Strassen Algorithm

Lam, Tao

3/24/17

1 Matrix Multiplication

We implement Strassen's algorithm to improve the standard matrix multiplication algorithm, $\Theta(n^3)$, for reasonably sized matrices to $\Theta(n^{\log 7})$. For sufficiently large values of n, Strassen's algorithm will run faster than the conventional algorithm. For small values of n, however, the conventional algorithm may be faster. We can define the cross-over point between the two algorithms to be the value of n for which we want to stop using Strassens algorithm and switch to conventional matrix multiplication. We seek to analytically and experimentally determine the cross-over point.

We define Strassen's algorithm as follows. Given matrices, A and B, of size n we split each into 4 sub-matrices of size $\frac{n}{2}$. We denote $A_{1,1}, A_{1,2}, A_{2,1}, A_{2,2}$ as the sub-matrices of matrix A and $B_{1,1}, B_{1,2}, B_{2,1}, B_{2,2}$ as the sub-matrices of matrix B, where $A_{1,2}$ denotes the sub-matrix on the top right of A. We compute the following seven products:

$$\begin{aligned}
M_1 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_2 &= (A_{1,1} + A_{1,2})B_{2,2} \\
M_3 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_6 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \\
M_7 &= (A_{1,1} - A_{2,1})(B_{1,1} + B_{1,2})
\end{aligned}$$

Then we can find the appropriate terms of the product by addition, such that $C = AB$ and $C_{1,2}$ denotes the top right section of C:

$$\begin{aligned}
C_{1,1} &= M_5 + M_4 - M_2 + M_6 \\
C_{1,2} &= M_1 + M_2 \\
C_{2,1} &= M_3 + M_4 \\
C_{2,2} &= M_5 + M_1 - M_3 - M_7
\end{aligned}$$

2 Analytical Cross-over

In order to efficiently design our variant of Strassen's algorithm, we are tasked with finding the analytical crossover point where it becomes more efficient to do conventional multiplication than another recursive step of Strassen's. To do this, we count the number of operations required by each algorithm, and attempt to find an optimum.

In the conventional multiplication of n x n matrices, there are n^2 elements in the matrix, each of which requires n multiplications and $n-1$ additions. Therefore, our operations function is:

$$f(n) = n^2(n + n - 1) = 2n^3 - n^2$$

For Strassen's, we see from above that there are 18 sub-matrix additions/subtractions, and 7 sub-matrix multiplications. Since Strassen's is a recursive algorithm, each of these 7 sub-matrices will either be multiplied using the conventional algorithm, or Strassen's depending on which was found to be faster. We need only to check one level of Strassen's, from $n = 1$ onwards because we are looking for the first point where it is better to multiply than to do another round of Strassen's:

We define Strassen's as:

$$s(n) = 18\left(\frac{n}{2}\right)^2 + 7f\left(\frac{n}{2}\right) = 18\left(\frac{n}{2}\right)^2 + 7\left(2\left(\frac{n}{2}\right)^3 - \frac{n}{2}^2\right)$$

In order to solve for the crossover point, we find where $s(n) = f(n)$. For even n, we find that $n_0 = 16$. For odd n, we solve for

$$s(n) = 18\left(\frac{n+1}{2}\right)^2 + 7\left(2\left(\frac{n+1}{2}\right)^3 - \left(\frac{n+1}{2}\right)^2\right) = f(n)$$

$$n \approx 37.17$$

so we round $n_0 = 37$.

In order to do a similar numerical analysis to help confirm this, we redefine Strassen's algorithm to take the minimum of conventional multiplication or the next round of Strassen's, so we define the operation function as

$$s(n) = \min\left\{f(n), 18\left(\left\lceil\frac{n}{2}\right\rceil\right)^2 + 7s\left(\left\lceil\frac{n}{2}\right\rceil\right)\right\}$$

We take the ceiling of the halved submatrix because if n was an odd number, the submatrices would need to be padded with a dimension in order to maintain its divisibility.

