Wrocław University of Science and Technology

Scientific Calculations - Task List 5

Linear Equations, Sparse Matrix

Marcel Jerzyk

January 6, 2020

1 Large Chemical Research Company and its problem

1.1 Problem

The task at hand is to solve a system of linear equations

$$Ax = b$$

for a coefficient matrix $A \in \mathbb{R}^{n \times n}$ and right sides vector $b \in \mathbb{R}^n$, $n \geq 4$. A is a *sparse, block matrix* with structure:

$$A = \begin{bmatrix} A_1 & C_1 & 0 & 0 & 0 & \dots & 0 \\ B_2 & A_2 & C_2 & 0 & 0 & \dots & 0 \\ 0 & B_3 & A_3 & C_3 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & B_{v-2} & A_{v-2} & C_{v-2} & 0 \\ 0 & \dots & 0 & 0 & B_{v-1} & A_{v-1} & C_{v-1} \\ 0 & \dots & 0 & 0 & 0 & B_v & A_v \end{bmatrix}$$

where $v = \frac{n}{l} \wedge l|n$ and $l \geq 2$ is the size of all square matrices inside inner blocks A_k, B_k, C_k.

- 0 is a zero square matrix of degree l,
- $A_k \in \mathbb{R}^{l \times l}$, $k = 1, ..., v$ is a dense matrix,
- $B_k \in \mathbb{R}^{l \times l}$, $k = 2, ..., v$ is like:

$$B_k = \begin{bmatrix} 0 & \dots & 0 & b^k_{1\,l-1} & b^k_{1\,l} \\ 0 & \dots & 0 & b^k_{2\,l-1} & b^k_{2\,l} \\ \vdots & & \vdots & \vdots & \vdots \\ 0 & \dots & 0 & b^k_{l\,l-1} & b^k_{l\,l} \end{bmatrix}$$

- $C_k \in \mathbb{R}^{l \times l}$, $k = 1, ..., v-1$ is diagonal matrix:

$$C_k = \begin{bmatrix} c^k_1 & 0 & 0 & \dots & 0 \\ 0 & c^k_2 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & c^k_{l-1} & 0 \\ 0 & \dots & 0 & 0 & c^k_l \end{bmatrix}$$

1.2 The Task

Our job is to:

1. Write a function that solves $Ax = b$ with Gauss Elimination (that takes into account specific character of matrix A) in two variants:
   (a) without choosing main element
   (b) with partial selection of the main element

2. Write a function determining the LU distribution of matrix A using Gauss Elimination (by again taking into account how our A looks) in two variants:
   (a) without choosing main element
   (b) with partial selection of the main element

   and LU should be effectively remembered.

3. Write a function that solves $Ax = b$ after determining LU from 2.

2 Solution

To solve given linear equations system we need to optimize classic algorithms that solves such tasks to work best with the specific structure of the given matrix. Matrix A is a sparse matrix and using Gauss Elimination and *LU* would take $\mathbb{O}(n^3)$ but after optimization we would like to reduce it to $\mathbb{O}(n)$.

2.1 Gauss Elimination Method

We are going to use Gauss Elimination Method to reduce unknowns from the linear equations system and to transform it to equivalent and simpler system - into *upper triangular matrix.*

These unknowns are going to be reduced in elimination step, in which at k step we calculate *Gauss-Elimination multipliers*

$$l_{ik} = \frac{A_{ik}^{(k)}}{A_{kk}^{(k)}}$$

where matrix $A^{(k)}$ is matrix A at k elimination step (for $i = k + 1, ..., n$). We use this multiplier to zero out a_{ik} by multiplying it after making subtraction of k row from i row. Simpler we can think of it as just calculating first unknown from first equation and then substituting it with for others in the system. Example:

$$\begin{gathered} x + 2y = 3 \\ 2x - 2y = 3 \\ \downarrow \\ 2(3 - 2y) - 2y = 3 \end{gathered}$$

Worth noting that this will not work if there will be 0 at k step at the diagonal. In that case we need to swap places in rows (or columns) and the right vector b to avoid in that diagonal situation like $a_{kk}^{(k)} = 0$. After the elimination is completed we need to use *back substitution algorithm*:

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

where rows $i = n, n-1, ..., 1$. The algorithm is $\mathbb{O}(n^3)$.

