The `trimmer` program

Sal Elder

November 13, 2021

1 Introduction

The purpose of this program is to shorten video files, without giving them the appearance of having been cut or sped up. For example, if you gave the program a video in which Alice crosses a room, and then Bob crosses the same room, then the program should output a video of Alice and Bob crossing the room together.

At first this may sound impossible, but it is actually similar to an existing process called "seam carving" [1]. The goal there is to shrink an image along the x-dimension (say) by some amount without losing or distorting important image features. So, for example, you can remove parts of the sky but not parts of people's faces.

Seam carving has also been applied to videos, by a natural extension of the concept from 2d to 3d [2]. That is, it has been used to retarget the width or height of videos. Here, I'm going to try to retarget the duration of a video instead. I haven't seen examples of that done with seam carving, although I have seen something similar with a different technique [3].

2 Algorithm overview

Here I give a summary of the seam-carving algorithm for images [1], in order to motivate the version for changing video duration.

First an energy $U(x, y)$ is assigned to each pixel, corresponding (ideally) to the importance of that pixel. Different "energies" can be used, such as the L_1 or L_2 norm of the image gradient. Then vertical "seams," $x(y)$, are removed from the image one at a time until it reaches the desired aspect ratio. A seam is a minimum-energy connected path of pixels from the top of the image to the bottom, containing exactly one pixel per row. An example of a seam is shown in Fig. 1(a).

2.1 Dynamic programming algorithm

A seam can be found with a simple dynamic programming approach. Define $\text{OPT}(x, y)$ as the minimum energy for a seam from $y = 0$ to $y = y$, which is constrained to include the point (x, y). Given this definition, it's easy to figure out $\text{OPT}(x, y+1)$: we simply choose the smallest of $U(x, y+1) + \text{OPT}(x', y)$ for neighboring x'. This is illustrated in Fig. 1(b). At the end, this gives us the minimum energy, but we can recover the seam itself in the usual way. When we select the point (x', y) to connect to $(x, y+1)$, we simply keep track of the x' we chose in a separate array, say $\text{PREV}(x, y+1) = x'$.

2.2 3d version

To shrink a 3d domain along its z-axis, we want to remove a continuous minimum-energy surface $z(x, y)$. Unfortunately, you can't find one with dynamic programming, so you have to use a graph cut algorithm. Moreover, to get it to run reasonably fast requires some tricks. Alternatively, you can reduce the problem to carving 1d seams, either by

- applying seam-carving separately to each z-plane
- taking the "maximum projection" $U(x, z) = \max_y U(x, y, z)$ and carving out a surface $z(x, y) = z(x)$.

