sum is represented accurately in floating-point
arithmetic for this case.

Because the numbers involved are modest in magnitude and well balanced,
roundoff error does not accumulate significantly, and even the naive
summation method works correctly.

This test establishes that all implementations are functionally correct
in simple, well-conditioned cases.
```

Test 2: Kahan Summation Improves over Naive Summation

My Output:

```
[Test] Kahan summation improves over naive summation
Reference sum = 1000000.0
naive_sum    = 0.0000000000000000e+00
kahan_sum    = 1.0000000000000000e+06
err_naive    = 1.000e+06
err_kahan    = 0.000e+00
```

My Interpretation:

The reference sum of this sequence is 1,000,000.0, as computed by math.fsum.

The naive summation returns 0.0, which is completely wrong. This happens because the large value 1e16 is added first, and then the small values 1.0 are repeatedly added. Due to floating-point rounding, these small additions do not affect the running total. When -1e16 is finally added, it cancels the large value, leaving zero.

In contrast, Kahan summation returns the correct result exactly. The compensation variable in Kahan's algorithm tracks the small errors that would otherwise be lost, allowing the accumulated sum to retain the effect of the small values.

The error for naive summation is 1.0e+06, while the error for Kahan summation is zero. This demonstrates that Kahan summation significantly improves numerical accuracy in ill-conditioned summation problems.

Test 3: Order Sensitivity with Large/Small Mixture

My Output:

```
[Test] Order sensitivity with large/small mixture
Expected sum (reference) = 1000000.0
Bad order results:
naive_sum   = 0.0000000000000000e+00
sorted_sum  = 1.0000000000000000e+06
kahan_sum   = 1.0000000000000000e+06
Good order naive result:
naive_sum   = 1.0000000000000000e+06
```

My Interpretation:

When the summation is performed in a bad order (large value first, then many small values, then a large negative value), naive summation again produces zero due to catastrophic cancellation.

Both sorted summation and Kahan summation recover the correct result of 1,000,000.0. Sorting by absolute value causes small numbers to be summed first, reducing loss of significance. Kahan summation achieves the same goal by compensating for lost low-order bits during addition.

When the summation order is changed to a good order (large value, then its negation, then small values), naive summation also succeeds. This shows that naive summation is highly order-sensitive, whereas the other methods are much more robust.

```
Test 4: Sorted Summation on Constructed Example

My Output:

[Test] Sorted summation on constructed example
Input values: [1e+20, 1.0, 1.0, 1.0, -1e+20]
math.fsum   = 3.0
naive_sum   = 0.0000000000000000e+00
sorted_sum  = 0.0000000000000000e+00

My Interpretation:

The true sum of this sequence is 3.0, as confirmed by math.fsum.

Both naive summation and sorted summation return 0.0, which is incorrect.
In this case, even sorting by magnitude does not help, because the large
positive and negative values still dominate the arithmetic and cause the
small values to be lost.

This example demonstrates that sorting alone is not a universal solution
to numerical error. While it often improves accuracy, it can still fail
in extreme cases.

In contrast, Kahan summation (as shown in earlier tests) is designed to
systematically track and correct lost precision, making it more reliable
across a wider range of inputs.

---

Overall Interpretation:

These tests show that floating-point summation is not associative and that
numerical results depend strongly on summation order.

Naive summation is fast but fragile. Sorting by magnitude can help, but
does not guarantee correctness. Kahan summation explicitly compensates for
roundoff error and provides a much more stable summation method.

