Numerical Linear Algebra for Computational Sciences and Information Engineering
CITHN2006

Direct Methods for Dense Linear Systems

by Nicolas Venkovic

Chair of Computational Mathematics
School of Computation, Information and Technology (CIT)
Technical University of Munich, Germany

1. *Write a function called* ***RowMajorForwardSubstitution*** *that implements forward substitution as described in slide #3.*

As we saw, to solve a lower triangular system $Lx = b$, we start by $x_1 = b_1/l_{11}$, which we then substitute in the expression to solve for x_2, and we keep moving **forward** like this. This yields the expression

$$x_i := \left(b_i - \sum_{j=1}^{i-1} l_{ij} x_j \right) / l_{ii} \quad \text{for} \quad i = 2, \dots, n$$

which is coded as follows:

[1]:
```
using LinearAlgebra, Random, Printf
```

[2]:
```
# Forward substitution with row-wise access to L
function RowMajorForwardSubstitution(L, b) # ~ trtrsRow(L, b), p. 67 in Darve␣
 ↪and Wootters (2021)
  n = length(b)
  x = Vector{Float64}(undef, n)
  for i = 1:n
    z = 0.
    for j = 1:i-1
      z += L[i, j] * x[j]
    end
    x[i] = (b[i] - z) / L[i, i]
  end
  return x
end;
```

One thing we notice in this implementation is that, the data from L is fetched **row-wise**, i.e., for a given index value i, the summation first uses l_{i1}, then l_{i2}, and then l_{i3}, and so on, until $l_{i,i-1}$.

However **Julia matrices** are stored in a **column-major** format, so that this pattern of data access results in multiple cache misses.

2. *Write a function called* `ColumnMajorForwardSubstitution` *that implements forward substitution fetching components of the lower triangular matrix in a column-wise fashion.*

Instead of following a row-wise data access, we prefer to compute x_i by streaming through the components of L in a **column-wise** fashion. To see how to do that, let us expose how the components of x are formed:

$$
\begin{aligned}
x_1 &:= b_1/l_{11} \\
x_2 &:= \left(b_2 - l_{21}x_1\right)/l_{11} \\
x_3 &:= \left(b_3 - l_{31}x_1 - l_{32}x_2\right)/l_{33} \\
x_4 &:= \left(b_4 - l_{41}x_1 - l_{42}x_2 - l_{43}x_3\right)/l_{44} \\
&\vdots \\
x_n &:= \left(b_n - l_{n1}x_1 - l_{n2}x_2 - l_{n3}x_3 - \cdots - l_{n,n-1}x_{n-1}\right)/l_{nn}.
\end{aligned}
$$

We can see that, once we are done evaluating x_i, the partial contributions $l_{i+1,i}x_i$, $l_{i+2,i}x_i$, ..., $l_{ni}x_i$ can all be added one after the other to x_{i+1}, $x_{i+2}, \ldots, x_n$, respectivelly. When doing so, the requirement is that x_{i+1} is fully evaluated before adding its the partial contributions to the subsequent component x_{i+1}, $x_{i+2}, \ldots, x_n$. Thus, the calculation can be re-ordered as follows:

$$
\begin{aligned}
&0. \quad x_1 := b_1,\ x_2 := b_2,\ \ldots,\ x_n = b_n \\
&1. \quad x_1 := x_1/l_{11} \\
&2. \quad x_2 := x_2 - l_{21}x_1,\ x_3 := x_3 - l_{31}x_1,\ x_4 := x_4 - l_{41}x_1,\ \ldots,\ x_n := x_n - l_{n1}x_1 \\
&3. \quad x_2 := x_2/l_{22} \\
&4. \quad x_3 := x_3 - l_{32}x_2,\ x_4 := x_4 - l_{42}x_2,\ \ldots,\ x_n := x_n - l_{n2}x_2 \\
&5. \quad x_3 := x_3/l_{33} \\
&6. \quad x_4 := x_4 - l_{43}x_3,\ \ldots,\ x_n := x_n - l_{n3}x_3 \\
&\quad\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\
&2n-2. \quad x_{n-1} := x_{n-1}/l_{n-1,n-1} \\
&2n-1. \quad x_n := x_n - l_{n,n-1}x_{n-1} \\
&2n. \quad x_n := x_n/l_{nn}
\end{aligned}
$$

where, clearly, the components of L are now accessed row-wise. This is coded as follows:

[3]:
```
# Forward substitution with column-wise access to L
function ColumnMajorForwardSubstitution(L, b) # ~ trtrs(L, b), p. 67 in Darve
 ↪and Wootters (2021)
  n = length(b)
  x = copy(b)
  for j = 1:n
```

```
    x[j] = x[j] / L[j, j]
    for i = j+1:n
      x[i] -= L[i, j] * x[j]
    end
  end
  return x
end;
```

3. *Compare the runtime of* `RowMajorForwardSubstitution` *and* `ColumnMajorForwardSubstitution` *for different matrix sizes.*

Let us now introduce a lower triangular matrix and measure the difference in runtime between the two approaches.

[4]:
```
function get_L(n)
  L = zeros(n, n)
  for j=1:n
    L[j,j] = 1.
    L[j+1:n,j] = rand(-2:2, n-j)
  end
  return L
end

Random.seed!(123467);
for n in (2_000, 20_000)
  L = get_L(n)
  x_exact = rand(0:9, n)
  b = L * x_exact
  dt1 = @elapsed x1 = RowMajorForwardSubstitution(L, b)
  dt2 = @elapsed x2 = ColumnMajorForwardSubstitution(L, b)
  @printf "n = %d\n" n
  @printf "Row-wise access: dt = %.2E, ||x - x_exact||_2 = %E\n" dt1 norm(x1 -␣
 ↪x_exact)
  @printf " Cache-friendly: dt = %.2E, ||x - x_exact||_2 = %E\n" dt2 norm(x2 -␣
 ↪x_exact)
end
```

```
n = 2000
Row-wise access: dt = 6.47E-03, ||x - x_exact||_2 = 0.000000E+00
 Cache-friendly: dt = 3.15E-03, ||x - x_exact||_2 = 0.000000E+00
n = 20000
Row-wise access: dt = 2.19E+00, ||x - x_exact||_2 = 0.000000E+00
 Cache-friendly: dt = 2.40E-01, ||x - x_exact||_2 = 0.000000E+00
```

4. *Write a function called* `get_LU` *that returns the L and U factors of a matrix A obtained by forward elimination without pivoting. Test your code by solving a linear system using the L and U factors.*

For a given matrix A with components $a_{ij} =: a_{ij}^{(0)}$, we saw that **forward elimination without**

pivoting goes a follows:

$$
\begin{aligned}
&\text{for } \; k = 1, \dots, n-1 \; \text{// Loop over Gauss transformations } G_1, \dots, G_{n-1} \\
&\quad \text{// Compute } A^{(k)} = G_k A^{(k-1)} \\
&\quad a_{ij}^{(k)} := a_{ij}^{(k-1)} \; \text{ for } \; i = 1, \dots, k \; \text{ and } \; j = i, \dots, n \\
&\quad a_{ij}^{(k)} := 0 \; \text{ for } \; j = 1, \dots, k \; \text{ and } \; i = j+1, \dots, n \\
&\quad \text{for } \; i = k+1, \dots, n \; \text{// Loop over rows acted on by } G_k \\
&\qquad m_i^{(k)} := a_{ik}^{(k-1)} / a_{kk}^{(k-1)} \\
&\qquad \text{for } \; j = k+1, \dots, n \; \text{// Loop over columns acted on by } G_k \\
&\qquad\quad a_{ij}^{(k)} := a_{ij}^{(k-1)} - m_i^{(k)} a_{kj}^{(k-1)}
\end{aligned}
$$

