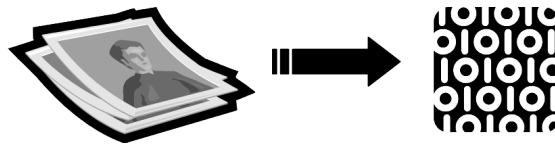


Mini-project: Image Compression



How can you compress an image into a set of bytes? This and other questions are answered in this project. You will work with some **advanced algorithms** during this project.

September 15, 2010

1 Assignment

To compress data is to store as little data as possible without losing any of the original information. One application is to store images as effective as possible. Your task is to implement a function for RunLengthCoding in MATLAB. Compression algorithms similar to the one presented here are for instance used in the .pdf and .gif file formats.

1.1 Specification

Start with a data matrix (*e.g.*, an image). The first four entries in the code vector (the compressed data) represent the size of the matrix:

$$h_1 h_0 w_1 w_0,$$

where the number of rows (the height) of the matrix is $h = 2^8 h_1 + h_0$ and the number of the columns (the width) is $w = 2^8 w_1 + w_0$. (This allows for the size of large matrices to be stored using bytes.) The rest of the code vector contains the matrix in compressed form one row at the time. The rows are compressed according to the following rules:

- A sequence of n identical values is represented with the code block $[(257 - n) x]$, where $257 - n \geq 129$.
- A sequence of n different values render the code block $[(n - 1) x_1 x_2 \dots x_n]$, where $n - 1 \leq 127$.
- The code 128 indicate the end of a row.

As new codes are obtained they are just appended to the code vector. Please, observe that a sequence that is to be coded as one block cannot contain more than 128 values in order for the code to be unambiguous. Divide longer sequences into shorter ones, *e.g.*, interpret 200 ones as 128 + 72 ones.

Example: The compressed form of the matrix

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 5 & 5 & 3 & 2 & 4 & 8 & 8 & 8 \\ 3 & 2 & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 8 & 8 & 2 & 3 \end{bmatrix}$$

is

$$\begin{bmatrix} 0 & 2 & 0 & 13 & 252 & 1 & 255 & 5 & 2 & 3 & 2 & 4 & 254 & 8 & 128 & \dots \\ \dots & 1 & 3 & 2 & 250 & 7 & 255 & 8 & 1 & 2 & 3 & 128 \end{bmatrix}$$

(The effectiveness of the compression improves with more repetitive data.)

1.2 Examination

- Produce a MATLAB function that takes a matrix and compresses it. Let the matrix be the argument to the function and return the compressed code vector. Assume that all values in the matrix are integers between 0 and 255.
- Create a matrix with integers between 0 and 255 and display it as a gray scale image. Use your function to compress the image. An alternative is to use the supplied test data described in Section 2.8.
- Compute the compression ratio for an image, *i.e.*, compare the size of the code vector to the size of the matrix.
- You shall, without problem, be able to run the four lines of code in the end of this compendium.
- If you have the time, investigate how the compression ratio varies with the color depth. The teaching assistant can help you generate test data. An alternative is to generate a random image with two colors, *e.g.*, let 10% of the image be black and the rest white.

2 Suggested Solution Outline

First read though the problem specification and the suggested solution outline and try to understand how it divides the problem into smaller subproblems, each of which is easier to solve than the original problem.

The steps below provide one way to split the main programming problem into logical subproblems. The subproblems are deliberately chosen so that it is possible to test the code before continuing with the next step. Do take these opportunities! Following the suggestions closely can result in many small functions solving small parts of the problem, however, it may not be the best, most efficient, nor smartest way to call these small functions to solve the whole problem. Some of the smaller functions (all?) are better suited to be pasted into the code where needed.

Note that this solution outline is intended for students with no or little previous programming experience. A skilled programmer would probably, based on

previous experience, divide the problem into different and/or fewer parts. If you consider yourself a skilled programmer and think that you have a better solution to the problem you are allowed to solve it your way. However, if you need help, the teaching assistant has more experience of the outlined solution.

2.1 Preallocate Memory

Dynamically expanding matrices in MATLAB can be very costly, hence it is important to preallocate memory for the code vector in order to achieve efficient MATLAB code. That is, create a row vector of zeros, *dest*, large enough to at least hold the compressed matrix. To keep track of where in the code vector to write information introduce a new variable, *destind*, to point to the next free element in the vector. When data is written to the code vector *destind* should increase to point out the next free element. To code a sequence of 5 ones, set *dest(destind)* to 257 - 5, increase *destind* with 1, set *dest(destind)* to 1, before finally increasing *destind* by 1 one more time.

Once the whole matrix is compressed the code vector occupy index 1 through *destind* - 1 in *dest*. It is this submatrix that is the compressed representation of the matrix.

It remains to decide how much memory should be preallocated for the code vector. In a worst case scenario every value is represented as a separate code block at the cost of two values. Furthermore, each row end needs a 128 and the size encoding requires another four positions. Hence, an upper bound on the size of the code vector is $2 \times width \times height + height + 4$.

To practise the preallocation technique, write a function with three numbers as arguments, allocates memory for 10 values, writes the three numbers to the matrix, and finally returns part of the memory where the numbers are stored. Example:

- The arguments (17, 8, 2) should return [17 8 2].

Write another function that returns an upper bound on the size of the code vector given the matrix to compress as argument. Some example output from one such function is given below (note that the limits used below are tighter than the one discussed above).

- The matrix `zeros(1, 1)` should yield *at least* 7.
- The matrix `zeros(1, 100)` should yield *at least* 139.
- The matrix `zeros(1, 500)` should yield *at least* 672.
- The matrix `zeros(100, 1)` should yield *at least* 304.
- The matrix `zeros(17, 43)` should yield *at least* 1007.