2.2 Gauss Elimination Method with Pivot

Gauss Elimination with Pivot is modification to previously mentioned *Gauss Elimination* thanks to which occasional "0" on the main diagonal in the matrix won't be longer a problem. First we search for largest element (furthest from 0 - absolute value) in a column and swapping rows so that these elements will be placed in a certain way:

$$|a_{kk}| = |a_{perm(k),k}| = max[|a_{ik}| : i = k, ..., n]$$

where $perm(k)$ is permutation vector in where we store order of swaps. In other words at k step, in k matrix column we take a row with absolute biggest value and we swap it with k row. This assures that on the main diagonal all elements won't be a 0. There's a small catch though - there's a possibility of a large number of potential swaps. We can narrow area on which we iterate by limiting it to the last row and column in which exist a non-zero value. That is:

$$lastColumn(row) = min[row + l, n]$$

$$lastRow(column) = min[n, l + l * \lfloor \frac{column + 1}{l} \rfloor]$$

where $i = 1, 2, ..., n$ is an iteration. The optimization works because of the structure of matrix that we are dealing with - *Gauss Elimination Method* in it's core zeroes out all values under the main diagonal, which is pointless in our example. The lastColumn and lastRow restrictions comes from simple regularity - (keep in mind that $l \geq 2$, which is the size of inner matrices). In general after first $l - 2$ columns (where only first l rows have non-zero value) next l columns have elements in first xl rows have non-zero values for $x \in (1, blocksamount)$, where $blockamount$ is just $\frac{n}{l}$ (size of matrix A divided on size of inner matrices). In the

last column n, last l rows with non-zero elements belong to the square matrix of the inner block A_v (because last block of A starts at $n - l$ column and $n - l$ row).
Summing it all up, we get $n * l$ (because of the block at the edge of the matrix) plus $(n-1) * l^2$ iterations giving $\mathbb{O}(n)$.

2.3 LU Distribution

LU distribution is also closely linked to *Gauss Elimination*. *LU* are two triangular matrices - **L**ower triangular matrix and **U**pper triangular matrix. As for the task - we can write our A in $Ax = b$ as a product of **L** and **U** matrices - $L * Ux = b$. We do that by transforming matrix A into upper triangular matrix U and remembering multipliers $z_{ij} = \frac{a_{ij}}{a_{jj}}$, where i are rows and j columns of matrix L. With this we can create lower triangular matrix L. So now we can solve it with this linear equation

$$L * b' = b$$
$$U * x = b'$$

As for **U**pper matrix we can reuse previously implemented *back substitution algorithm* and for the **L**ower (new but similar) *forward substitution algorithm*:

$$x_i = \frac{b_i - \sum_{j=i+1}^{i-1} l_{ij} x_j}{l_{ii}}$$

where rows $i = 1, 2, ..., n$. Solving linear equations with this algorithm takes $\mathbb{O}(n^2)$ and determining *LU distribution* takes $\mathbb{O}(n^3)$, but...

... we can determine the *LU distribution* for matrix A with $\mathbb{O}(n)$ for constant l because we can use *Gauss Elimination Method* with a modifications:

- instead of zeroing out a_{ij} - give it them the previously mentioned multiplier value $z_{ij} = \frac{a_{ij}}{a_{jj}}$,
- save permutation vector in *Gauss Elimination Method with pivot* to solve the $Ax = b$ later on.

