Assignment 1

Image decompressor
Due: 9:00AM, Monday 11th April

The first two assignments for FIT3042 this year form a single project with two phases. In the first phase, you will write a decoder for a simple video format called `rleplay` – that is, you'll extract each frame from the image. In the second phase, you will write a player that can take this sequence of decompressed frames and display them to the screen.

Requirements

For Assignment 1, you must write a program called `rledecode` that decodes an `rleplay` file and produces a sequence of images in PPM format.

`rledecode` must take two *command line arguments*: the first is the name of the `rleplay` file that is to be decoded, and the second is either a dash (i.e. minus sign) or a prefix for the files that are to be output.

If the second argument is *not* a dash, each output file should be named *prefix-xxxx*.ppm, where *prefix* is the prefix specified on the command line and *xxxx* is a 4-digit integer indicating which frame it is. The first frame should be 0001, the next 0002, and so on.

If second argument is a dash, the PPMs should be sent to `stdout`, separated by the integer (not string!) -1. This will make it easier to write the video player in Assignment 2 as it will only have to accept pixel maps on `stdin` – you will be able to pipe the output of Assignment 1 directly into Assignment 2 and it should run.

As well as this basic functionality, you must implement two *optional*[1] command line arguments:

`--scale` *scalefactor* – scales each image by a factor of *scalefactor* both horizontally and vertically, by using bilinear interpolation. The *scalefactor* must be a positive integer; note that a *scalefactor* of 1 would do nothing.

`--tween` *tweenfactor* – uses bilinear interpolation to insert *tweenfactor* frames between each pair of frames in the input. The *tweenfactor* must be a positive integer.

Therefore, this command line:

`rledecode video.rle MyVideo --scale 2`

would decode the video `video.rle` into files called `MyVideo-0001.ppm`, `MyVideo-0002.ppm`, and so on, and would scale each frame by a factor of two before outputting to PPM. Meanwhile, this command line:

`rledecode cats.rle - --tween 1`

would decode the video `cats.rle` and output the resulting PPMs to `stdout`, inserting one tweened frame between each adjacent pair of frames in the input.

Note that the options are introduced by two dashes (--). This option format is compatible with the GNU `getopt_long()` library function, so you may use that to help you parse the command line if you want to. You'll find more information about this function in its manual page.

If you would like more information about pixel maps, run-length encoding, and bilinear interpolation, read on. The remainder of this document includes useful reference material and links to help you complete this assignment.

---

[1]"Optional" doesn't mean you can choose not to implement them. It means the user doesn't have to use them.

Pixel maps

As you probably know, video images are made up of a collection of coloured dots or pixels. To represent these digitally, we use a sequence of numbers representing the intensity of red, green, and blue colouration of each pixel - by convention, that order is typically used. For most video files, 8-bit integers are used to represent colour intensity, where 0 (the smallest value that can be represented as an 8-bit unsigned integer) represents the lowest intensity, and 255 the highest. We use 8 bits for each of the three colours, so we end up using 24 bits per pixel.

This chart shows some example combinations of red, green, and blue, and the colours they create:

Color values	Colour
(0,0,0)	(black)
(0, 255,0)	(green)
(128, 128, 0)	(vomit)
(255, 255, 255)	(white)

To represent an entire image in a file, all we need to do is store a sequence of colour values in a defined order, and supply a little bit of extra information about the image size and shape (either explicitly or implicitly). One very simple format for doing so is the Portable PixMap format, or PPM, used in the widely-distributed `netpbm` image processing tools. There are two versions of PPM - one encodes the numbers directly, the second writes the numbers as ASCII text and is known as *plain PPM*. A very small example plain PPM is below[2]. The first line is a so-called *magic number* that identifies the file as a plain PPM file, the second line contains a comment, the third line records the width and height of the image in pixels, and the last line contains the maximum value for a channel (note that this example uses 15 rather than the more common 255).

Pixels are represented in the file left-to-right, then top-to-bottom order.

```
P3
# feep.ppm
4 4
15
 0  0  0    0  0  0    0  0  0   15  0 15
 0  0  0    0 15  7    0  0  0    0  0  0
 0  0  0    0  0  0    0 15  7    0  0  0
15  0 15    0  0  0    0  0  0    0  0  0
```

We could write a video player that simply reads and displays a sequence of PPM files sufficiently quickly. However, there are two problems with this: first, it's kind of inconvenient to have a video spread out over thousands, possibly hundreds of thousands of individual files, and secondly the PPM format takes up a lot of space just to store a single image.

