CS 3430 S26: Scientific Computing
Assignment 4

Vladimir Kulyukin
SoC – CoE – USU

January 31, 2026

Learning Objectives

1. Forward and back substitution for triangular systems, and how numerical error propagates through iterative arithmetic.
2. LU decomposition with partial pivoting, including why row permutations are necessary and how they affect the solution of linear systems.
3. Archimedes method for computing π as an early numerical algorithm, with guaranteed convergence but slow convergence performance.
4. Related rates as a bridge between calculus and numerical computation, and how approximations of π influence physical predictions.
5. Image blurring and deblurring as linear inverse problems, and why mathematically correct methods can be numerically unstable and computationally expensive.

Introduction

In this assignment, we will explore solving triangular linear systems, LU decomposition with pivoting, numerical approximation of π, related rates, and image blurring and deblurring as the inverse linear system problem. Although these topics may appear unrelated at first glance, they are unified by a single theme: how mathematical algorithms behave when they meet floating-point arithmetic and real data.

We will begin by implementing forward and back substitution and observing how numerical errors propagate through triangular systems. We will then build on this foundation by solving linear systems using LU decomposition, where issues of conditioning, pivoting, and implementation conventions play a central role.

Next, we will step back in time and implement Archimedes' method for approximating π. This method is historically important and mathematically correct, yet surprisingly slow in terms of convergence, which highlights the difference between correctness and practical performance.

We will then connect numerical approximation to physical reasoning through a related-rates problem in fluid mechanics, showing how approximations of π propagate into real-world quantities such as rates of change in physical systems.

Finally, we will apply linear algebra to images. By modeling image blurring and deblurring as linear systems $Ax = b$, we will see that inverse problems are fundamentally fragile: while blurring is easy and stable, deblurring amplifies noise, introduce side effects, and becomes computationally expensive. This is linear algebra live, but with all of its numerical warts exposed.

A recurring theme throughout the assignment (and the course!) is that clean mathematics often meets messy reality. Some methods are theoretically elegant but brittle, while others are more

robust but less exact or beautiful. Learning when and why this happens is a central goal of scientific computing.

Please review Lectures 6 and 7 PDF and/or your class notes before starting this assignment. In particular, pay close attention to and get comfortable with:

- the structure of lower- and upper-triangular matrices;
- the general formulas for forward and back substitution;
- the role of pivots and what happens when a pivot is zero or very small;
- the LU decomposition;
- the basic geometric ideas behind Archimedes' approximation of π.

Problem 1: Forward and Back Substitution (1 point)

In this problem, we will implement two fundamental algorithms from numerical linear algebra: *forward substitution* and *back substitution*. These algorithms are the computational workhorses behind LU decomposition and many other modern linear system solvers.

The file `cs3430_s26_hw_4_prob_1.py` contains the stubs of two functions we will implement.

1) `back_substitution(U, b)`

This function assumes that we have the following linear system, where the coefficient matrix is upper-triangular:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

The general back substitution formula we derived in Lecture 6 is:

$$x_i = \frac{1}{a_{ii}} \left[b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right].$$

Your function must solve $Ux = b$ using this formula.

2) `forward_substitution(L, b)`

This function assume that we have the following linear system, where the coefficient matrix is lower-triangular:

$$\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

The forward substitution formulas we derived in Lecture 6 are:

$$y_1 = \frac{b_1}{a_{11}},$$

and for $i = 2, 3, \ldots, n$,

$$y_i = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} y_j \right].$$

Your function must solve $Ly = b$ using these formulas.

Important restrictions:

- You **may not** use `np.linalg.solve` inside these functions. We use it in the unit tests as the ground truth.
- You must implement the algorithms explicitly using loops.
- Your code must check for incompatible shapes and zero (or near-zero) pivots and raise a `ValueError` when appropriate.

The file `cs3430_s26_hw_4_prob_1_uts.py` contains unit tests you will run to test your implementations. These tests include small, medium, and larger systems (up to 50×50).

