A Fast, Intuitive and Accurate Correspondence Method

Jan Dlabal
Stanford University
California
dlabal@alumni.stanford.edu

Abstract

The correspondence problem is an important one in computer vision, because it lets you extract depth information from two images taken from slightly different viewpoints, much like how the human brain figures out depth information given the input from our two eyes. There are many methods to solve this problem, such as normalized cross correlation, which starts with two rectified images, and scans across the corresponding line in the second image and roughly speaking picks the most similar pixel (in terms of its value and the value of the pixels around it) as the correspondence for a given input pixel. This method works well on images with lots of diverse pixels but is notoriously error-prone on images that have continuous similar or repeating regions, which is a huge fraction of real world images. Therefore, while the NCC method is very intuitive and easy to implement, it is lacking in accuracy. It also requires to score each pixel against the full line in the second image, which is quite slow. In my project I show that we can do much better in terms of accuracy and speed, without sacrificing the intuitiveness of cross correlation. The novel method I present uses normalized cross correlation metrics as a way to score correspondences, but it doesn't necessarily always pick the local maximum. My method thus significantly improves on the results of NCC by incorporating non-local constraints that preserve logical ordering between two correspondences, while also improving runtime from the very slow $O(n^3)$ by doing some divide and conquer.

1. Introduction

The correspondence problem is an important problem in computer vision. It is basically the problem that our brain has to solve when it gets the input from our two eyes and figures out 3D depth information that lets us interact with the world around us. Naturally, computers need to do something similar to be able to really understand what they're seeing. There are of course different solutions to understanding depth such as lasers/radars, however such solutions are usually expensive and not always practical. Solving the correspondence problem essentially gives you depth information from just two normal cameras, which are nowadays extremely cheap and easy to use in almost any application.

In class, we've seen a few ways of solving this problem, but none of them took into account non-local information, and thus many of them are prone to getting stuck in very contrived solutions that a human would immediately see as errors.

In this project, I try to come up with a novel algorithm that does incorporate some non-local information so that the results obtained make sense, while also steering clear of using an overly complicated solution for this problem that has such a nice intuitive interpretation.

2. Problem Statement

2.1. General problem

The correspondence problem in itself is straightforward to state. We start with two rectified images of the same scene, from slightly different viewpoints. An example of such two images follows. Note that the image on the right is taken from a viewpoint slightly to the right of the original one as the gray container now almost touches the left side of the frame while in the original picture there is a noticeable border.

I obtained a dataset of such images from the Middlebury vision group web site, as cited later.

2.2. Dictionary result representation

Now, given a pair of such images, the problem is to match each pixel coordinate in the first image with a pixel coordinate in the second image, where a "match" is defined as the location of the same physical point in the second image. So, for example, the pixel corresponding to the upper left corner of the letter R in the first image should be matched with the same upper left corner of the letter R in the second image.
Given this definition, a correspondence method takes in a pair of images and produces a dictionary as its result, where the dictionary's keys are the pixel coordinates in image 1, and the dictionary's values are the pixel coordinates in image 2. This is done for every pixel in the first image. Note that some pixels of the first image may not match to anything, such as the leftmost columns of the first image in the above example, since they simply do not appear in the second image. Similarly, not all pixels in the second image might not be matched for the same reason.
Our goal is thus to come up with an intuitive, fast and accurate method to construct this mapping dictionary.

3. Algorithm description

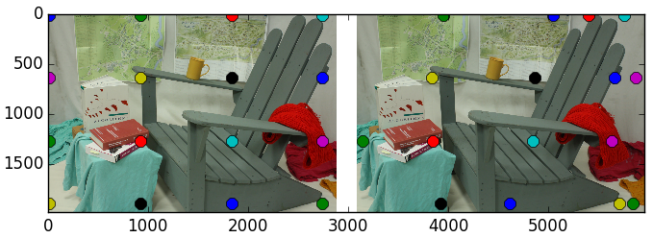
3.1. Overview

In broad strokes, our algorithm starts by getting the pair of images to compute correspondences on. It then picks `NUM_CORRESPONDENCES` initial points to compute correspondences for, where `NUM_CORRESPONDENCES` is a parameter of the algorithm (see discussion in last algorithm section for more details on parameter settings). It then uniformly distributes the desired number of correspondences throughout the initial correspondence image (essentially, a grid with `NUM_CORRESPONDENCES` points is created). The algorithm then computes the correspondences for these grid points as best as it can, using NCC as its basis but also using non-local constraints as described in the next section. Finally, based on the correspondences, the algorithm splits up the original images and recursively calls itself on the split up parts of the original images, reconstructing the full correspondence result along the way.