We also saw that, **if no breakdown happens**, we obtain the upper triangular matrix $U = G_{n-1} \cdots G_1 A =: A^{(n-1)}$. Moreover, the lower triangular matrix $L := G_1^{-1} \cdots G_{n-1}^{-1}$ is a by-product of the procedure, i.e., we have

$$
L = \begin{bmatrix} 1 & & & \\ m_2^{(1)} & \ddots & & \\ \vdots & & 1 & \\ m_n^{(1)} & \cdots & m_n^{(n-1)} & 1 \end{bmatrix}
$$

such that $LU = A$. A first implementation to obtain the LU factors of A by forward elimination is as follows:

[5]:
```
# "outer-product" implementation of forward elimination with row-wise data␣
 ↪access
function get_LU(A) # ~ getrfOuter!(A), p. 74 in Darve and Wootters (2021)
  n, _ = size(A)
  L = zeros(n, n); L[diagind(L)] .= 1.
  U = copy(A)
  for k = 1:n-1
    for i = k+1:n
      m = U[i, k] / U[k, k]
      for j = k+1:n
        U[i, j] -= m * U[k, j]
      end
      L[i, k] = m
    end
    U[k+1:n, k] .= 0.
  end
  return L, U
end;
```

which we can test with a matrix as follows:

[6]:
```
function get_A(n)
  A = zeros(n, n)
  for j=1:n
    A[:,j] = rand(-2:2, n)
    A[j,j] = 1.
  end
  return A
end;

Random.seed!(123467)

for n in (10, 100, 1_000)
  A = get_A(n)
  L, U = get_LU(A)
  println(maximum(abs.(L * U - A)))
end
```

```
9.82927766795898e-15
1.8189894035458565e-12
5.0391690820106305e-11
```

We also saw that $Ax = b$ can be solved in two triangular solves, i.e.,

1. Solve for z such that $Lz = b$,
2. Solve for x such that $Ux = z$.

[7]:
```
Random.seed!(1)

for n in (10, 100, 1_000)
  A = get_A(n)
  x_exact = rand(0:9, n)
  b = A * x_exact
  L, U = get_LU(A)
  z = LowerTriangular(L) \ b # Forward substitution
  x = UpperTriangular(U) \ z # Backward substitution
  @printf "n = %d, ||x - x_exact||_2/||x_exact||_2 = %E\n" n norm(x - x_exact) /
  ↪ norm(x_exact)
end
```

```
n = 10, ||x - x_exact||_2/||x_exact||_2 = 5.237667E-15
n = 100, ||x - x_exact||_2/||x_exact||_2 = 1.322292E-11
n = 1000, ||x - x_exact||_2/||x_exact||_2 = 1.595005E-11
```

5. *Write a function called* ***RowMajor_LU_InPlace!*** *that computes the L and U factors of A, in-place, by forward elimination without pivoting. Verify your code.*

Clearly, U can be computed in-place, i.e., the components of $A^{(1)}, \ldots, A^{(n-2)}, A^{(n-1)} = U$ can be stored within A from one Gauss transformation to another.

Also, for a given transformation G_k, the components $a^{(k)}_{k+1,k}, \dots, a^{(k)}_{nk}$, which are set to zero by construction, can be used to store the components of the k-th column of L below the diagonal, i.e., $m^{(k)}_{k+1}, \dots, m^{(k)}_n$.

Then, no extra memory needs to be alocated, and upon completion of the forward elimination procedure, A contains simultaneously the components of U in its upper-triangular part, and the non-trivial components of L.

