To run the code

```
python solution.py
```

Report

Algorithm description

Gaussian Elimination with partial pivoting

Gaussian Elimination (GE) is a method of solving systems of linear equations of the form Ax = b. It does this by applying row operations on every row of the augmented (A | b) matrix until it becomes row echelon form, or in other words, becomes an upper triangular matrix.

The row operation in question, is Ei ← (Eij / Ejj) * Ej, which turns every number below the current row's pivot to 0s (Ejj is the pivot). Do this to every row, and we'll get a matrix that looks like an upper triangular matrix.

Once we have the upper triangular matrix we perform backsubstition, which solves for all elements of our x solution vector backwards. This is trivial since the last equation will have 1 unknown and one equation, the second to last will have 2 unknowns and one equation, but can substitute in the unknown we just solved for, and so on, all the way up to the first index, and the system is solved!

```
def backsubstitution(A : List[List[float]]) → List[float]:
  N, M = len(A)-1, len(A[0])-1
  x : List[float]  = [0.0 for _ in range(N+1)]
  x[N] = A[N][M]/A[N][N]
  for i in range(N,-1,-1):
    Σaijxj = sum([ A[i][j] * x[j] for j in range(i+1, N+1)])
    x[i] = (A[i][M] - Σaijxj)/A[i][i]
  return x
```

At least in theory. In practice, there are issues if Ejj is 0 or is very close to 0. So the division (Eij / Ejj) will return a very big number or worse, infinity.

Big numbers are not inherently bad, but in a numerical context using floating point numbers, they are very bad. Floating point numbers become more sparse as their magnitude grows. So any operation that results in a big number will be inherently inaccurate.

Parital pivoting is a way to address this, by ensuring that the pivot is always the biggest number available along that column.

So right before doing the titular elimination step of Gaussian Elimination™, we find the row with the absolute biggest number in the column, then swap it with the current index.

```
max_row = k
max_val = abs(A[row_idx[k]][k])
for i in range(k+1, N):
  if abs(A[row_idx[i]][k]) > max_val:
    max_val = abs(A[row_idx[i]][k])
    max_row = i
if max_row != k: row_idx[k], row_idx[max_row] = row_idx[max_row], row_idx[k]
```

```
for i in range(k+1, N):
  m = A[row_idx[i]][k] / A[row_idx[k]][k]
  for j in range(k, M):
      A[row_idx[i]][j] = A[row_idx[i]][j] - m * A[row_idx[k]][j]
```

Of course, this messes up the order of the solution, so at the final step, before doing backsubstitution, we revert all of our row swaps.

```
result = [A[row_idx[i]][:] for i in range(N)]
return result
```

Gauss Seidel

The Gauss-Seidel method is an iterative method for solving systems of linear equations. Unlike Gaussian Elimination, which gives you the answer in one go (albeit with a lot of computation), Gauss-Seidel

starts with a guess and gradually refines that guess until we're satisfied with the answer.

The core idea is fairly intuitive: for each equation in our system, we'll solve for one variable, assuming we know the values of all other variables. Of course, we don't actually know those other values when we start, so we just use our current best guess.

The iterative formula looks like this:

$$x_i^{(k)} = (b_i - \Sigma_{j=1}^{i-1} a_{ij} x_j^{(k)} - \Sigma_{j=i+1}^{n} a_{ij} x_j^{(k-1)})/a_{ii}$$

The clever bit is that when updating x_i, we use the NEWEST values of x_j for j < i (that's the x_j^(k) part) and the PREVIOUS iteration's values for j > i (the x_j^(k-1) part). This makes Gauss-Seidel converge faster than methods that only use previous iteration values for everything.

We've implemented two different ways to check for convergence:

1. The simpler approach is to check if the solution isn't changing much between iterations: $||x^{(k)} - x^{(k-1)}||_\infty < \varepsilon$
2. The more rigorous approach is to check the residual vector: $||Ax^{(k)} - b||_\infty < \varepsilon$

This measures how well our current solution actually satisfies the original equations.

Our function lets you choose which approach to use with the `use_residual` parameter. As we'll see in the results, checking the residual gives better quality solutions but takes WAY more computation time, especially for large systems.

The beauty of Gauss-Seidel is that for certain types of matrices (diagonally dominant or symmetric positive-definite), it's guaranteed to converge. And for large, sparse systems, it can be much faster than direct methods like Gaussian Elimination. However, for poorly conditioned matrices, it might converge very slowly or not at all.

Results

The experimental results demonstrate interesting performance and accuracy characteristics of both Gaussian Elimination with partial pivoting and the Gauss-Seidel method across different system sizes.

Small System (5×5)

For the small 5×5 system with the known solution [8, 6, 7, 5, 3]:

Gaussian Elimination with partial pivoting:

- L1 error: 3.11e-15
- L2 error: 2.04e-15
- L∞ error: 1.78e-15
- Solve time: 2.60e-05 seconds

Gauss-Seidel with |x - x_last| convergence criterion:

- L1 error: 1.53e-11
- L2 error: 9.05e-12
- L∞ error: 7.90e-12
- Solve time: 6.72e-05 seconds

Gauss-Seidel with Ax - b residual criterion:

- L1 error: 0.0
- L2 error: 0.0
- L∞ error: 0.0
- Solve time: 0.048 seconds

For the small system, Gaussian Elimination achieves near machine precision (errors ~10^-15) and is the fastest method. The Gauss-Seidel method with the standard convergence criterion produces acceptable accuracy (errors ~10^-12) but takes about 2.6× longer. The residual-based Gauss-Seidel approach reports perfect accuracy (likely due to numerical precision in the output formatting) but at a significant time cost (1800x slower than Gaussian Elimination).

