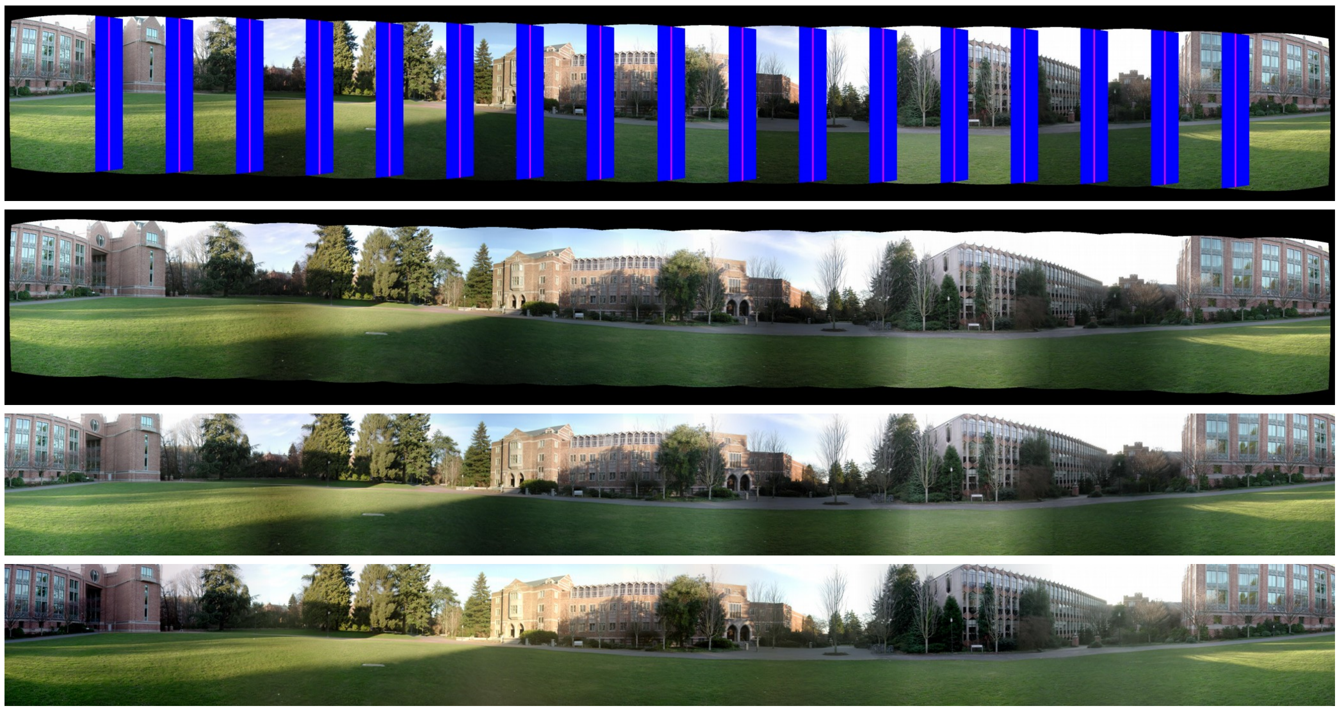
Project 2: Panorama Mosaic Stitching

CSE559A - Computer Vision



Daniel de Cordoba Gil

10/04/2016

The first step in our project was to make the code to compute the inverse map to warp the image. This part was pretty simple having the equations available in the class slides. The greatest challenge was to apply the rotation, as I did not quite understand how to use the matrix M to rotate and translate some pixel coordinates. I actually started solving the problem manually, and with this I understood what was going on and solved the problem.

This is the code generated:

```
// *** BEGIN TODO ***
// Compute Euclidean coordinates
p[0] = sin(xf) * cos(yf);
p[1] = sin(yf);
p[2] = cos(xf) * cos(yf);

// Apply rotation
p = r * p;

// Project to z=1 plane
xt = p[0] / p[2];
yt = p[1] / p[2];
zt = p[2] / p[2];

// Distort with radial distortion
float r2 = pow(xt, 2) + pow(yt, 2);
float r4 = pow(r2, 2);
xt = xt * (1 + k1*r2 + k2*r4);
yt = yt * (1 + k1*r2 + k2*r4);
// *** END TODO ***
```

In this problem, we also applied radial distortion, using the appropriate formula. This element of the homework was optional, and it should give us extra credit. We will use the k1 and k2 parameters to correct the radial distortion..

To run the code, we run into difficulties with the pictures and converting them from .jpg to .tga. I am not sure why but when converting them, they would turn 90 degrees (only for the images we took). We had to modify the original .jpg images to fix this.

We then computed the focal distance. This was done with the formula:

focal length in pixels = (image width in pixels) * (focal length in mm) / (CCD width in mm)

Finding the values was easy thanks to jhead and a bit of Google. We obtained a result of:

focal length = 720 pixels * 10.0mm / 22.3mm = **322.87 pixels**

To make it easier, we will use the value 325 pixels, as the difference will not be noticeable.

Therefore, to compute the spherical warp for our images we used the commands:

```
./Panorama sphrWarp ../test_sets/room/room01.tga room01.warp.tga 325
./Panorama sphrWarp ../test_sets/room/room02.tga room02.warp.tga 325
…
```

Next step was computing the sift features of the warped images. We had many problems with the provided package to compute sift, as no matter what we did it would output an error when running it, and we invested many hours trying to find out the reason. It turned out that a package was missing in our computer to run some specific 32 bits programs. After fixing it, we could finally compute the sift features:

```
./sift <../PackageProject_2/PanoramaSkel/room01.warp.pgm >room01.warp.key
./sift <../PackageProject_2/PanoramaSkel/room02.warp.pgm >room02.warp.key
…
```

Next, we computed the matches between adjacent pictures (no problems found in this step):

```
./Features matchSIFTFeatures room01.warp.key room02.warp.key 0.8 room0102.txt 2
./Features matchSIFTFeatures room02.warp.key room03.warp.key 0.8 room0203.txt 2
…
```

Having the features descriptors and the matches for adjacent images, we had to compute every pair of adjacent images using RANSAC. What this means is calculating the x and y translation from one picture to the other.

