Bachelor thesis
Computer Science

Radboud University

Approximate Nearest Neighbor Field Computation via k-d Trees

Author:
Jeftha Spunda
s4174615

First supervisor/assessor:
dr. Fabian Gieseke
fgieseke@cs.ru.nl

Second assessor:
prof. dr. Tom Heskes
t.heskes@science.ru.nl

August 15, 2016

Abstract

An Approximate Nearest Neighbor Field (ANNF) describes the coherency between two images A and B by approximating the nearest neighbor from image B for every pixel patch in image A. In this thesis we propose an algorithm using k-d trees and PCA to efficiently compute an ANNF between two images. This approach is then compared to a state-of-the-art method called PatchMatch which tackles this problem in a different way. Because both methods exploit different aspects of the data, it is not directly clear which method is more suited for ANNF computation. This research aims to provide a better insight in this area. What we find is that PatchMatch yields reasonable accuracy about 3-4 times as fast as our approach, but when given enough time a k-d tree + PCA will surpass accuracy of PatchMatch.

Contents

Chapter 1

Introduction

1.1 Nearest Neighbor Field

A problem in computer vision is matching image patches between two images A and B. More specifically, finding, for every pixel patch in image A, the most similar pixel patch in image B. This means that we are finding the nearest neighbor for every patch in an image. The result is a *nearest neighbor field* that describes the mapping from image A to image B as seen in Figure 1.1 below.

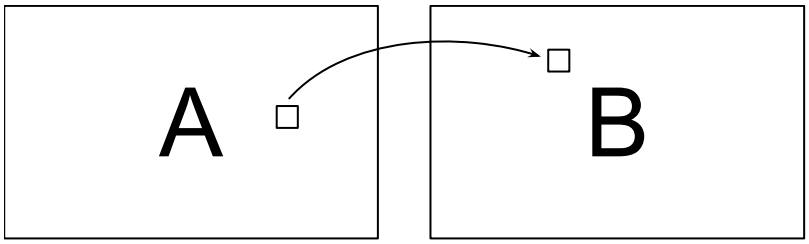

Figure 1.1: The arrow represents the nearest neighbor mapping between the pixel blocks in A and B. Applying this to every pixel block in A creates a nearest neighbor field between A and B.

A nearest neighbor field has many different applications. It is mainly used in computer graphics. For example in image retargeting, completion and reshuffling [1] (See Figure 1.2), Super-Resolution (upscaling an image while maintaining high quality detail) [4] and image denoising [2].

Figure 1.2: Example of application of an ANNF. It is used in image editing tools and makes use of the PatchMatch algorithm to compute an ANNF. (Image from PatchMatch paper [1])

Computing a nearest neighbor field is a computationally expensive task. This is because every pixel patch in image A has to be compared with every other pixel patch in image B. In an image of size 1920*1080 pixels there are 2067604 overlapping patches of 3 by 3 pixels in each image. Each pixel in a patch is represented by three values (RGB) that determine the color. This means that a pixel patch of size 3 by 3 is a vector of 27 dimensions. For a pixel patch of size 8 by 8 this vector grows to a size of 192 dimensions.

Speeding up the construction of a nearest neighbor field is done by settling for approximate nearest neighbors, instead of exact nearest neighbors. This gives rise to the term *Approximate Nearest Neighbor Field* (ANNF).

Much research has already been done in this field. Some popular algorithms that are used to compute an ANNF are PatchMatch and Coherency Sensitive Hashing (CSH).

PatchMatch is a randomized algorithm that generates an ANNF by incremental updates. It begins with a random field. With every iteration it then goes through two phases: propagation and random search.

Propagation attempts to improve the current position in the field by using information from its direct neighbors. Random search attempts to do the same, only this time by randomly by testing a sequence of candidate offsets. This typically converges after 4 to 5 iterations.

Furthermore, some image editing tools are provided with PatchMatch that use this algorithm to achieve effects as shown previously in Figure 1.2.

