Math 245: Numerical Methods and Mathematical Computing

Algorithms for naïve Gaussian elimination and the $A = LU$ factorization, using 0-based indexing and semi-open intervals

Brenton LeMesurier, Last Revised November 8, 2017

This notebook describes these core algorithms of linear algebra using the same indexing conventions as in most modern programming languages: Python, C, Java, C++, javascript, C#, Objecive-C, Swift, etc. (In fact, almost everything except Matlab and Fortran.)

The key elements of this are:

- Indices for vectors and other arrays **start at 0**.
- Ranges of indices are described with *semi-open intervals* $[a, b)$ in cases where the order does not matter.
  This "index interval" notation has two virtues: it emphasizes the mathematical fact that the order in which things are done is irrelevant (such as within sums), and it more closely resembles the way that most programming languages specify index ranges. For example, the indices i of a Python array with n elements are $0 \le i < n$, or $[0, n)$, and the Python notations `range(n)`, `range(0,n)`, `:n` and `0:n` all describe this. Similarly, in Java, C C++ etc., one can loop over the indices $i \in [a, b)$ with `for(i=a, i<b, i+=1)`
- When indices musr run in increasing order — i = a, then a+1, ... then b — I use

  `for i from a to b`
- When indices musr run in decreasing order — $i = b$, then $b - 1, \ldots$ then a — I use

  for i from a to b This is the one place that the indexing is a bit tricky in Python:
  The index values i are $b \ge i > a - 1$, which in Python is `range(b, a-1, -1)`.
- Often though, the order does not matter

Notes on pseudo-code notation

- I mark the end of `for` loops with a line `end for`, and likewise with `if ... end if` and so on. Python omits these `end` markers, instead relying on indentation, but it can help with readability.
- I use the equals sign only when the quantity at left has a unique, fixed value, so that it is a mathematically valid equality.
- If instead the quantity at left can vary during the computation, I indicate assignment with a left-arrow, as with $i \leftarrow i + 1$, because this is not the same as mathematical equality: $i = i + 1$ is false!

Python code

I include Python code for comparison for just the three most basic algorithms: "naive" LU factorization and forward and backward substitution, without pivoting. The rest are good exercises for learning how to program loops and sums.

The naive Gaussian elimination algorithm

Version 1: Strict mathematical notation

In this careful version, the original matrix A is called $A^{(0)}$, and the new versions at each stage are called $A^{(1)}$, $A^{(2)}$, and so on to $A^{(n-1)}$, which is the row-reduced form also called U; likewise with the right-hand sides $b^{(0)} = b$, $b^{(1)}$ up to $b^{(n-1)} = c$.

```
for k in [0, n-1)
   for i in [k+1, n)
     l_{i,k} = a^{(k)}_{i,k} / a^{(k)}_{k,k}
      for j in [k+1, n)
        a^{(k+1)}_{i,j} = a^{(k)}_{i,j} - l_{i,k} a^{(k)}_{k,j}
      end for
     b^{(k+1)}_i = b^{(k)}_i - l_{i,k} b^{(k)}_k
   end for
end for
```

Actually this skips formulas for some elements of the new matrix $A^{(k+1)}$, because they are either zero or are unchanged from $A^{(k)}$: the rows before $i = k$ are unchanged, and for columns before $j = k$, the new entries are zeros.

Version 2: Algorithmic notation, with variable quantities

In software all those super-scripts can be ignored, just updating the arrays `A` and `b`. However then quantities like $a_{i,k}$ have variable values, so I use teh above-mentioned assignment notation "$\leftarrow$".

```
for k in [0, n-1)
   for i in [k+1, n)
     l_{i,k} = a_{i,k} / a_{k,k}
      for j in [k+1, n)
        a_{i,j} ← a_{i,j} - l_{i,k} a_{k,j}
      end for
     b_i ← b_i - l_{i,k} b_k
   end for
end for
```

The direct LU factorization with L unit lower triangular

One advantage of this algorithm is that the notati is purely mathematical: there are no mutiple versions needing notation like the $a^{(k)}_{i,k}$ above.