We started computing the alignPair function, which had to get nRANSAC random matches between two features, and for each compute the distance between both features, and count the number of matches whose distance was within RANSACthresh of such distance. With this, we could get the number of inliers.

This was the code generated for this:

```
// BEGIN TODO
// Initialize random seed
srand (time(NULL));

// Declare variables
int maxInliers = 0;
vector<int> posMaxInliers;

// Repeat nRANSAC times
for (int i = 0; i < nRANSAC; i++)
{
    // Select random matching pair
    int pos = rand() % matches.size();
```

```
        // Calculate translation
        int xt = f1[matches[pos].id1-1].x - f2[matches[pos].id2-1].x;
        int yt = f1[matches[pos].id1-1].y - f2[matches[pos].id2-1].y;

        // Count inliers
        M[0][2] = xt;
        M[1][2] = yt;
        vector<int> inliers;
        int numInliers = countInliers(f1, f2, matches, m, f, M, RANSACthresh,
inliers);

        // Save inliers if greater than maxInliers
        if (numInliers > maxInliers)
        {
            maxInliers = numInliers;
            posMaxInliers = inliers;
        }
    }
    printf("Number Inliers: %d\n", maxInliers);
    // Compute M for test with maximum inliners
    leastSquaresFit(f1, f2, matches, m, f, posMaxInliers, M);
    // END TODO
```

We can see that a random match is acquired in every iteration, and the inliers are counted for that match. If the number of inliers is greater than the maximum number of inliers, we save the new maximum and the inliers vector.

To end, we compute the M, which is the translation matrix (M can normally be used for rotation and other transformations but here we only use it for translation) for all the inliers.

The two functions used that we will explain now are countInliners and leastSquaresFit.

countInliners receives an M and checks the translation for every match, and compares it to the translation according to M. If the distance between these tow translations is smaller than RANSACthresh, then the match is considered an inlier and it is counted and saved as such:

```
    inliers.clear();
    int count = 0;

    for (unsigned int i=0; i<(int) matches.size(); i++) {
        // BEGIN TODO
        float xd = f1[matches[i].id1-1].x - f2[matches[i].id2-1].x - M[0][2];
        float yd = f1[matches[i].id1-1].y - f2[matches[i].id2-1].y - M[1][2];
        if (sqrt(pow(xd, 2) + pow(yd, 2)) <= RANSACthresh)
        {
            count++;
            inliers.push_back(i);
        }
        // END TODO
    }
```

leastSquaresFit just computes the average translation of all the inliers, once the best match has been found in the alignPair function. It simply adds up every translation for x and y and in the end it divides such number by the number of inliers to get the average translation for x and y, and saves it into M:

```
    double u = 0;
    double v = 0;

    for (int i=0; i<inliers.size(); i++) {
        double xTrans, yTrans;

        // BEGIN TODO
        // compute the translation implied by the ith inlier match
        // and store it in (xTrans,yTrans)
        xTrans = f1[matches[inliers[i]].id1-1].x - f2[matches[inliers[i]].id2-1].x;
        yTrans = f1[matches[inliers[i]].id1-1].y - f2[matches[inliers[i]].id2-1].y;
        // END TODO

        u += xTrans;
        v += yTrans;
    }

    u /= inliers.size();
    v /= inliers.size();

    M[0][0] = 1;
    M[0][1] = 0;
    M[0][2] = u;
    M[1][0] = 0;
    M[1][1] = 1;
    M[1][2] = v;
    M[2][0] = 0;
    M[2][1] = 0;
    M[2][2] = 1;
```

Once the code was finished, we executed it with:

```
./Panorama alignPair room01.warp.key room02.warp.key room0102.txt 200 2 sift
./Panorama alignPair room02.warp.key room03.warp.key room0203.txt 200 2 sift
…
```

The last step was to stitch and crop the images to create a panorama.

We first find the final size of the images once stitched together. The best way to do so is to get every corner and calculate all the places it could end after every translation. Because the corners are the extremes of the images, we know that the maximum and minimum value for x in any corner after every translation will also be the maximum value for any x in the picture, and the same happens with y. Therefore, to calculate the bounds of our acc image we just get the MAX and MIN x for all the corners after stitching every image.