Using a numerical analysis starting from $n = 1$, we have the following number operation results:

n	$Conventional\_Ops$	$StrassenVariant\_Ops$	$Faster$
1	1	1	$Conventional$
2	12	12	$Conventional$
3	45	45	$Conventional$
4	112	112	$Conventional$
5	225	225	$Conventional$
6	396	396	$Conventional$
...			$Conventional$
14	5292	5292	$Conventional$
15	6525	6525	$Conventional$
16	7936	7872	$Strassen$
17	9537	9537	$Conventional$
18	11340	11097	$Strassen$
...			$Conventional$
...			$Strassen$
32	64512	59712	$Strassen$
33	70785	70785	$Conventional$
34	77452	71961	$Strassen$
35	84525	83511	$Strassen$
36	92016	83511	$Strassen$
37	99937	99937	$Conventional$
38	108300	99997	$Strassen$
39	117117	112900	$Strassen$
40	126400	112900	$Strassen$
41	136161	134505	$Strassen$

For even n's, we see that the point we switch over to Strassen Variant is at $n = 16$. When even n's are greater than or equal to this point, it is more efficient to use at least one step of Strassen's than it is to multiply conventionally. The pattern of even n's using Strassen's continues as n increases. We see the first odd n to use Strassen's is 35, and the actual cutoff to be around 37 or 39. This confirms our analytical work. We think the anomaly here could be a result of how close the cutoff point is to a power of 2, but there could be other factors at play. Figure 1 shows a graph of our results.

3 Strassen Implementation

We chose to represent our matrices using vectors as opposed to arrays to allow for the flexibility of providing a matrix dimension at run-time. While there is a time trade-off to use a vector over a simple array, we found this added convenience well worth it. We also note that vectors come in useful if we are to pad matrices with 0s. An array's size is fixed at compilation time and the size of a vector can change dynamically as the program executes.

Note that a space naive implementation could declare a new $\frac{n}{2}$ matrix for each sub-matrix when computing matrices A, B and their sub-matrices $A_{1,1}, A_{1,2}, A_{2,1}, A_{2,2}, B_{1,1}, B_{1,2}, B_{2,1}, B_{2,2}$; however, we can avoid this by simply indexing into the original matrices, A and B. We can determine the start index for both the row and the column indexed into the original matrix by doing some simple algebraic