RLE image compression

As far as the second problem goes, we can reduce this by taking advantage of a common property of image data - it is very repetitive. We can take advantage of this property using a simple scheme called run-length encoding.

---

[2]You'll be using standard PPM, using integers rather than ASCII strings, but this example is in plain PPM so that you can read it.

As Wikipedia explains:

> For example, consider a screen containing plain black text on a solid white background. There will be many long runs of white pixels in the blank space, and many short runs of black pixels within the text. Let us take a hypothetical single scan line, with B representing a black pixel and W representing white:
>
> WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWW
>
> If we apply the run-length encoding (RLE) data compression algorithm to the above hypothetical scan line, we get the following:
>
> 12W1B12W3B24W1B14W
>
> This is to be interpreted as twelve Ws, one B, twelve Ws, three Bs, etc. The run-length code represents the original 67 characters in only 18.
>
> (https://en.wikipedia.org/wiki/Run-length_encoding#Example)

The run-length encoding scheme we will use for our files is based on this principle, but is slightly more sophisticated: the Variant Packbits scheme designed by Michael Dipperstein.

Video file format

Our video file format is unique. Like PPM, it contains some header information which describes the dimensions of the frames that make up the images. It then contains data representing each individual frame of the video.

To assist in the run-length encoding, within each frame we will store the red channel for the whole frame, then the green, then the blue. You should apply the PackBits algorithm to the data for each channel, for each frame, independently.

A Backus-Naur form[3] description of the file format is presented below:

```
<Video-file> ::= <header> <frames>
<header> ::= <id-string> <properties>
<id-string> ::= "RLEPLAYv1\n"
<properties>::= <width> " " <height> "\n"
<width>::= <pos-int>
<height>::= <pos-int>
<pos-int> ::= /* a positive integer rendered as an ASCII string */
<frames> ::= <frame> <frames> | "E"
<frame> ::= "K" <key-frame-data>
<key-frame-data> :: <red-frame-data><green-frame-data><blue-frame-data>
<red-frame-data> :: <packbit-data>
<green-frame-data>:: <packbit-data>
<blue-frame-data>:: <packbit-data>
<packbit-data>:: /* binary unsigned integers with
                   variant PackBits compression applied, see
                   http://en.wikipedia.org/wiki/PackBits and
                   http://michael.dipperstein.com/rle/index.html*/
```

<red-frame-data> stores the information for the red channel, <green-frame-data> for the green channel, and <blue-frame-data> for the blue channel. Each channel has an 8-bit value

---

[3]For those who haven't studied grammars, Backus-Naur form is a way to describe the syntax of a computer language. Kernighan and Ritchie's book on C defines the syntax of C using Backus-Naur form. If you're not familiar with it, there's an explanation at https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form.

for each pixel.

Each channel's data is stored in row-major order, so the first value corresponds to the top left of the image, the second value the pixel immediately to the right, and so on, until you start the next row.

As noted, each channel's data is compressed using *Variant PackBits* compression by Michael Dipperstein. You'll find the algorithms for Variant Packbits compression and decompression at (`http://michael.dipperstein.com/rle/index.html`).

There are a couple of sample `rleplay` files, and in one case the PGM files from which it was produced, on Moodle. There is also an Ubuntu package containing a tool, `ppmtorle`, which can compress a series of files in PPM format (all of the same size, and with a `maxval` of 255) into a single `rleplay` file. To use it, you must first create a series of PPM files in the same directory using the following naming convention:

`prefix-xxxx.ppm`

where `prefix` is a prefix common to all the files, and `xxxx` is a 4-digit integer from 0000 up to (potentially) 9999. The files must all be the same size.

To make a `rleplay` file, run `ppmtorle` as follows:

`ppmtorle prefix outfile min-index max-index`

where `outfile` is the name for the output file, `min-index` is the number of the first image in the sequence you wish to convert, and `max-index` is the last.

Bilinear interpolation

Lets say you have a 32×32 pixel bitmap for a desktop icon, you do not have a 64×64 pixel version, and you need to scale up what you have to 64×64 pixels. How would you go about it?