This problem illustrates the difference between mathematical correctness
and numerical stability: even simple arithmetic like addition can fail
badly on real machines if numerical error is ignored.
```

Problem 4: Numerical Differentiation, Error, and Richardson Extrapolation (1 point)

In Lectures 1 and 2, we discussed numerical differentiation, truncation error, roundoff error, and the fact that smaller step sizes do not always produce more accurate numerical results. In this problem, you will experimentally investigate these effects using the *central divided difference* method and *Richardson extrapolation*.

Part A: Central Divided Difference

Implement the function

`central_diff(f, x, h)`

in `cs3430_s26_hw_1_prob_4.py` so that it computes the central divided difference approximation of the derivative of `f` at `x`:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

Use this function to approximate the derivative of

`f(x) = sin(x)`

at `x = 1.0` for a range of step sizes

`h = 1e-1, 1e-2, ..., 1e-12.`

The exact derivative is `cos(1.0)`.

In your comments, explain:

- how the error behaves as `h` decreases initially;
- why the error eventually stops decreasing and begins to increase;
- how this illustrates the tradeoff between truncation error and roundoff error.

Part B: Richardson Extrapolation

Implement the function

`richardson_from_central(f, x, h)`

that applies Richardson extrapolation to the central divided difference approximation.

Your implementation must use the formula

`R(h) = (4*D(h/2) - D(h)) / 3`

where `D(h)` denotes the central divided difference with step size `h`.

Use this function to approximate the derivative of

`f(x) = exp(x)`

at `x = 0.0` for the same range of step sizes as above. The exact derivative is `exp(0.0)`.

In your comments, explain:

- why Richardson extrapolation dramatically improves accuracy for moderate values of `h`;
- why the Richardson approximation eventually stops improving;
- why Richardson extrapolation can become no better (or worse) than the original central difference for very small `h`.

Part C: Interpretation

Run the unit tests in

`cs3430_s26_hw_1_prob_4_uts.py`

and paste the **full unit test output** into comments at the end of that file.

Then, in **plain ASCII comments**, interpret the output at the same level of detail as in Problems 1, 2, and 3. Your interpretation must clearly identify:

- the truncation-dominated regime;
- the roundoff-dominated regime;
- the approximate step size where accuracy is maximized;
- why numerical differentiation has an optimal step size.

Important: This problem reinforces a central lesson of scientific computing: more sophisticated methods and smaller step sizes do not guarantee better answers. Numerical accuracy depends on balancing mathematical error with floating-point limitations on real machines.

My interpreatation for this problem is below.

```
Test 1: Central Difference on sin(x) at x = 1.0

My Output:

[Test] Central Difference on sin(x) at x = 1.0
h        | approx               | error
-------------------------------------------
 1.0e-01 |  0.5394022521697600 |  9.001e-04
 1.0e-02 |  0.5402933008747335 |  9.005e-06
 1.0e-03 |  0.5403022158176896 |  9.005e-08
 1.0e-04 |  0.5403023049677103 |  9.004e-10
 1.0e-05 |  0.5403023058569989 |  1.114e-11
 1.0e-06 |  0.5403023058958567 |  2.772e-11
 1.0e-07 |  0.5403023056738121 |  1.943e-10
 1.0e-08 |  0.5403023084493697 |  2.581e-09
 1.0e-09 |  0.5403023028982545 |  2.970e-09
 1.0e-10 |  0.5403022473871033 |  5.848e-08
 1.0e-11 |  0.5403011371640787 |  1.169e-06
 1.0e-12 |  0.5402900349338324 |  1.227e-05

My Interpretation:

For h = 1e-1 down to about h = 1e-5, the error decreases by roughly
a factor of 100 each time h is reduced by a factor of 10.

This confirms that the central difference method has O(h^2)
truncation error, exactly as predicted by theory.

After h is approximately 1e-5, the error stops improving and then increases.
This happens because roundoff and cancellation errors dominate
when h becomes too small.

This demonstrates that smaller h does not always mean better accuracy.
There is an optimal range of h where truncation and roundoff errors
are balanced.

