A Flop Count Tool with an Example in PLU Factorization

Shiyi Chen
Tutor: Leonardo T. Rolla

December 25, 2020

Abstract

We develop a new flop tool which can help to count floating point operation(flops) in MATLAB/GNU Octave. The first two sections describe the flops and PLU factorization. In the following sections, we introduce the PLU factorization — a method for solving linear equations. We discuss several possible ways to count flops on a PLU factorization script in Matlab/Octave. One way we tried is using the flop tool developed by Hang Qian. Although his tool does not give us correct result due to its limitation on counting operations involving varying variable sizes, his code gives the flop count formulas we needed in developing the new tool. Our new tool is compatible with GNU Octave and it is licensed under GNU LGPL.

Contents

1 Flops

Flop stands for floating-point addition, subtraction, multiplication, and division[2]. Flops can be used to quantify the amount of work needed to be done for an algorithm to finish executing.

For example, below is a sample Matlab/Octave code which generates a 5 by 5 matrix and multiplies the first two rows and the first column:

```
A = ones(5,5);
B = A(1:2,:) * A(:,1);
```

Above code takes 2*5*2 flops in total due to the matrix multiplication. For a more general situation, Table 1 summarizes the flops needed for some operations:

Operation	Dimension	Flops
`a * x`	a is a scalar, x is a n by 1 array	n
`x * y`	x and y are both n by 1 arrays	$2n$
`A * x`	A is m by n array, x is n by 1 array	$2mn$
`A * B`	A is m by r array, B is r by n array	$2mnr$
`A .* B`	A and B are both m by n arrays	$2mn$
`A + B`	A and B are both m by n arrays	mn
`A - B`	A and B are both m by n arrays	mn
`A\b`	A is m by m array, b is m by 1 array	about $\frac{2m^3}{3}$

Table 1: Flop count formulas

For the last row in Table 1, `A\b` returns the solution of the linear equation $Ax = b$. For the solving the n by n linear systems, flops roughly equals to PLU factorization of the matrix A which is about $\frac{2m^3}{3}$.

2 PLU factorization

PLU factorization is one way to solve the linear system with respect to square matrices. In PLU factorization, a square matrix into three parts, a lower triangular matrix L, an upper triangular matrix U, and a permutation matrix P which keeps the track of row exchange. After the factorization, the relation $PA = LU$ holds. To solve the linear equation $Ax = b$, we only need to calculate Pb then solve $LUx = Pb$. This process is much easier than solving $Ax = b$ directly because U is upper triangular while L is lower triangular. The PLU factorization includes alternatively applying row exchange and row replacement, in the mean while, uses P, L, and U to keep the track of those changes, PLU stops at $N-1$ rounds while N represents the size of the square matrix.

An intuitive way of thinking row exchange and row replacement is considering these two elementary operations as two kind of matrix multiplications with elementary matrices. Suppose E is the row replacement matrix and Q is the row exchange matrix, since $PA = LE^{-1}EU$ and $QPA = QLQQU$, at each iteration we can set the new P, L, U as following[3]:

$$\begin{cases} P_{k+1} = P_k \\ L_{k+1} = L_k E_k^{-1} \\ U_{k+1} = U_k \end{cases} \quad \text{or} \quad \begin{cases} P_{k+1} = QP_k \\ L_{k+1} = QL_kQ \\ U_{k+1} = QU_k \end{cases}$$

We start PLU program by doing row exchanges and row replacements using matrix multiplication. Although matrix multiplication is intuitive to understand, it involves too many flops. Here is why: using all matrix multiplications in algorithm, a row replacement takes one matrix multiplication while a row exchange takes four matrix multiplications. As shown in Section 1, matrix multiplication is not efficient, so we may do row exchange by switching the row and row replacement by substituting the targeting rows. Script 1 takes a square matrix A as input and matrices P, L, U as outputs.

The script consists of three functions. The main function uses a `for` loop to perform row exchange and row replacement alternatively for $N-1$ times, `exchange` and `replace` perform row exchange and row replacement respectively.