3.2. Convex optimization formulation

This is the core of the algorithm. As described in the overview, the goal of this step is to compute the best possible correspondence result for a given subset of pixel locations distributed on a grid in the first image. An example of the grid distribution of points to compute correspondences for is given below:

Note that intuitively, there are non-local ordering properties that hold for these grid points. For example, the yellow point in the first line must appear first in the corresponding line in the second image, the black point must follow it, and the blue point must follow both the yellow and the black point. The normal normalized cross correlation algorithm of course does not take this into account and thus sometimes does very poorly. An example of this is shown below:



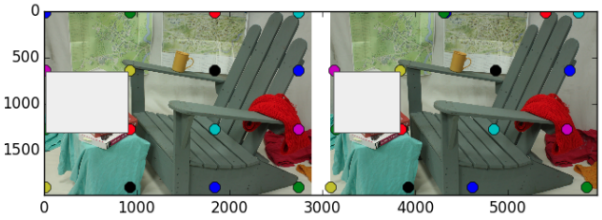
In the above output the of the original normalized cross correlation algorithm, you can see for example that there are large errors with the pink point in the second line and the yellow dot in the last line, as they are on the complete opposite side compared to where they should be given the physical distribution of the objects/points.

To combat this issue we formulate the problem as follows. For each grid point in the original image, we create n binary variables (n being the width in pixels of the images), where setting one of them to true means that that's the column the grid point was matched to. We create constraints such that at least one and at most one of these variables can be set to true, which correspond to picking a unique column.
Next, we use the normal cross correlation algorithm to compute the similarity scores of all pixels in the corresponding row for each grid point. The problem can then be formulated as maximizing the total similarity score by

assigning the variables we've just created.
The one remaining issue is how to formulate the constraint that the points have to be in order. This can be done as follows.
Note that the first pixel in a given line will have a set of binary variables associated to it like `[0,0,1,0,0,...]`, and the second pixel might have a set of binary variables associated to it like `[0,0,0,0,1,0,...]`, and so on. We basically need to make sure that the second pixel doesn't set a variable to 1 before the first pixel has already set a pixel to 1. Thus, we simply create a constraint for each $1 \leq i \leq n$ that says that the sum of the first i elements of pixel 1's choice variables is larger than the corresponding sum of the first i elements of pixel 2's choice variables, and so on, for all pixels in a given line.
This setup is exactly equivalent to having an ordering constraint. The one issue with it is that it requires creating $(k-1)n$ constraints where k is the number of pixels in a given row. However, this can be sped up quite easily by doing an approximation where we simply skip over some of the constraints (i.e. instead of i going from 1 to n in the formulation above, i goes from 1, to for example 11, to 21, adding `SKIP` each time). This approximation works very well in practice and the skip parameter will be discussed more in detail in the parameter settings section.
One final thing to mention is that of course using binary choice variables is infeasible to do in practice, because to find the correct result we would have to enumerate all possibilities which would be extremely slow. Instead, we simply relax the problem, making the choice variables take real values. Then, once the convex problem is solved, we recover the final (binary-valued) solution by setting the argmax for a given pixel's choice variables to 1 and setting all others to 0. Again, this approximation is widely used and works quite well in practice as we'll see in the results section.

3.3. Recursive step

As described in the previous section, the convex optimization part can give us a very good result for a small number of points. However, it is not tractable to simply run all of the pixels through this system, as the number of constraints, as described before, is $(k-1)n^2$ total, where k is the number of pixels to compute corresondences for per line and n is the width of the image.
To avoid this issue, then, we simply split up the image based on the correspondences we obtained into rectangles, and then recursively run the original algorithm on the subimages, until we got to such a point that there are few enough points to just solve via brute force.
An example of the subgrid idea is shown below:



In the picture above, intuitively the pixels in the white rectangle in the left image must have correspondences in the rectangle in the white rectangle of the image on the right.
Note that of course in some cases the alignment will not be perfect, and the subgrid might not necessarily be a rectangle. In such cases we simply take the smallest rectangle containing the true subgrid we obtain as the subgrid to use.

