CS319: Scientific Computing (with C++)

CS319 Lab 8: Linear Systems 2

Week 10 (14+15 March, 2024)

Goal:

To develop expertise in *operator overloading* and to demonstrate this by developing a new implementation of the Jacobi and Gauss-Seidel methods from Lab 7.

Deadline:

Submit your work by 17:00, Friday 22 March. You should upload a *single archive file*, such as a `zip` file, that contains all the necessary source files. Submit your solution through the "`Lab 8`" section on Canvas. You should include all necessary files for your program to compile: *even if they are unchanged from the versions your downloaded from the website*. **Include your name and ID number in each file.**

Recall Jacobi's method

In Lab 7 you developed implementations of the Jacobi and Gauss-Seidel algorithms for solving a linear system of N equations in N unknowns: *find* $x_1, x_2, \dots, x_N$, *such that*

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N &= b_N. \end{aligned}$$

We expressed this as a matrix-vector equation: *Find* $\mathbf{x}$ *such that*

$$A\mathbf{x} = \mathbf{b},$$

where A *is a* $N \times N$ *matrix, and* $\mathbf{b}$ *and* $\mathbf{x}$ *are (column) vectors with* N *entries*.

Recall Jacobi's method

Then **Jacobi's method** is: choose $\mathbf{x}^{(0)}$ and set

$$
\begin{aligned}
x_1^{(k+1)} &= \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1N}x_N^{(k)}) \\
x_2^{(k+1)} &= \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \cdots - a_{2N}x_N^{(k)}) \\
&\vdots \\
x_N^{(k+1)} &= \frac{1}{a_{NN}}(b_N - a_{N1}x_1^{(k)} - \cdots - a_{N,N-1}x_{N-1}^{(k)})
\end{aligned}
$$

There is also a matrix-version of this iteration. We set D and T to be the matrices

$$
d_{ij} = \begin{cases} a_{ii} & i = j \\ 0 & \text{otherwise.} \end{cases} \qquad t_{ij} = \begin{cases} 0 & i = j \\ -a_{ij} & \text{otherwise.} \end{cases}
$$

So $A = D - T$. Then *Jacobi's method* can be written neatly in matrix form:

$$
x^{(k+1)} = D^{-1}(b + Tx^{(k)}). \tag{1}
$$

Recall Jacobi's method

This week in classes, we learned about **operator overloading**.

- We overloaded the assignment operator `=` for `Vector`s.
- We overloaded the `Vector` addition and subtractions operators.
- We overloaded the `Matrix`-`Vector` multiplication operator.

Our next step is to implement Jacobi's method with overloaded versions of the vector addition operator, `+`, and multiplication operator, `*`, for matrices and vectors, and in just a few lines:

```
while ( (Rnorm > TOL) && (count<max_its))
{
  count++;
  xk1 = Dinv*(b+T*xk); // set x = inverse(D)*(b+T*x)
  R = A*xk-b;
  Rnorm = R.norm();
  xk = xk1;
}
```

Of course, this depends on already having defined the matrices `T` and `Dinv` first.

Write a C++ program that has a function which implements and test's Jacobi's method, based on the `Vector` and `Matrix` classes from Week 10. If working from code you wrote for Lab 7, here are some changes you need to make.

- Include the `Vector10.h` and `Matrix10.h` header files. Add `Vector10.cpp` and `Matrix10.cpp` to the project.
- In the `main()` function, define `A` as a `Matrix` object, and b, x and xk as `Vector` objects.
- Those classes take care of DMA in their constructors, so we don't need `new` or `delete` operations in the `main()` function.
- We haven't yet learned to overload the subscript operator, so, instead of, say, `A[0][0]=9` we will need `A.setij(0,0,9)`. To minimise typing, you might prefer to initialise `A`, `b`, `x` and `xk` in loops.
- Rewrite the `Jacobi` function so that it takes and returns `Matrix` and `Vector` arguments, as appropriate. Note that we no longer have to pass `N` as an argument, since we can use the `size()` method to determine its value for any `Matrix` or `Vector()` object.
  Here is a reasonable function prototype:

Assignment: Part 1

```
Vector Jacobi(Matrix A, Vector b, Vector xk,
    unsigned &count, unsigned max_its, double TOL);
```

- We no longer need the `norm(double *x, unsigned N)` function any more, since, for any vector, `x`, we can call `x.norm()`
- Similarly, we don't need to `diff()` function any more: we can set `d=x-y`, and then call `d.norm()`.
- And we don't need the `print_vec()` function any more.

