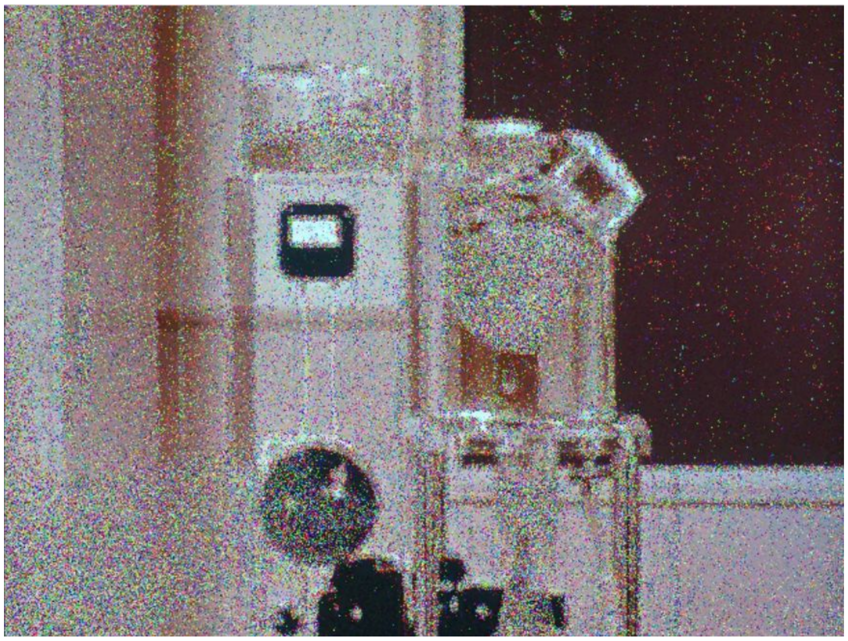
Algorithmic Distortion of Video Files on the Per-pixel Scale

Justin Trupiano
University of Colorado
June 2020

Random Guess. This algorithm starts with a completely randomized image. Each of the 307,200 pixels (640x480) in the image is assigned a random color value–three numbers, representing the RGB channels, between 0 and 255. For each frame of the video, all of the pixels from the randomly generated image are checked against the corresponding pixel of the original video. If the difference between the two values is within a range specified by the *dist* variable, that value is left unaltered. For example, if the randomly assigned value is 127 and the original pixel value is 120, that pixel is left unchanged. If the difference between the two values is greater than the *dist* value, the randomly assigned value is again randomized with a number between 0 and 255. The less movement from frame to frame of the original video, the higher the chance that this randomization process will result in the correct values for a given pixel. This results in an emergent image where the video is relatively consistent, and chaos where changes from frame to frame are most drastic.

The included video uses a *dist* value of 10. This was determined a good balance between the two extremes. Higher values of *dist* result in unchanging randomly generated images as values more distant from the original video pixels would be determined 'correct.' Lower values of *dist* result in ever-changing randomly generated static images. For example, a *dist* value of 0 would require the randomly generated pixel to exactly match the original video, a 1 in 16777216 chance (16777216 or 256^3 is the number of possible unique colors in 24-bit color space).

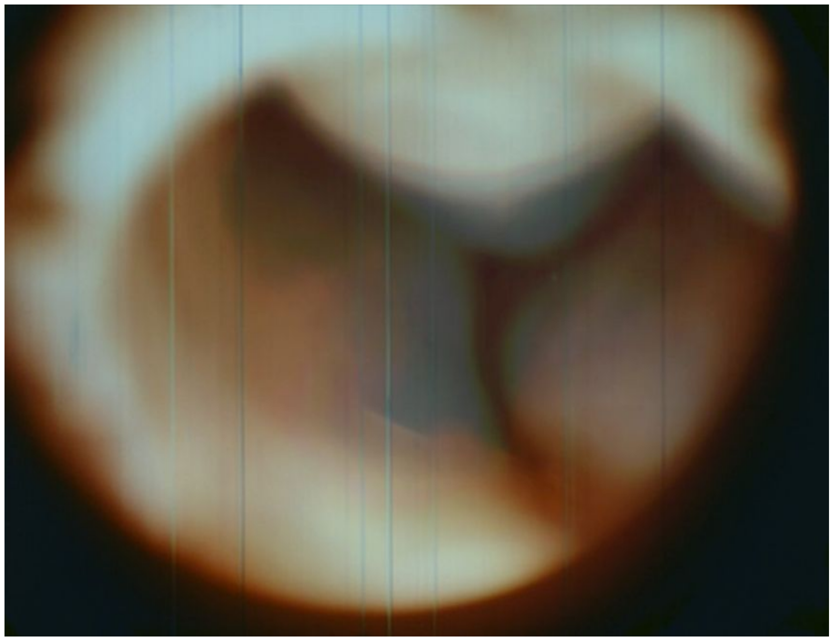


```
void randomGuess()
{
  loadPixels();
  for (int y = 0; y < height; y++) {

      int loc = x + y*width;

      float tR = red(testImage[loc]);
      float tG = green(testImage[loc]);
      float tB = blue(testImage[loc]);

      float mR = red(originalImage[loc]);
      float mG = green(originalImage[loc]);
      float mB = blue(originalImage[loc]);


      float rDist = abs(mR - tR);
      float gDist = abs(mG - tG);
      float bDist = abs(mB - tB);

      float dist = 10;
      float r = 0;
      float g = 0;
      float b = 0;

      if (rDist > dist)
        r = random(255);

      if (gDist > dist)
        g = random(255);

      if (bDist > dist)
        b = random(255);

      testImage[loc] += color(r,g,b);

      originalImage[loc] = testImage[loc];
  }
  updatePixels();
}
```

