MA5233 Computational Mathematics

Homework Sheet 2

Simon Etter, 2019/2020

Recommended deadline: 29 August 2019
Final deadline: 5 September 2019, 7pm

1 LU factorisation

Along with this homework sheet comes a file `sheet2.jl` which contains three incomplete function definitions `mylu()`, `tril_solve()` and `solve()`. Complete these functions.

You are only allowed to use basic arithmetic operations for this exercise. In particular, you are not allowed to use advanced linear algebra functions like `lu()` or `\`.

Hints.

- The material presented in the lecture may not be enough for you to do this exercise. This is deliberate, and you are encouraged to find alternative references which explain the LU and triangular-solve algorithms in more detail. However, make sure the code is truly yours and do not copy & paste someone else's work.
- `sheet2.jl` also contains tests for all of the above functions. You may expect full marks for this exercise if your implementation passes these tests.
- Like most programming languages, Julia passes arrays by reference, i.e. after an assignment `b = a`, the two variables `a` and `b` will point to the same memory location and modifying one array will also modify the other. This is illustrated in the following example.

  ```
  julia> a = [1]      # Create a vector
         b = a        # Let both a and b point to same memory location
         b[1] = 2     # Modify b
         println(a)   # This also modifies a!
  [2]
  ```

  This pass-by-reference behaviour may lead to subtle bugs in your code. For example in `mylu()`, you may want to initialise the `U` factor with `A` and then gradually overwrite `U` as the factorisation proceeds. If you initialise `U` using `U = A`, overwriting parts of `U` will also overwrite `A` which may cause the tests to fail. This can be avoided by writing `U = copy(A)` instead.
- `zeros(n1,n2,...)` initialises an `n1 x n2 x ...` array of all zeros.
- In Julia, `I` represents the identity matrix of any arbitrary size. Example:

```
julia> ones(3,3) + I
3×3 Array{Float64,2}:
 2.0  1.0  1.0
 1.0  2.0  1.0
 1.0  1.0  2.0
```

However, `I` is not mutable, i.e. writing e.g. `I[2,1] = 1` will result in an error. Use `Matrix{Float64}(I,(n,n))` to convert `I` to a mutable `n x n` identity matrix.

- The `rand()` function used in the tests produces pseudo-random numbers: calling `rand()` repeatedly produces a *deterministic* sequence of number which satisfies many of the statistical properties of a truly random sequence. `Random.seed!(42)` resets this sequence to a fixed initial state:

```
julia> Random.seed!(42);

julia> rand()
0.5331830160438613

julia> rand()
0.4540291355871424

julia> Random.seed!(42);

julia> rand()
0.5331830160438613
```

Resetting the random number generator in tests is good practice since it makes the tests reproducible.

2 Accuracy and performance assessment

This task compares the accuracy and performance of solving linear systems $Ax = b$ by means of the following three factorisations.

- The unpivoted `mylu()` factorisation implemented in Exercise 1.
- The pivoted `lu()` factorisation provided by the `LinearAlgebra` package.
- The `qr()` factorisation provided by the `LinearAlgebra` package.

More precisely, you are asked for the following.

1. Report the relative inf-norm errors in the solutions x computed via the three factorisations mentioned above for the following parameters.
   - Matrices: `rand(n,n)` and `wilkinson_matrix(n)` (provided in `sheet2.jl`).
   - Exact solution: `x = rand(n)`.
   - Right-hand side: `b = A*x`.

- Problem size: $n = 100$.

Note that you are asked for the errors for two different matrices A, i.e. overall you should report $2 \times 3 = 6$ numbers.

2. Relate the results of Task 1 to our discussion in class.
3. Report the runtimes of the three factorisations for problem sizes $n = 100$ and a larger n such that `lu()` takes about 0.1 seconds on your machine.
4. Comment on the findings of Tasks 1 and 3. Which algorithm would you recommend for solving general dense linear systems?

Hints.

- In order to make the `lu()` and `qr()` functions accessible in your code, you have to load the `LinearAlgebra` package by writing `using LinearAlgebra`.

```
julia> # Before you load the LinearAlgebra package:
       lu(A);
ERROR: UndefVarError: lu not defined [...]

julia> # Now load the LinearAlgebra package
       using LinearAlgebra

julia> lu(A);  # No error this time
```

- Have a look at the two macros `@time`, `@elapsed`.

Solution.

1. We observe the following relative errors in x:

```
Random matrix:
mylu: 3.0378228066045873e-12
  lu: 4.2377953447692144e-15
  qr: 9.052567520516398e-15

Wilkinson matrix:
mylu: 0.1802227655890217
  lu: 0.19914840146142077
  qr: 9.952435696324384e-15
```

2. *Random matrix.* As expected, the pivoted `lu()` and `qr()` factorisations reach about the same accuracy while the unpivoted `mylu()` has a significantly larger error.

Wilkinson matrix. Only `qr()` reaches an acceptable accuracy. The large errors of both `mylu()` and `lu()` may be explained by taking a closer look at the LU factorisation e.g. for $n = 5$:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 \\ -1 & -1 & -1 & 1 & 0 \\ -1 & -1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 4 \\ 0 & 0 & 0 & 1 & 8 \\ 0 & 0 & 0 & 0 & 16 \end{pmatrix}.$$

We see that the entries in the last column of U are given by $U(i,n) = 2^i$, which implies that $\|U\| \approx 2^n$ for all norms.[1] We recall from the lecture that the error in the solution to $Ax = b$ computed via LU factorisation is small relative to $\|U\|$, but since $\|U\|$ is very large this error may not be small at all.

3. We observe the following runtimes.

```
100 x 100 system:
mylu: 0.000402833 sec
  lu: 0.000989124 sec
  qr: 0.00461598 sec

2000 x 2000 system:
mylu: 20.224803007 sec
  lu: 0.079908177 sec
  qr: 0.701310025 sec
```

4. The pivoted LU factorisation is considered better than the QR factorisation for the following reasons.

   - The accuracy of the LU factorisation is typically comparable to that of the QR factorisation. As mentioned in class, counterexamples like Wilkinson's matrix for which the LU factorisation performs badly are exceedingly rare in practice. In particular, we have seen in Task 1 that the LU factorisation is as accurate as the QR factorisation for random matrices.
   - The runtimes given in Task 3 indicate that the LU factorisation runs a factor 5-10 faster than the QR factorisation.

   It is clearly a very bad idea to use our own, unpivoted LU factorisation: its results are significantly less accurate than those of the pivoted LU factorisation, and for large matrices it is slower than `lu()` by a factor of over 200.

[1] More precisely, $\|U\|_\infty = 2^n$. Estimates for the other norms follow by norm equivalence.