Triangular systems

Some systems of equations are very easier to solve than others. Suppose the system is $L\mathbf{x} = \mathbf{b}$, but L is a lower triangular matrix. The associated system of equations looks like this:

$$\begin{aligned}
l_{11}x_1 \qquad\qquad\qquad\qquad &= b_1 \\
l_{21}x_1 + l_{22}x_2 \qquad\qquad &= b_2 \\
l_{31}x_1 + l_{32}x_2 + l_{33}x_3 \qquad &= b_2 \\
&\vdots \\
l_{N1}x_1 + l_{N2}x_2 + \cdots + l_{NN}x_N &= b_N.
\end{aligned}$$

To solve this,

- first set $x_1 = b_1/l_{11}$.
- Now substitute this into the second equation to get $x_2 = (b_2 - l_{21}x_1)/l_{22}$.
- Next we use $x_3 = (b_3 - l_{31}x_1 - l_{32}x_2)/l_{33}$,
- and so on.

In fact, this is quite like Jacobi's method, except we don't have to iterate.

Assignment: Part 2

Since we write `L*x= b`, it is reasonable to write `x=b/L`.

Exercise

Overload the "/" operator so that, if L is **lower** *triangular, then x is computed as outlined above.*

We will make this operator a `friend` of the `matrix` class, meaning that it is not a member of the class, but is "known" to it.

Modify the `Matrix10.h` header file to include the following function prototype in the class definition:

```
friend vector operator/(vector u, matrix L);
```

Note that we are explicitly passing both arguments.

Assignment: Part 2

Then, in the `Matrix10.cpp` file, add the code for the operator function. The first line might be

```
vector operator/(vector b, matrix L){
   int N = L.size();
   vector x(N); // x solves L*x=b
.
. // you add the rest of the code here
.
}
```

Important: the `friend` keyword appears only in the function prototype, and not in the function definition itself.

Gauss-Seidel, again

Recall that the **Gauss-Seidel method** is choose $\mathbf{x}^{(0)}$ and set

$$x_1^{(k+1)} = \frac{1}{a_{11}}(b_1 - a_{12}x_2^{(k+1)} - a_{13}x_3^{(k)} - \cdots - a_{1N}x_N^{(k)})$$

$$x_2^{(k+1)} = \frac{1}{a_{22}}(b_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k+1)} - \cdots - a_{2N}x_N^{(k)})$$

$$\vdots$$

$$x_N^{(k+1)} = \frac{1}{a_{NN}}(b_N - a_{N1}x_1^{(k+1)} - \cdots - a_{N,N-1}x_{N-1}^{(k+1)})$$

In the same way as we did for Jacobi's method, we can write this in a succinct matrix-vector form: we set L and U to be the matrices

$$l_{ij} = \begin{cases} a_{ij} & i \geq j \\ 0 & \text{otherwise.} \end{cases} \qquad u_{ij} = \begin{cases} 0 & i \geq j \\ -a_{ij} & \text{otherwise.} \end{cases}$$

So $A = L - U$. Then the *Gauss-Seidel method* can be written as

$$Lx^{(k+1)} = b + Ux^{(k)}. \tag{2}$$

Note that this involves solving a linear system where L is the coefficient matrix. However, we have overloaded the "/" operator to do just that.

Assignment: Part 3

1. Implement the Gauss-Seidel method using your overloaded "/" operator;
2. Write a program that uses both the Jacobi and Gauss-Seidel methods to solve the same linear system;
3. Verify that the Gauss-Seidel method is more efficient (assuming they both converge);

About this lab assignment

This is the final lab assignment for CS319. It is a little different from the previous ones. Although, like the others, it contributes 10% to your final grade, getting all 10 marks requires demonstrating a range of skills developed in the module.

Here are some ways you can demonstrate your skills in C++ programming.

- The code you have developed so far solve the 3×3 linear system from Lab 7. However, the methods presented should work for any $N \times N$ linear system, providing that the system matrix is strictly diagonally dominant. That means
$$|a_{ii}| \geq \sum_{j=1, j \neq i}^{N} |a_{ij}|.$$
Consider developing your code so that the user chooses N, and then a suitable linear system is constructed (and solved) for that N.
- Read the data for the linear system from a file; write the solution to a file (maybe as a NumPy array).

About this lab assignment

- Add a `diag()` method to the `Matrix` class to extract the diagonal of a matrix. Overload the `+` and `-` operators the `Matrix` class. Use these to simplify the implementation of the `Jacobi()` method.