Move Toward Color. This algorithm starts with a completely randomized image. Each of the 307,200 pixels (640x480) in the image is assigned a random color value–three numbers, representing the RGB channels, between 0 and 255. For each frame of the video, all of the pixels from the randomly generated image are checked against the corresponding pixel of the original video. If the two values are equal, that value is left unaltered. If the two values are not equal, the random value is adjusted in the direction of the original video. For example, if the random value is 127 and the value from the original video is 200, the random value will be increased. The amount by which it is increased is determined by the *step* variable. *Step* represents a percentage of the difference between the two numbers. If *step* is 0.01, the incorrect value will move toward the correct value by 1% of the total difference. If *step* is 1.0 the value will be adjusted toward the correct value by 100%.

The included videos represent a range of *step* values between 0.001 and 0.75. The lower the *step* value the longer it takes for pixels to match the original video, resulting in a ‘burn-in’ effect, as the original video continues to change the lower *step* value doesn’t allow enough time for the algorithmically generated images to catch up. The larger the *step* value the faster individual pixels match the corresponding pixels, resulting in a slight stutter effect as the algorithmically generated image is only a few frames behind the original.

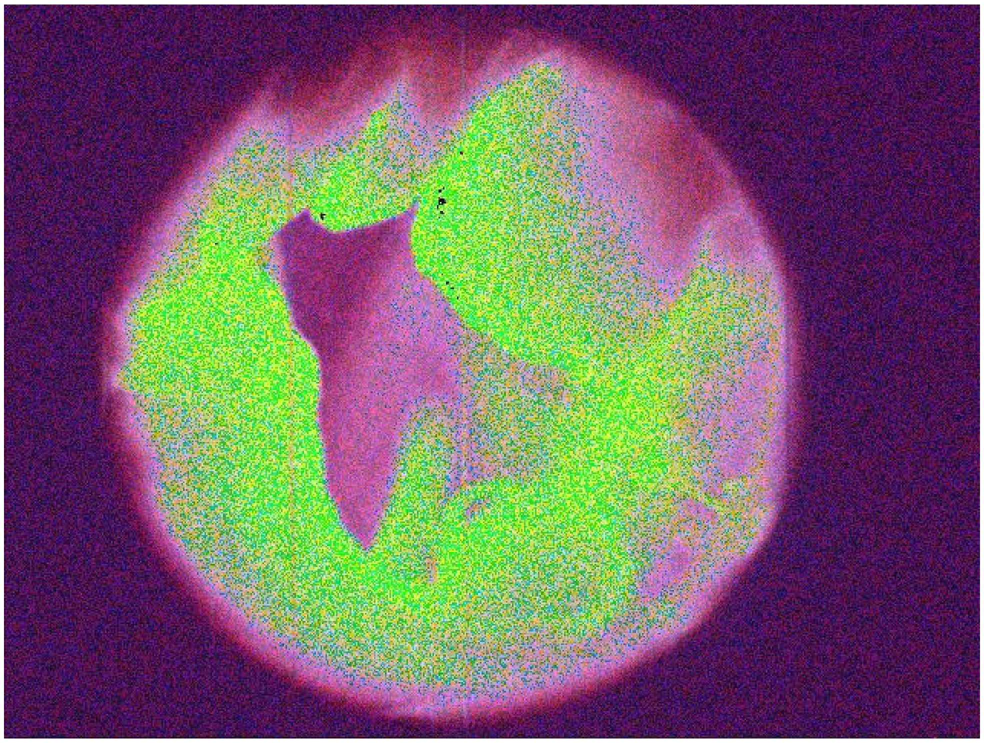
Below is a still from *moveTowardColor_0_025.mov*. This version uses a *step* value of 0.025 and shows an interesting mix of the described phenomena. The *step* value is low enough to see the ‘burn-in’ effect, but still fast enough to show the movement of the original video.