A Short Write-Up on Error in Forward and Back Substitution (Required)

When we run the unit tests on, we may notice the following behavior:

- Back substitution typically produces very small numerical differences when compared to NumPys solver.
- Forward substitution may produce a noticeably larger difference (for example, a norm on the order of 10^{-2} for a 50×50 system), even though all tests still pass.

This behavior does **not** indicate a bug. In scientific computing, *nonzero numerical error is normal*. The key question is not "Is the error exactly zero?" but rather "Is the error controlled and understood?" This is the point of writing this short write-up.
Below is my write-up that you can use as a template. I wrote it after running the unit tests on my machine.

```
Observed Numerical Behavior in Forward and Back Substitution

After running the unit tests for Problem 1, I observed that both back substitution
and forward substitution produced results that closely matched NumPys
solutions. However, the magnitude of the forward error differed across problem
sizes and between the two methods.

For small systems, my solutions and NumPy solutions matched almost exactly.
I expected it to happen, because very few floating-point operations are
performed, so rounding error has little opportunity to accumulate/propagate.

For medium systems, I also observed small numerical differences. They
were slightly larger, but they remained well within acceptable tolerances.
In particular, my solutions still satisfied np.allclose, indicating agreement
up to machine precision.

For large systems (50 x 50), I obserbved a clear difference: back
substitution  produced a very small forward error (around 10^-8), whereas
forward substitution produced a larger error (around 10^-2).

Here is my theory. In forward substitution, each  y_i depends on all previous
values y_1,...,y_{i-1}; in back substitution, each x_i depends on all
```

```
subsequent values x_{i+1}, ..., x_{n}.

Although both forward and back substitution compute each unknown using
previously computed values, the numerical directionality differs, because
of when division by very small pivots occurs in randomly generated matrices.
On my machine, I observed that small diagonal entries in randomly generated
lower- and upper-triangular matrices often appeared closer to the top left
corner of the matrix. I played with various random seeds, but didn't notice
any changes in this pattern. So, in forward substitution, these matrices
amplified rounding error at the beginning of the computation, which allowed
that error to propagate through all subsequent steps.
Consequently, forward substitution often exhibited larger accumulated
numerical error. In back substitution, on the other hand, this limited
how far amplified error could spread, because these divisions occurred
much later in the computation.

My conclusion is that, on my machine, the random lower-triangular matrices
tend to be more poorly conditioned than upper-triangular matrices.

The ground-truth NumPy solutions were not exact. I expected that as well.
NumPy performs the same floating-point arithmetic, so the comparison
measures differences between two floating-point computations, not deviation
from an analytical solution.

Passing np.allclose meant that each component more or less agreed with
machine precision. A norm of approximately 2.5x10^-2 in a 50-dimensional
system was not alarming, because it reflected a normal floating-point
behavior.

In summary, the observed numerical behavior was consistent with theoretical
expectations and demonstrated how algorithmic structure and floating-point
arithmetic interact in real scientific computing. The poor conditioning
of lower-triangular matrices (on my machine!) was a small
(but very instructive) discovery to me!
```

Problem 2: LU Decomposition with Pivoting (1 point)

In this problem, we will explore one of the most important ideas in numerical linear algebra: **LU decomposition with row pivoting**. This is the algorithmic backbone of many modern linear system solvers.

Given a square matrix $A \in \mathbb{R}^{n \times n}$, LU decomposition with pivoting factors A as:

$$PA = LU,$$

where:

- P is a permutation matrix (row reordering),
- L is a unit lower-triangular matrix (ones on the diagonal),
- U is an upper-triangular matrix.

This factorization allows us to solve a linear system

$$Ax = b$$

by converting it into two triangular systems.

Why Pivoting Matters

Even if a linear system can be solved *without* row permutations on paper, numerical solvers often introduce pivoting anyway. Why?

- Pivoting improves numerical stability.
- It avoids division by very small numbers.
- It reduces amplification of floating-point error.

As a result, **SciPy may permute rows even when it is not strictly necessary for correctness**. This is not a bug – it is a feature.