The first row and column "$k = 0$" are special: they do fit the general pattern, but the sums are empty, so it is safer and a bit clearer to handle them separately:

```
for j in [0, n)
   u_{0,j} = a_{0,j}
end for
for i in [1, n)
   l_{i,0} = a_{i,0}/u_{0,0}
end for
```

```
for k in [1, n)
   for j in [k, n)
      u_{k,j} = a_{k,j} − Σ_{s∈[0,k)} l_{k,s} u_{s,j}
   end for
   for i in [k+1, n)
      l_{i,k} = ( a_{i,k} − Σ_{s∈[0,k)} l_{i,s} u_{s,k} ) / u_{k,k}
   end for
end for
```

Python translation

Since module `numpy` is used so often, it is a widely-used convention to give it the nickname `np`.

```
In [1]: import numpy as np
```

Also, note:

- the convenient function `sum`, which adds the elements of an array, and
- the "slicing" notation with columns;
  in general `a:b` denotes the semi-open interval $[a, b)$, as in `range(a,b)`.
  Thus `A[b:c, d:e]` is the smaller array with elements `A[i, j]` for $i \in [b, c)$ and $j \in [d, e)$.

In [2]:
```
def luFactorize(A):
    '''
    Dolittle's method for direct calculation of an LU factorization LU of A,
    with L unit lower triangular and U upper triangular.
    Warning: nothing is done to detect or avoid division by zero!
    '''
    n = len(A)  # Function len counts the rows of A.
    L = np.eye(n, n)  # The identity matrix or "I"; my apologies for the pun.
    U = np.zeros((n, n))
    # Do the first row and column [k=0] separately, as they involve no sums.
    for j in range(0, n):
        U[0, j] = A[0, j]
    for i in range(1, n):
        L[i, 0] = A[i, 0]/U[0, 0]
    # Now the general case.
    for k in range(1, n):
        for j in range(k, n):
            U[k, j] = A[k, j] - sum(L[k, 0:k] * U[0:k, j])
        for i in range(k+1, n):
            L[i, k] = (A[i, k] - sum(L[i, 0:k] * U[0:k, k]))/U[k, k]
    return (L, U)
```

Forward substitution with a unit lower triangular matrix

For L unit lower triangular, solving $Lc = b$ by forward substitution is

$c_0 = b_0/l_{0,0}$
for i in [1, n)
$$c_i = b_i - \sum_{j \in [0,i)} l_{i,j} c_j$$
end for

In [3]:
```
def solveUnitLowerTriangular(L, b):
    n = len(b)
    c = np.zeros(n)
    c[0] = b[0]/L[0,0]
    for i in range(1, n):
        c[i] = b[i] - sum(L[i,0:i] * c[0:i])
    return c
```

Backward substitution with an upper triangular matrix

$x_{n-1} = c_{n-1}/u_{n-1,n-1}$
for i from n-2 down to 0
$$x_i = \frac{c_i - \sum_{j \in [i+1,n)} u_{i,j} x_j}{u_{i,i}}$$
end for

Counting backwards in Python

For the Python implementation, we need a range of indices that change by a step of -1 instead of 1. This can be done with an optional third argument to range: `range(a, b, step)` generates a succession values starting with `a`, each value differing from its predecessor by `step`, and stopping **just before** `b`.
That last rule requires care when the step is negative: for example, `range(3, 0, -1)` gives the sequence $\{3, 2, 1\}$. So to count down to zero, one has to use $b = -1$! That is, to count down from $n - 1$ and end at 0, one uses

```
range(n-1, -1, -1)
```

In [4]:
```
def solveUpperTriangular(U, c):
    n = len(c)
    x = np.zeros(n)
    x[n-1] = c[n-1]/U[n-1,n-1]
    for i in range(n-2, -1, -1):
        x[i] = ( c[i] - sum(U[i,i+1:n] * x[i+1:n]) )/U[i,i]
    return x
```