3.4. Speed saving approximations and parameter settings

There are a couple of parameters of my algorithm that can be adjusted; we will discuss them here.

3.4.1 `NUM_CORRESPONDENCES`

Where this parameter comes into play was described in section 3.2. I tested a few values of this parameter and found that setting it to 25 works well. In theory, increasing or decreasing it should not have much of an effect on accuracy; however, if it is set too small or too large it will have the effect of slowing down the computation. This is because if it is set too large, the number of points we will be attempting to solve by convex optimization will be too large, and if it is set too low, the grid will not be split up enough and unnecessarily many recursive calls will occur.

3.4.2 `SKIP`

Again, this parameter's use was described in section 3.2. Basically, a higher `SKIP` means that the convex optimization code will run faster, but it will possibly be less accurate because it might skip enforcing an important ordering constraint. Generally, I found that even a very high setting such as 20 (i.e., only every 20th correspondence constraint is being enforced) actually works quite well. In practice, I wrote a function that starts by setting `SKIP` to a high value, then takes a result of the convex optimization part and checks that all the points are ordered correctly. If they are ordered correctly, no further work is needed and the generated solution is returned.
On the other hand, if the returned result has some points out of order, the current value of the `SKIP` parameter is halved,

and the convex optimization part is rerun. This entire process is repeated up until we get a valid result (there is guaranteed to be such a result if we enforce all the constraints, i.e. once we get to `SKIP` = 1).

4. Experimental Setup

4.1. Dataset

As mentioned earlier, I am using the Middlebury vision dataset, which is made up of a large number of pairs of rectified images and corresponding depth information.

4.2. Gold correspondence generation

For each pair of images, we use the depth information provided to generate a gold correspondence result. We know that:

$$\text{disparity} \propto \frac{B\dot{f}}{z}$$

where z is the depth (this is given by the dataset), the focal length is given as well in the metadata of the dataset and so is the distance between camera centers. With some calibration, then, we can use the depth information to reconstruct the correspondence locations in the dictionary format specified in the problem statement section.

4.3. Evaluation metrics

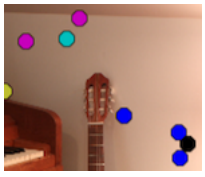
Given that we can compute a gold correspodence result as described in the previous section, we can use a simple evaluation metric, consisting of computing the average absolute error in the column assigned to all correspondences (note that since the images are rectified the row is always equal – and correct – in between the two images so it is ignored when computing accuracy statistics).
I also wrote a Python script that takes a given pair of images and a dictionary result from a correspondence method, randomly samples a set number of points from the dictionary result, and creates a color coded plot, where if two points have the same color they are a correspondence. This method was implemented mostly as a debugging aid and a sanity check to see that the results are making sense. However, the plots are also very useful in seeing the huge advantage in accuracy that this method presents over using the standard correlation based correspondence algorithms.

5. Results

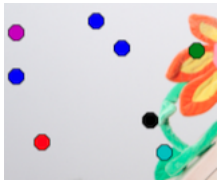
I ran my algorithm on the dataset from Middlebury mentioned earler. Since this dataset consists of very high resolution images (2820 x 1920), regenerating every single correspondence for all pixels takes a long time on a laptop, but I have nevertheless obtained full results on a few images, and I also ran many more images by just getting a subset of the pixels' correspondences. Both of these gave me very encouraging results especially when comparing against normalized cross correlation, which was my target – essentially this algorithm is not much more complicated than normalized cross correlation yet gets much better results.