Large System (100×100)

For the large 100×100 system:

Gaussian Elimination with partial pivoting:

- L1 error: 4.85e-07
- L2 error: 6.18e-08
- L∞ error: 1.61e-08
- Solve time: 0.016 seconds

Gauss-Seidel with |x - x_last| convergence criterion:

- L1 error: 4.85e-07
- L2 error: 6.18e-08
- L∞ error: 1.61e-08
- Solve time: 0.004 seconds

Gauss-Seidel with Ax - b residual criterion:

- L1 error: 4.85e-07
- L2 error: 6.18e-08
- L∞ error: 1.61e-08
- Solve time: 6.56 seconds

For the larger system, the accuracy of both methods is virtually identical across all error metrics (approximately 10^-7 for L1, 10^-8 for L2 and L∞). However, the performance differences become striking:

- Standard Gauss-Seidel is 4× faster than Gaussian Elimination
- Residual-based Gauss-Seidel is 400× slower than Gaussian Elimination

The speed comparison matrix reveals the relative performance clearly:

```
[1.0, 0.2681, 412.9778]  // GE vs all
[3.7302, 1.0, 1540.5043] // GS vs all
[0.0024, 0.0006, 1.0]    // GS-residual vs all
```

Computational Cost Analysis

The results align with theoretical expectations:

1. Scaling Properties:
    - Gaussian Elimination has $O(n^3)$ complexity, which becomes apparent as we move from the small to large system

    - Gauss-Seidel's per-iteration $O(n^2)$ complexity, combined with rapid convergence for this well-conditioned system, makes it more efficient for the larger problem.
2. **Convergence Criteria Impact**:
    - The standard $|x - x_{last}|$ criterion is computationally efficient
    - The residual-based criterion ($||Ax - b||$) dramatically increases computational cost, making it impractical for most applications despite its theoretical advantages
3. **Error Behavior**:
    - For the small system, direct methods provide superior accuracy
    - For the large system, both methods achieve comparable accuracy, suggesting that for well-conditioned matrices, iterative methods can be just as accurate as direct methods

Performance and Accuracy Implications

The choice between these methods should consider:

1. **System Size**: For small systems ($n \leq 50$), Gaussian Elimination is typically preferred for its combination of speed and accuracy. For larger systems, Gauss-Seidel becomes increasingly attractive.
2. **Matrix Properties**: While not directly tested in this experiment, Gauss-Seidel's convergence heavily depends on the matrix being diagonally dominant or having other favorable properties. Gaussian Elimination with partial pivoting is more robust for general matrices.
3. **Accuracy Requirements**: If machine precision is required, direct methods are preferable for small to medium systems. For large systems where approximate solutions are acceptable, iterative methods offer a good balance of accuracy and efficiency.
4. **Convergence Criteria**: The standard difference-based convergence criterion offers an excellent compromise between accuracy and speed for iterative methods. The residual-based criterion, while theoretically sound, imposes excessive computational costs that negate the advantages of iterative methods.

These findings suggest that for large-scale scientific computing applications with well-conditioned matrices, iterative methods like Gauss-Seidel are preferable, while direct methods remain valuable for smaller systems or when exact solutions are required.

Challenges

The Python implementation was incredibly straight forward and involved no real roadblocks.

I, however, also spent 2 days trying to get this to work in lean4, which was a monumentally difficult task that I ultimately failed. Here are the core reasons why I failed to implement Gaussian Elimination and Gauss Seidel in a pure, dependently typed, and functional language:

- Indexing an array is non-trivial: Lean4 requires absolute certainty that programs are valid. Thus it is not enough to simply "prove" by construction that an i is a valid index for some array A (say, by declaring i as i in range(len(A))). More accurately, it is possible to prove via construction, but that proof needs to be carried around (as a boolean proof/statement) should i pass off something to a subroutine (like backsubstitution).
- Nesting uniform length arrays into another array is also non-trivial: Lean4 does not care that I explicitly declared something as a nested array containing uniform length arrays. len(A[0]) is not at all guaranteed to be the same as len(A[1]). It is my job to prove this, every single time A is passed into a new scope, always.
- Reading a file is non-trivial: as a pure functional language, lean4 cannot simply "read a file", because "read a file" involves invoking side effects. Therefore, it cannot simply do read("somepath") and return, say, a string. It must, instead, encapsulate that read operation, and everything it touches, around a Monadic IO type. This disables about half the features available to the langauge, such as: `#eval` for printing stuff or `#check` for seeing types. This made testing and development completely hellish.

So after the 2 days of spam programming this in Lean4, I promptly gave up the next day. The python version by comparison, was completed in a single sitting.