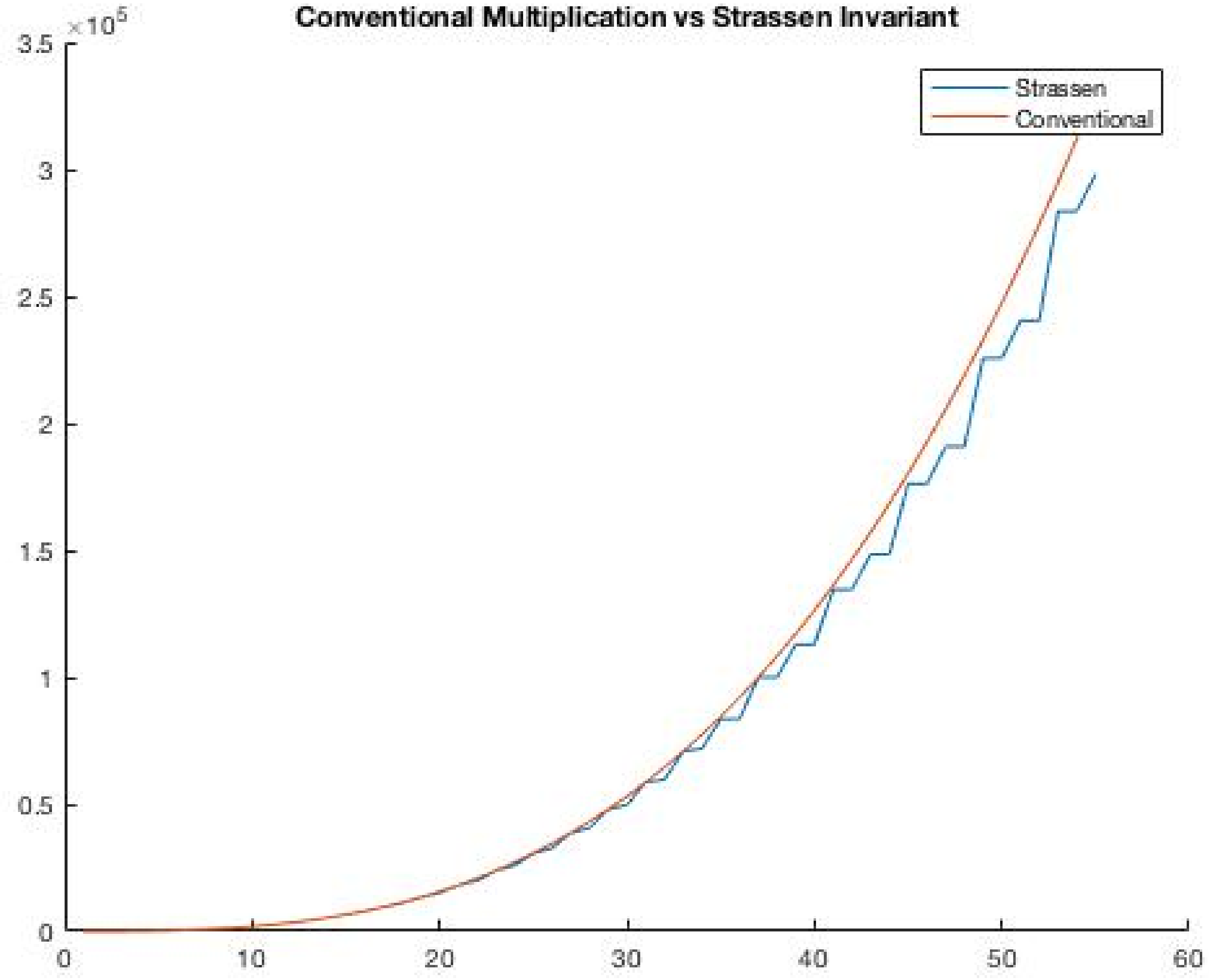

Figure 1: Numerical Analysis, note intersection 37

manipulation at each recursive call and passing these values to subsequent recursive function calls to find the smaller sub-matrices appropriately. We can similarly do the same when computing matrix C and its sub-matrices $C_{1,1}, C_{1,2}, C_{2,1}, C_{2,2}$. Doing this saves considerably on memory as we do not have to declare new sub-matrices and also saves considerably on time as we do not have to copy over values from one matrix to another, both savings are up to a constant factor and we save at each recursive call.

We also tried the following: we noted that we could maybe get away with not having to initialize all 7 matrices $M_1, M_2, ..., M_7$ and 2 additional matrices to store results l_term, r_term. Rather, we need only initialize 4 matrices M_1, M_2, M_3, M_4 and the 2 additional matrices l_term, r_term and overwrite appropriate matrices at each step when computing $C_{1,1}, C_{1,2}, C_{2,1}, C_{2,2}$. Doing so again would have saved a constant factor of extra space and we would have saved at each recursive call as well. However, we quickly found out this would not work due to the ordering of the Strassen function calls and the fact that passing a matrix reference would be altered incorrectly in subsequent recursive calls.

4 Experimental Cross Over

To find the optimal cross-over point, we iterated over cross-over values from 15 to n, where n denotes the dimension of the matrix and selected for the cross-over point which took the least amount of time. Note that we started from 15 based on the results of our analytical analysis and filled in the matrix randomly with 0s and 1s. We conducted multiple trials and took the average. The results are contained in *output/cross_over_timings.txt* (see repo).