3 Flop count — explorations and inspirations

3.1 Count flops using formulas

Let the size of square matrix be n, here is how we count `LU.m`:

- Find all lines involving operations and execution times, locations below are shown in Script 1:

  line 08: `for i = 1:sz-1`
  "$-$" executed once.
  line 19: `q = i - 1;`
  "$-$" executed `n-1` times.
  line 26: `q = i + 1;`
  "$+$" executed `n-1` times.
  line 29: `a = U(j,i)/u;`
  "$/$" executed $\sum_{i=1}^{n-1}(n-i)$ times.
  line 32: `U(j,q:sz) = U(j,q:sz) - a*U(i,q:sz);`
  "$-$" and "$*$" executed $\sum_{i=1}^{n-1}(n-i)$ times respectively.

- Using the table in section one, we have the following formula:

$$\begin{aligned}\text{flops} &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(2(n-j)+1)+n-1+n-1+1 \\ &= \frac{2}{3}n^3-\frac{n^2}{2}+\frac{11}{6}n-1.\end{aligned}$$

This is the best method for PLU alone, but we went a more general tool.

3.2 Existing flop count tool

The current flop count tool in Matlab was written by Hang Qian. This flop count tool, a Matlab function named "FLOPS", analyzes the .M file line by line and tells users the flops at each line of the script and total flops in the terminal. "FLOPS" provides a wide range of applications, it supports both function and script. This tool also supports some operations other than `+`, `-`, `*`, `/`, including some functions like `lu`, `sum`, `prod`, `sin`. Moreover, the user can self-define the flop count rules in an

EXCEL file, the new rules override the current rules if there exists such rules previously defined in the flop count tool program. Otherwise, the tool adds the user defined rules in flop count.

Following the user guide, we generate the random matrices with sizes from 1 to 40, run "FLOPS" and get the flops. Comparing the leading term $\frac{2}{3}n^3$, the result given by "FLOPS" is far less than the leading term. The reason for this error is that the PLU script written by us has hit a limitation of "FLOPS" as stated in the user guide:

- Variable classes and matrix sizes cannot change anywhere in the codes. A variable, once created, cannot be deleted (but its value may change as long as it does not undermine matrix decomposition, say chol). A MAT file that stores all variables is required[1].

The general situation for this tool to fail is that the vector sizes change within the file, this is because "FLOPS" supposes the size of the variables unchanged in the whole file and the script takes the final sizes as their actual sizes which is not the case in `LU.m`. Below is part of `LU.m` that causes the problem, in line 5, we have used slicing in our code, although it does not change the size of the variables, the slicing part `U(i,q:sz)` changes in every iteration in the for loop. Since "FLOPS" only deals with fixed sizes, the result given by this tool is inaccurate.

```
1 for j = q : sz
2     a = U(j,i)/u;
3     L(j,i) = a;
4     U(j,i) = 0;
5     U(j,q:sz) = U(j,q:sz) - a*U(i,q:sz);
6 end
```

Here is a code that shows the same problem:

```
1 U = ones(5,5); b = ones(5,1); i = 1;
2 a = U(i : 5, 1 : 5);  c = a * b;
3 i = 3;
4 a = U(i : 5, 1 : 5);  c = a * b;
```

As long as variable sizes change in the script, "FLOPS" will have a great chance to fail.

3.3 Count with help of original file

We need a stable yet easy method that can give us a correct flop count to analyze the complexity of the algorithm in the future. The first method is suitable for the counting in PLU factorization, it may not be suitable for future usage since we may dealing with scripts involving large amount of computation; the second method cannot deal with the situations in which variable sizes changing in the algorithm. Here is a more practical way we count the flops:

- Create a variable named `flop_counter`, declare it as global variable since it needs to be updated within multiple functions, set it as 0.

- Make a copy of the original file, say `LU_tmp.m`, whenever there is an operation, add a flop count formula after the command which involves the operations. For example, the formula for line 57 will be:

  ```
  flop_counter = flop_counter + 2*(sz-q).
  ```

- Run the code.

The variable `flop_counter` should be the amount of flops the code takes.