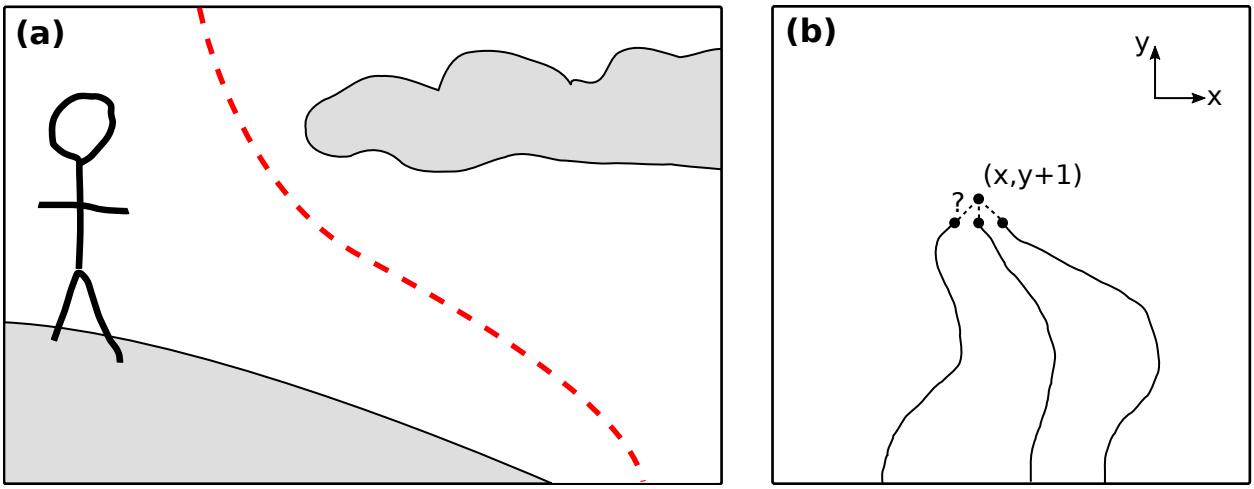

Figure 1: **Seam carving. (a)** An example of a seam. **(b)** Illustration of a dynamic programming algorithm used to find seams. Points represent pixels.

All of these variants are described in Ref. [2].

I don't want to bother with the graph cut method, so I'll use one of the reductions to carving 1d seams from 2d images. I've tried the first type previously, but it produces very prominent "slices" in the video. An example frame is shown in Fig. 2(a). (This is analogous to the "jittering" or "temporal incoherence" described in Ref. [2].) Basically, $z(x, y)$ is continuous along one axis but totally discontinuous along the other.

Instead, the program I show here will use the "max projection" version. A sample frame is shown in Fig. 2(b). Note for typical inputs, I expect projecting along y to work better than projecting along x. That's because different events tend to happen on the left or right of a frame rather than the top or bottom.

3 Program outline

Here is the skeleton for the program. (Small numbers indicate the page number that a code block is defined on or added to.)

2 ⟨* 2⟩≡

```
⟨Include headers 4b⟩
using namespace std;

⟨Declarations 4c⟩
⟨Definitions 6a⟩

int main(int argc, const char * argv[]) {
        ⟨Parse arguments 3⟩
        ⟨Load video 7a⟩
        ⟨Apply algorithm 7b⟩
        ⟨Output video 7c⟩
        return 0;
}
```

Defines:
`main`, never used.

Figure 2: **Continuity of 1d seams.** (a) Example output when seam carving is applied separately to planes cut through the video volume. Seams are continuous as a function of x but not y. (b) Example output when seam carving is applied to the max-energy projection along the y axis. Seams are continuous as a function of both x and y. This is the method described in this document.

4 Input/output

Let's decide on the usage pattern for this program. Rather than encode and decode video myself, I'd rather pipe to and from ffmpeg. That is, I'll assume raw pixel bytes are coming in on `cin`, and I'll send the resulting pixels to `cout` at the end. The program itself will accept arguments for the video width, video height, and fraction of time to keep. For example, the user may call the program like this:

```
ffmpeg -i myInput.mpg -pix_fmt rgb24 -f rawvideo -
| ./trimmer 360 240 0.5
| ffmpeg -f rawvideo -pix_fmt rgb24 -s 360x240 -r 24 -i - output.mpg
```

Some notes on `ffmpeg`. Options apply to the next input or output specified. The `-r` flag is the framerate. The `-` means to pipe to/from standard output/input. It's syntactic sugar for `pipe:`, i.e. the unnamed pipe.

Note that `cin >>` and `cout <<` do formatted input and output, so `cin >>` will ignore "whitespace," and `cout << '\n'` outputs *two* bytes on Windows. We can use `cin.get` and `cout.put` to do unformatted input and output of a single character at a time, but I've found doing it that way is much slower than using a buffer with `cin.read` and `cout.write`.

As an aside, the test I ran was to pipe pixel data from `ffmpeg` into a "passthrough" program, and pipe that program's output into a file. I either wrote the passthrough program like

```
for(char c; cin.get(c);) cout.put(c);
```

or

```
while(cin.read(buffer, framesize)) cout.write(buffer, framesize);
```

Getting back to the code, first confirm that the user has provided the correct number of arguments.

3 ⟨*Parse arguments* 3⟩≡ (2) 4a▷

```
string expected {"Expected arguments: [width] [height] [fraction kept]\n"};
if (argc != 4) {
        cout << expected;
        return 1;
```

```
}
```

Defines:
`expected`, used in chunk 4a.
Uses `fraction` 4a, `height` 4a, and `width` 4a.