CSH is an extension to PatchMatch which speeds up PatchMatch by 3 to 4 times. It uses hashing to seed the initial patch matching and, alike PatchMatch, uses image coherence for propagation. This hashing allows for faster propagation and thus results in a speedup compared to PatchMatch.

1.2 Goal

In this thesis we propose an algorithm for computing an ANNF between two images. This approach uses a traditional k-d tree to represent the patches, combined with PCA to reduce dimensionality. Measuring performance is done by running our algorithm on pairs of images and reporting the runtime for ANNF construction and plotting it against the accuracy of the matches between pixel patches.

The main goal is to assess to what extent using our algorithm is viable for ANNF computation. A critical step in speeding up a nearest neighbor field construction is reducing the dimensionality of the pixel patches. There are a number of ways to achieve this, and for this research we opt for PCA. We will show that PCA can significantly decrease runtime, but at the cost of accuracy.

What we aim to find is a good balance between accuracy loss and runtime by trying different values for fitting the PCA model and changing the number of dimensions that the pixel vector is reduced to.

1.2.1 Comparison to PatchMatch

PatchMatch [1] is a state-of-the-art method for ANNF construction. In this research we compare our algorithm's performance to that of PatchMatch's to see how using a traditional k-d tree approach holds up against a method like PatchMatch. We do so by running a series of tests on several image pairs and plotting performances side by side. PatchMatch includes a method that allows for image reconstruction. Given an image B and an ANNF from image A to image B, image A can be reconstructed. This is done by picking a pixel value based on a voting mechanism. A pixel is voted upon by all the patches in which it is included to determine its final value.

To further visualize the comparison between our algorithm and PatchMatch, we will feed ANNFs generated by our algorithm into PatchMatch's reconstruction method. This allows us to evaluate how suited our approach is for image reconstruction.

Chapter 2

A k-d tree based ANNF computation

In this chapter we give a more exact definition of an ANNF and provide an in-depth description of our method and show the different aspects and phases that our implementation consists of. We do so by first showing a quick overview of our method, before discussing each aspect in more detail. To make everything more explicit, pseudocode can be found in Section 2.3.

2.1 Approximate Nearest Neighbor Field

A nearest neighbor field (NNF) describes the correspondences between two images by showing how every pixel patch in image A is mapped to its nearest neighbor from image B.

Let A and B be two RGB images represented by a 3-d matrix of size $h \times w \times 3$ (Figure 2.1) and let AP, BP be the sets of all patches in image A and B, respectively. Let $p \in AP$ be a pixel patch of height and width n, flattened to a vector of size $3n^2$ and let f be a function such that $f : \mathbb{R}^{3n^2} \times BP \to \mathbb{R}^3$ where $f(p, BP) = (x, y, d)$. x and y represent the coordinates of the pixel which is in the top left corner of the pixel patch $q \in BP$ that is the nearest neighbor of patch p. d is the distance between p and q, for some distance metric.

In this research we are using the Euclidean distance metric, given by:

$$d(u, v) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_i - q_i)^2}$$

For two vectors u and v of dimensionality i.

Now we define an NNF as a predicate over $\mathbb{R}^3$, such that:

$$NNF(AP, BP) = \forall_{p \in AP, (x,y,z) \in \mathbb{R}^3}[f(p, BP) = (x, y, z)]$$

Reshaping NNF(AP, BP) into a 3-d matrix of size $h - n + 1 \times w - n + 1 \times 3$[1], gives us the final representation of a nearest neighbor field that we are using in this research.