Lets look at a 2×2 to 4×4 scaling example. The numbers represent grayscale intensities.
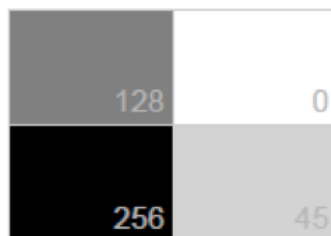
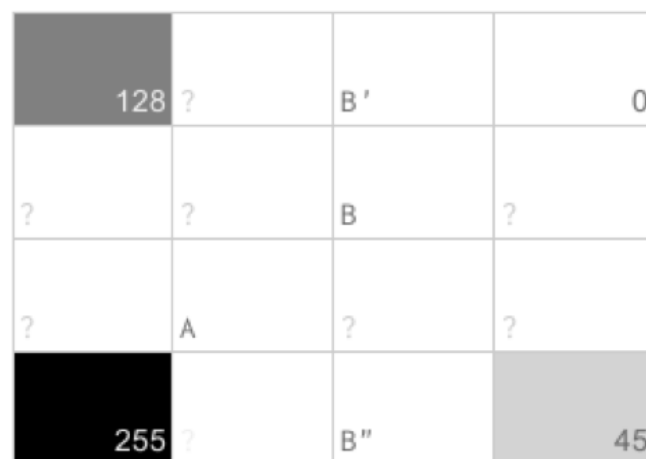

Figure 1: Original image   Figure 2: Target size

How do we fill in these pixels? One way of doing this is to retain our original pixels and "fill in the gaps". Lets design a strategy for filling in the pixels. This means choosing a mathematical function that will let us compute which colour value to put into each new pixel, based on the existing coloured-in pixels around it. This kind of strategy is called *interpolation*.

Lets look at pixel B in the right-hand image. It would make sense to weight B highest on the original pixel closest to it, and weight other pixels based on distance and direction.

So what sort of function should we use to determine the value for intermediate pixels? If we use a linear function, values will be influenced uniformly with increasing distance in a given direction. On the other hand, if we wuse a quadratic, cubic, or other higher order function, the influence of our original pixels drops off quicker with increasing distance. This can be useful- it could produce sharper images – but it would be harder to compute.

For this assignment, we will start with a linear function – that is, we will use *linear interpolation*. We will perform a linear interpolation in each dimension, once for x, once for y. Linear interpolation in two spatial dimensions is called *bilinear interpolation*.

So, we want B to depend quite heavily on our 0 and 45 intensity pixel (which we will now abbreviate as $0i$, $45i$, etc.) in the horizontal direction, and a little less so on the $128i$ and $255i$ pixels. In the vertical direction, we want it to depend more on the $0i$ and $128i$ pixels than the $45i$ and $255i$ pixels. Since this is a linear interpolation, we can define the influence of each of the pixels in the horizontal direction to be in inverse proportion to their horizontal distance to the pixel B.

Lets first interpolate in the horizontal direction.

We can think of B' as the horizontal projection of B along the $128i - 0i$ axis. B' would receive a horizontal value of $\frac{1}{3} * 128i + \frac{2}{3} * 0i = 128/3i = 42.6667i$. B'' would receive a horizontal value of $(\frac{1}{3} * 255 + \frac{2}{3} * 45)i = 115i$.

Now lets bring vertical values into the picture. B' is twice as close to B as B'' is. So, using our influence rule, the value of B should be $\text{floor}(\frac{2}{3} * 128/3 + \frac{1}{3} * 115)i = 66$. (The choice of floor over rounding or ceiling is arbitrary).
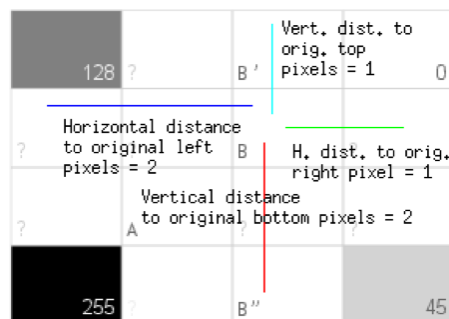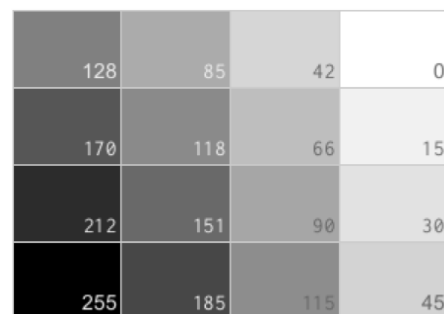

Figure 3: Vertical and horizontal distances

Figure 4: Interpolated image

Note that we need to scale our image by an integral factor, which can be odd. So a 5×3 image with $3\times$ scaling would become 15×9; how do we space our original pixels? If we were to map $(1, 1)$ and $(5, 1)$ to $(1, 1)$ and $(15, 1)$ respectively, the intervening original pixels would sit at non-integer coordinates, and worse, the spacing for the horizontal and vertical axes would be different.

