# CS 3430: S26: Scientific Computing
# Assignment 5

Vladimir Kulyukin
School of Computing
College of Engineering
Utah State University

February 7, 2026

## Learning Objectives

After completing this assignment, you should be able to:

1. Use **generators** to represent infinite numerical processes in a controlled and compositional way.
2. Implement and compare multiple algorithms for computing $\pi$ and $e$, including continued fractions, series expansions, and modern high-performance formulas.
3. Understand the dramatic impact of *problem formulation* on convergence speed and computational cost.
4. Use **arbitrary-precision arithmetic** (`mpmath` and `decimal`) to separate mathematical convergence from representation limits.
5. Reason explicitly about **precision budgeting**, numerical verification, and what it means for a computed number to be "correct."

## Introduction

In this assignment, we dive into one of the foundational problems in scientific computing: how to compute mathematical *constants* such as $\pi$ and $e$. I deliberately italicized the term *constants.* To a computer scientist, a constant is an entity (e.g., a number) that never changes. To a mathematician, a constant is a number that never changes – until, of course, we need more digits, after which the same number stubbornly remains a constant. This assignment lives precisely in that never-ending tension.

These numerical constants are fixed in mathematics, yet there are infinitely many algorithms for computing them. Some are simple and slow; others are sophisticated and astonishingly fast. Scientific computing asks not only whether an algorithm is correct, but also how fast it converges, how it interacts with finite-precision arithmetic, and (this is important!) when its output can be trusted.

A unifying theme of this assignment is the use of **generators**. Generators allow us to treat an infinite process (e.g., continued fractions, infinite series, or hypergeometric expansions) as a sequence of finite approximations. In this sense, generators are the wings that safely carry us across the infinite abyss of the $\pi$ and $e$ mantissas.

We will begin with continued fractions for $\pi$ and $e$, progress through the classical Leibniz and Machin formulas, and then work the remarkable series of Ramanujan and the modern Chudnovsky method. Finally, we will push these methods by computing and verifying hundreds (and, hardware permitting, thousands!) of digits of $\pi$.

Please review **Lecture 8**, the **Lecture 8 Addendum (Generators)**, and **Lecture 9** or your class notes before starting this assignment. These lectures contain the mathematical and computational ideas on which every problem below is based.

## Problem 1: Continued Fractions for $\pi$ and $e$ (1 point)

In this problem, we will implement generators for continued fraction expansions of $\pi$ and $e$.

You will write *four* generators:

- `pi_cf_mp` and `e_cf_mp`, using `mpmath`;
- `pi_cf_dec` and `e_cf_dec`, using `decimal`.

Each generator must yield successive convergents of the corresponding continued fraction.

The unit tests will help you verify:

- structural correctness (generators exist and yield values);
- numerical correctness against exact rational oracles;
- correct interaction with precision contexts.

### Required Short Write-Up

After running the unit tests, write a short report discussing:

- how quickly the convergents approach machine precision;
- whether `mpmath` and `decimal` behave differently;
- how many iterations are needed before further progress becomes meaningless.

Save your write-up as a **multi-line comment at the beginning of** `cs3430_s26_hw_5_prob_1.py`. My writeup is in the zip.

## Problem 2: Leibniz vs. Machin (1 point)

In this problem, you will compare two historically important algorithms for computing $\pi$:

- the Leibniz series;
- Machins formula.

Both methods use the same Taylor expansion of $\arctan(x)$, but they behave very differently.

You will implement generator-based versions of both algorithms using `mpmath`. The unit tests compare convergence rates directly and print numerical evidence of the difference.

No write-up is required for this problem. The printed unit-test output is intended to be read and interpreted. I included my observations on this problem in the zip. It is strictly FYI.

## Problem 3: Ramanujan and Chudnovsky (1 point)

In this problem, we will encounter two of the fastest known series for $\pi$:

- a Ramanujan-type series;
- the Chudnovsky series implemented using binary splitting.

You will implement both as generators using `mpmath`. The unit tests demonstrate that:

- Ramanujans series converges dramatically faster than Machins;
- the Chudnovsky series converges even faster, producing dozens of correct digits per term.

This problem emphasizes that mathematical depth can translate directly into computational power-but only if representation and precision are handled correctly.

### Why Binary Splitting Is Given to You

You are provided with a binary-splitting implementation of the Chudnovsky formula rather than being asked to derive or implement it yourself. This choice is deliberate. Binary splitting is a modern, highly optimized algorithm whose correctness depends on careful symbolic rearrangement, exact integer arithmetic, and subtle performance considerations. While mathematically elegant, its implementation is marginal to our experimental goals in this assignment.