Based on the output, we see that for matrix dimension 65 and onward, the Strassen Variant beats out the conventional matrix multiplication algorithm. We also noted that for dimensions 55, 56, 58 and 61 the Strassen Variant won out as well. We were careful to use matrices that were not just powers of 2, but also even and odd by testing every matrix dimension. Based on the analytical analysis, these results make sense; however, the disparities in results could be due to memory and computer factors. As a result, we conclude that the experimental optimal cross-over point lies at 64.

By avoiding excessive memory allocation and deallocation as well as avoiding copying large blocks of data unnecessarily, as described above in the implementation of Strassen, we were able to encounter little to no errors. Our implementation was able to handle matrices of size up to 4096. Note that caution must be taken when using larger integer values for such large matrices, as calculations risk integer overflow.

5 Padding for Strassen's

There are four different ways we chose to handle padding for Strassen's.

First, in order to handle matrix multiplication when n is not a power of 2, one can simply pad the matrix with 0s such that the dimension reaches the closest, next power of 2. We can take advantage of C++'s vector declaration, where all the values are initialized to 0. Thus, we need only declare a matrix whose size is the next power of 2 and then fill in the matrix with the values read in from the file. We can keep track of the index differences, such that we can then extract our desired matrix from the padded matrix. This is not the best way to pad, as we see later.

Second, given a crossover point of n_0 and a matrix dimension of n, we do not need to pad the matrix to the next largest power of 2. Instead, we can just pad the matrix to a dimension such that Strassen's can run all the way until the recursive step has dimension n_0. In order to do this, we find the pad amount $p(n, n_0)$:

$$p(n, n_0) = \min_k \left\{ n \leq n_0 * 2^k \right\} - n$$

We increase k until the inequality holds, which gives us the next dimension that will hit the crossover recursion size, then subtract the current dimension to find the pad amount. Therefore, if we have a 4 x 4 matrix, and the crossover step is 3, then the matrix dimension will be increased to 6 x 6, where the excess rows and columns are 0's. Note that this method is the same as the first for n_0 power of 2.

The third way we handle padding is to pad after each recursive step if n is an odd number, and stop if less than or equal to the cross over. This seems efficient at first glance, but each recursive step has a 4 times increase in the number of matrices to pad (although each matrix is smaller).

Finally, we can handle padding in a way that we look for the next number that can divide down neatly to less than or equal to the crossover:

$$\min_k \left\{ c * 2^k - n \right\}$$

where $c \in [n_0/2, n_0]$, and add all necessary padding to the matrix at the very beginning. This is more efficient than the first way since the constraint is "relaxed", and it is far more likely to reach some double multiple of a number c then it is to reach the particular cross over multiple. This runs in $O(n\log n)$ time as we search through $\frac{n_0}{2}$ values and each value takes $\log(n)$ time to compute. Which is negligible when compared to the time to compute the matrix itself.

Numerically, we run code on each method of padding to the following graph on sizes from 64 to 2048, using a crossover of 64. We get the following results.

In Figure 2, we see the padding required to get to the next power of 2. This is clearly not very efficient, because as soon as the dimension is slightly above the power of 2, it quickly jumps to a large value.

In Figure 3, where we only pad when the number is odd, we see the same jump, but to a lesser degree. The magnitude is much less than that of the power of 2 padding.

Finally, in Figure 4, where we add the necessary padding in the beginning, we see a similar pattern, but ever so slightly lesser. We decide to implement the padding algorithm in Figure 3, by checking each next number to see if it can divide neatly down to a number less than n_0. We note that gives us a faster completion time than our naive power of 2 algorithm as seen in the tables.