3 Results

After assuring that everything is correctly implemented with a bunch of test (that we will get $x = (1, 1, ..., 1)^T$ and that b is correctly determined) we are ready to perform the test for $n \times n$ matrices with block size $l = 4$ on all four implemented algorithms.

n	Gauss		LU	
	Optimized	Pivoted	Optimized	Pivoted
16	1.2e-5	1.9e-5	1.2e-5	1.6e-5
1000	0.001515	0.001932	0.001542	0.001899
2500	0.008042	0.0092	0.008239	0.009025
5000	0.030319	0.032711	0.030747	0.032327
7500	0.067175	0.070991	0.069084	0.070217
10000	0.134709	0.142636	0.136977	0.142198
25000	1.068279	1.086184	1.084745	1.086767
50000	4.386979	4.42937	4.365844	4.417811
75000	9.821247	9.81228	9.787487	10.032837
100000	17.38349	17.693363	17.487092	17.636127

Table 1: Time Comparison (s)

Table 1 shows that there is no meaningful time difference between all functions, with the biggest difference of $\pm 0.31s$ in the $n = 100000$.

n	Gauss		LU	
	Optimized	Pivoted	Optimized	Pivoted
16	0.002	0.005	0.007	0.007
1000	0.35	0.214	0.305	0.313
2500	0.248	0.534	0.763	0.782
5000	0.496	1.068	1.526	1.564
7500	0.744	1.602	2.289	2.346
10000	0.992	2.136	3.052	3.128
25000	2.48	5.341	7.63	7.82
50000	4.959	10.681	15.259	15.64
75000	7.439	16.022	22.888	23.46
100000	9.918	21.362	30.518	31.281

Table 2: Memory Comparison (MB)

Now, in *Table 2*, we see variations in the results - *Gauss* takes less memory than the method with two matrices, even while using pivot. In the *LU* adding pivot doesn't impact the memory as much as in *Gauss*, where it adds about double the memory usage.

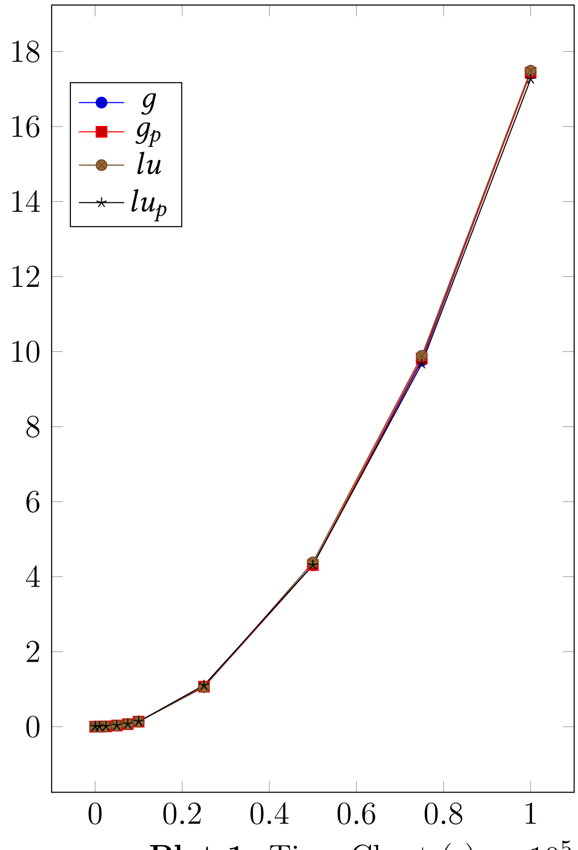
While we are at it - this is how memory/time for Julia based optimization and to unoptimized $x = A\backslash b$ look like.

n	Julia Optimization		No Optimization	
	Time	Memory	Time	Memory
16	6.2e-5	0.04	0.001138	0.003
1000	0.000984	1.523	0.017216	7.645
2500	0.002684	3.779	0.110942	47.722
5000	0.005994	7.54	0.565814	190.811
7500	0.010503	11.302	1.681591	429.268
10000	0.045574	15.063	3.747541	763.092
25000	0.037366	37.631	53.360656	4768.753
50000	0.077359	75.244	-	OutOfMemory
75000	0.133018	112.857	-	OutOfMemory
100000	0.169378	150.47	-	OutOfMemory