The code is the following:

```
        // *** BEGIN TODO #1 ***
        // add some code here to update min_x, ..., max_y
                if (i == 0) {
            firstCorner[0][0] = corners[0][0];
            firstCorner[0][1] = corners[0][1];
            firstCorner[0][2] = corners[0][2];
            firstCorner[1][0] = corners[1][0];
            firstCorner[1][1] = corners[1][1];
            firstCorner[1][2] = corners[1][2];
            firstCorner[2][0] = corners[2][0];
            firstCorner[2][1] = corners[2][1];
            firstCorner[2][2] = corners[2][2];
            firstCorner[3][0] = corners[3][0];
            firstCorner[3][1] = corners[3][1];
            firstCorner[3][2] = corners[3][2];
        }
        if (i == n-1) {
            lastCorner[0][0] = corners[1][0];
            lastCorner[0][1] = corners[1][1];
            lastCorner[0][2] = corners[1][2];
            lastCorner[1][0] = corners[0][0];
            lastCorner[1][1] = corners[0][1];
            lastCorner[1][2] = corners[0][2];
            lastCorner[2][0] = corners[3][0];
            lastCorner[2][1] = corners[3][1];
            lastCorner[2][2] = corners[3][2];
            lastCorner[3][0] = corners[2][0];
            lastCorner[3][1] = corners[2][1];
            lastCorner[3][2] = corners[2][2];
        }

        min_x = (float) MIN(min_x, corners[0][0]);
        min_x = (float) MIN(min_x, corners[1][0]);
        min_x = (float) MIN(min_x, corners[2][0]);
        min_x = (float) MIN(min_x, corners[3][0]);

        min_y = (float) MIN(min_y, corners[0][1]);
        min_y = (float) MIN(min_y, corners[1][1]);
        min_y = (float) MIN(min_y, corners[2][1]);
        min_y = (float) MIN(min_y, corners[3][1]);

        max_x = (float) MAX(max_x, corners[0][0]);
        max_x = (float) MAX(max_x, corners[1][0]);
        max_x = (float) MAX(max_x, corners[2][0]);
        max_x = (float) MAX(max_x, corners[3][0]);

        max_y = (float) MAX(max_y, corners[0][1]);
        max_y = (float) MAX(max_y, corners[1][1]);
        max_y = (float) MAX(max_y, corners[2][1]);
        max_y = (float) MAX(max_y, corners[3][1]);
        // *** END TODO #1 ***
```

Here we are also saving the first and last corners, as they will be used later in our code.

This was also one of the hardest sections to finish, as we did not quite understand what was happening when a corner was multiplied by pos (corners[0] = pos * corners[0];), but one it was clear, solving this part was a piece of cake.

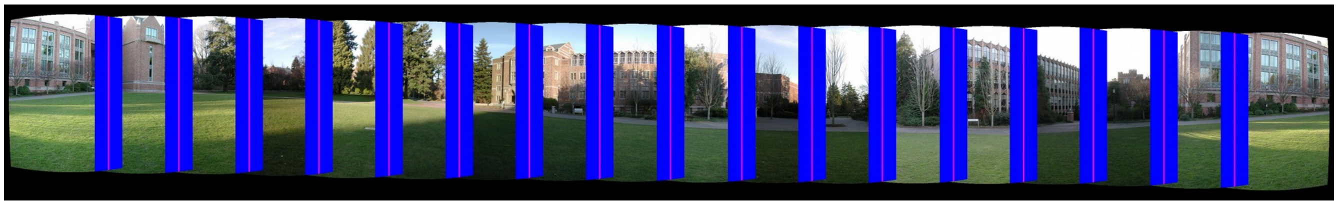
After this, probably the most complicated section was completed. This is the function AccumulateBlend.

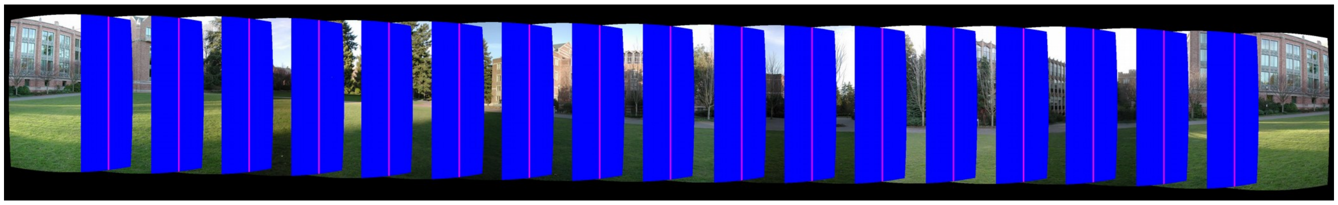
We have added many extra features to our code, and we think it deserves extra credit. In the first place, a set of variables on top of the function can be changed to show special features of our code. These are:

```
    // Used to show blending sections
    bool see_blend = false;
    // Used to activate optimal blending
    int optimal_blend = 1; // 0:simple, 1:optimal, 2:average
    // Used to see images for debugging and pause the program every iteration
    bool debug_mode = false;
```

see_blend allows the user to see the merges between pictures: where they start, where they end, and the middle point. This is really useful to decide visually the blendWidth parameter.

For example, in the following images see_blend is true and we tried a blendWidth of 100 and 200:



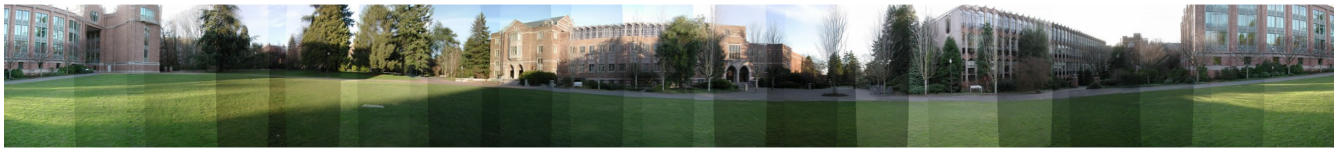


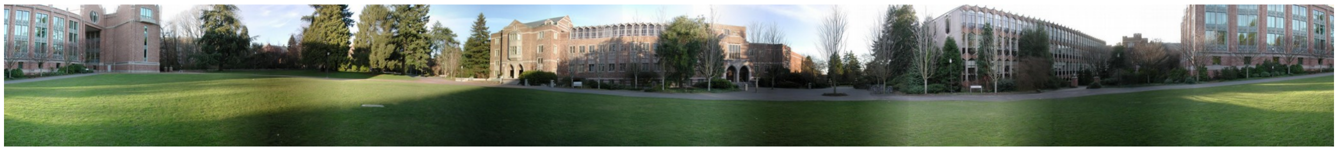
Note: the above images' height drift has not been corrected.