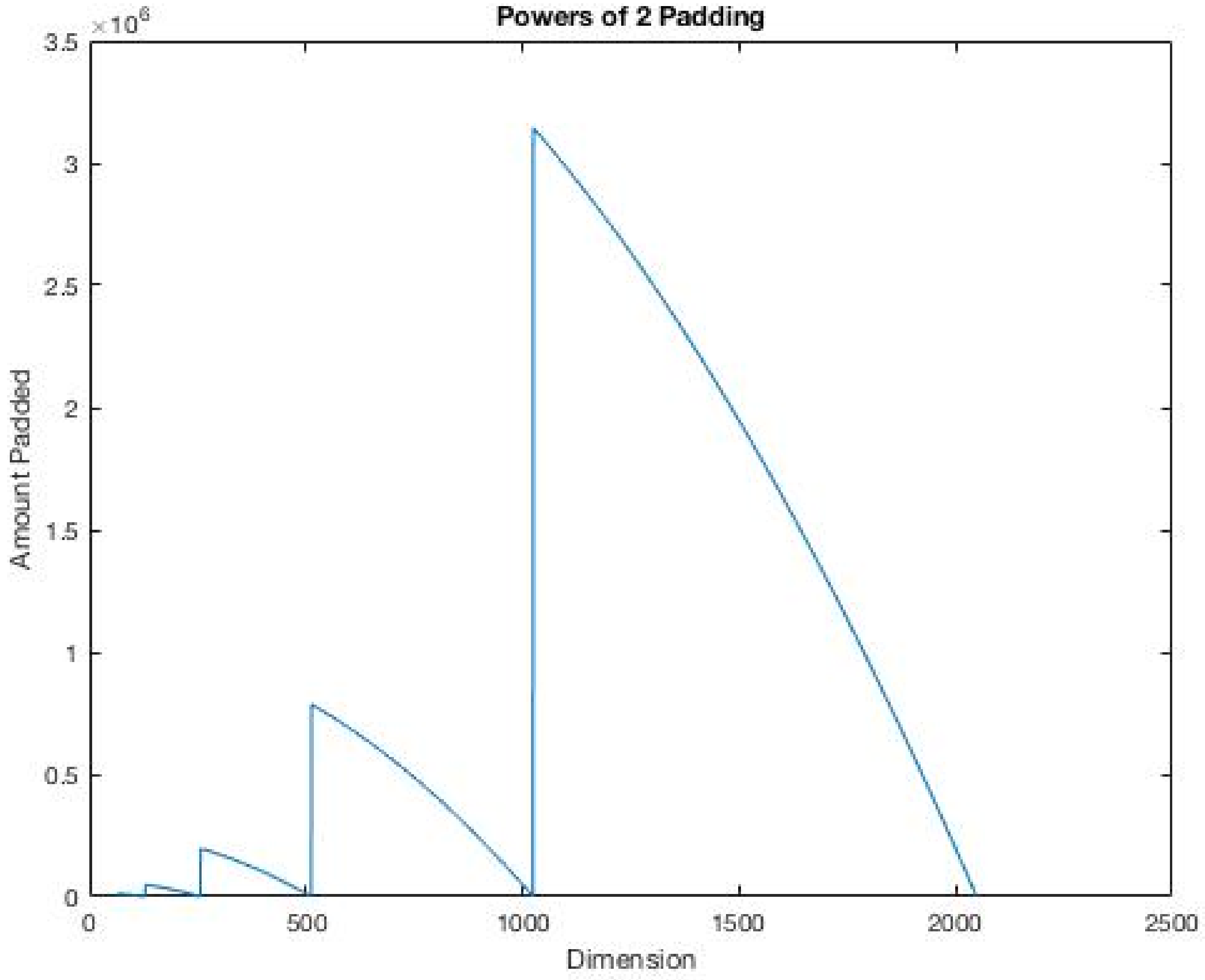

Figure 2: Padding to next power of 2

