MATH3511/6111: Scientific Computing
Mathematical Sciences Institute, Australian National University
Semester 1, 2022

Lab 4: Linear Solvers, Sparse Matrices

Lindon Roberts

Due date: 9am, Monday 16 May (week 11)

This lab is based on earlier versions by Stephen Roberts, Graeme Chandler, Jimmy Thomson, Linda Stals and Kenneth Duru.

1 Norms and Conditioning

The NumPy function `np.linalg.norm` calculates different vector and matrix norms.

```python
import numpy as np

# Create an example vector
x = np.sin(np.arange(10))
print("x =", x)
print("||x||_2 =", np.linalg.norm(x, ord=2))
print("||x||_1 =", np.linalg.norm(x, ord=1))

# Create an example matrix
A = np.arange(36).reshape((6,6))
print("A =", A)
print("||A||_1 =", np.linalg.norm(A, ord=1))
print("||A||_2 =", np.linalg.norm(A, ord=2))
```

Lab Book 1. By reading the documentation for `np.linalg.norm`, calculate the infinity norm of x and the Frobenius norm of A (for x and A as given in the above code). What is the 'default' choice for `ord` (i.e. what do you get if you just calculate `np.linalg.norm(x)` or `np.linalg.norm(A)`)? [4 points]

The condition number of a matrix is defined as $\kappa(A) = \|A\| \cdot \|A^{-1}\|$ (but this definition can be generalised to non-square and/or non-invertible matrices). It is implemented in the function `np.linalg.cond`.

```python
print("A =", A)
print("kappa(A) =", np.linalg.cond(A))  # using operator 2-norm, but other norms available
```

As we know (and saw in Lab 1), ill-conditioned matrices tend to be difficult to solve. Vandermonde matrices (see the polynomial interpolation lectures) are often ill-conditioned, for example. The code below demonstrates what happens when you solve a linear system with an ill-conditioned matrix (even if you use a good, backwards stable, algorithm).

```python
# Make a set of interpolation nodes (e.g. equally spaced in [0,2])
n = 20
xdata = np.linspace(0, 2, n+1)

# Build the Vandermonde matrix (ill-conditioned)
A = np.zeros((n+1,n+1))
for i in range(n+1):
    A[:,n-i] = xdata**i

# Solve Ax=b for a right-hand side b where we know the true solution
xtrue = np.arange(n+1)
b = A @ xtrue
x = np.linalg.solve(A, b)  # solve using Gaussian Elimination with partial pivoting
print("xtrue =", xtrue)
print("x =", x)
print("Condition number =", np.linalg.cond(A))
```

2 Direct Linear Solvers

One way to solve linear systems is using direct methods, which usually means Gaussian Elimination. The function `np.linalg.solve` that we used above uses Gaussian Elimination with partial pivoting.

Also, SciPy gives us functions which explicitly calculate the LU factorisation $A = PLU$ and can then solve linear systems using this factorisation. Use this if you need to solve multiple linear systems with the same A (but different right-hand sides), or if you want to compute your factorisation in advance (to speed up future linear solves).

Warning! SciPy uses the convention $A = PLU$, which is different to $PA = LU$ as used in the lectures. What is the relationship between the two different P matrices?

```
import scipy.linalg as linalg

# Create an example 6*6 linear system (related to elastic membranes)
n = 6
A = np.diag(np.ones(n-1), -1) - 2*np.diag(np.ones(n)) + np.diag(np.ones(n-1), 1)
b = np.ones(n)

# Calculate LU factorisation
lu, piv = linalg.lu_factor(A)

# Solve a linear system using a previously computed LU factorisation
x = linalg.lu_solve((lu, piv), b)
```

Lab Book 2. Run the above code. What is the solution x? Check this by measuring the size of the residual $\|Ax - b\|_2$. By reading the documentation and the lecture notes, explain how `lu` and `piv` store the information in P, L and U. [5 points]

