Chapter 1.1: Counting flops, big Oh, block-matrix multiplication

As we go through the course, certain things will be taken for granted because they are contained in the prerequisites; for example, today I will talk about vectors and matrices, vector-vector, matrix-vector, and matrix-matrix multiplication, and assume you know what I am talking about.

Flops and Big Oh notation. Suppose we want to understand how fast an algorithm is. For example, we want to do a very simple computation in MATLAB, e.g., computing the inner product of two vectors $u$ and $v$ of length $n$: $u^T \cdot v = \sum_{i=1}^{n} u(i) \cdot v(i)$. We might want to write a function which takes as input a number $n$ and two column vectors, each with $n$ components, and returns the (inner) product of these vectors.

Here is the simplest way to do this in MATLAB, with the help of a for loop.

```matlab
function product = vecvec(n, u, v);
   if size(u,1) == [n,1],
      error('u has the wrong format');
   end;
   if size(v,1) == [n,1],
      error('v has the wrong format');
   end;
   product = 0; % initializes the product; this is how we comment out things
   for i = 1 : n
      product = product + u(i) * v(i);
   end
end
```

This code is pretty self-explanatory; it calculates the inner product via the “for loop”, after a quick check is performed to make sure the inputs are consistent (that $n$ is indeed the length of $u$ and $v$).

How long will it take to calculate this? Of course, it will depend a bit on the speed of your processor, but on the same processor, it will depend on the number of floating point operations, or flops, that the code will execute.

Floating point operations are “the big 4”, $+,-,\ast,/$, but occasionally we will also refer to things like $\sqrt{}$ as a floating point operation. (In reality, many systems spend a lot more on multiplication/division than one addition/subtraction, so sometimes only the former are counted, but here we will count everything.)

If we go back to the function, there is a simple way to check, empirically, how many operations are executed: we can keep track of the number of operations in a new variable, $op\_count$, which we will also include among the outputs:

```matlab
function [product, op_count] = vecvec(n, u, v);
   if length(u) == n,
      error('u has the wrong format');
   end;
   if length(v) == n,
```

error('v has the wrong format');
end;

product = 0;  % initializes the product; this is how we comment out things
op_count = 0;
for i = 1 : n
    product = product + u(i) * v(i);
    op_count = op_count + 2;
end

Note that we increment op_count by 2 each time, which reflects the fact that the line before it
has one addition and one multiplication, thus 2 flops. The update of the variable product is the
only line of the code where we do actual computations.

How fast is this code? How much longer will it take to run if we let n = 100? n = 200? n = 800?
The output op_count can be used to answer these questions, individually, but if we want, we can
find a formula for the answer, with relative ease.

Each execution of the for loop has 2 operations, and there are n executions, so the answer is
2n. Therefore the cost is linear.

It will take roughly twice as long to perform the same calculation if n = 200, 8 times as long if
n = 800, and 200 times as long if n = 20,000. (If we increase n too much, other issues will arise,
like the cost of transporting data from the slow, storage memory to the fast, computing memory,
but this is the subject of a different kind of course.)

We will refer to this vector multiplication algorithm as being a $O(n)$ algorithm, which simply
means that the amount of work done is proportional to n.

For HW 0, you will have to do the same exercise for matrix-vector, respectively, matrix-matrix
multiplication, and count the number of operations performed.

In general, we will deal with algorithms using for loops, matrix-vector, matrix-matrix, and vector-
vector multiplications, and so the number of flops will always be proportional to some polynomial
in n, thus, in Big-Oh notation, $O(n^p)$ for some p integer. Mostly we will deal with $p = 1, 2, 3$.
We will sometime request more than just a general magnitude of the flop count, we will request the
actual constant of proportionality, as we will with LU, Cholesky, and QR. (And in HW 0, as
well!)

One last thing that may be of interest: although the simple-minded matrix multiplication of two
square $n \times n$ matrices is an $O(n^3)$ algorithm, this is not the fastest possible matrix multiplication
algorithm. The complexity of matrix multiplication is conjectured to be roughly $O(n^2)$, although
much less can currently be shown. Fast matrix-multiplication algorithms has complexities from
$O(n^{\log_2 7})$ to roughly $O(n^{2.35})$. If that intrigues you, read the last two paragraphs of 1.1.

**Block-matrix multiplication.** Some of the fast matrix multiplication algorithms mentioned in
the paragraph above use block-matrix multiplication; also, we will use it very occasionally as well
in problems in this class. As such, it’s worth reviewing.

Suppose that we want to multiply two matrices A, B, which are $m \times n$, respectively $n \times p$, and
suppose that $m_1 + m_2 = m$, $n_1 + n_2 = n$, $p_1 + p_2 = p$. Split the matrices A and B by splitting their
rows and columns into two sets, as follows:

$$
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},
$$
where $A_{11}$ is $m_1 \times n_1$, $A_{12}$ is $m_1 \times n_2$, $A_{21}$ is $m_2 \times n_1$, $A_{22}$ is $m_2 \times n_2$, and similarly, $B_{11}$ is $n_1 \times p_1$, $B_{12}$ is $n_1 \times p_2$, $B_{21}$ is $n_2 \times p_1$, and $B_{22}$ is $n_2 \times p_2$.

Then the multiplication of the two matrices $A$ and $B$ can also be done block-by-block, i.e.,

\[
A \cdot B = \begin{pmatrix}
A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22}
\end{pmatrix}.
\]

(Note first that we can indeed multiply all the block-matrices in the way presented on the right hand side, and the partition allows for that. For example, $A_{21} \cdot B_{11}$ is a $m_2 \times p_1$ matrix, and so is $A_{22} \cdot B_{21}$, so it’s ok to add them.)