5.1. Example image results

I will now show examples of several images' computed correspondences, so that it is clear even by visual inspection that my algorithm performs quite well. For each of the results shown in figures 1,2,3, a randomly selected subset of correspondences is shown and color coded so that it is easy to see which location maps to which other location. The reason for doing this randomized selection plotting is simply so that the resulting image is not too cluttered with points so that the results are still clear.
Note that upon visual inspection, the results are of course not perfect but they are nevertheless very good. Especially interesting are the results on fairly uniform surfaces that we nevertheless get correct. For example, if running just normalized cross correlation, you would expect the following points in the piano image to have many errors; however our algorithm gets them almost exactly right, and the general physical layout of the points is preserved for sure, which will result in a more accurate depth result:

A similar pattern can be seen in the other images as well, for example in the flowers image there are several groups of pixels that are simply all on the same background wall; again, these would be hard to get correctly using the traditional methods:

Note that, again, while the result is not perfect, the correspondences are certainly in the approximately correct physical location, and there are no pixels that are misaligned. Again, this will much improve the depth result obtained from our method compared to using normalized cross correlation.

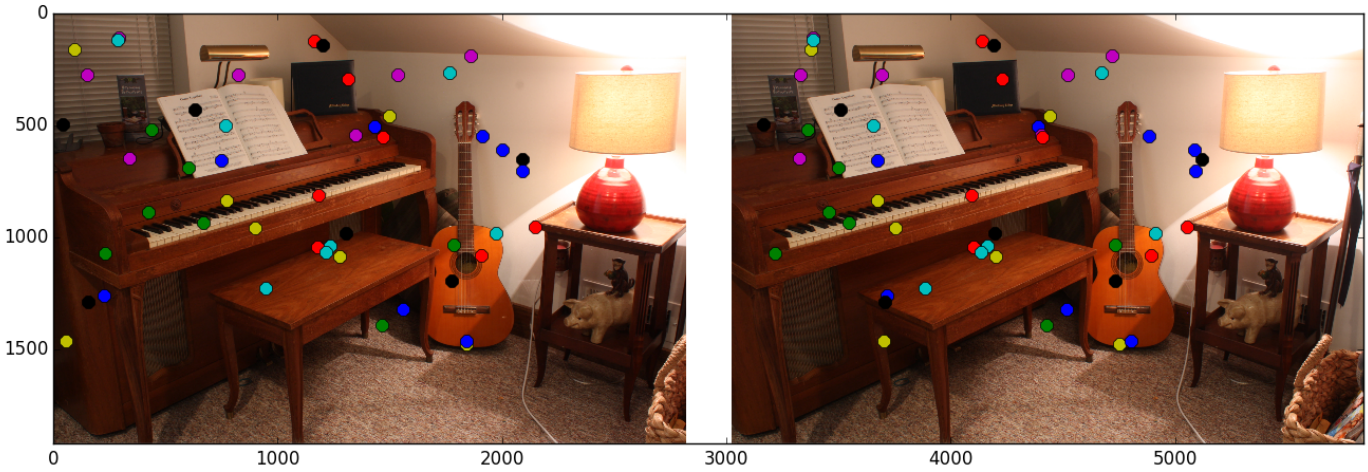


Figure 1. Piano image result

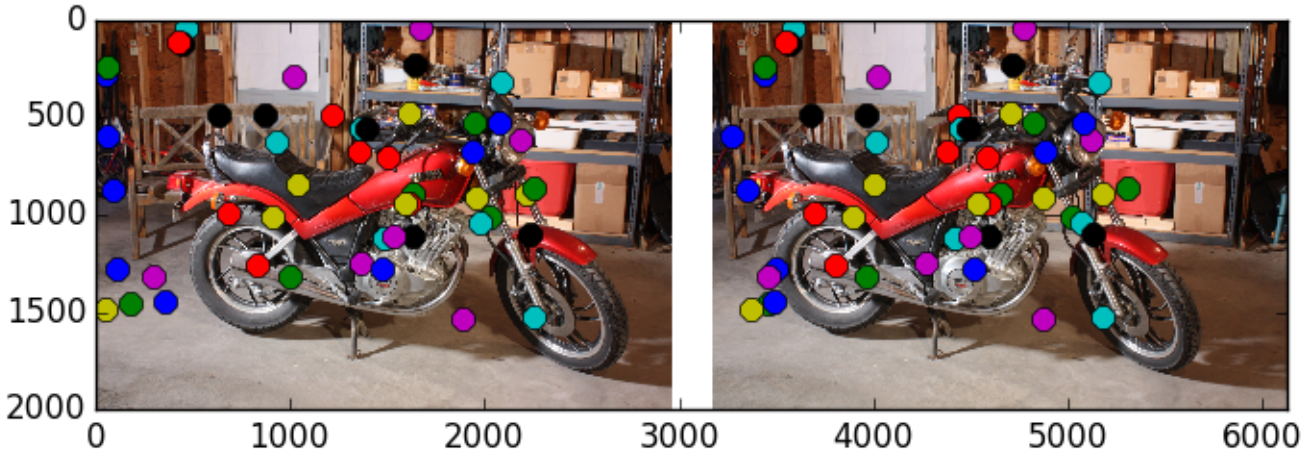


Figure 2. Motorcycle image result

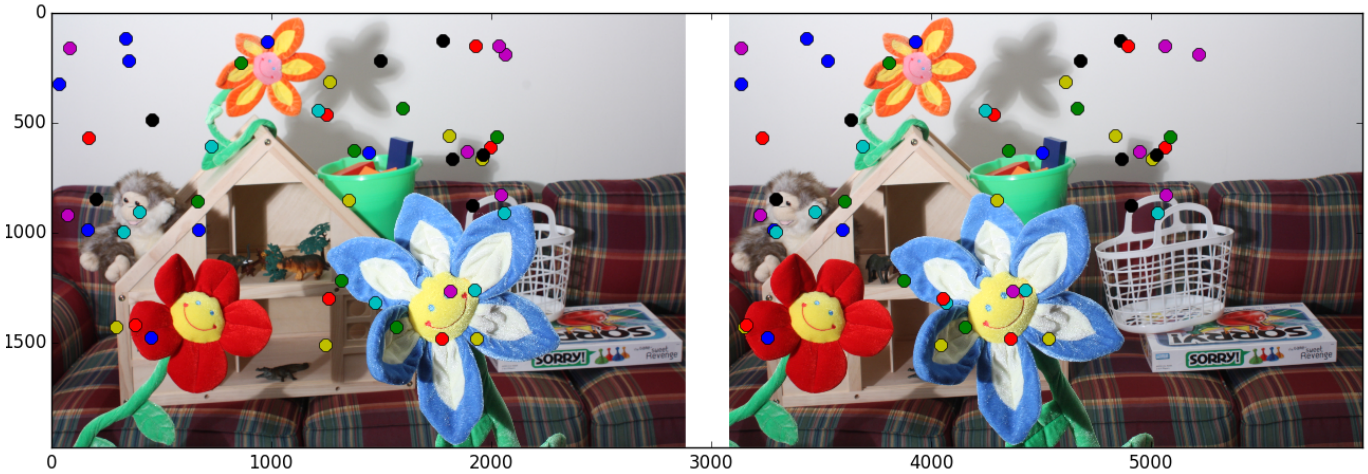


Figure 3. Flowers image result

5.2. Statistics

In addition to doing some visual inspection I computed some statistics on the results we obtain both through our