As it can be seen, it is easy to see the width of the intersection and decide which blend is better.

Setting debug mode to false will pause after every image is stitched to acc and save a temporary image with the contents of acc and the image saved. This is useful for debugging.

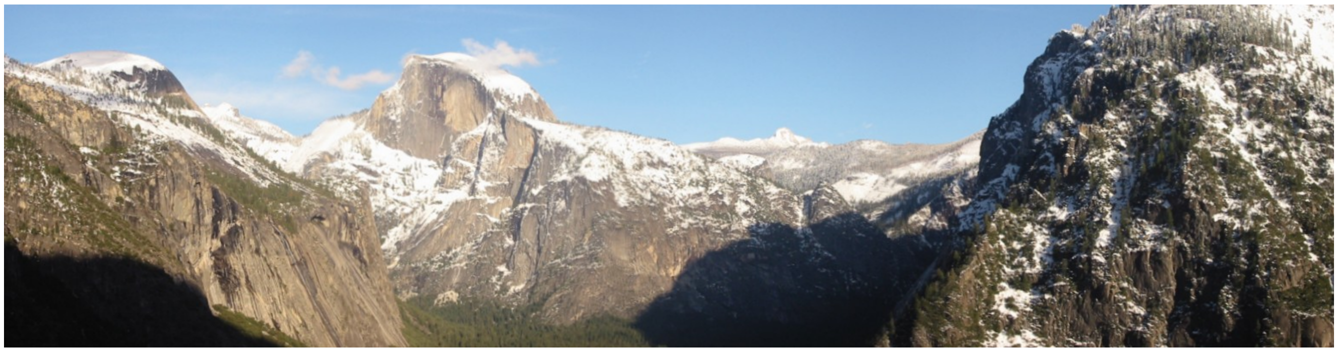
Finally, setting optimal_blend to 0 will disable our blending method and implement a really simple method, which has a clearly worse result. The comparison can be seen in the following panoramas:

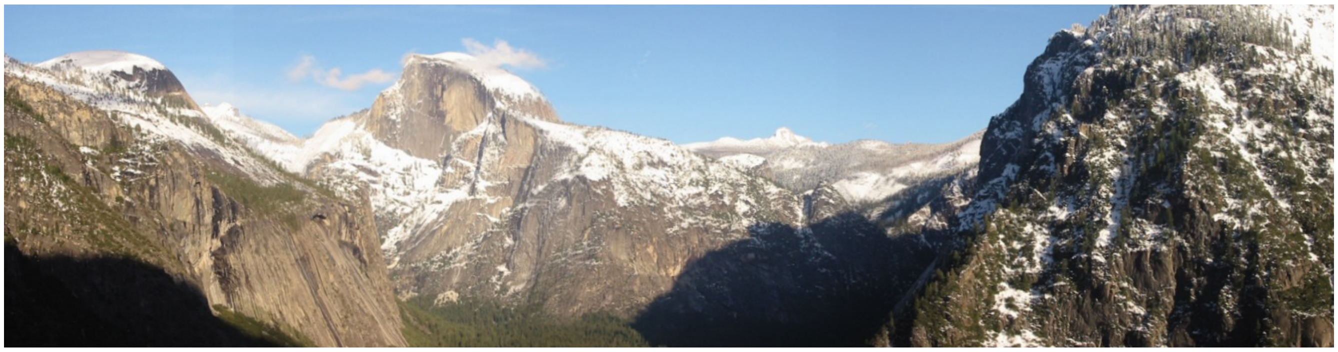




The first one is blended with the regular method: setting a weight of 1 and allowing some transparency. The weight is set poorly: weight is always one, and therefore we see the differences.

Setting optimal_blend to 2 will disable the optimal blend, but will enable the regular feathering. The difference is subtle, but it makes a huge difference if both pictures are watched side by side (the HQ version is better for spotting the differences):





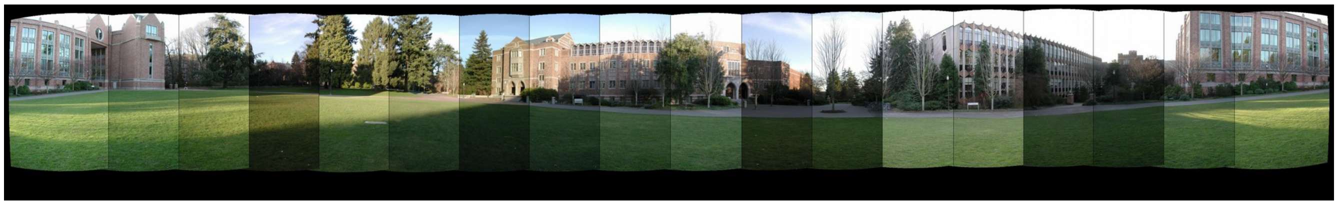
Some artifacts and repetition are found in the second picture (the regular feathering blending) not visible in the first one (optimal blending).

Because of all the features added, the code is a bit messy and complicated, and it will not be included here as it would take at least two pages.

The steps followed for the optimal blend are explained in the following page:

First, the beginning and end of the intersection between the new image and the accumulator will be calculated. This gives us two x values: intersection_start and intersection_end, from which we can calculate if the image will be added to the left or right of the previous image, as well as the middle point of the blending intersection.