We don't really want to change the spacing between pixels, so we assume a unit distance between pixels along both directions (though in reality pixels are slightly wider than they are tall). This means we will end up with scaled images that might be very slightly smaller in width or height or both in terms of pixels than you'd get if you simply multiplied. Our 5×3 bitmap would scale to a 13×7 bitmap. And this is fine!

Here is a simple algorithm for interpolation by integer factor n. ASSUMPTION: Our image is at least 2×2 in size. If it's not, we will not be able to interpolate in two dimensions.

Say you are scaling a $w \times h$ image by factor n.

1. ~~Create~~ an empty bitmap of size $n * (w - 1) + 1 \times n * (h - 1) + 1$.

2. ~~Populate~~ the bitmap with our original pixels. A pixel located at (x, y) on the original bitmap will be located at $(n * (x - 1) + 1, n * (y - 1) + 1)$ on the new bitmap. Let us call the set of original pixels with new coordinates O_P.

3. ~~Pick~~ the pixel at the bottom-right corner of the image $(n * (x - 1) + 1, n * (y - 1) + 1)$. Call this the *pivot pixel*. Note: We will process the whole row and column defining one pivot at a time. The number of pivots we need is the length of the original diagonal in pixels.

4. For all pixels (x, y) such that $x = P_x$ or $y = P_y$, where (P_x, P_y) is in set O_P, we will need to interpolate all pixels in the box defined by original pixels now located between $(x, y), (x - n + 1, y), (x, y - n + 1), (x - n + 1, y - n + 1)$.

    (a) Let us call this set of interpolated pixels to be determined pixel set IP. Further, we will denote the pixels as P^{BR}, P^{BL}, P^{TR}, P^{TL} respectively, for the Bottom-Right, Bottom-Left, Top-Right, and Top-Left pixels.

    (b) For each pixel P in I, compute the horizontal linear interpolation from the top two pixels of the box, P^{TL} and P^{TR}.

    First, we find the horizontal intensity effect from the top left pixel: multiply (intensity value of P^{TL}) by [(the horizontal distance of P to P^{TR}) divided by (the distance between the two top pixels P^{TR} and P^{TL})]

    Next, we find the effect from the top right pixel: multiply (the intensity value of P^{TR}) by [(the horizontal distance of P to P^{TL}) divided by (the distance between the two top pixels P^{TR} and P^{TL})]

    (c) Add the values from step 4b to get total horizontal effect from the top pixels

    (d) Calculate the horizontal linear interpolation for the bottom pixels

    (e) The intensity of our pixel is then [(horizontal effect of top pixels) * (distance of pixel P to the line joining bottom pixels)] + [(horizontal effect of original bottom pixels) * (distance of pixel P to the line joining original top pixels)]

5. Go back to step 4 and repeat by setting the new pivot to $(P_x, P_y) = (P_x - n + 1, P_y - n + 1)$. Continue until $P_x \leq 0$ or $P_y \leq 0$. (This condition could be made stronger)

6. Terminate

Notes

It is important that you get the basic functionality working as this will allow you to use your program in Assignment 2. It is less important to implement stretching and tweening, as the focus of Assignment 2 will be displaying the PPMs rather than on interpolation.

Submission instructions

You must submit a single archive file in gzipped tar format (`.tar.gz`) through Moodle. See `info tar` (or consult the internet) for information how to create a gzipped tar archive.

You must include a working `Makefile` with your submission. It should be possible for the marker to build all the executables by changing to the root directory of your submission and running `make` at the command line. The use of `make` and `Makefiles` will be covered in lectures in Week 4.

You should include a `README` file, in text format, that describes (at a minimum):

- your name and student ID number

- how to compile and run the program

- what functionality is and is not supported

- and any known bugs or limitations.

This information is to help your marker build and run your program. That means that it is to your advantage if it is accurate: don't claim that you have implemented something that's not actually there. You'll lose marks for not following the submission instructions. Remember, you want your marker to feel happy and generous!

Marks breakdown

Marks for this assignment are broken down as follows:

- Basic functionality, including functional correctness, memory management, error handling, robustness, security, usability etc.: **8 marks**

- Extended functionality (scaling and tweening): **5 marks**

- Code quality (breakdown into functions, use of headers, readability, adequacy of comments, etc.): **5 marks**

- Adherence to submission guidelines; use of `make`: **2 marks**