We can use a string stream as an intermediary type between `const char *` and `int` or `double`.

4a ⟨*Parse arguments* 3⟩+≡ (2) ◁3

```
istringstream width_string {argv[1]};
istringstream height_string {argv[2]};
istringstream fraction_string {argv[3]};
double fraction; int width, height;
if (
        !(fraction_string >> fraction) ||
        !(width_string >> width) ||
        !(height_string >> height)
) {
        cout << expected;
        cout << "I couldn't parse the arguments (int int double).\n";
        return 1;
}
if (fraction <= 0 || fraction > 1) {
cout << "I require 0 < fraction <= 1.\n";
return 1;
}
```

Defines:
`fraction`, used in chunks 3 and 7b.
`fraction_string`, never used.
`height`, used in chunks 3 and 5–7.
`height_string`, never used.
`width`, used in chunks 3 and 5–7.
`width_string`, never used.
Uses `expected` 3.

Add the needed libraries for basic IO so far.

4b ⟨*Include headers* 4b⟩≡ (2) 5b▷

```
#include <iostream>
#include <string>
#include <sstream>
```

Next we will store the pixel data coming into the standard input. We can proceed to define a suitable container type `Video` with some basic members we can expect to use.

4c ⟨*Declarations* 4c⟩≡ (2)

```
class Video {
        ⟨Video members 5a⟩
};
```

Uses `Video` 6a.

5a ⟨*Video members* 5a⟩≡ (4c) 7d▷

```
public:
        Video(istream&, int width, int height);
        int length() const; // in frames
        unsigned char& get(int frame, int row, int col, int channel);
private:
        int frames, rows, cols;
        ndarray<unsigned char> pixels;
        ndarray<double> energy;
```

Uses `cols` 6a, `frames` 6a, `get` 6a, `height` 4a, `length` 6a, `ndarray` 5b, `rows` 6a, `Video` 6a, and `width` 4a.

Rather than build one here, I'm simply going to include the definition of my very basic class for n-dimensional arrays. It supports initialization with a given shape, and element access, and that's about it. Perhaps in the future I'll write an improved ndarray as another literate program.

5b ⟨*Include headers* 4b⟩+≡ (2) ◁4b 6b▷

```
#include "ndarray.h"
```

Defines:
`ndarray`, used in chunks 5a, 6a, 8b, 9a, and 11b.

The definitions of these members are straightforward. Note that because we don't know the number of frames when the constructor is called, we have to store the bytes in a temporary container and later transfer them into an ndarray.

6a ⟨*Definitions* 6a⟩≡ (2) 7e▷

```
Video::Video(istream& input, int width, int height)
: rows{height}, cols{width} {
        vector<unsigned char> all_bytes;
        int framesize = width*height*3; // bytes in a single frame
        char* buffer = new char[framesize];
        while(input.read(buffer, framesize)) {
                all_bytes.insert(all_bytes.end(), buffer, buffer + framesize);
        }
        delete[] buffer;
        clog << "Setting up: ";
        // determine how many frames we have
        frames = all_bytes.size() / (3*rows * cols);
        // then put the data into pixels
        ndarray<unsigned char> reshaped {frames, rows, cols, 3};
        int indx = 0; // index into 1d vector
        for (int f=0;f<frames;++f) {
        for (int r=0;r<rows;++r) {
        for (int c=0;c<cols;++c) {
        for (int chan=0;chan<3;++chan) {
                reshaped.get(f,r,c,chan) = all_bytes[indx];
                ++indx;
        }}}}
        pixels = reshaped;
        ⟨Calculate voxel energy 9a⟩
}
int Video::length() const {return frames;}
unsigned char & Video::get(int frame, int row, int col, int channel) {
        return pixels.get(frame, row, col, channel);
}
```

Defines:
  buffer, never used.
  cols, used in chunks 5a and 7–11.
  frames, used in chunks 5a and 8–11.
  framesize, never used.
  get, used in chunks 5a, 7e, and 9–11.
  length, used in chunks 5a and 7.
  rows, used in chunks 5a, 7e, 9a, 11b, and 12a.
  Video, used in chunks 4c, 5a, and 7–9.
Uses height 4a, ndarray 5b, and width 4a.