new improved algorithm and through the legacy normalized cross correlation version to see how they compare – I used the metrics as described in section 4.3. This computation was only done on the grid points as described in 3.2, because it was not feasible to get the old algorithm to return a full result given its large $O(n^3)$ complexity and the large size of our pairs of images in the dataset. See figure 4 for the graph.

There are several interesting things to note here. First, it is clear that we are much better in basically every case. The only image in which we do very slightly worse (Shelves) was one which happened to have a lot of differently colored pixels, so one that was easy for the normalized cross correlation algorithm as well. Another interesting image was Jadeplant, where we reduced the average error by around 3x. Upon closer inspection of this image, it can be seen that its background is very uniform and almost completely black, which is a hard case for the normalized cross correlation and provides a good reason to use our improved algorithm instead.

Finally, I also computed the total average improvement of my method as compared to normalized cross correlation and found that on average the error my method makes is about 36% lower than the error the NCC method makes, which certainly is a significant improvement.

6. Conclusions and future directions

Based on the results that can be seen in section 5, clearly we see that this algorithm is a significant step up from the previous work of normalized cross correlation. It is also much faster, given its use of divide and conquer, and therefore is just more suitable to use in practice in general. It is also still a fairly intuitive method, giving us the confidence that the model is a good one and not just a coincidental result of a complicated model performing well due to overfitting the input data. I therefore believe that my goal of creating an intuitive, fast, and accurate correspodence method was achieved.