[1] there are $h - n + 1 * w - n + 1$ *overlapping* patches of size n in an image of size $h \times w$

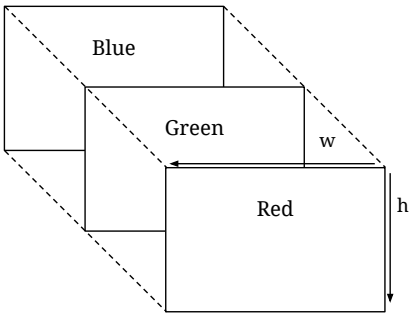

Figure 2.1: RGB image representation

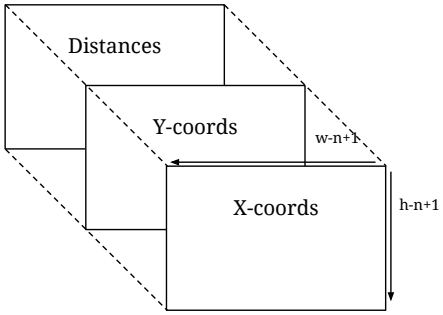

Figure 2.2: NNF representation where $h \times w$ are original image dimensions and n is patch height and width.

The NNF has 3 layers (numbered 0, 1 and 2). The first layer contains all the x-coordinates of the patches in B. So, the NNF on index $(x, y, 0)$ contains the x-coordinate of the patch in image B of which the patch in image A on position (x, y) (in image (A)) is the nearest neighbor. Similarly, the second layer contains all the y-coordinates of the patches in B. Finally, the third layer contains all the Euclidean distances (Figure 2.2).

Approximation

Because computing an NNF as described above is very computationally complex as image size increases, approximation is required. This is done by reducing the dimensionality of the patches in AP and BP. Let AP' and BP' be the sets that contain dimensionality reduced patches and $p' \in AP'$.

Computing $f(p', BP')$ will now return a 3-tuple that is not guaranteed to match $f(p, BP)$. This is due to the data loss caused by dimensionality reduction. The more dimensionality reduction is applied, the less likely it becomes that $f(p', BP') = f(p, BP)$.

We now define an *Approximate* NNF (ANNF) as a predicate over $\mathbb{R}^3$, such that:

$$ANNF(AP, BP) = NNF(AP', BP')$$

2.2 In-depth algorithm overview

In short, our implementation can be described in the following way. First, the images A and B that the ANNF will be built for are read. This is followed by dividing the images into patches. When the patches have been created, dimensionality reduction in the form of *PCA* is applied.

After dimensionality reduction, the algorithm builds the *k-d tree* from all the patches from image B and then finds the nearest neighbor in this tree for every patch from image A. Reshaping and rearranging the output yields the first two fields of the ANNF (x and y coordinates). Lastly, the L_2 distances between patches from A and their nearest neighbor in B are computed in the *original* patch representation (so not in the reduced dimension space created by PCA) and added to the ANNF.

Furthermore, the algorithm provides functionality to export the ANNF to .mat format for easy importing in MATLAB, used for the reconstruction of an image.

Code has been written in Python, making use of NumPy, SciPy, scikit-image and scikit-learn libraries. A more detailed explanation follows below.

2.2.1 Image preparation

Image representation Before any sort of computation can be performed on images, it is necessary to read them from a file and represent them in the right data structure. For this we use the *scikit-image* library. Calling `skimage.data.imread(filename)` loads an image from file and returns it in the form of an *ndarray* (N-dimensional array). This array is shaped as shown in Figure 2.1.

Extracting patches Because nearest neighbor computations are performed between patches, we have to be able to divide an image into overlapping pixel blocks of size n. In order to achieve this efficiently we use the *scikit-learn* library which provides methods to extract patches from an array. Calling `sklearn.feature_extraction.image.extract_patches_2d(image , (patch_height, patch_width))` extracts all overlapping blocks of size `patch_height * patch_width` from `image` (in our case `image` means the ndarray from `skimage.data.imread(filename)` and `patch_height` and `patch_width` are both n).

This creates a list of all pixel patches of size $3n^2$ in an image (3 layers deep, because every pixel has is represented by 3 values (RGB)). Every pixel patch is represented by its top left pixel. Meaning that the i-th *patch* is a pixel block in which the i-th *pixel* is the top left pixel.