I used to recalculate the energy every time we removed a frame, but this was the slowest part of the program, so I'm calculating once at the time we load the pixels. I'll define that part later when we get to the seam carving section of the program.
Since I used vector above, I need to include the header for it.

6b ⟨*Include headers* 4b⟩+≡ (2) ◁5b 9d▷

```
#include <vector>
```

Having this class makes the input part of the `main` a one-liner.

7a ⟨*Load video* 7a⟩≡ (2)
```
Video input_video {cin, width, height};
```

Defines:
`input_video`, used in chunk 7.
Uses `height` 4a, `Video` 6a, and `width` 4a.

To reduce the video by the desired amount, we simply shrink it by one frame at a time and repeat as needed.

7b ⟨*Apply algorithm* 7b⟩≡ (2)
```
int frames_to_remove = input_video.length() * (1.0 - fraction);
for (int k=0; k < frames_to_remove; ++k) {
        if (k > 0) cerr << "\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b";
        cerr << "Frame: " << setw(4) << (k+1) << "/" << setw(4) << frames_to_remove;
        input_video.carve_frame();
}
clog << "\n";
```

Defines:
`frames_to_remove`, never used.
Uses `carve_frame` 8b, `fraction` 4a, `input_video` 7a, and `length` 6a.

The last step of the program is to output the pixel data for the reduced-duration video. We simply loop over the indices, since my ndarray doesn't support the range-for yet.

7c ⟨*Output video* 7c⟩≡ (2)
```
input_video.dump_bytes(cout);
```

Uses `dump_bytes` 7e and `input_video` 7a.

7d ⟨*Video members* 5a⟩+≡ (4c) ◁5a 8a▷
```
public: void dump_bytes(ostream &);
```

Uses `dump_bytes` 7e.

7e ⟨*Definitions* 6a⟩+≡ (2) ◁6a 8b▷
```
void Video::dump_bytes(ostream & out) {
        for (int f=0;f<length();++f) {
        //for (int r=0;r<rows;++r) {
        //for (int c=0;c<cols;++c) {
        //for (int chan=0;chan<3;++chan) {
        out.write((char *)&pixels.get(f), rows*cols*3);
        //      out.put(pixels.get(f,r,c,chan));
        }//}}}
}
```

Defines:
`dump_bytes`, used in chunk 7.
Uses `cols` 6a, `get` 6a, `length` 6a, `rows` 6a, and `Video` 6a.

5 Seam carving

That completes the input and output parts of the program; now we need only go back and define the `carve_frame` operation. The purpose of this operation is to reduce a video by a single frame.

It takes some thought to figure out the right way to resize the video. The important observation is that for every `row,col` pair, there's exactly one `frame` for which `frame,row,col` is deleted. (On the other hand, there may not be exactly one pixel deleted for all pairs of `frame,row` or `frame,col`.)

An easy way to handle it is to break the task into two parts: marking pixels for deletion for all `row,col`, and then applying the reshape operation to remove one frame. When we reshape, we'll simply build up a new array ignoring the unwanted elements. That will be easier than trying to move elements around in-place.

In fact, since we're going to carve based on a maximum projection along axis `row`, we only need to store the frame to delete as a function of the column.

8a ⟨*Video members* 5a⟩+≡ (4c) ◁7d 9b▷

```
public: void carve_frame();
```

Uses `carve_frame` 8b.

8b ⟨*Definitions* 6a⟩+≡ (2) ◁7e 9c▷

```
void Video::carve_frame() {
        ndarray<double> opt {frames, cols};
        ndarray<int> prev {frames, cols};
        ndarray<int> to_delete {cols}; // mark frame to delete

        ⟨Base case 10a⟩
        ⟨Apply recurrence 10b⟩
        ⟨Mark seam for deletion 11a⟩
        ⟨Reshape pixel array 11b⟩
}
```

Defines:
`carve_frame`, used in chunks 7b and 8a.
`opt`, used in chunks 10 and 11a.
`prev`, used in chunks 10b and 11a.
`to_delete`, used in chunk 11.
Uses `cols` 6a, `frames` 6a, `ndarray` 5b, and `Video` 6a.

