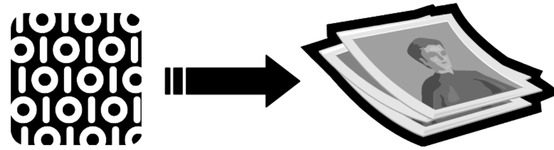
TSRT04 – INTRODUCTION TO MATLAB

Mini-project: Image Decompression



How can you translate a set of bytes into an image? 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 RunLengthDecoding in MATLAB. This decoder should be able to restore a matrix stored with the RunLengthCode defined below. Any errors in the code vector (the data on compressed form) should be reported. Compression algorithms similar the one presented here are for instance used in the .pdf and .gif file formats.

1.1 Specification

The code vector representing a matrix (*e.g.*, an image) is defined by the following rules:

- The four first positions in the code vector denote the size of the decompressed 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 data in the matrix follows after the size in the code vector. The rows are compressed separately, and the code for every row ends with the code 128. Each row is coded using one or more code blocks as described next.
- If the first value in a code block, n, is in the interval 0 to 127 the next $n+1$ values in the code vector should be copied to the current location in the output matrix.
- If the first value in a code block, n, is in the interval 129 to 255 the next value is to be repeated $257 - n$ times in the output matrix starting at the current location.

Example: The compressed vector

$$\begin{matrix} [0 & 2 & 0 & 13 & 252 & 1 & 255 & 5 & 2 & 3 & 2 & 4 & 254 & 8 & 128 & \ldots \\ & & & & \ldots & 1 & 3 & 2 & 250 & 7 & 255 & 8 & 1 & 2 & 3 & 128] \end{matrix}$$

represents the matrix

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

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

The following errors may occur in the code vector (make sure your code handles them):

- The code 128 appears unexpected.

- The decompressed data does not fit in the matrix.
- The code 128 was expected but was not found.
- The code vector contains, compared to the given size, too many or too few rows.

1.2 Examination

- Produce a MATLAB function that takes a code vector representing an image and decompresses it. Let a code vector be the argument to the function and return the image matrix. Assume that all values in the matrix are integers between 0 and 255. Errors in the compressed vector must render an error message indicating the type of error and where it occurred.
- Decompress a code vector into a gray scale image matrix. (Each element in the matrix represent the color of a pixel in the image. 0 represents a black pixel and 255 a white one.) Display the image. You may use the test data provided, see Section 2.6.
- You shall, without problem, be able to run the four lines of code in the end of this compendium.
- Compute the compression ratio for an image, *i.e.*, compare the size of the code vector to the size of the matrix.

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 Preallocating Memory

As a first step, create an empty matrix (*i.e.*, a matrix with only zeros) to hold the decompressed data. The four first values in the code vector gives the size of this matrix. Write a function that takes a code vector as the only argument and returns a correctly sized matrix. For now, ignore the rest of the code vector.

It is important to keep track of where to find the next value in the code vector during the decompression phase, therefore introduce a variable, k, to keep track of the next position to access. Initialize k to 1 and increase k by one each time a value is read at index k in the vector. After extracting the matrix size, k should be 5 to point to the start of the compressed data.

Before continuing to step two test your function with the examples below to make sure it works.

- The input [0 1 0 10] should yield a 1×10 matrix.
- The input [2 8 1 72] should yield a 520×328 matrix.

2.2 Decompress a Homogeneous Sequence

Now append the code for a sequence identical values (*i.e.*, on the form $(257 - n), x)$ to the four values representing the matrix size. Write a function that decompresses the sequence and inserts the result at the beginning of the output matrix.

It is as important to keep track of where to insert the decompressed values as to know where to read the next value in the code. Therefore, introduce a variable, r, to indicate the active row in the output matrix. Let r be 1 for now. Introduce another variable, c, to point to the column to which data should be written. Initialize c to 1, and increase c by one each time a value is written to the cth column of the output.

Some examples:

- [0 1 0 7 254 9] should yield [9 9 9 0 0 0 0].
- [0 1 0 7 200 9] should produce an appropriate error message.
- [0 1 0 7 254] should produce an appropriate error message.

The better error message produced, the easier it is to find and correct any bugs in your code.

2.3 Decompress a Heterogeneous Sequence

Write a function that decompresses the code for different values (*i.e.*, on the form $(n - 1), x_1, x_2, \ldots, x_n)$. Some examples to help you test this functionality:

- [0 1 0 7 2 6 19 27] should yield [6 19 27 0 0 0 0].
- [0 1 0 2 2 6 19 27] should produce an appropriate error message.

- [0 1 0 7 2 6 19] should produce an appropriate error message.

2.4 Recognize the Two Code Sequences

Combine the results from the two previous steps to a function handling both types of code blocks, *i.e.*, both homogeneous and heterogeneous sequences, following the first four numbers in the code vector.

2.5 Decompress a Whole Row

Now write a function to decompress a whole row in the output matrix. There are two different ways to determine that a whole row has been decompressed; either compare the value of c to the width of the output matrix, or detect when the value 128 ends a row. The error messages will probably look a bit different depending on which criteria is used.

Some example to test your code:

- [0 1 0 7 2 6 19 27] should produce an appropriate error message.
- [0 1 0 7 2 6 19 27 128] should produce an appropriate error message.
- [0 1 0 7 2 6 19 27 253 12 128] should yield [6 19 27 12 12 12 12].
- [0 1 0 7 2 6 19 27 253 128 128] should yield [6 19 27 128 128 128 128].
- [0 1 0 7 2 6 19 27 240 128 128] should result in an appropriate error message.

2.6 Decompressing Several Rows

Finally, decompress a whole matrix one row at the time. Use r to indicate which row is currently processed. Initialize r to 1 and increase it by one each time another row is completed. Stop when the last row in the matrix has been decompressed. This introduces new possible errors in the code vector; too many and too few rows.

Before testing your function on some larger compressed matrices, use the examples below to make sure you get the error messages you expect.

- [0 3 0 7 2 6 19 27 253 12 128] should produce an appropriate error message.
- [0 1 0 7 2 6 19 27 253 12 128 2 6 19 27] should produce an appropriate error message.
- [0 2 0 4 3 1 2 75 6 128 3 7 75 6 7 128] should yield $\begin{bmatrix} 1 & 2 & 75 & 6 \\ 7 & 75 & 6 & 7 \end{bmatrix}$
- [0 1 0 7 2 6 19 27 253 12 128 2 6 19 27 253 12 128] should produce an appropriate error message.

Decompressing medium sized image matrices should be no problem with a fairly efficient function. Two image matrices, as well as a function to compress them with, are provided for you to test your decompression 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 image. 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 `tsrt04compress` to compress the images (see the online help). 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 `decompress`):

```
>> snowboardImage = snowboard(12);
>> comp = tsrt04compress(snowboardImage);
>> decomp = decompress(comp);
>> all(all(decomp == snowboardImage))

ans =

     1

>>
```