This is coded as follows:

[8]:
```
# In-place "outer-product" implementation of forward elimination with row-wise data access
function RowMajor_LU_InPlace!(A) # ~ getrfOuter!(A), p. 74 in Darve and Wootters (2021)
  n, _ = size(A)
  for k = 1:n-1
    for i = k+1:n
      A[i, k] /= A[k, k]
    end
    for i = k+1:n
      for j = k+1:n
        A[i, j] -= A[i, k] * A[k, j]
      end
    end
  end
end;
```

[9]:
```
Random.seed!(12345)
n = 1_000
A = get_A(n)
A_LU = copy(A)
RowMajor_LU_InPlace!(A_LU)
U = UpperTriangular(copy(A_LU))
L = LowerTriangular(copy(A_LU)); L[diagind(L)] .= 1.
println(maximum(abs.(L * U - A)))
```

```
3.6419578464119695e-10
```

Looking at the way the non-trivial components of $A^{(k)} = G_k A^{(k-1)}$ are set, we see the inner-most loop of the initial implementation of forward elimination iterates in a row-wise fashion over components of $A^{(k-1)}$, i.e., we set

$$a^{(k)}_{ij} := a^{(k-1)}_{ij} - m^{(k)}_i a^{(k-1)}_{kj} \quad \text{by iterating over} \quad j = k+1, \dots, n$$

for $i = k+1, \dots, n$.

Therefore, for a given k such that each $1 \leq k < n$, we have

$$a^{(k)}_{ij} = \cdots = a^{(n-1)}_{ij} = u_{ij} \quad \text{for} \quad i \in \{1, \dots, k+1\} \quad \text{and} \quad j \in \{i, \dots, k+1\} \tag{1}$$

where u_{ij} denotes the components of the upper-triangular factor U. Therefore, we have

$$u_{ij} = a_{ij}^{(i-1)} \quad \text{for} \quad i \in \{1, \dots, j\}.$$

6. *Write a function called* `ColumnMajor_LU_InPlace!` *that does in-place LU factorization without pivoting, with a column-wise data access pattern.*

Instead of iterating over $G_1, \dots, G_{n-1}$ in an outer-most loop, we wish to form U column-by-column. That is, given a column j such that $1 < j \leq n$, assume the non-trivial components of the $j-1$ first columns of U are known, and we want to form u_{ij} for $i = 1, \dots, j$. Then we have

$$\begin{aligned}
u_{1j} &= a_{1j}^{(0)} \\
u_{2j} &= a_{2j}^{(1)} = a_{2j}^{(0)} - m_2^{(1)} a_{1j}^{(0)} \\
u_{3j} &= a_{3j}^{(2)} = a_{3j}^{(1)} - m_3^{(2)} a_{2j}^{(1)} \\
&\vdots \\
u_{j-1,j} &= a_{j-1,j}^{(j-2)} = a_{j-1,j}^{(j-3)} - m_{j-1}^{(j-2)} a_{j-2,j}^{(j-3)} \\
u_{jj} &= a_{jj}^{(j-1)} = a_{jj}^{(j-2)} - m_j^{(j-1)} a_{j-1,j}^{(j-2)}
\end{aligned}$$

where $a_{3j}^{(1)}, \dots, a_{jj}^{(j-2)}$ are not computed yet. However, in general, for $k < j$, we can also evaluate $a_{ij}^{(k)}$ by

$$\begin{aligned}
&\text{// Forming } a_{ij}^{(k)} \text{ from } a_{ij}^{(0)}, a_{ij}^{(1)}, \dots, a_{ij}^{(k-1)} : \\
&\text{for } \; l = 1, \dots, k \\
&\quad a_{ij}^{(l)} := a_{ij}^{(l-1)} - m_i^{(l)} a_{lj}^{(l-1)}
\end{aligned}$$