In lab 1, we saw that solving a linear system with the 15×15 Hilbert matrix can produce a very bad solution. There, we saw a large error when solving $Ax = b$ via $x = A^{-1}b$, but the same thing can happen using `np.linalg.solve` (Gaussian Elimination with partial pivoting).

```
n = 15
A = np.zeros((n,n))
#######
# TODO: build n*n Hilbert matrix as per lab 1
#######
xtrue = np.ones((n,))
b = A @ xtrue              # RHS of Ax=b with known solution xtrue
x = np.linalg.solve(A, b)  # Gaussian Elimination with partial pivoting
print("Relative error of np.linalg.solve =", np.linalg.norm(x - xtrue) / np.linalg.norm(xtrue))
```

Lab Book 3. Explain why the `np.linalg.solve` applied to the Hilbert matrix (as per the above code) produces such a bad answer. [5 points]

Lab Book 4. Based on the LU factorisation code given in lectures, write your own function which implements Gaussian Elimination with partial pivoting (i.e. produces the factorisation $PA = LU$). Your function should produce P, L and U as dense $n \times n$ matrices. For an example linear system of your own, show that your function gives the same answer as `scipy.linalg.lu`. [5 points]

Lab Book 5. Write your own function which, given a pre-computed factorisation $PA = LU$, solves a linear system $Ax = b$. For the same example linear system as the previous question, show that your function gives the same answer as a built-in NumPy/SciPy routine. [3 points]

We can also see from this example that A is sparse (many entries are zero), and that both L and U are also sparse. However, we can easily check that A^{-1} is not sparse, using `np.linalg.inv`.

A common alternative factorisation is the QR factorisation: if $A \in \mathbb{R}^{n\times n}$, we calculate $A = QR$, where Q is an $n \times n$ orthogonal matrix ($Q^T Q = I$) and $R \in \mathbb{R}^{n\times n}$ is upper triangular. NumPy can calculate QR factorisations using `np.linalg.qr`. Depending on the specific algorithm, this process takes approximately $2n^3 + \mathcal{O}(n^2)$ flops, so is approximately 3 times slower to compute than an LU factorisation with partial pivoting. Given $A = QR$, we can solve a linear system $Ax = b$ by solving the upper triangular system $Rx = Q^T b$ using backward substitution. Fortunately, SciPy has a function for forward/backward substitution:

```
# Create an example 6*6 linear system (related to elastic membranes)
n = 6
A = np.diag(np.ones(n-1), -1) - 2*np.diag(np.ones(n)) + np.diag(np.ones(n-1), 1)
b = np.ones(n)

# Calculate QR factorisation
Q, R = np.linalg.qr(A)

# Solve a linear system using a QR factorisation
x = linalg.solve_triangular(R, Q.T @ b, lower=False)
```

Lab Book 6. Using the above code, check that $A = QR$ and $Q^T Q = I$, at least up to rounding error. Do not just print both sides of the equality, use matrix norms to print out summary information (this approach is necessary for large matrices). [2 points]

However, one advantage of QR is that it exists for all matrices $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ (not just square matrices). In this case, there are two ways of writing the factorisation:

- Full QR: $A = QR$ where $Q \in \mathbb{R}^{m\times m}$ is orthogonal ($Q^T Q = I$) and $R \in \mathbb{R}^{m\times n}$ is upper triangular;
- Reduced QR: $A = QR$ where $Q \in \mathbb{R}^{m\times n}$ is rectangular but has orthonormal columns ($Q^T Q = I_{n\times n}$), and $R \in \mathbb{R}^{n\times n}$ is square and upper triangular.

In most cases you only need the reduced QR (which is faster to compute). This is very useful for finding the best solution to overdetermined linear systems. Since an overdetermined system $Ax = b$ may not have any solution, we instead ask for the choice of x which minimises the norm of the residual:

$$\min_{x\in\mathbb{R}^n} \|Ax - b\|_2.$$

We use the reduced QR to solve an overdetermined system $Ax = b$ in the same way as above.

```
# Create an example 10x6 overdetermined linear system
m = 10
n = 6
A = np.sqrt(np.arange(m*n)).reshape((m, n))
b = np.ones(m)

# Calculate reduced QR factorisation
Q, R = np.linalg.qr(A, mode='reduced')

# Solve a linear system using a QR factorisation
x = linalg.solve_triangular(R, Q.T @ b, lower=False)

print("Residual =", np.linalg.norm(A@x - b))
```

Note: the solution of a least-squares problem is also the solution to the square linear system $(A^T A)x = A^T b$, and so we could instead solve this using Gaussian Elimination or the conjugate gradient method.[1] However, the matrix $A^T A$ often has a much larger condition number than A, so it is usually better to solve the least-squares problem directly using QR factorisations.