Patch representation After patch extraction, every patch is flattened to a $3n^2$-dimensional vector to prepare it for nearest neighbor computation. We do so by using *NumPy*. NumPy is a Python package that allows for very efficient operations performed on large arrays and matrices. Using `numpy.reshape()` we can flatten a pixel patch as displayed in Figure 2.3.

2.2.2 Dimensionality reduction

Because dimensionality is a very important parameter in decreasing the runtime, it is necessary to implement a form of dimensionality reduction in addition to using a k-d tree. For this

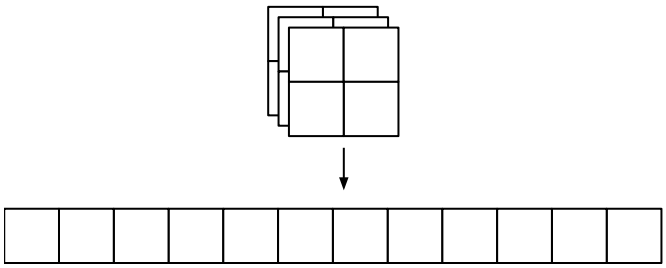

Figure 2.3: Pixel patch of size 2 converted to a 12-dimensional vector

research we are using the PCA implementation from *scikit-learn* to reduce the dimensions of pixel patches. Calling `pca = sklearn.decomposition.PCA(n_components=x)` creates a PCA object which, after fitting, is able to reduce data to x components.

Choosing the number of components in a PCA reduction can greatly affect the outcome of a nearest neighbor computation, so this needs to be chosen carefully. The greater the number of components, the more data is kept, but the slower the computation will be. Because a k-d tree loses its effectiveness as the number of dimensions grows, we have decided to limit the number of components to between 2 and 10 for our tests.

Fitting As mentioned above, a PCA model has to first fit a model. Fitting this model is not done on the entire set of patches, but on a *random subset* of both images A and B prior to the reduction. The insight here is that to create a PCA model for images that represents the data well , the entire image is not required. We have found that choosing a relatively small random subset (10% of patches from both A and B) suffices. It is important that this is a random subset, because that allows the model to fit on patches taken from all over the image. In many images, the top left corner does not represent the data well, so taking the first 10% of patches to fit PCA on, would result in a poor fit.

Increasing the size of this subset will not necessarily significantly improve accuracy (as will become evident in the next chapter). At some point, the accuracy stagnates. This insight allows us to save a lot of computation time in the PCA reduction step of the algorithm.

We use NumPy to generate a random subset. For image A, this is done by calling `rand_subset_a = patches_a[numpy.random.choice(patches_a.shape[0], patches_a.shape[0]*x, replace=False), :]`. Note that x should be between 0 and 1 here and it denotes the size of the random subset that is returned. For example, if $x = 0.1$, 10% of patches is selected. Replacement is set to False, because it is not desired to fit the model on potentially duplicate patches.

We repeat this process for image B and concatenate the result to the random subset of image A. Calling `pca.fit(rand_subset_a + rand_subset_b)` readies the PCA model for transformation.

Transformation Now that the PCA-model has been fitted, the actual dimensionality reduction is done by calling `pca.transform(patches_a)` and `pca.transform(patches_b)`.