Around the intersection_middle point is where the blending will be performed, and it will have a width of blendWidth, centered at this point. The idea is that in this point, the weight for both images will be 0.5, and therefore they will be equally transparent. As the x's come closer to one image or the other and further from the intersection_middle, the weight changes so that over one a weight w is applied and over the other a weight (1 – w) is applied. At a distance of blendWidth/2 from the intersection_middle, the weight of one image will become zero and the other will become 1, and therefore, only one of both images will be shown. For x > intersection_middle + blendWidth/2 or x < intersection_middle - blendWidth/2, the weight will be maintained at 0 or 1, and therefore only one of both pictures will be shown. In consequence, if we set blendWidth to 0, it is the same as splitting the images in the middle of the intersection, and only showing one or the other, as it can be seen in the following picture.



Another advantage of this method is that normalizing is not necessary. As the sum of the weights in every pixel blended will always be equal to one. Anyway we have implemented the method NormalizeBlend, which saves the acc image into a CbyteImage, and normalizes the values in case the blending function used is changed. The code is:

```
    // *** BEGIN TODO ***
    // fill in this routine..
    for (int y = 0; y < acc.Shape().height; y++) {
        for (int x = 0; x < acc.Shape().width; x++) {
            if (acc.Pixel(x, y, 3) < 1) {
                img.Pixel(x, y, 3) = (uchar) 0;
            } else {
                img.Pixel(x, y, 0) = (uchar) (acc.Pixel(x, y, 0) / acc.Pixel(x, y,
3));
                img.Pixel(x, y, 1) = (uchar) (acc.Pixel(x, y, 1) / acc.Pixel(x, y,
3));
                img.Pixel(x, y, 2) = (uchar) (acc.Pixel(x, y, 2) / acc.Pixel(x, y,
3));
                img.Pixel(x, y, 3) = (uchar) 255;
            }
        }
    }
    // *** END TODO ***
```

The last step is to correct the drift and crop the image so no transparent pixel are shown. To correct the drift, we make the translation of the y pixels depend on the x pixel, with is done through the parameter A[1][0]. The slope of the correction is obtained with the expression:

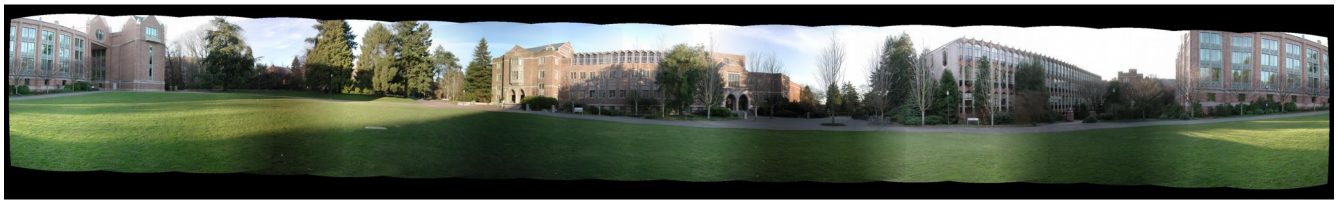
```
      // Different distance if panorama was made turning left or right
    float distCornersX = lastCorner[0][0] - firstCorner[0][0];
    if (abs(distCornersX) < abs(lastCorner[1][0] - firstCorner[1][0])) {
        distCornersX = lastCorner[1][0] - firstCorner[1][0];
    }
    A[1][0] = (lastCorner[0][1]-firstCorner[0][1]) / (distCornersX);
```

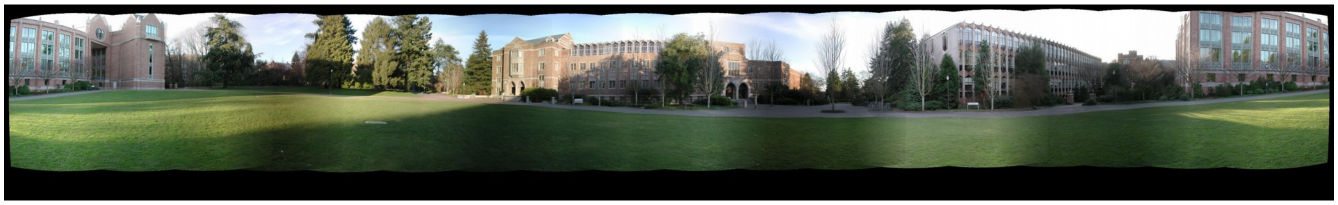
To end, we have designed a way of cropping the image and removing margins. This is done manually, selecting the percentage of the pictures that makes the unwanted frame of the image, and the offset that we want to translate the image. Therefore, for every image, we have some parameters that are optimal to crop the image so it looks good:

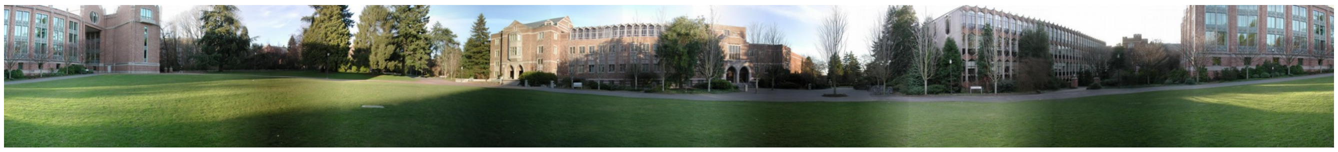
```
//      // Yosemite (blendWidth=150)
//      float margin_x = 0.02, margin_y = 0.11; // Margin to remove
//      float offset_x = 0.00, offset_y = -0.03; // Offsets for x and y

//      // Campus (blendWidth=200)
//      float margin_x = 0.005, margin_y = 0.14; // Margin to remove
//      float offset_x = 0.00, offset_y = 0.05; // Offsets for x and y

//      // Room (blendWidth=100)
//      float margin_x = 0.015, margin_y = 0.22; // Margin to remove
//      float offset_x = 0.00, offset_y = -0.065; // Offsets for x and y

    // No cropping
    float margin_x = 0.00, margin_y = 0.00; // Margin to remove
    float offset_x = 0.00, offset_y = 0.00; // Offsets for x and y

    CShape cShape(floor(mShape.width*(1.0-2.0*margin_x)), floor(mShape.height*(1.0-
2.0*margin_y)), nBands);
    printf("Final size: %dx%d\n", cShape.width, cShape.height);
    CByteImage croppedImage(cShape);

    // Compute the affine deformation
    CTransform3x3 A;

        // *** BEGIN TODO #2 ***
    // fill in the right entries in A to trim the left edge and
    // to take out the vertical drift

    A[0][2] = (margin_x + offset_x) * mShape.width ;
    A[1][2] = (margin_y + offset_y) * mShape.height;
```