Overdetermined linear systems arise frequently in data fitting (called 'linear' or 'ordinary' least-squares problems). NumPy also has built-in functions to solve least-squares problems (see if you can find them, and check that you get the same answer from using QR factorisations).

3 Iterative Linear Solvers

The other way to solve linear systems is to use an iterative method, where we produce a sequence of vectors x_0, x_1, x_2, ... which converge to the true solution. This can be useful if we are happy to get an inaccurate solution quickly (rather than a high-accuracy solution eventually), or can only access the matrix via matrix-vector products (i.e. we can only calculate Ax given an x in some way, and we don't have access to all the entries of A). The most famous iterative linear solver is the Conjugate Gradient method (CG), which is suitable when A is symmetric positive definite.

```python
def conjugate_gradient(A, b, x0, tol):
    # Initialise variables
    x = x0.copy()

    # Run the main CG loop
    k = 0
    residual_norm_history = []
    residual_norm_history.append(np.linalg.norm(r))
    while k < len(b) and np.linalg.norm(r) >= tol:  # stop when k=n or small residual
        # Complete CG iteration here

        # Store the norm of the current residual
        residual_norm_history.append(np.linalg.norm(r))
        k += 1

    return x, np.array(residual_norm_history)
```

Lab Book 7. Complete the above code to get an implementation of CG (based on 'classic' version in Algorithm 2 of the lecture slides). Make sure you only interact with A via performing exactly one matrix-vector product in each iteration. The below code can be use to test your `conjugate_gradient` function by plotting the norms of the residuals $\|r_k\|$ for a test symmetric positive definite system. Using the below test code for one choice of n, verify that CG converges in at most n iterations for an $n \times n$ matrix. [3 points]

```python
import matplotlib.pyplot as plt

# Create example symmetric positive definite linear system
n = 10
A = -np.diag(np.ones(n-1), -1) + 2*np.diag(np.ones(n)) - np.diag(np.ones(n-1), 1)
b = np.arange(n)
x0 = np.zeros((n,))

# Solve with CG
x, resids = conjugate_gradient(A, b, x0, tol=1e-5)

print("Final residual ||Ax-b|| =", np.linalg.norm(A@x - b))
```

[1]This linear system is sometimes referred to as the 'normal equations'. What conditions on A make the matrix $A^T A$ symmetric positive definite (so you can use CG)?

```python
# Plot decrease in residuals
plt.figure(1)
plt.clf()
plt.semilogy(resids, 'o-')
plt.grid()
plt.xlabel('k')
plt.ylabel('||rk||')
plt.show()
```

We know that CG converges in at most n iterations. However, if the eigenvalues of A are all similar, or clumped into a small number of distinct regions, then CG will converge to a good solution after (potentially) many fewer than n steps.

Lab Book 8. Construct two 20×20 symmetric positive definite matrices with condition number $\kappa(A) = 10^5$: one with all eigenvalues distinct, and one with exactly 5 distinct eigenvalues (each with multiplicity 4). You will need to explain how you know you have satisfied these requirements. Compare the performance of CG on both matrices. [4 points]

The below code can be used to generate a matrix with a desired set of eigenvalues (and a random set of eigenvectors).

```python
# Set the random number generator (for reproducibility)
np.random.seed(0)

# Build a random n*n orthogonal matrix of eigenvectors
n = 10
Q = np.linalg.qr(np.random.rand(n, n))[0]

# Desired set of eigenvalues (need all > 0 for positive definite matrices)
evals = np.arange(1, n+1)

# Construct symmetric positive definite linear system with desired eigenvalues
A = Q.T @ np.diag(evals) @ Q
b = np.arange(n)
x0 = np.zeros((n,))
```

4 Sparse Matrices

If a matrix has many zero entries, then it is often better to store it as a *sparse matrix*. Instead of storing all the entries of the matrix, we only store the nonzero entries and their locations. The SciPy module `scipy.sparse` has lots of routines for creating and manipulating sparse matrices.