The file `cs3430_s26_hw_4_prob_2.py` contains a method stub:

```
lu_decompose_and_solve(A, b)
```

You will implement this method to:

1. Compute the LU decomposition of A using `scipy.linalg.lu`.
2. Extract matrices P, L, and U.
3. Correctly solve the system $Ax = b$ by:
   (a) computing $b' = P^T b$,
   (b) solving $Ly = b'$ via forward substitution,
   (c) solving $Ux = y$ via back substitution.

The method must return the tuple:

$$(P, L, U, x).$$

Why We Must Compute $b' = P^T b$: A Concrete Example

Consider the linear system

$$Ax = b, \quad \text{where} \quad A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

This system has a perfectly valid solution, but the leading diagonal entry $a_{11} = 0$ makes it unsuitable for direct elimination.

A numerical solver fixes this by swapping the two rows of A. This swap is represented by the permutation matrix

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Multiplying by P swaps the rows:

$$PA = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

At this point, the system being factored is no longer $Ax = b$, but rather

$$PAx = Pb.$$

However, SciPy defines LU decomposition in the form

$$A = PLU,$$

not

$$PA = LU.$$

Thus, the original system

$$Ax = b$$

becomes

$$PLUx = b.$$

Multiplying both sides by P^T gives us

$$LUx = b',$$

where

$$b' = P^T b.$$

This is an implementation convention rather than a mathematical necessity. Understanding this distinction is an important part of learning numerical linear algebra, where theoretical algorithms are often adapted to practical software interfaces.

Unit Tests and an Expected Failure

The unit test file `cs3430_s26_hw_4_prob_2_uts.py` contains several tests. Most of them are expected to pass if your implementation is correct.

However, one test is intentionally included that **is expected to fail**:

`test_lu_structure_medium_expected_failure`

This test checks whether:

$$PA \approx LU$$

holds numerically for a medium-sized random matrix using `np.allclose`.

You will observe that this test fails, even though:

- the LU decomposition is correct,
- the computed solution x matches NumPy's solution,
- all other tests pass.

An Expected Test Failure (and Why It Happens)

One of the unit tests for this problem intentionally checks whether

$$PA \approx LU$$

for a medium-sized random matrix. You will observe that this test fails, even when:

- the LU decomposition is correct,
- the solution vector x matches NumPy's solution,
- all other tests pass.

This failure is expected. LU decomposition with pivoting is numerically stable, but it is not an exact algebraic identity in floating-point arithmetic. The matrices L and U are constructed using

many floating-point operations, each of which introduces small rounding errors. When these errors accumulate, the product LU can differ noticeably from PA, even though the factorization is mathematically valid.

The point of this failure is to show you that the purpose of LU decomposition is not to reproduce PA exactly, but to allow us to solve the original system $Ax = b$ accurately. Indeed, the computed solution x satisfies $Ax \approx b$ to machine precision.

This test illustrates an important takeaway in scientific computing:

> In numerical linear algebra, insisting on exact matrix identities is often a mistake; what matters is whether the computed solution solves the original system $Ax = b$.

Run the tests. Observe the failure. Think carefully about why it happens (if it happens on your machine as it does on mine). Internalize it and file it away in your brain for the future.

Problem 3: Archimedes Method for Computing π (1 point)

Long before calculus, Archimedes discovered a remarkable way to compute bounds on π using only geometry. And, nothing else! His method approximates the circumference of a circle by inscribing and circumscribing regular polygons and then increasing the number of polygon sides.

In this problem, we will implement Archimedes method on a unit circle (radius 1) and observe how numerical bounds on π converge.

Let:

- s_n be the side length of an *inscribed* regular n-gon;
- t_n be the side length of a *circumscribed* regular n-gon.