Above, I've outlined the steps of the DP algorithm. Since the algorithm will make reference to the image energy, now is a good time to define that part of the constructor. The L_1 norm of the intensity gradient is a typical choice for resizing images. Instead of taking the gradient with respect to all three axes, here I only take it with respect to time. This way we say objects are only important if they are moving. Note that when taking this derivative, we need to pay attention to edge cases. For convenience I reference the intensity of the pixels instead of accessing the three color channels separately.

Also note that we're calculating the max-projection of the energy along the `row` axis, that is

$$U(\texttt{frame}, \texttt{col}) = \max_{\texttt{row}} U(\texttt{frame}, \texttt{row}, \texttt{col}), \tag{1}$$

where $U(\texttt{frame}, \texttt{row}, \texttt{col}) = |\partial \texttt{intensity} / \partial \texttt{frame}|$.

9a ⟨*Calculate voxel energy* 9a⟩≡ (6a)

```
ndarray<double> temp {frames, cols};
for (int r=0; r<rows; ++r) {
⟨Print loading bar 12a⟩
for (int f=0; f<frames; ++f) {
for (int c=0; c<cols; ++c) {
        double df = intensity(f,r,c) - intensity(f == 0 ? 1 : f-1, r,c);
        temp.get(f,c) = max(abs(df), temp.get(f,c));
}}}
energy = temp;
```

Uses `abs` 9d, `cols` 6a, `frames` 6a, `get` 6a, `intensity` 9c, `ndarray` 5b, and `rows` 6a.

We can define the intensity of a pixel as the sum of the RGB components. Recall that the RGB components are `unsigned char`; we mustn't let them overflow.

9b ⟨*Video members* 5a⟩+≡ (4c) ◁8a

```
double intensity(int,int,int);
```

Uses `intensity` 9c.

9c ⟨*Definitions* 6a⟩+≡ (2) ◁8b

```
double Video::intensity(int f, int r, int c) {
        double res {0.0};
        for (int chan=0; chan<3;++chan) res += get(f,r,c,chan);
        return res;
}
```

Defines:
`intensity`, used in chunk 9.
Uses `get` 6a and `Video` 6a.

Let's also include headers for the absolute value and max functions...

9d ⟨*Include headers* 4b⟩+≡ (2) ◁6b 12b▷

```
#include <cmath>
#include <algorithm>
```

Defines:
`abs`, used in chunk 9a.

In Section 1 we saw how seam-carving works for an image in terms of x and y. Now we need to translate that to our videos in terms of frame and column. Since we were shrinking x in the exposition above, we identify x →frame, y →column. That is, `opt[f,c]` is:

- the minimum energy for a path
- in the maximum-energy projection along `row`
- defined on `col = 0, ..., c`
- with the constraint that at `col=c`, `frame=f`.

Similarly, assuming `c > 0`, the definition of `prev[f,c]` is

- the value of `f` when `c-1`
- in the path described above.

Note that we will leave the `c=0` values of the `prev` array undefined but still present in the array. We could have shifted the indices by one to save a little memory, but that would have made the definition of `prev[f,r,c]` less intuitive.

The base case `opt[f,0] = energy[f,0]` immediately follows from the definition of `opt`.

10a ⟨*Base case* 10a⟩≡ (8b)

```
for (int f=0; f < frames; ++f)
        opt.get(f,0) = energy.get(f,0);
```

Uses `frames` 6a, `get` 6a, and `opt` 8b.

The recurrence translates to $\texttt{opt}[\texttt{f}, \texttt{c}+1] = \min_{|\texttt{f}'-\texttt{f}|<=1} \texttt{opt}[\texttt{f}', \texttt{c}] + \texttt{energy}[\texttt{f}, \texttt{c}]$, with $\texttt{prev}[\texttt{f}, \texttt{c}+1] = \texttt{f}'$. The main thing to note before writing the code is to be careful about edge cases. I.e., `f-1`, `f`, `f+1`, and `c+1` may not always be valid indices.