2.2 Compress a Homogeneous Sequence

Now try to solve the problem to compresses a sequence (represented as a row vector) of at least two identical values. Assume that the sequence ends with

another number (disregard it for now), or the end of the matrix, and that the sequence is shorter than 129 numbers so that splitting the sequence is not needed.

Write a function to solve the problem. First, preallocate as much memory as you will need according to Section 2.1. Use a variable, c , to keep track of which column in the matrix should be studied next. If you compare the value in column c to the value in column $c - 1$ this allows you to use an easier test for the end of the row, than with a look ahead approach. Two examples to use for testing:

- [1 1 1 1 1 5] should yield [252 1].
- [1 1 1 1 1] should yield [252 1].

2.3 Compress a Heterogeneous Sequence

The next step is to compress a sequence of at least one different numbers. Assume that the sequence ends in two identical numbers (to be handled later) or the end of the row. A difficulty here is that the length of the sequence is stored first in the code block. One way to handle this is to save the value of *destind* (described in Section 2.1) in another variable and then increase the value by 1. (Right now *destind* is always 1 to start with, but do not use that.) After that, copy the sequence of different values to the code vector while determining the length of the sequence. When the end of the sequence is found return to the beginning of code block and enter the length code using the saved index.

Unfortunately, the end of the row test is a little trickier than for homogeneous sequences. It is necessary to look one value ahead to detect that the sequence ends in two identical values. Therefore the last value in a row needs special attention. It is up to you if you consider handling this in an optimal way is worth the extra effort.

Some examples:

- [3 2 4 5 5] should preferably yield [2 3 2 4], or else [3 3 2 4 5].
- [3 2 4] should yield [2 3 2 4].
- [3 2 2] should preferably yield [0 3], or else [1 3 2].

The better compression method is used in the subsequent examples, and in the problem specification above.

2.4 Distinguish Between Different Sequences

Combine the results from the two special cases above, homogeneous and heterogeneous sequences, to a function handling both types of sequences and rows with width one. Test the function with previous examples as well as with

- [3] should yield [0 3].
- Make sure that it works for the examples in Sections 2.2 and 2.3 as well!

2.5 Compress a Whole Row

Write a function that compresses a whole row. For now leave out the matrix size and the 128 to indicate end of row. Note that long sequences can be handled easily by making sure no too long sequences are detected. (No sequence may be longer than 128 values.)

Some useful examples:

- `[1 1 1 1 1]` should yield `[252 1]`.
- `[1 1 1 1 1 5 5]` should yield `[252 1 255 5]`.
- `ones(1, 129)` should yield `[129 1 0 1]`.
- `[3 2 4]` should yield `[2 3 2 4]`.
- `[1 1 1 1 1 3 2 4 5 5]` should yield `[252 1 2 3 2 4 255 5]`.
- `[1 1 1 1 1 5 5 3]` should yield `[252 1 255 5 0 3]`.
- `[1 1 1 1 1 5 5 3 2 4]` should yield `[252 1 255 5 2 3 2 4]`.
- The vector `repmat(135, 1, 300)` should only return values larger than 128. Make sure the result is reasonable too.
- The vector `repmat([1 2 3], 1, 100)` should only return values smaller than 128. Make sure the result is reasonable.

2.6 Numbers in Base $2^8 = 256$

The functions `mod` and `floor` are useful to represent matrix size with values less than 256. Write a function with one argument x ($0 \leq x < 256^2$) that returns a vector with two values, `[y_1 y_2]`, such that $x = 2^8 y_1 + y_2$.

The math is not supposed to be the difficulty here so if you have not figured out how to do it in few minutes ask the teaching assistant for help.

Some values to test your function with:

- 17 should yield `[0 17]`.
- 256 should yield `[1 0]`.
- 260 should yield `[1 4]`.
- 530 should yield `[2 18]`.

2.7 Add Matrix Size and Row Ends

Combine the results from above to a function that given an matrix with one row produces a complete code vector. The first four elements code vector represents the matrix size, and the last value should be 128 to indicate the end of the row.

Test your function with

- `[1 1 1 1 1]` should yield `[0 1 0 5 252 1 128]`.
- The vector `repmat(135, 1, 530)` should give a result starting with 0, 1, 2, 18 and ending with 128.

2.8 Compressing Several Rows

It is finally time to compress a matrix with several rows. Start by introducing a variable, r , to indicate the row being processed. Initialize r to 1, and change all accesses to the source matrix so that r is used as row index. Test the function with some old test cases to make sure it still works.

Simply introducing a loop over r where r goes from 1 to the number of rows in the matrix should produce a working function. All previous code except for code handling preallocation and size encoding should be put inside the loop.

Compressing medium sized image matrices should be no problem if your function is fairly efficient. Two image matrices, as well as a function to decompress code vectors, are provided to help you test the compression function. To get access to the images and the function use the following command

```
>> initcourse TSRT04
```

in MATLAB to add them to the MATLAB path. The function `snowboard` provide the images. (Read the online help for information on how to use it.) Display the image in gray scale and investigate the effect changing the color depth has.

```
>> colormap('gray'); imagesc(snowboard(256))
```

Use the function `tsrt04decompress` to decompress the code vectors your function produce. (See the online help for usage.) Choose an image, and compress and decompress it. The decompressed image should be identical to the original image. The code below can be used to test this (assuming you function is called `compress`):

```
>> snowboardImage = snowboard(12);
>> comp = compress(snowboardImage);
>> decomp = tsrt04decompress(comp);
>> all(all(decomp == snowboardImage))
```

```
ans =
```

```
1
```

```
>>
```