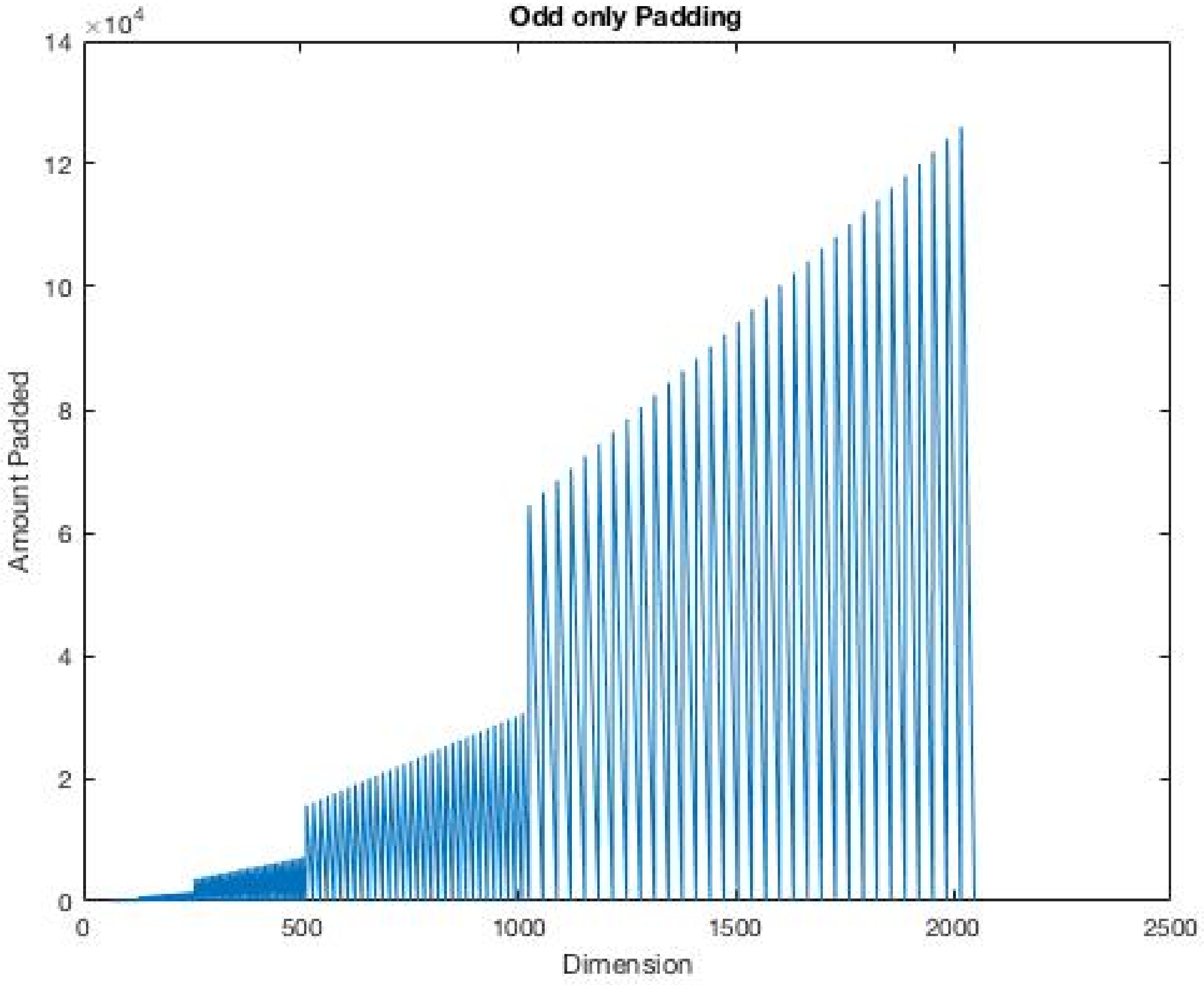

Figure 3: Padding only when odd