Archimedes showed that:

$$\frac{ns_n}{2} \;\leq\; \pi \;\leq\; \frac{nt_n}{2}.$$

Starting from a regular hexagon ($n = 6$) on the unit circle:

$$s_6 = 1, \qquad t_6 = \frac{2}{\sqrt{3}},$$

we repeatedly double the number of sides using the following recurrence relations (cf. Lecture 7):

$$s_{2n} = \sqrt{2 - \sqrt{4 - s_n^2}},$$

$$t_{2n} = \frac{4\left(\sqrt{1 + \frac{t_n^2}{4}} - 1\right)}{t_n}.$$

Each doubling produces tighter lower and upper bounds on π.

The file `cs3430_s26_hw_4_prob_3.py` contains methods that:

- initialize Archimedes method at $n = 6$;
- iteratively double the number of sides up to $n = 12288$;
- compute lower and upper bounds on π at each step;
- print a table of bounds similar to the one shown in the lecture PDF.

One of the unit tests in `cs3430_s26_hw_4_prob_3_uts.py` prints the full Archimedes table. This test is *observational*: it always passes and exists so that you can inspect numerical behavior rather than verify a specific value.

In `cs3430_s26_hw_4_prob_3.py`, you will implement Archimedes' method.

Part 1: Iterative Bounds

Write code that:

- starts with $n = 6$,
- repeatedly doubles the number of sides,
- computes π_{lower} and π_{upper} at each step,
- continues until a specified maximum number of sides (e.g., $n = 12288$).

Your implementation must preserve the following properties:

- monotone increase of the lower bound;
- monotone decrease of the upper bound;
- enclosure of π at every step.

Part 2: Midpoint Approximation

Implement the method:

```
archimedes_pi(n: int) -> float
```

This method should:

- compute the lower and upper bounds for a given n;
- return their midpoint:

$$\pi_{\text{est}} = \frac{\pi_{\text{lower}} + \pi_{\text{upper}}}{2}.$$

We will compare this midpoint estimate to `math.pi` in the unit tests.

Unit Tests

The unit tests in `cs3430_s26_hw_4_prob_3_uts.py` verify:

- correct initial bounds for $n = 6$;
- monotone squeeze behavior;
- enclosure of π at every iteration;
- convergence of the gap $\pi_{\text{upper}} - \pi_{\text{lower}}$;
- correctness of the midpoint estimate for $n = 96$;
- printing of a table that shows the evolution of the bounds.

Short Write-Up (Required)

After running the unit tests, write a short write-up on the printed table and save in as a multi-line comment in `cs3430_s26_hw_4_prob_3.py`. My write-up is below.

My Write-Up

This is the table produced by my implementation.

```
START: test_print_archimedes_table
Archimedes' Method Table (up to n = 12288):
     n        pi_lower        pi_upper              gap
-------------------------------------------------------------
     6  3.000000000000  3.464101615138 4.641016151378e-01
    12  3.105828541230  3.215390309173 1.095617679432e-01
    24  3.132628613281  3.159659942098 2.703132881627e-02
    48  3.139350203047  3.146086215131 6.736012084595e-03
    96  3.141031950891  3.142714599646 1.682648755044e-03
   192  3.141452472285  3.141873049980 4.205776945216e-04
   384  3.141557607912  3.141662747055 1.051391434461e-04
   768  3.141583892149  3.141610176600 2.628445058583e-05
  1536  3.141590463237  3.141597034323 6.571086575313e-06
  3072  3.141592106043  3.141593748817 1.642773807653e-06
  6144  3.141592516588  3.141592927874 4.112854785632e-07
 12288  3.141592618641  3.141592725623 1.069818020838e-07
```

The table shows lower and upper bounds on π obtained from inscribed and circumscribed polygons with increasing numbers of sides.

The first thing I observed is a clear monotone squeeze behavior. The lower bound increases at every step, while the upper bound decreases. At no point do the bounds cross, and π always remains enclosed between them. This confirms that the method is mathematically correct and numerically stable.

As the number of sides doubles from 6 to 12288, the gap between the bounds shrinks steadily. However, the convergence is relatively slow. Even after doubling the number of sides eleven times, the gap is on the order of 10^{-7}. This shows that Archimedes method gains accuracy gradually and requires very large polygons to achieve high precision.