---

Test 2: Richardson Extrapolation on exp(x) at x = 0.0

My Output:

[Test] Richardson Extrapolation on exp(x) at x = 0.0
h        | D_err        | R_err
--------------------------------
 1.0e-01 |  1.668e-03 |  2.084e-07
 1.0e-02 |  1.667e-05 |  2.085e-11
 1.0e-03 |  1.667e-07 |  1.471e-13
 1.0e-04 |  1.667e-09 |  8.151e-13
 1.0e-05 |  1.210e-11 |  4.701e-12
 1.0e-06 |  2.676e-11 |  4.726e-11
 1.0e-07 |  5.264e-10 |  5.264e-10
 1.0e-08 |  6.077e-09 |  6.077e-09
 1.0e-09 |  2.723e-08 |  1.012e-07
 1.0e-10 |  8.274e-08 |  8.274e-08
 1.0e-11 |  8.274e-08 |  8.274e-08
 1.0e-12 |  3.339e-05 |  1.074e-04

My Interpretation:
```

```
For moderate values of h (from 1e-1 to about 1e-3),
Richardson extrapolation dramatically reduces the error.

For example, at h = 1e-2, the central difference error is about 1.7e-05,
while the Richardson error is about 2.1e-11, an improvement of roughly
six orders of magnitude.

This happens because Richardson extrapolation cancels the leading
O(h^2) truncation error term, producing a higher-order approximation.

The smallest Richardson error occurs near h = 1e-3.
Beyond this point, the Richardson error stops improving and eventually
becomes worse than the central difference error.

This occurs because Richardson extrapolation combines two noisy
finite-difference estimates. When roundoff error dominates,
the subtraction in the Richardson formula amplifies floating-point noise.

This test clearly shows the transition from a truncation-dominated
regime to a roundoff-dominated regime.

The key lesson is that even higher-order methods have an optimal
step size, and using h that is too small can reduce accuracy instead
of improving it.
```

Problem 5: Romberg Integration and Symbolic vs Numerical Computation (1 point)

In Lectures 1 and 2, we discussed numerical integration, Richardson extrapolation, and the distinction between *symbolic* and *numerical* computation. In this problem, we will implement *Romberg integration* in NumPy and compare its performance against both the trapezoidal rule and symbolic computation using SymPy. Romberg integration is a systematic way to improve numerical integration by combining the trapezoidal rule with repeated Richardson extrapolation.

Part A: Trapezoidal Rule

Implement the function

`trap(f, a, b, n)`

that approximates the integral

$$\int_a^b f(x)\,dx$$

using the trapezoidal rule with `n` subintervals.

- Your implementation must be vectorized using `numpy`.
- The trapezoidal rule should serve as the base method for Romberg integration.

Part B: Romberg Integration

Implement the function

`romberg(f, a, b, K)`

that constructs a Romberg table of size $(K+1) \times (K+1)$.

- The first column must use trapezoidal approximations with $n = 2^k$ subintervals.
- Higher columns must be computed using the Romberg recurrence:

```
R[k, j] = R[k, j-1] + (R[k, j-1] - R[k-1, j-1]) / (4**j - 1)
```

Return the full Romberg table.

Part C: Numerical Experiment with π

Use your Romberg implementation to approximate

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi.$$

- Print the full Romberg table.
- Report the best estimate and its absolute error relative to `math.pi`.
- Compare the Romberg result to a plain trapezoidal estimate.

Part D: Symbolic Reference with SymPy

Use `sympy` to compute an exact derivative symbolically and evaluate it numerically using `lambdify`.

- Compute a symbolic derivative.
- Convert it to a numerical function.
- Use this value as a high-quality reference.

Part E: Interpretation

Run the unit tests in:

`cs3430_s26_hw_1_prob_5_uts.py`

Paste the **full unit test output** into comments at the end of that file.
Then, **in plain ASCII comments**, interpret the output at the same level of detail as in Problems 1–4. Your interpretation must explain:

- why the trapezoidal rule converges slowly;
- how Romberg integration accelerates convergence;
- why Richardson extrapolation improves accuracy;
- how symbolic computation differs from numerical approximation;
- why Romberg can achieve near machine-precision accuracy.

Takeaway: This problem reinforces a central theme of scientific computing: Mathematically correct methods can behave very differently when implemented on real machines with finite precision. Romberg integration works not because it is "more mathematical," but because it is *numerically smarter*.

Below is my interpretation for Problem 5 that you can use as an example.