There are several 'formats' for storing sparse matrices in SciPy (based on how the nonzero indices and values are saved in memory). In general, it is easiest to create sparse matrices using the COO or LIL format, then convert to CSC or CSR formats for performing linear algebra tasks (like matrix multiplication or solving linear systems).

Suppose we wish to create a 10×10 sparse matrix where the nonzero entries are on the diagonal, first row, and first column:

$$A = \begin{bmatrix} 1 & 2 & 3 & \cdots & 10 \\ 2 & 2 & 0 & \cdots & 0 \\ 3 & 0 & 3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 10 & 0 & 0 & \cdots & 10 \end{bmatrix}.$$

We can create this by making lists of the nonzero column indices, row indices, and entries as follows:

```
import scipy.sparse as sparse

# Column indices of the nonzero values
# They are ordered as: first column (top to bottom), first row (second to last column),
    diagonal (2nd entry to last entry).
# Note: the '+' operator appends lists: [1,2] + [3,4,5] gives [1,2,3,4,5]
col_idx = [0]*10 + list(range(1, 10)) + list(range(1, 10))

# Row indices of the nonzero values
row_idx = list(range(10)) + [0]*9 + list(range(1, 10))

# Values of the nonzero entries
values = list(range(1, 11)) + list(range(2, 11)) + list(range(2, 11))

# Form the sparse matrix
nrows, ncols = 10, 10  # dimensions of A
A = sparse.coo_matrix( (values, (row_idx, col_idx)), shape=(nrows, ncols), dtype=float )

# Convert to CSR format for efficient linear algebra (if desired)
A = A.tocsr()
```

There are lots of things we can do with sparse matrices:

```
# Dimensions of a sparse matrix (same as NumPy arrays)
print("Dimensions =", A.shape)

# How many nonzero entries?
print("Number of nonzero entries =", A.nnz)

# The "sparsity" of a matrix is the fraction of entries which are nonzero
print("Sparsity =", A.nnz / (A.shape[0] * A.shape[1]))

# Convert to a regular NumPy array
# This is useful to check our code for col_idx, row_idx and values
# WARNING: do not do this for very large matrices (or you will run of memory)
A_as_dense_matrix = A.toarray()
print(A_as_dense_matrix)
```

There are many other things you can do, like calculate A^T or the maximum/minimum elements of A, for example. The SciPy documentation has more information about the available functions.

Standard linear algebra arithmetic works in the way you would expect:

```
# Create a sparse identity matrix the same size as A
I = sparse.eye(nrows)

A_plus_two_I = A + 2*I
print(A_plus_two_I.toarray())  # check answer correct by printing A+I as a dense matrix

# Matrix-vector multiplication
x = np.arange(nrows)
print("Ax =", A @ x)
print("A^T x =", A.T @ x)

# Matrix-matrix multiplication
A_times_I = A @ I
print(A_times_I.toarray())
```

You can also view and change entries of A using standard NumPy slicing.

```
print("Entry (1,1) is", A[1,1])
print("Entry (-1, -2) is", A[-1, -2])

# Change an entry from zero to nonzero
A[-1, -2] = -5.5
print("Entry (-1, -2) is", A[-1, -2])
```

Warning! I get a SciPy warning message saying it is slow to change the sparsity structure of a CSR matrix. In general, you should do all your matrix construction first (in COO or LIL format) and only change to CSR/CSC when you are finished.

A very useful way to visualise sparse matrices is to plot the sparsity pattern of the matrix (i.e. where the nonzero entries are located). This can be done using a 'spy' plot:

```
import matplotlib.pyplot as plt

plt.figure(1)
plt.clf()
plt.spy(A)
plt.show()
```

5 Sparse Linear Solvers

Since we do not explicitly store all the nonzero entries of a sparse matrix, it is very common to use iterative linear solvers to solve linear systems with a sparse A. However, sparse LU factorisation routines exist too: they are the same as regular Gaussian Elimination, but they do pivoting in a much more sophisticated way (to ensure both good numerical stability and making sure that the factorisation remains sparse). Fortunately, SciPy implements both of these categories of linear solvers (as well as other routines for eigenvalues, etc.).