Our focus here is not algorithmic heroics, but scientific computing practice: precision budgeting, convergence testing, validation against ground truth, and understanding when numerical results can be trusted. In real scientific work, such tools like Chudnovsky are treated as givens: they are just computational tools that enable experimentation rather than objects of reinvention.

For those of you who are mathematically inclined, I have included an optional slide deck titled `CS3430_S26_ChudnovskyBinSplit.pdf` in the zip. This deck explains, step by step, how the Chudnovsky formula can be restructured into a form suitable for the so called *binary splitting*. The slides show how large products and sums over integer intervals can be recursively decomposed, leading to the fast and numerically stable algorithm used in this assignment.

The purpose of this deck is *not* to require you to re-derive or re-implement the algorithm. Binary splitting is a modern, state-of-the-art technique that is far more sophisticated than what I could reasonably ask you to reproduce in one homework setting. Instead, I tried my best to compile this deck to demystify this method and to clarify why it scales so well in practice.

Bottom line: treat the binary-splitting Chudnovsky implementation as a trusted numerical engine and focus on what scientific computing actually demands: precision budgeting, convergence behavior, and verification against independent reference data.

## Problem 4: Verifying the Mantissa of $\pi$ at Scale (2 points)

In this final problem, we will push the Chudnovsky algorithm to large-scale verification. You do not need to write any code for this problem. This is a pure scientific computing experiment.

I included in the zip the file `pi_mantissa_99999.txt` containing the first **99,999 digits of the mantissa of $\pi$**. I downloaded it from `http://www.geom.uiuc.edu`.

Your task is to:

- compute $\pi$ using the binary-splitting Chudnovsky generator;
- extract a specified number of mantissa digits;

- verify correctness using cryptographic hash digests.

Rather than asking “How many digits can I compute?”, this problem asks a deeper question: *How many digits can I justify as correct?* Let us dive into some tools at our disposal.

## Verifying Large Numerical Results with Hashes

The file `cs3430_s26_hw_5_prob_4.py` contains the following function.

```
def sha256_of_digits(digits: str) -> str:
    h = hashlib.sha256()
    h.update(digits.encode("ascii"))
    return h.hexdigest()
```

SHA-256 stands for *Secure Hash Algorithm, 256-bit.* It is a cryptographic hash function standardized by the U.S. National Institute of Standards and Technology (NIST). A hash function takes an input of arbitrary length (for example, a string with thousands of digits) and deterministically produces a fixed-size output in this case, a 256-bit value, typically written as a 64-character hexadecimal string. The defining property of a cryptographic hash is not secrecy but *fingerprinting*: even a one-character change in the input produces a completely different output, while the same input always produces the same hash.

In the above function, `hashlib.sha256()` creates a new SHA-256 hash object. This object maintains an internal state that is updated as data is fed into it.

Then, `digits.encode("ascii")` converts the Python string `digits` into a sequence of bytes. Hash functions operate on bytes, not on Python strings. Since the mantissa consists only of characters `0--9`, ASCII is sufficient. UNICODE is a waste here.

The method `h.update(...)` feeds the byte sequence into the hash algorithm. This updates the internal state of the hash object.

Finally, `h.hexdigest()` returns the final hash value as a hexadecimal string. This string is the *digest*: a compact fingerprint of the original input.

Here is a brief example.

```
>>> from cs3430_s26_hw_5_prob_4 import *
>>> sha256_of_digits("314")
'748064be03a08df81e31bd6f9e7e7c4cc9f84b4401b9a3c6e85b7ff816d3ba68'
>>> sha256_of_digits("315")
'377adeb4cd4096adc7ca64b533938cffc6294a9b3534f883b2336a26252cda9a'
```

Although the inputs differ by only one digit, their SHA-256 digests are completely different. This sensitivity is intentional and is known as the *avalanche effect.* It ensures that even a single incorrect digit in a long numerical computation is detected immediately.

In sum, hashing allows us to reason about correctness at scale. Instead of asking are these 100,000 digits identical? we ask a far simpler question: do these two cryptographic fingerprints match? This shift from elementwise comparison to digest comparison is a hallmark of scientific data-intensive computing.

## Deep Convergence Test

The unit test that I want you to work with is `TestChudnovskyDeepConvergence`. The method `test_hash_matches_until_failure` gives you the tools to test:

- the number of requested digits,
- the working precision,
- and the number of Chudnovsky terms,

until hash verification fails.