```
Test 1: Romberg Integration for pi

My Output:

[Test] Romberg integration for pi
Romberg table:
[[3.         0.         0.         0.         0.         0.
  0.        ]
 [3.1        3.13333333 0.         0.         0.         0.
  0.        ]
```

```
 [3.13117647 3.14156863 3.14211765 0.         0.         0.
  0.        ]
 [3.13898849 3.1415925  3.14159409 3.14158578 0.         0.
  0.        ]
 [3.14094161 3.14159265 3.14159266 3.14159264 3.14159267 0.
  0.        ]
 [3.14142989 3.14159265 3.14159265 3.14159265 3.14159265 3.14159265
  0.        ]
 [3.14155196 3.14159265 3.14159265 3.14159265 3.14159265 3.14159265
  3.14159265]]
Best estimate: 3.1415926535897216
Error vs pi: 7.149836278586008e-14

My Interpretation:

The first column of the Romberg table corresponds to the trapezoidal rule
with increasing numbers of subintervals. These values approach pi slowly,
as expected for an O(h^2) method.

The higher columns are obtained by Richardson extrapolation. Starting in
column 1, the values rapidly converge toward pi.

By the time we reach R[6,6], the estimate is
3.1415926535897216, which differs from pi by about 7.15e-14.

This demonstrates the power of Romberg integration: it systematically
cancels leading error terms in the trapezoidal rule and achieves very
high accuracy with relatively few function evaluations.

---

Test 2: Symbolic Reference Using SymPy

My Output:

[Test] Symbolic reference using SymPy
Symbolic derivative df_sym = exp(x)*sin(x) + exp(x)*cos(x)
Reference derivative at x = 1.0 = 3.7560492270947274

My Interpretation:

SymPy computes the derivative symbolically, producing the exact analytic
expression exp(x)*sin(x) + exp(x)*cos(x).

This expression is then evaluated numerically at x = 1.0 using lambdify,
giving a reference value of approximately 3.7560492270947274.

This value is not subject to truncation error from finite differences.
It serves as a high-quality reference for comparing numerical methods.

---

Test 3: Trapezoidal Rule vs Romberg Convergence

My Output:

[Test] Trapezoidal rule vs Romberg convergence
Trapezoid estimate: 3.141551963485655
Romberg estimate: 3.1415926535897216
True value (pi): 3.141592653589793
Trap error: 4.069010413809693e-05
Romberg error: 7.149836278586008e-14

My Interpretation:
```

```
The trapezoidal rule approximation differs from pi by about 4.07e-05,
which is relatively large despite using many subintervals.

In contrast, the Romberg estimate differs from pi by about 7.15e-14,
which is near machine precision for double-precision floating-point numbers.

This confirms that Romberg integration dramatically outperforms the plain
trapezoidal rule by using Richardson extrapolation to cancel lower-order
error terms.

Overall, this experiment shows that algorithmic design matters: both
methods are mathematically correct, but Romberg integration is far more
numerically efficient and accurate.
```

What To Submit

Zip your 10 files in CS3430_S26_HW1.zip and submit it in Canvas. Your zip must contain the following 10 files.

These files must have your implementations.

1. `cs3430_s26_hw_1_prob_1.py`;
2. `cs3430_s26_hw_1_prob_2.py`;
3. `cs3430_s26_hw_1_prob_3.py`;
4. `cs3430_s26_hw_1_prob_4.py`;
5. `cs3430_s26_hw_1_prob_5.py`.

These files must have your interpretation comments.

1. `cs3430_s26_hw_1_prob_1_uts.py`;
2. `cs3430_s26_hw_1_prob_2_uts.py`;
3. `cs3430_s26_hw_1_prob_3_uts.py`;
4. `cs3430_s26_hw_1_prob_4_uts.py`;
5. `cs3430_s26_hw_1_prob_5_uts.py`.

All 10 files must be present. Submissions missing any required file will be considered incomplete.