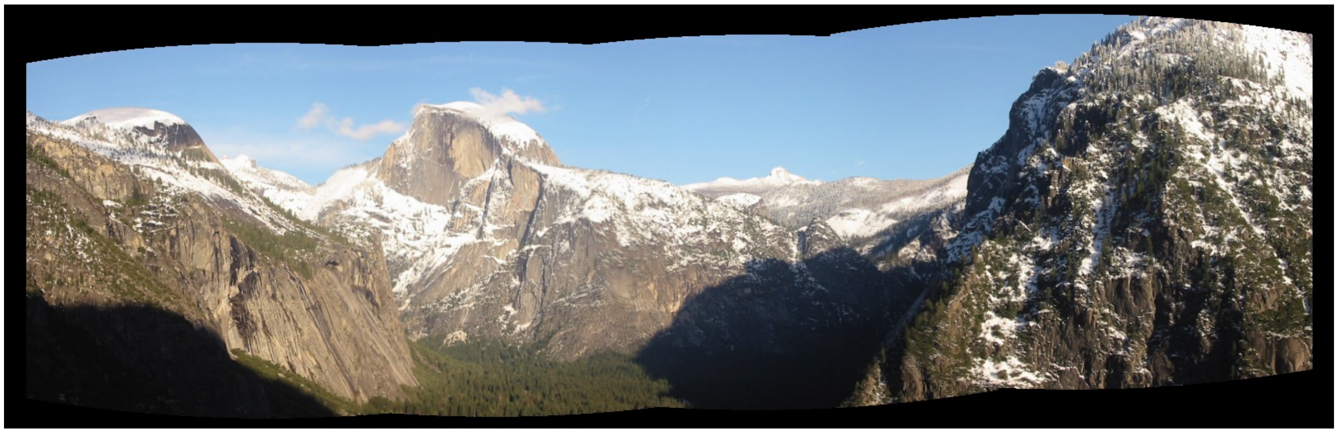
As it can be seen above, for each image (Yosemite, Campus, Room) we have some optimal features that determine how the image can best be cropped and translated. There is also a No cropping option which leaves the image as original, and only corrects the drift. This is also non required and could be a reason for extra credit.

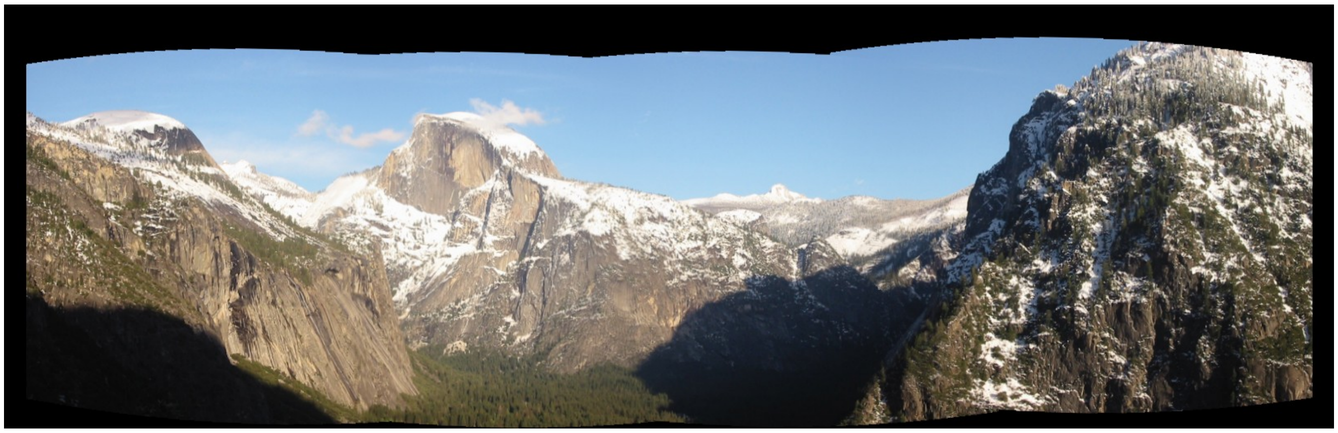
The difference can be seen in the following images, which shows the drifting correction and then the cropping correction:

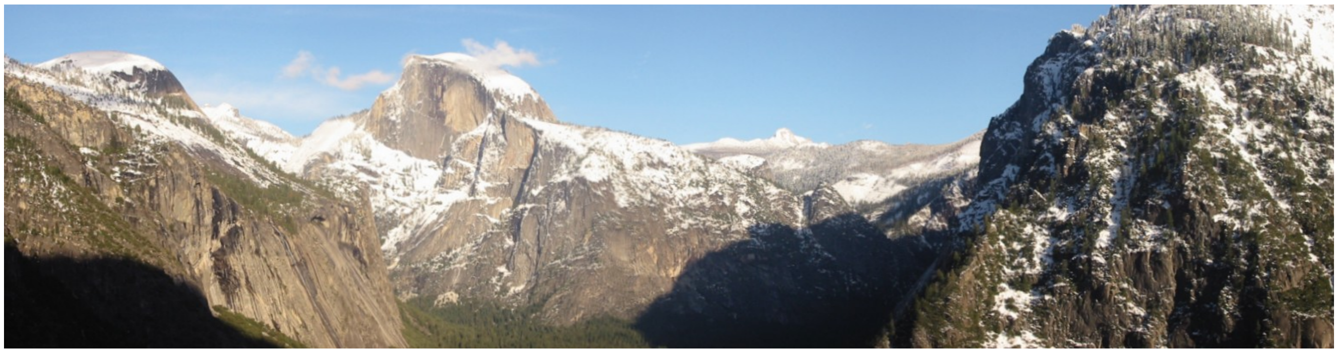


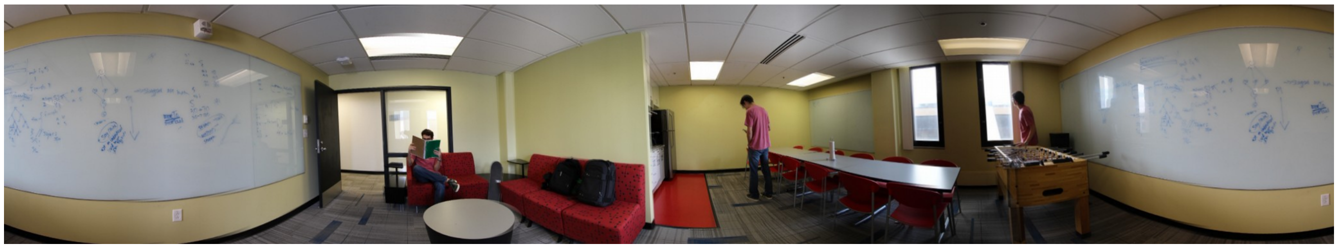






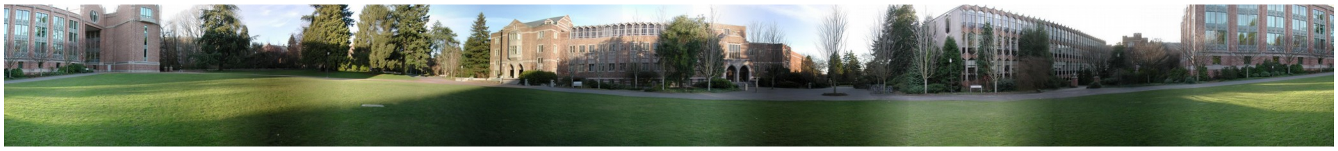


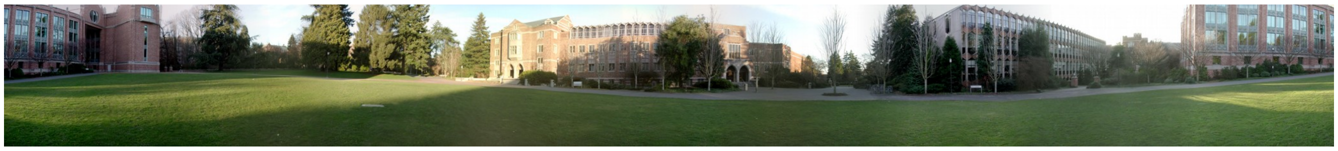




As it can be seen in the last picture, the same person (me) appears several times in the same panorama, which is also a reason for extra credit.

The last extra feature implemented has been a correction of the color. The results are seen in the next images:



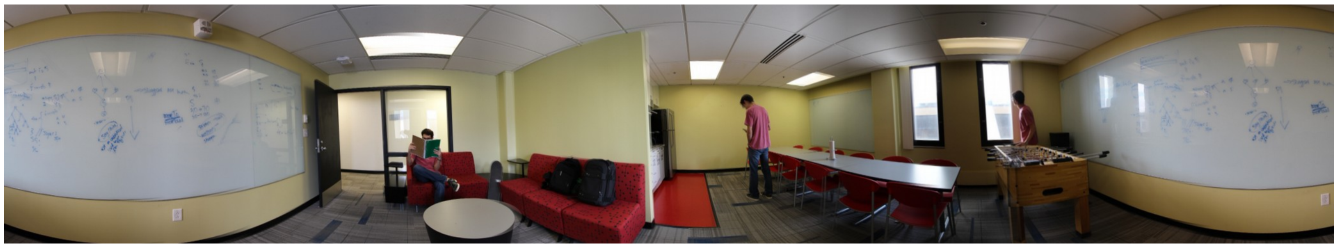


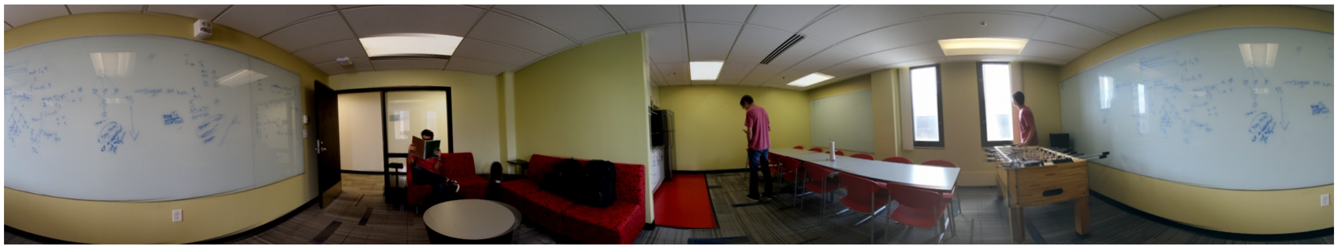
It can be seen that the second image is quite better, although it is not perfect, but the grass looks a lot more uniform. The method used has been to measure the RGB components of the pixels in the intersection of acc and img, for both acc and img, and add the difference to the new image to correct it, and make it darker or lighter to match its neighbor.

