Numerical Methods - Project 1

Thomas Simcox

October, 2021

1a. Cholesky Factorization Algorithm

My Cholesky Factorization algorithm can be found on Github at the following link: [insert link here].

My algorithm is written in Julia, and consists of just one function called `chol()`, which calls one auxilary function I wrote called `isPositiveDefinite()` which returns a `Bool` based on whether or not a passed in matrix is postive-definite. Below is a code snippet of my `chol()` function:

```
function chol(M::Matrix{Int64})::Union{Matrix{Float64}, Nothing}
   # if the matrix isn't positive definite, return log indicating so
   !isPositiveDefinite(M) && return println("Matrix is not positive definite.")

   # destructure dims of array into variables
   (m, n) = size(M)

   # create empty matrix consisting of floats
   U = zeros(Float64, m, n)
   j = 1

   # calculate Cholesky decomposition
   U[j, j] = sqrt(M[j, j])
   for col in eachcol(M)
      for (i, v) in enumerate(col)
         # this is just a series of if checks that determines the calculation we need to do based on
         # whether i < j, i > j, or i === j
         i > j ? U[i, j] = v / U[j, j] :
         i < j ? U[i, j] = 0 :
         i === j && i === 2 ?
         U[i, j] = sqrt(v - (U[i, j-1])^2) :
         ""
      end
      j += 1
   end
   # return our new matrix U in upper-triangular form
   z = U[1, 2]
   U[1, 2] = U[2, 1]
   U[2, 1] = z
   return U
end
```

As shown in the function's signature, I decided to take advantage of Julia's type narrowing capabilities and restricted the function's argument to be of type `MatrixInt64`. And, I made sure that the function will return a `MatrixFloat64`, or `Nothing`. I specified this union return type because if the passed in matrix is not positive definite, I return a `prinln()`, and otherwise, I return the matrix factorization `U`.

1b. Testing

Test matrix: $\begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}$

Julia REPL:

```
julia> include("chol.jl")
2×2 Matrix{Float64}:
 1.0  0.0
 0.0  2.0
```

Test matrix: $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$

Julia REPL:

```
julia> include("chol.jl")
Matrix is not positive definite.
```

Test matrix: $\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$

Julia REPL:

```
julia> include("chol.jl")
2×2 Matrix{Float64}:
 1.41421   0.707107
 0.0       1.22474
```

Test matrix: $\begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix}$

Julia REPL:

```
julia> include("chol.jl")
Matrix is not positive definite.
```

Test matrix: $\begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}$

Julia REPL:

```
julia> include("chol.jl")
Matrix is not positive definite.
```

Test matrix: $\begin{bmatrix} 5 & -1 \\ -1 & 3 \end{bmatrix}$

Julia REPL:

```
julia> include("chol.jl")
2×2 Matrix{Float64}:
  2.23607   -0.447214
  0.0       1.67332
```

1c. & 1d.

Running the commands `z.L * z.U - B` and `norm(z.L * z.U - B)` resulted in the following matrix and float, respectively:

```
julia> z.L * z.U - B
3×3 Matrix{Float64}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> norm(z.L * z.U - b)
0.0
```

Both of these results make sense, and show that the factorization was accurate. Had there been any rounding or truncating error, we would have seen it in the above calculations.

1e. Julia REPL outputs for matrices in `prog1c.dat`

(Skipping ones that are not pos-def per instructions)

I hard-coded the matrices in the above file, but here's the method I wrote for displaying the result of the `cholesky()` method for each big matrix (if it exists):

```
# container for big matrices in prog1.dat
bigMatrices = [D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8]

# if matrix in bigMatrices has a cholesky decomp (i.e. doesn't throw), display it
map(x -> try display(cholesky(x)) catch e println("Matrix not pos-def.") end, bigMatrices);
```

And here's the resulting REPL output:

```
julia> include("chol.jl")
Cholesky{Float64, Matrix{Float64}}
U factor:
3×3 UpperTriangular{Float64, Matrix{Float64}}:
 3.74166  8.55236  14.1648
  .       1.96396   3.49149
  .        .        0.408248

Matrix not pos-def.

Cholesky{Float64, Matrix{Float64}}
U factor:
4×4 UpperTriangular{Float64, Matrix{Float64}}:
 1.0  2.0  3.0   4.0
  .   5.0  6.0   7.0
  .    .   8.0   9.0
  .    .    .   10.0

Cholesky{Float64, Matrix{Float64}}
U factor:
5×5 UpperTriangular{Float64, Matrix{Float64}}:
 1.22491  0.86684   1.12125    0.616258  0.643779
```

```
  .       0.840492  0.522972   0.410345  0.774078
  .        .        0.794623  -0.137927  0.0562672
  .        .         .         0.407503  0.082128
  .        .         .          .        0.293004

Matrix not pos-def.

Matrix not pos-def.

Matrix not pos-def.

Cholesky{Float64, Matrix{Float64}}
U factor:
6×6 UpperTriangular{Float64, Matrix{Float64}}:
 0.562231  0.573649   0.0814434   0.226344   -0.223845  -0.0343774
  .        0.500819  -0.358015   -0.0600356   0.253891   0.491001
  .         .         0.787863   -0.115487   -0.294512  -0.384284
  .         .          .          0.67844    -0.417802   0.42769
  .         .          .           .          0.616474  -0.174615
  .         .          .           .           .         0.237077
```