In the future, there are several further directions that we could take to keep improving on the algorithm. For example, even though the algorithm performed well in all of my practical tests, it does have one theoretical weakness which could be remedied if it becomes an issue (again, I have not seen this being a problem but to make the algorithm more robust this could be one thing to look into). At each step of the recursion, the decision the algorithm makes about where an individual point goes is essentially not reversible – this means that if the initial grid gets solved really badly, all points within the resulting subgrids will probably get a pretty bad result as well, if the subgrid that was decided on turned out to be wrong in the first place. To remedy this, one could imagine that instead of having an evenly spaced grid, one could make the algorithm look at a line, pick out the most unique pixels in it (i.e. if all pixels are blue except one that is red, pick the red one) and compute the correspondence grid based on those. Given this process, it would be less likely we accidentally subdivide the image into bad grids since the pixels we'd be computing the correspodences on would hopefully be characteristic enough to get a very precise result on them. This would also help address any occlusions; presumably this would filter the initial points such that if any of them were not present in the other image, those points would be ignored. Currently, occlusions are not really addressed; however, I have not seen this to be an issue in practice, most likely given that the dataset I was using was from camera pictures taken quite close together so it is unlikely to have many occlusions.

Nevertheless, the key take home message is that this method works very well in practice – there are some small improvements and robustness additions that could be made but I think the results section shows clearly that the core idea is a good one.

Finally, the full implementation is available for you to try and play with – see the first link in the references section.

7. References

1. The implementation is available in full on my Github profile: `https://github.com/houbysoft/correspondence-231a`
2. The class lecture describing prior methods like NCC: `http://web.stanford.edu/class/cs231a/lectures/lecture6_affine_SFM.pdf`
3. The Middlebury dataset: `http://vision.middlebury.edu/stereo/data/scenes2014/`
4. Python library used to solve convex problems: `http://www.cvxpy.org/`
5. chpatrick's library for reading PFM matrix files into Python: `https://gist.github.com/chpatrick/8935738`

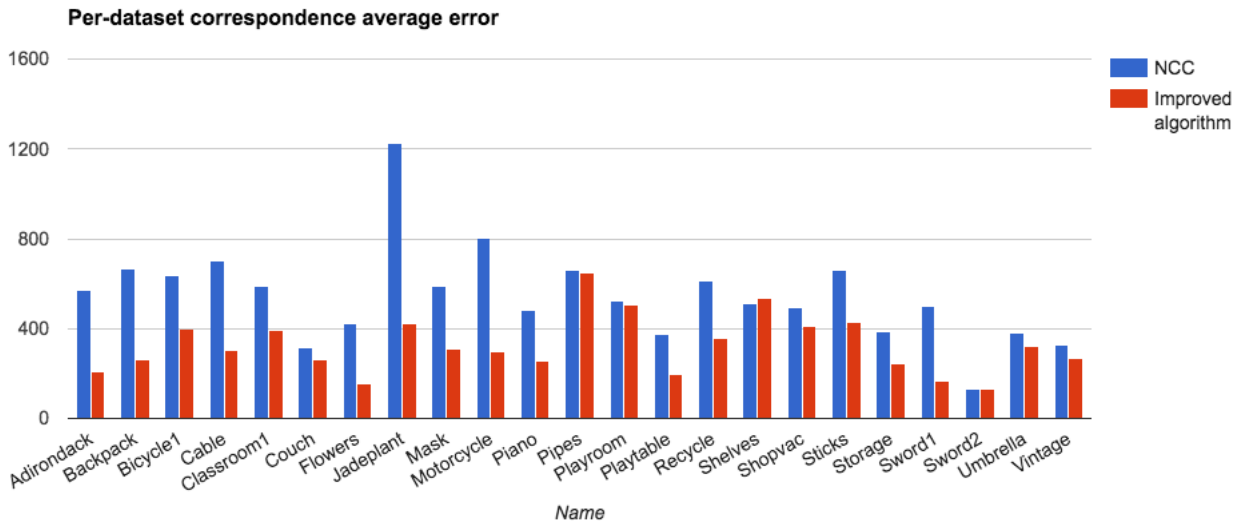


Figure 4. Error statistics