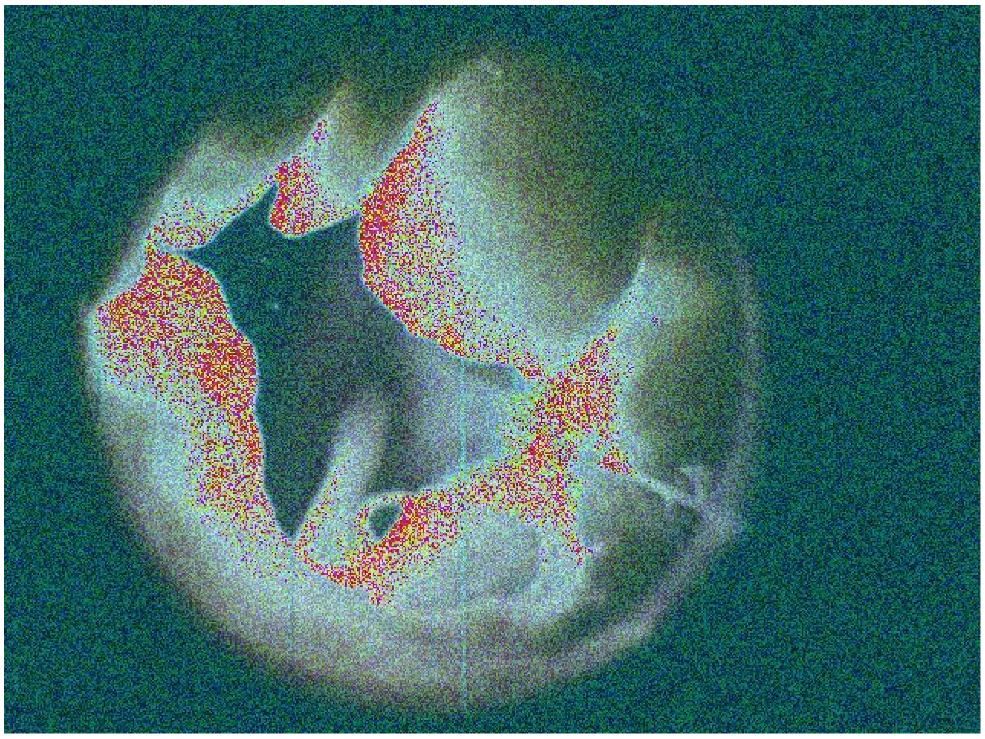
```
void moveTowardColor(float step)
{
  loadPixels();
  for (int y = 0; y < height; y++) {

      int loc = x + y*width;

      float tR = red(testImage[loc]);
      float tG = green(testImage[loc]);
      float tB = blue(testImage[loc]);

      float mR = red(pixels[loc]);
      float mG = green(pixels[loc]);
      float mB = blue(pixels[loc]);

      if (tR != mR)
        tR += (mR - tR) * step;

      if (tG != mG)
        tG += (mG - tG) * step;

      if (tB != mB)
        tB += (mB - tB) * step;

      testImage[loc] = color(tR, tG, tB);

      pixels[loc] = testImage[loc];
  }
  updatePixels();
}
```

Randomize Additive Pixels. This algorithm takes three numbers as input, one for each color channel (RGB). These numbers can range from 0-255. Every frame, each pixel in the original image is increased by a random number between 0 and the specified number. The included videos represent a range of possible value combinations for each channel in steps of 0, 127, and 255.

The first image below is taken from the video where Red and Blue are set to 127 while Green is set to 0. This means that in each frame, every pixel gets a random value between 0 and 127 added to its Red and Blue channels, while there is no change to its Green channel. This results in a purple hue to the overall image as the Red and Blue channels are being increased.



The larger the number the more distortion of that channel. For example, the following example uses the values R0 G127 B127, randomly increasing the Green and Blue channels. In some places, this acts to highlight the Red channel, even though the Red values are left unaltered. This is because color values cannot be greater than 255, so if a value exceeds 255 it overflows to 0. In the case of the image below, the pixels which are close to white (~R255, ~G255, ~B255) have a chance of overflowing in the Green and Blue channels, resulting in low values for Green and Blue and high values in the Red channel (~R255, ~G0, ~B0).



```
void randomizeAdditivePixels(int rAmount, int gAmount, int bAmount)
{
  loadPixels();
  for (int y = 0; y < height; y++) {
      int loc = x + y*width;

      float r = red(pixels[loc]);
      float g = green(pixels[loc]);
      float b = blue(pixels[loc]);

      r = random(rAmount);
      g = random(gAmount);
      b = random(bAmount);

      pixels[loc] += color(r,g,b);
  }
  updatePixels();
}
```