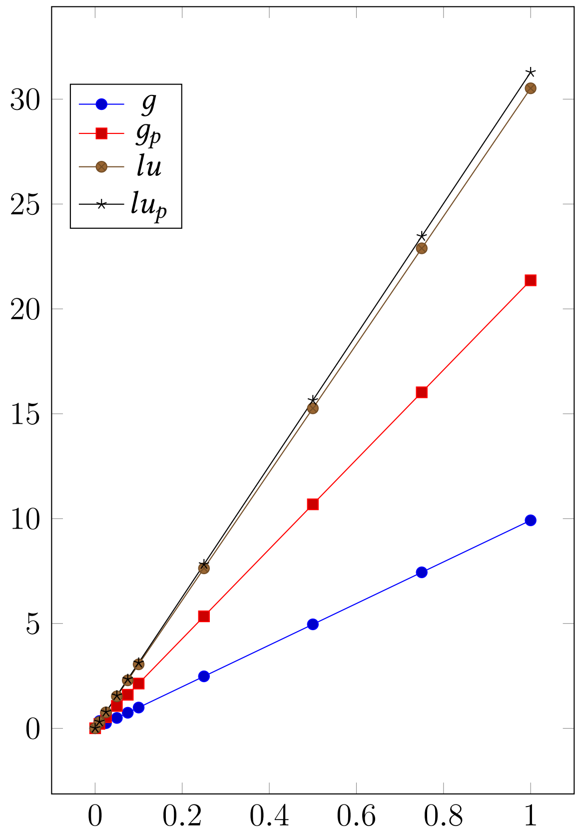
Table 3: Comparison of julia opt. and no at all

Our algorithms performs compared to not optimized problem solution for a sparse Matrix (in *Table 3*) are very much favored in case of time efficiency, memory usage and even in completing the task at all - on a 25000×25000 matrix it already used above $4GB$ of RAM with $\approx 53s$ to complete and failed to complete next iterations for bigger matrix size. The build-in implementation for *SparseArrays* in Julia crushes the task with fractions of a second to complete, but with memory about $\approx$ 7 times more memory usage on average compared to our *Gauss Method w/o choosing pivot.*

The plots on the right side of this page (notice the 10^5 x scale) further assures us that the only difference between the methods is memory wise. In the time chart the curve is stacked with all functions on each other, because the time difference is not noticeable.



Plot 1: Time Chart (s)



Plot 2: Memory Chart (MB)

4 Conclusions

There are few, so for readability let's list them:

- Optimization is necessary to complete the task on large matrices (bigger than value $n \approx [25000 - 50000]$ and beyond),
- Optimization have enormous impact on both time efficiency and memory usage, with our optimization by limiting the amount if iterations we reduce the computational complexity from $\mathbb{O}(n^3)$ to $\mathbb{O}(n)$
  - On the graph we can see that the time is squared - the impact comes from referencing elements in the *SparseArrays* structure
- Proposed optimizations (back/forward substitution algorithm and pivot) for *Gauss* and *LU matrices* make them equally good in terms of time efficiency,
- *Gauss* without pivot takes the least memory as it doesn't need to remember *perm* nor additional matrices like in other cases (double for pivot, triple for *LU*).
- *LU* with and without pivot takes the same amount of memory

5 Implementations

5.1 User Interface

Program has implemented a simple user interface with commands to satisfy the task requirements:

- "*test*" - to run the tests for all algorithms - *gauss*, *lu* and *b determination* - with the given test data (they are compared to Julia implemented answer) with atol (error limit) equal to 0,
- "*b [size]*" - to determine "b" for matrix "A" and saving it to "b.txt",
- "*compare [all / gauss / lu]*" - to run comparison test of functions
- "*calculate [gauss / lu] [pivoted / not] [calc b / retrieve b] [size]*" - to save calculation result with given options to file

5.2 Managing Functions

They are called to run calculating functions and parse results further to get the result. They were helpful to not stack duplicated code and pass on *perm* or *newmatrices* into arguments. For example, to solve LU with pivot, we use *gemPivoted* algorithm with LU set to true, retrieve both *perm* and new A, put them into *triangularUpper* to perform forward substitution and get new b and put it into *triangularLower* to get x after last backward substitution algorithm.