This method gives us the same result as the computation done in section 3.1.

Compared to the method in section 3.2, this method requires manually adding codes. We want an easy method, without adding any lines by hand, which gives us the correct result.

4 The new tool

Inspired by Sections 3.2 and 3.3, we decide to start a tool which can count flops correctly and does not require manual coding. We initially coded in Octave and later we made it compatible with Matlab. For Octave users, version 4.4.2 and above is suggested. The principle for this tool is the same as the method presented in Section 3.3 — adding flop count

formulas line by line, but we let the flop count tool do the job.

The new tool is available in GitHub https://github.com/Eleven7825/flopTool under GNU Lesser General Public License with version 3.0 or later.

4.1 The user guide

To count flops, we need three things: m-file (function and script are both acceptable), `flop_update.m`, and `flop_script.m`. We will use a script called `fileName.m` as an example for the description. An step by step tutorial script - `easy_runtest.m` with an example script is available in the GitHub repository.

1, Prepare all code within one m-file. If the script calls multiple functions in different scripts, put all functions in one m-file.
2, Put this file in the same folder with `flop_update.m`, `flop_script.m`, go to that folder, execute `flop_script("fileName.m")` or `flop_script("fileName")`.

3, Notice the command window, `Begin generating flop count script for example ...` indicates the starting to generate the temporary flop count file. The prompt `done!` indicates the process terminates and there will be a temporary file called `fileName_tmp.m` generated under the same folder. If warning appears, the program will tell the user which line in the original file causes the problem. The user needs to check the line in the original file and fix the error. There are two types of warning:

1) `Warning: In line XX, unrecognized pattern` indicates the flop analyzer crashed at `XX` line. For example, there may be some brackets missing which causes the program failing to read the variables. In this case, the program will jump that line and continue counting the rest of the script. The user needs to check whether the brackets are missing in the line, then try running `flop_script.m` again.

2) `Warning: In line XX, can't find left variable, assigning value 1 to it.` In this case, program does not find variables corresponding to operators in this line. The program will automatically assign 1 to that unknown variable and continue counting in this line. The user need to add brackets around the variable they want the program to count at the original file and run `flop_script.m` again, the temporary file will be automatically overwritten for the second run.

4, Run the temporary file, if the m-file is a function, enter the temporary function signature, for example `fileName_tmp(A)`; if the file is a script, click run. The temporary code should have the same output in the terminal but with a slower speed. After executing, a variable called `flop_counter` should be created in the workspace.

4.2 Process, Implementation, Limitations, and Comparison

Process The new tool passes `fileName` as parameter, the program finds the file with this name under the same folder and reads the entire m-file into n by 1 text cell array - each line in the cell contains a line of code, this is done by function `readText` which is part of Qian's work.

Next, the program will pass each non-empty, non-comment element in the cell to `fpExpt` function. This function will detect all the operations within the line, then classify them into two categories: `lu, sin, cos, prod, sum`, etc will be passed into the first category; basic addition, subtraction, multiplication, division will be passed through the second category. `fpExpt` — the main tool for the function will catch variables corresponding to the operators based on the operator's categories; then assign the captured variables to the main function in string format. Finally, the main function will pass the operators, variables to the flop count formula, generate a new text cell and write the new cell into the temporary m-file.

When the user runs the newly generated m-file, the variables will be calculated into numerical format based on the expression originally in string format. The numerical variables will be passed into `flop_update.m`, an independent function. `flop_counter` will be updated on a global level based on sizes of input variables. `flop_update` formulas are copied from "FLOPS". Future distribution should follow the GNU Lesser General Public License and remains both license notices.

Implementations The core part of this tool is `fpExpt`, which functions as a operation signs and variable names recognizer.

1) Method to get variable names.
First, the `fpExpt` function will change the non-numeric, non-alphabetic,

non-underscore characters into blank spaces. Next, it will call the string split function to split the command string into a text cell. The string cell contains the variable names needed to be recognized.

2) Method for `fpExpt` to determine the sequence of operation
The program will detect the operations with a higher priority first. The priority is raise power > multiply or divide > plus or minus. The program will add brackets around two variables after detection. If there are additions or subtractions in later detection, the bracket will tell where the variables which `fpExpt` need to recognize starts and ends.