The method is stable, because it relies on square roots and well-behaved recurrence relations rather than subtraction of nearly equal numbers or ill-conditioned linear systems. At the same time, it is slow compared to methods like Newton–Raphson, which converge much faster but rely on calculus. On the other hand, Archimedes' method never overshoots π: unlike fast iterative methods (e.g., NewtonRaphson), it trades convergence speed for certainty.

Problem 4: Related Rates, Geometry, and π (1 point)

In this problem, we combine geometry, calculus, and numerical computation to study a classic *related rates* problem from fluid mechanics. This problem also gives us an opportunity to explore how numerical approximations of π affect a physically meaningful quantity.

Physical Setup

Consider a water tank in the shape of an **inverted circular cone** with:

- base radius R meters,
- height H meters.

Water is pumped into the tank at a constant rate

$$\frac{dV}{dt} \quad \text{(in cubic meters per minute)}.$$

At a given moment, the water reaches height h meters. Our goal is to compute the rate at which the water level is rising,

$$\frac{dh}{dt},$$

at that moment.

Mathematical Model

Let us set up a model for this fluid mechanics problem. By similar triangles, the radius of the water surface is related to the water height by

$$r = \frac{R}{H}\, h.$$

The volume of water in the tank is therefore

$$V = \frac{1}{3}\pi r^2 h = \frac{1}{3}\pi \left(\frac{R}{H}\right)^2 h^3.$$

Differentiating with respect to time t and applying the chain rule gives:

$$\frac{dV}{dt} = \frac{dV}{dh}\frac{dh}{dt}.$$

Solving for $\frac{dh}{dt}$ yields the desired rate of change of the water level in meters per minute.

In `cs3430_s26_hw_4_prob_4.py`, you will implement a function that:

- symbolically differentiates the volume formula using `SymPy`;
- evaluates $\frac{dh}{dt}$ numerically for given parameters;
- allows the numerical value of π to be supplied explicitly.

This design allows you to compare results obtained using:

- `math.pi`;
- your Archimedes approximation `archimedes_pi(n)`.

Unit Tests

The unit tests in `cs3430_s26_hw_4_prob_4_uts.py`:

- compute $\frac{dh}{dt}$ using `math.pi`;
- compute $\frac{dh}{dt}$ using `archimedes_pi(96)`;
- compare the two results numerically.

You should observe that the two values are close but not identical. This difference reflects the numerical accuracy of the approximation to π, not an error in the calculus or geometry.

While no write-up is required for this problem, I do want you to reflect on whether the result obtained with Achimedes' method is different from the result obtained with math.pi. It was, on my machine, but the difference was small. The takeaway is this: we can compute trillions of digits in the mantissa of π. However, insisting on exact constants or exact identities is often a wrong approach. What matters is whether the computed approximation is *accurate enough for the physical problem being modeled*. If it is, we are done!

Problem 5: Image Blurring and Deblurring as an Inverse Problem (1 point)

In this problem, we revisit images from HW 3 and view image blurring and deblurring through the lens of LU decomposition, numerical stability, and the dangers of inversion.

You are given several real images in the `imgs/` directory:

- `BirdOrnament.jpg`
- `Elephant.jpg`
- `hive_1.png`
- `june.jpg`
- `nt_01.jpg`
- `road_1.png`
- `road_2.png`
- `road_3.png`
- `road_4.png`
- `sudoku.jpg`

These are the same images (or a subset of them) that you used in HW 3 for edge detection.

Blurring as a Linear System

We can model *row-wise blurring* of an image as a linear system. Each row of a grayscale image is treated as a vector $x \in \mathbb{R}^n$, and blurring is modeled as

$$Ax = b,$$

where:

- x is the original (sharp) image row;
- b is the blurred image row;
- A is a tridiagonal **blur matrix** that mixes neighboring pixels.