A boolean has been added to enable or disable this feature:

```
// Correct exposure differences
    bool color_correct = true;
```

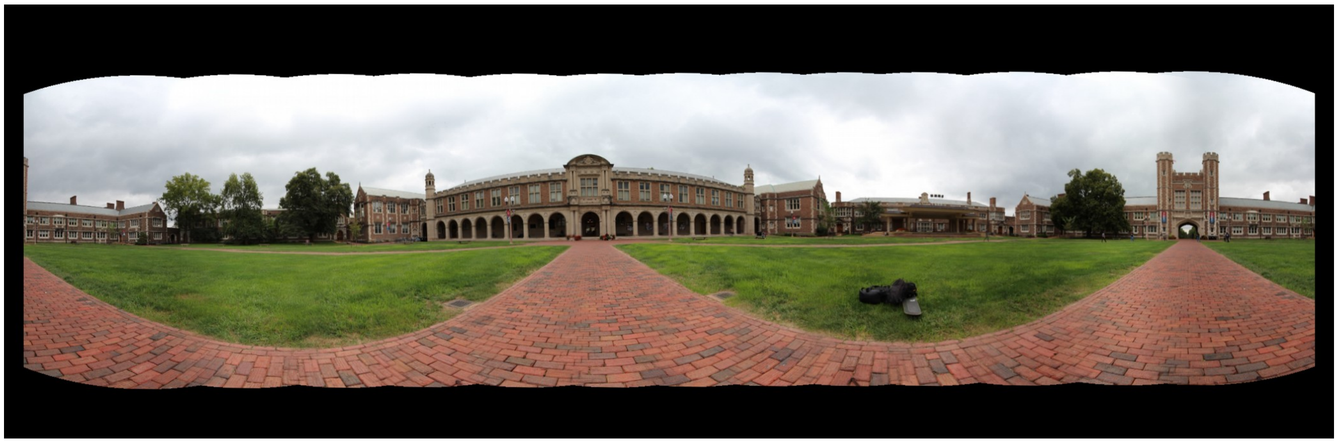
We have also recolored the image made by us:

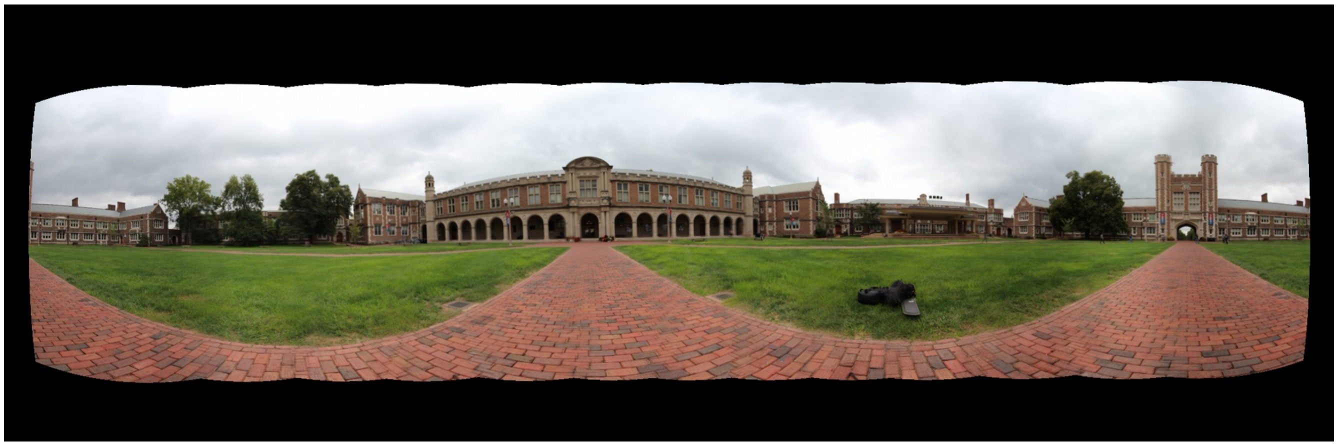




Unfortunately, pictures inside don’t look so good, and it is recommended to use only the exposure correction in the pictures that have clear changes in the exposure, as this modifies the color of the whole picture. Therefore, we will disable this option by default.

In the next pictures, the difference between correcting radial distortion or not is shown:





The difference is really small, but the first one does not have a radial distortion correction, and the second one does. To see the difference better, check the HQ images.

To end, a summary of all I have done to deserve extra credit (sorry for insisting on this, but I have to compensate my marks in the previous assignment, and I am aiming for 3pts extra credit).

- Compute radial distortion when warping the image.
- Making the code really flexible allowing to enable and disable the various kinds of blending, and adding the functionality of seeing the intersections of the pictures.
- Implementing a more complex and better blending way that is clearly better, even perfect (I am even surprised on how well it works, for example in the Yosemite picture I cannot tell its a panorama).
- Allowing options for cropping images so the transparent background is not shown.
- Correcting the color to remove the artifacts created by the changes of exposure.
- I have the same person multiple times in the panorama.