Therefore, the components u_{ij} for $i = 1, \dots, j$ may be computed only using the components $a_{1j}^{(0)}, \dots, a_{jj}^{(0)}$ and

$$\begin{aligned}
& m_2^{(1)} && \text{for } \; u_{2j} \\
& m_3^{(1)}, m_3^{(2)} && \text{for } \; u_{3j} \\
& \quad \vdots \\
& m_{j-1}^{(1)}, \dots, m_{j-1}^{(j-2)} && \text{for } \; u_{j-1,j} \\
& m_j^{(1)}, \dots, m_j^{(j-2)}, m_j^{(j-1)} && \text{for } \; u_{jj}
\end{aligned}$$

which are the non-trivial components of the $j-1$ first columns of the lower-triangular factor L. While the evaluation of $m_2^{(1)}, \dots, m_n^{(1)}$ from $a_{11}^{(0)}, \dots, a_{n1}^{(0)}$ is a given, the evaluation of $m_{j+1}^{(j)}, \dots, m_n^{(j)}$ such that $1 < j < n$ is done by

$$\begin{aligned}
m_{j+1}^{(j)} &= a_{j+1,j}^{(j-1)} / a_{jj}^{(j-1)} = a_{j+1,j}^{(j-1)} / u_{jj} \\
&\vdots \\
m_n^{(j)} &= a_{n,j}^{(j-1)} / a_{jj}^{(j-1)} = a_{n,j}^{(j-1)} / u_{jj}
\end{aligned}$$

where, simimlarly,

$$\begin{aligned} a_{j+1,j}^{(j-1)} &\text{ can be formed from } a_{j+1,j}^{(0)}, \dots, a_{j+1,j}^{(j-2)} \\ &\vdots \\ a_{n,j}^{(j-1)} &\text{ can be formed from } a_{n,j}^{(0)}, \dots, a_{n,j}^{(j-2)} \end{aligned}$$

Therefore, the non-trivial components of the j-column of the lower-triangular and upper-triangular factors of A, i.e., L and U, respectively, can be formed using only data from the j-th column of A, and the components in the previously formed columns of L. This can be done with an inner-most loop which iterates in a column-wise fashion of the components of A. This is done as follows:

[10]:
```
# In-place implementation of forward elimination with column-wise data access
function ColumnMajor_LU_InPlace!(A) # ~ getrfAxpy!(A), p. 75 in Darve and␣
 ↪Wootters (2021)
  n, _ = size(A)
  for j = 1:n
    for k = 1:j-1
      for i = k+1:n
        A[i, j] -= A[i, k] * A[k, j]
      end
    end
    for i = j+1:n
      A[i, j] /= A[j, j]
    end
  end
end;
```

7. *Compare the runtime of* `RowMajor_LU_InPlace!` *and* `ColumnMajor_LU_InPlace!` *for different matrix sizes.*

The effect on runtime is as follows:

[11]:
```
Random.seed!(1)

for n in (10, 100, 1_000)
  A = get_A(n)
  rand(0:9, n);
  A_LU = copy(A)
  dt = @elapsed RowMajor_LU_InPlace!(A_LU)
  U = UpperTriangular(copy(A_LU))
  L = LowerTriangular(copy(A_LU)); L[diagind(L)] .= 1.
  @printf "n = %d\n" n
  @printf "Row-wise access: dt = %g, component-wise max error = %E\n" dt␣
 ↪maximum(abs.(L * U - A))
  A_LU = copy(A)
  dt = @elapsed ColumnMajor_LU_InPlace!(A_LU)
  U = UpperTriangular(copy(A_LU))
  L = LowerTriangular(copy(A_LU)); L[diagind(L)] .= 1.
```

```
  @printf " Cache-friendly: dt = %g, component-wise max error = %E\n" dt␣
 ↪maximum(abs.(L * U - A))
end
```

```
n = 10
Row-wise access: dt = 4.955e-06, component-wise max error = 7.105427E-15
 Cache-friendly: dt = 0.0799702, component-wise max error = 7.105427E-15
n = 100
Row-wise access: dt = 0.000521807, component-wise max error = 8.390133E-11
 Cache-friendly: dt = 0.000672226, component-wise max error = 8.390133E-11
n = 1000
Row-wise access: dt = 0.825525, component-wise max error = 7.033041E-11
 Cache-friendly: dt = 0.39375, component-wise max error = 7.033041E-11
```