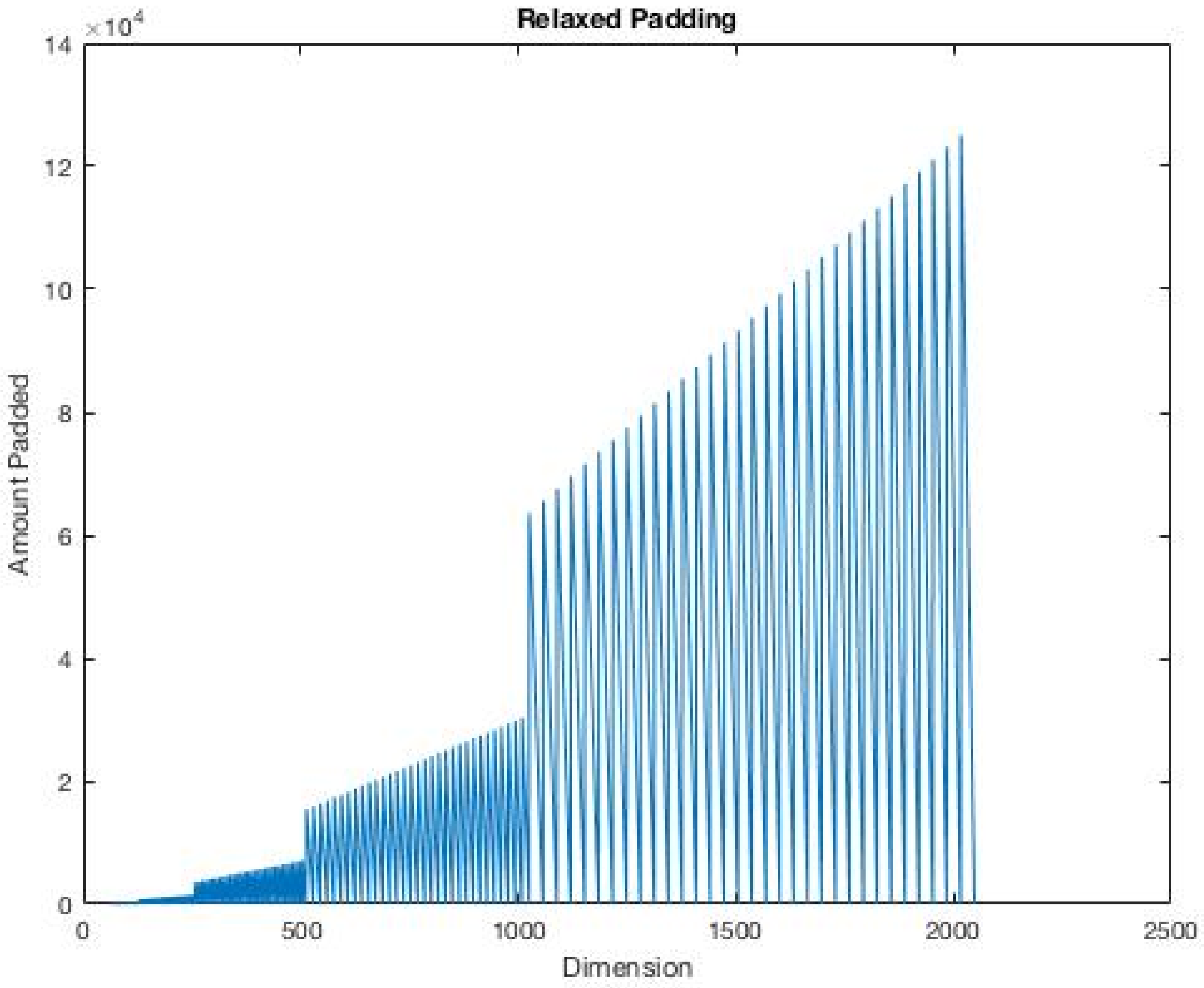

Figure 4: Relaxed padding, until can divide to some $c < n_0$