3) Method for `fpExpt` to decide whether a pair of brackets functions as slicing or indicator of the operational orders
`fpExpt` considers brackets around operators as order indicators while other situations slicing.

4) The method to find the other corresponding bracket location for given brackets.
Function `fpExpt` will call `findbrak` function, this function will find another side of the bracket and return the location of the other side of the bracket.

5) Method to determine whether an operation symbol is between two quote marks or not
This is done by function `isBetPr`. This function will find all the quote mark locations, create an array contains all indexes which are in the quote marks. After a comparison with the operation symbol's indices, it determines whether the operators are in two quote marks or not.

Limitations 1) The flop count script will be more time consuming than the original file.

2) The new tool inherits the counting formulas from "FLOPS", it has the same limitations for counting operations[1].

3) Outside functions are not parsed, users need to put all functions in

[1]For example, logical operations(`|`, `&`), relation operations(`>`,`>=`) are not counted. Transpose signs and indexing are recognized but not count into total flops. Different from Qian's, we count negations ($A = -A$) as subtractions with corresponding sizes. In this case, we consider the left variable unrecognizable, thus, value 1 was assigned to the left variable. In Matlab/Octave, the flop count is equivalent to `ones(size(A))-A`.

the same file.

4) For `lu`, `sum`, etc, only one argument is supported.

Comparison with "FLOPS" Compared to "FLOPS", the new tool may be more convenient to use in the sense that it does not need to save or load MAT files or profiling. We hope future users can find it useful. Our code incorporates part of code by Qian. Table 2 was made to help the future users decide which tool to use based on their purposes.

The new tool	"FLOPS"
The new tool may be more convenient for counting with changing input variable sizes in the sense that it does not need profiling, saving or loading MAT files.	Need profiling, saving and loading MAT files.
Longer elapse time for the temporary counting tool, which makes our tool not suitable for longer scripts.	Same elapse time with the original file.
Cannot recognize functions containing multiple arguments.	Can accept some multiple arguments like sum(A,2).
Cannot recognize functions handles.	Can recognize bsxfun.
Cannot know exact flops in each line.	Display flops at each line and their executed rounds.
Support variable changes and slicing.	Does not support variable changes and slicing.
Support multiple functions in a script but not nested.	Support nested function.
Change the rules in TXT file.	Change the rules in EXCEL file.
Support Octave $>$ 4.2.0 and Matlab.	Only support Matlab.
Generate transparent auditable code in a separate .M file.	Display the flop count on the command window.

Table 2: A comparison with "FLOPS"

5 Linear regression — run test on the new tool

The script `runtest.m` functions as a test code for our new tool and also testifies how well the flops curve of self-written PLU script approximates the leading term $\frac{2n^3}{3}$. An important feature for `runtest` is that users have freedom to specify the parameters if they want to change certain variables in the workspace.

5.1 The user guide for runtest.m

To test this script, after click run, detailed execution information will appear on the command window. The current parameters in the workspace will also be displayed. For the first run, the program will take the default values. The user can change the default values in the workspace, then the program will take user-defined value for the execution. Table 3 displays all default values and their functionalities.

Parameters	Discription	default
`n`	The number of matrices to be tested (`double`).	6
`foldername`	The name of targeting output folder (`string`).	"output"
`fileformat`	The format pass to the `saveas` function (`string`).	"epsc"
`filename1`	The name for the plot (`string`).	"PLU-flops1.eps"
`show_plot`	0 represent popping the figure window, 1 otherwise (`double`).	0

Table 3: The default variables for runtest.m

The `runtest.m` mainly does the following:

1) Call flop_script.m and create the temporary flop count file called LU_tmp.m under the current folder.

2) Generate a set of random matrices with sizes from $2^0, 2^1$ to 2^{n-1}. n is by default 6.

3) Set the flop_counter as 0, perform flop count on these matrices by calling `LU_tmp(A)`, while A is overwritten repeatedly by those matrices with increasing sizes.