This failure is not a bug. It marks the boundary where representation limits overtake mathematical convergence. I included my writeup on this test in the zip.

But what I want you to do is to write your own test method `test_my_exploratory_hash_matches` in that class and then write your own report. Read my comments in the stub of this method on what you can do with this test.

### Required Short Write-Up

Write a short report on your observations. You may want to discuss in it:

- how correctness depends on precision budgeting;
- why increasing `mp.dps` blindly is not a principled strategy;
- what the hash-based verification teaches you about numerical epistemology.

Save this write-up as a **multi-line comment at the beginning of** `cs3430_s26_hw_5_prob_4.py`.

## What to Submit

- All completed `cs3430_s26_hw_5_prob_*.py` files, where `*` $\in \{1, 2, 3, 4\}$.
- Short write-ups embedded as comments in Problems 1 and 4.
- No separate PDF or text write-ups are required.

Happy hacking! And, enjoy flying across infinity with generators!

## Some Optinal Parting Thoughts on the Margin

### Leibniz and the Cost of Slowness

You might wonder why the Leibniz series for $\pi$ appears in this assignment at all, given how slowly it converges. The answer is historical as much as mathematical.

Gottfried Wilhelm Leibniz discovered the series

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

in the late 17th century, at a time when calculus itself was still being invented. From a modern perspective, this formula is inefficient: obtaining even a few correct digits of $\pi$ requires thousands or millions of terms. Yet for Leibniz, the series was a profound conceptual breakthrough. It demonstrated, for the first time, that $\pi$a geometric quantitycould be expressed as an infinite arithmetic process.

Leibniz did not have computers, floating-point arithmetic, or generators. Each term had to be computed by hand. What mattered was not speed, but existence: the mere fact that $\pi$ could be generated algorithmically was revolutionary. Archimedes computed $\pi$ without numbers in the modern sense, relying entirely on geometry and enclosure. Leibniz abandoned geometry altogether, recasting $\pi$ as an infinite arithmetic process – slow, but conceptually transformative.

From the standpoint of scientific computing, the Leibniz series teaches a lasting lesson. A method can be mathematically correct, elegant, and historically important, yet computationally impractical. Correctness alone is not enough. Convergence rate, numerical stability, and algorithmic structure ultimately determine whether a method is usable in practice.

In this assignment, Leibniz serves as a baseline: a reminder that mathematics answers the question *can this be computed?*, while scientific computing must answer *should this be computed this way?*

## Ramanujan and Explosive Convergence

If Leibniz represents the patience of early calculus, Srinivasa Ramanujan represents something entirely different.

Ramanujan, a self-taught mathematician from India working in the early 20th century, discovered a remarkable collection of formulas for $\pi$, many of which converge at astonishing speed. One such series takes the form

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!\,(1103 + 26390k)}{(k!)^4\, 396^{4k}}.$$

Each additional term contributes roughly eight correct digits of $\pi$. With only a handful of terms, one achieves precision that would require millions of iterations of the Leibniz series.

What makes Ramanujans work especially remarkable is not just its efficiency, but its origin. Ramanujan had little formal mathematical education and worked largely in isolation. His notebooks are filled with formulas written down without proofs. Yet, many of them were later verified and remain among the most efficient known methods for computing mathematical constants.

Ramanujans life illustrates a deep and sometimes uncomfortable distinction: formal education and raw talent are not the same thing. Modern mathematics relies on training, rigor, and formal proof, but Ramanujan demonstrated that insight, intuition, and numerical experimentation can precede (and sometimes outrun!) formal justification.

To this day, professional mathematicians and historians of science and mathematics debate how to interpret Ramanujans work. Some of his formulas were rigorously proved only decades after his death; others required entirely new mathematical frameworks. In some cases, multiple proofs exist; in others, the most natural derivations remain elusive.

This leads to a striking asymmetry in mathematics and scientific computing:

> What works cannot always be proved, and what can be proved rigorously can sometimes be computationally useless.

Scientific computing permantenly lives in this tension. We routinely rely on methods that work spectacularly well in practice, even when the full theoretical justification is subtle, incomplete, or still evolving. At the same time, we learn (sometimes the hard way) that perfectly proved methods may converge too slowly to be useful.

Ramanujan's formulas remind us that computation is not merely an application of theory. It is also an experimental science, where insight, structure, formulatiion, and formalization matter as much as proof.