5.3 Algorithms

Algorithm implementations are on the next page. There, in gem for example, is a special boolean LU that determines if we should zero out the values or put in the $z = \frac{a_{ij}}{a_{jj}}$ multipliers. *Perm* is the permutation vector - at if it determines if we are using it or not to proceed further. Not always all return values are used - it depends on the algorithm we had chosen.

Code 1: gaussEliminationMethod(A, b, n, l, LU)

```
for i <- 1 to n-1 do
  lastC <- min(i+l, n)
  lastR <- min(l + l*floor(i+1/l), n)

  for k <- i+1 to lastR do
    if A[i][i] = 0
      error "Diag Coef = 0"
    end

    z <- A[i][k]/A[i][i]

    if !LU A[i][k] <- 0
    if  LU A[i][k] <- z

    for c <- k+1 to lastC do
     A[k][c] <- A[k][c] - z * A[i][c]
    end

    if !LU b[k] <- b[k] - z*b[i]
  end
end
return A
```

Code 2: triangularUpper(A, b, n, l, perm, pLU)

```
test
for k <- n downto 1
  sum <- 0
  if !perm
    last <- min(k + l, n)
    for c <- k+1 to last
      sum <- sum + A[c, k] * x[c]
    end
    x[k] <- (b[k] - sum) / A[k, k]
  end

  if perm
    last <-
     min(2*l + l*floor(perm[k]+1 /
                          l), n)
    for c <- k+1 to last
     sum = sum + A[c, perm[k]] * x[c]
    end
    if !pLU then
     x[k] <- (b[perm[k]] - sum) /
              A[k, perm[k]]
    end
    if  pLU then
     x[k] <- (b[k] - sum) /
              A[k, perm[k]]
    end
  end
end
return x
```

Code 3: gemPivoted(A, b, n, l, LU)

```
perm <- {1, ..., n}
for p to n-1 do
  lastC <-
    min(l + l*floor(p+1 / l), n);
  lastR <-
    min(2*l + l*floor(p+1 / l), n);


  for k <- p+1 to lastC
    maxIndex <- p
    maxVal <- abs(A[p, perm[p]]

    for i <- k to lastC
      absVal <- abs(A[p, perm[i]])
      if absVal > maxVal then
        maxIndex <- i
        maxVal <- absVal
      end
    end

    if maxVal = 0 then
      error "Zero in Col"
    end

    swap(perm[p], perm[maxIndex])

    if  LU then A[p, perm[k]] <- z
    if !LU then A[p, perm[k]] <- 0

    for c <- p+1 to lastR
      A[c, perm[k]] -= z *
                 A[c, perm[p]]
    end

    if !LU then
      b[perm[k]] -= - z * b[perm[p]]
    end
  end
end
return perm, A
```

Code 4: triangularLower(A, b, n, l, perm)

```
for k <- 1 to n
  sum <- 0
  if !perm
    last <- max((floor(k-1 /
                 l) * l) - 1, 1)
    for c <- last to k-1
      sum = sum + A[c, k] * x[c]
    end
    x[k] <- b[k] - sum
  end

 if perm
   last <- max(l * floor(perm[k]-1 /
                   l) - 1, 1)
   for c <- last to k-1
     sum = sum + A[c, perm[k]] * x[c]
   end
   x[k] = b[perm[k]] - sum)
 end
end
return x
```