Lab Book 9. Write code which generates the below $n \times n$ symmetric positive definite matrix A (where n can be changed to any value) as a sparse matrix in CSR format. For $n = 5$ check your code by converting A to a dense matrix and printing it out. [4 points]

Hint: look at the documentation for `sparse.diags`.

$$A = \begin{bmatrix} 7/2 & -4/3 & 1/12 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ -4/3 & 7/2 & -4/3 & 1/12 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 1/12 & -4/3 & 7/2 & -4/3 & 1/12 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 1/12 & -4/3 & 7/2 & -4/3 & \ddots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \ddots & 7/2 & -4/3 & 1/12 & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots & -4/3 & 7/2 & -4/3 & 1/12 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1/12 & -4/3 & 7/2 & -4/3 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 1/12 & -4/3 & 7/2 \end{bmatrix}.$$

If $n = 5$, you should get

$$A = \begin{bmatrix} 7/2 & -4/3 & 1/12 & 0 & 0 \\ -4/3 & 7/2 & -4/3 & 1/12 & 0 \\ 1/12 & -4/3 & 7/2 & -4/3 & 1/12 \\ 0 & 1/12 & -4/3 & 7/2 & -4/3 \\ 0 & 0 & 1/12 & -4/3 & 7/2 \end{bmatrix}.$$

Up to a missing factor depending on the grid size h, when multiplied by a vector of equally spaced values $(u(x_0), \ldots, u(x_n))$ the matrix A represents a finite-difference approximation to $-u''(x) + u(x)$.

Given a sparse matrix A, we can solve the linear system $Ax = b$ with

```
x = sparse.linalg.spsolve(A, b)  # A should be in CSR or CSC format
```

Alternatively, if we just want to build a sparse LU factorisation of A (with sparsity-respecting pivoting), so we can solve many linear systems later, we can do:

```
# Build the factorisation
sparse_LU_factorisation = sparse.linalg.splu(A)
# Now, use the factorisation to solve a linear system
x = sparse_LU_factorisation.solve(b)  # solve Ax=b
```

Lab Book 10. With `b=np.arange(n)`, compare the time taken to solve $Ax = b$ with `spsolve` and `np.linalg.solve` for $n = 10, 100, 1000$ and comment on your results. For `np.linalg.solve`, you will have to first convert A to a dense matrix (do not count this towards the time taken). [5 points]

Hint: the code below measures how long it takes to run a piece of code. If you get a memory error for creating a dense matrix with $n = 1000$, use a smaller value.

```
import time

# Measure how long it takes to run a piece of code (in seconds)
t_start = time.time()
# RUN CODE HERE
t_end = time.time()
time_taken = t_end - t_start
print("Total time taken (seconds) =", time_taken)
```

5.1 Sparse Iterative Methods

One of the advantages of an iterative linear solver like CG is that it you only need to have `A @ x` defined, which means you can use both dense and sparse matrices (but sparse matrices can much more quickly calculate `A @ x`, and need less memory to store A). Do the same timing test as above but using your implementation of CG (not for your lab book). You shouldn't need to make any changes to your CG code, just give it the dense or sparse version of A.

SciPy implements a number of useful iterative solvers for $Ax = b$ including CG and others like MINRES (symmetric but not positive definite matrices) and GMRES (any A). These are in `sparse.linalg.cg`, `sparse.linalg.minres`, `sparse.linalg.gmres`, etc. Investigate the different choices and compare the number of iterations they need for different matrices. Note that these functions allow A to be sparse or dense (or even a function that computes matrix-vector products).

Many of these routines also allow you to add a preconditioner to speed up convergence (a matrix $P \approx A$ which is easy to invert). Common choices include $P = \text{diag}(A)$ (Jacobi preconditioning) or incomplete LU factorisation. SciPy implements incomplete LU factorisations (`sparse.linalg.spilu`), for example.