The blur matrix A has the form:

$$A = \begin{bmatrix} 1 & \alpha & 0 & \cdots & 0 \\ \beta & 1 & \alpha & \cdots & 0 \\ 0 & \beta & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \alpha \\ 0 & \cdots & 0 & \beta & 1 \end{bmatrix},$$

where α and β control how strongly each pixel is influenced by its neighbors.

In the file `cs3430_s26_hw_4_prob_5.py`, you are given helper functions to:

- construct this blur matrix;
- blur a single row;
- blur an entire image row by row.

Blurring is a **forward problem**. It is numerically stable and always succeeds.

Deblurring as an Inverse Problem

Deblurring attempts to recover x from b by solving

$$Ax = b.$$

This is an **inverse problem**. We will attempt to solve it using LU decomposition (with forward and back substitution) *without regularization.*

In this problem, you will:

1. Blur each image row-wise.
2. Attempt to deblur each row using LU decomposition.
3. Reassemble the image from the recovered rows and look at its quality.

Important Note: Deblurring may fail. LU decomposition can encounter zero or near-zero pivots, especially when noise is present. When this happens, your code should raise an exception rather than silently producing nonsense.

Expected-Failure Unit Test

One of the unit tests for this problem is *expected to fail.* It deliberately attempts to deblur noisy data using LU decomposition and checks that the solver raises an error. Sometimes, the most honest outcome is failure.

Why Deblurring Can Be Slow

You may notice that some images take much longer to deblur than others. This happens because:

- Each image row requires solving a linear system;
- Larger images $\rightarrow$ more rows $\rightarrow$ more LU decompositions;
- Ill-conditioned rows cause: very small pivots, slow numerical convergence, possible solver failure.

In practice, image deblurring is almost always done using least-squares or regularized methods, which are faster and more stable than exact solvers. In this assignment, however, we deliberately used LU decomposition to expose the numerical difficulties of inverse problems.

It is worth noting that LU decomposition can be parallelized. Modern numerical linear algebra libraries distribute LU computations across many processors using block-based algorithms and specialized hardware. However, such parallelism typically requires access to high-performance computing resources (clusters, GPUs, or optimized vendor libraries).

In high-performance settings, image-based linear systems are often solved using domain decomposition: the image is partitioned into smaller subimages (e.g., 30×30 blocks), and each block is processed independently. Since these subproblems are independent, they can be assigned to different CPU cores, GPUs, or compute nodes and solved in parallel. This strategy significantly reduces wall-clock time and is commonly used in large-scale image processing and scientific computing pipelines. But: such parallelization typically requires access to high-performance compute resources and specialized software.

A broader lesson of this assignment is that being faithful to the mathematics often comes at a computational cost. Algorithms such as LU decomposition or exact inversion of linear operators are mathematically correct, but running them efficiently at scale may require substantial hardware resources, such as parallel CPUs, GPUs, or compute clusters. Mathematics tells us what is possible; hardware often determines what is practical.

Exact continuous mathematics is free on paper, but expensive on silicon.

What to Submit

Zip your solutions into `cs3430_s26_hw_4.zip` and upload it in Canvas. Your zip should contain:

1. `cs3430_s26_hw_4_prob_1.py` with a short write-up addressing your numerical observations. Please save this write-up in `cs3430_s26_hw_4_prob_1.py` as a multi-line comment at the beginning of the file.
2. `cs3430_s26_hw_4_prob_2.py`.
3. `cs3430_s26_hw_4_prob_3.py` with your write-up on the printed table saved as a multi-line comment at the beginning of the file.
4. `cs3430_s26_hw_4_prob_4.py`.
5. `cs3430_s26_hw_4_prob_5.py` file with the blurred and deblurred images saved by your code.

Happy Hacking! Enjoy watching the π squeeze and the slow blur!

References

1. L. Goldstein, D. Lay, D. Schneider, N. Asmar. *Calculus and its Applications*.
2. E. Burger. *Zero to Infinity: A History of Numbers*. The Great Courses Publishing, 2007.
3. www.sympy.org.