10b ⟨*Apply recurrence* 10b⟩≡ (8b)

```
for (int c=0; c+1<cols; ++c) {
for (int f=0; f<frames; ++f) {
        opt.get(f,c+1) = opt.get(f,c);
        prev.get(f,c+1) = f;
        if (f-1 >= 0 && opt.get(f-1,c) < opt.get(f,c+1)) {
                opt.get(f,c+1) = opt.get(f-1,c);
                prev.get(f,c+1) = f-1;
        }
        if (f+1 < frames && opt.get(f+1,c) < opt.get(f,c+1)) {
                opt.get(f,c+1) = opt.get(f+1,c);
                prev.get(f,c+1) = f+1;
        }
        opt.get(f,c+1) += energy.get(f,c+1);
}}
```

Uses `cols` 6a, `frames` 6a, `get` 6a, `opt` 8b, and `prev` 8b.

At this point we have computed enough information to determine which seam to delete. The procedure is to find $\min_{\texttt{f}} \texttt{opt}[\texttt{f}, \texttt{cols} - 1]$, then follow through `prev` and delete all the (f, c) traced out.

11a ⟨*Mark seam for deletion* 11a⟩≡ (8b)

```
// find endpoint of seam
int f_min = 0;
for (int f=1; f < frames; ++f) {
        if (opt.get(f,cols-1) < opt.get(f_min,cols-1))
                f_min = f;
}
// trace through seam and mark pixels
for (int c=cols-1; c >= 0; --c) {
        to_delete.get(c) = f_min;
        f_min = prev.get(f_min, c);
}
```

Defines:
f_min, never used.
Uses cols 6a, frames 6a, get 6a, opt 8b, prev 8b, and to_delete 8b.

Finally, we need to update the `pixels` ndarray so as not to include the pixels we've marked for deletion. We simply build a replacement, inserting all pixels but the ones we've decided to remove. We also have to remove the corresponding points from the `energy` array.

11b ⟨*Reshape pixel array* 11b⟩≡ (8b)

```
ndarray<unsigned char> new_pixels {frames-1, rows, cols, 3};
ndarray<double> new_energy {frames-1, cols};
for (int c=0; c<cols; ++c) {
        int f_new = 0; // frame index into replacement arrays
        for (int f_old = 0; f_old < frames; ++f_old) {
                // f_old: frame index into existing arrays
                if (f_old == to_delete.get(c)) continue;
                for (int chan=0; chan<3; ++chan) {
                for (int r=0; r<rows; ++r) {
                        new_pixels.get(f_new,r,c,chan) =
                                pixels.get(f_old, r, c, chan);
                }}
                new_energy.get(f_new,c) =
                        energy.get(f_old,c);
                ++f_new;
        }
}
// maintain class invariant
frames = frames-1;
pixels = new_pixels;
energy = new_energy;
```

Defines:
replacement, never used.
Uses cols 6a, frames 6a, get 6a, ndarray 5b, rows 6a, and to_delete 8b.

It's not critical, but a loading bar makes a program much nicer to use. We can use the backspace character to write over previous prints. Sometimes it doesn't work with `clog` since it's buffered, but `cerr` is unbuffered so we can use that for things we may want to overwrite, like percentages.

12a ⟨*Print loading bar* 12a⟩≡ (9a)

```
int percentage = 100 * (r+1) / rows;
if (r > 0) cerr << "\b\b\b\b";
cerr << setw(3) << percentage << "%";
if (r == rows-1) clog << "\n";
```

Uses `rows` 6a.

We need a header for the `setw` function, which forces the number to print with 3 columns, so we get, e.g., " 1%" or " 90%".

12b ⟨*Include headers* 4b⟩+≡ (2) ◁9d

```
#include <iomanip>
```

6 Bibliography

References

[1] S. Avidan and A. Shamir, "Seam carving for content-aware image resizing." Siggraph (2007).

[2] M. Rubinstein et al., "Improved Seam Carving for Video Retargeting." Siggraph (2008).

[3] Y. Pritch et al., "Nonchronological Video Synopsis and Indexing." IEEE Transactions on Pattern Analysis and Machine Intelligence (2008).