4) Plot the relationship between the size of A and flops on the log-log scale, create a subfolder called output within the current folder if such folder does not exist, save the newly generated file in this folder with default with format EPS.

5) Do linear regression on the log-log scale without considering the first three points, plot all the points in blue circles while the regression line in red. Save the plot in EPS in the output subfolder, display the regression relationship formula.

5.2 Results

The following result was produced by setting n = 10. The linear regression gives us flops = $0.620505 \cdot N^{3.012631}$. And the plot of the `runtest` is shown in Figure 1. When n = 6, the regression formula is flops = $0.599507 \cdot N^{3.024663}$. This seems consistent with the formula we get from Section 3.1 which indicates the leading term is $\frac{2}{3}N^3$.

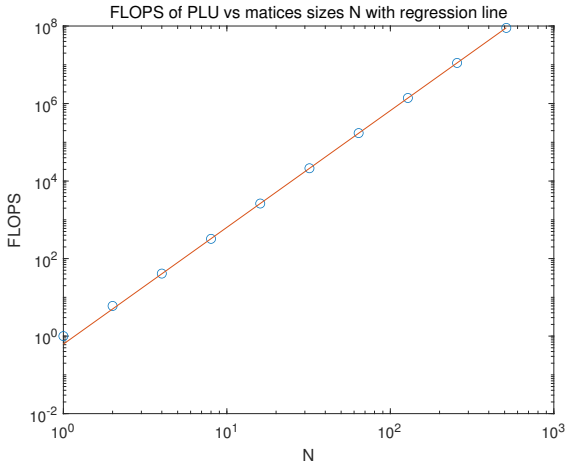

Figure 1: PLU-flops2

6 Conclusion

In this report, we describe an important factor to quantify the complexity of an algorithm - flops. Then, we move on to describe several ways to count flops of PLU factorization - a method to solve the linear equations. One way we tried is the flop counting tool in Matlab/Octave developed by Hang Qian. Although this tool ("FLOPS") does not give us the accurate flop count ("FLOPS" fails when there are variables with varying sizes in the script), "FLOPS" helps us to develop the new flop count tool.

In Section 5, we conclude that our new tool gives the exact result comparing to the accurate count. Moreover, our new tool can work on Octave which is a free software under GPL. However, our tool has long elapse time, this defect may be more significant for longer scripts. There are many possible improvements for our flop tool, for example, solve the long elapse problems, support for functions containing multiple arguments, make the flops at each line available, also, `flop_counter=0` should be done by the user rather than by the generated code. We cannot solve all those problems at once because of the time limitation. We made our tool available under GNU Lesser General Public License and all source codes can be found at https://github.com/Eleven7825/flopTool. We hope future users could have a choice of flop count tools best fitting their needs.

References

[1] Hang Qian (2020). Counting the Floating Point Operations (FLOPS) (https://www.mathworks.com/matlabcentral/fileexchange/50608-counting-the-floating-point-operations-flops), MATLAB Central File Exchange. Retrieved November 5, 2020.

[2] Gene H. Golub. *Matrix Computations*. 1983.

[3] Leonardo T. Rolla. PLU Fractorization. 2020.

Script 1 The shortened code for PLU factorization

```
%PLU factorization

function [P,L,U] = LU(A)
...Find size of the matrix A
...Initialize P_0,L_0,U_0

% proccess of PLU
for i = 1 : sz-1
    [P1,L1,U1] = exchange(P,L,U,i);
    [P2,L2,U2] = replace(P1,L1,U1,i);
    ... Update P1, U1, L1
end
end

%row exchange function
function [P,L,U]=exchange(P,L,U,i)
global sz
...Find maximum absolute entry in a column
q = i-1;
...Perform exchange
end

%row replacement function
function [P,L,U] = replace(P,L,U,i)
global sz
q = i+1;
u = U(i,i);
for j = q : sz
    a = U(j,i)/u;
    L(j,i) = a;
    U(j,i) = 0;
    U(j,q:sz) = U(j,q:sz) - a*U(i,q:sz);
end
end
```