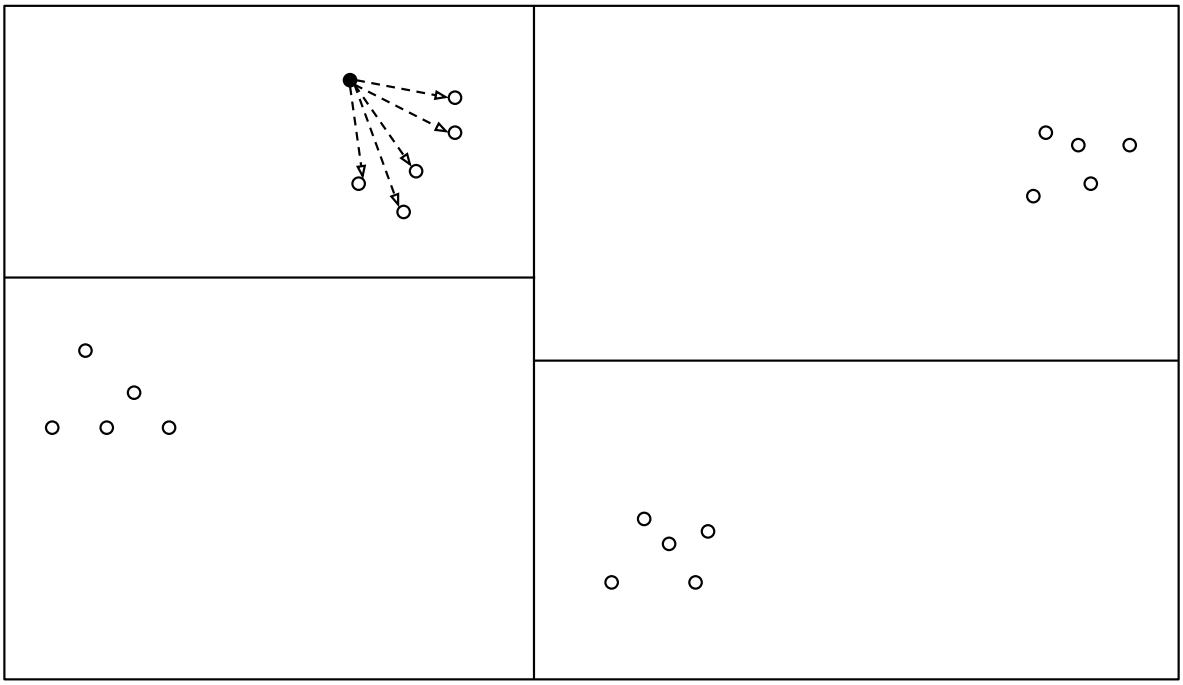

Figure 2.4: Nearest neighbor selection in a 2-dimensional k-d tree, leaf size is 5. A ∘ represents a data point in the tree and the • represents the query point. Note that in the leaf, the L_2 distance is computed for every point. The shortest arrow points to the nearest neighbor.

2.2.3 Finding the nearest neighbors

After the dimensionality reduction has been applied, we build a k-d tree for all the patches in image B and traverse the tree for every patch in image A to find its nearest neighbor. Note that we are interested in just *one* nearest neighbor when computing an ANNF. We do this using the scikit-learn library.

Tree building The k-d tree is built by calling
`neighbors = sklearn.neighbors.NearestNeighbors(n_neighbors= self.nearest_neighbors, algorithm="kd_tree").fit(patches_b)`
. The tree is built by taking median splits, so building is done in $O(n\ log\ n)$ time. We set the leaf size in the tree to its default, which is 20. This means that median splitting stops when there are 20 patches in a leaf of the tree.

Tree traversal Finding the nearest neighbor is done by traversing the k-d tree for every patch in image A. In our implementation this is realized by calling `neighbors.kneighbors(patches_a)`.

When we get to a leaf in the tree a switch is made to brute force, meaning that the L_2 distance is computed to each of the 20 patches in the leaf and the patch with the shortest distance to the query patch is the nearest neighbor. Figure 2.4 visualizes this for a k-d tree built for 2-dimensional data points where the leaf size is 5.

2.2.4 Building the ANNF

The nearest neighbor search results in a list of indices, which tells us which patch in image A is mapped to which patch in image B. We now create the ANNF by taking all the x and y-coordinates from the patches in B and rearranging them such that the ANNF on position [x][y][0] contains the x-coordinate of the patch in B that the patch on position [x][y] in image A is mapped to. Similarly, position [x][y][1] contains the y-coordinate of the patch in B (recall Figure 2.2).

Distance computation The third field of the ANNF contains all the distances between patches in image A and their nearest neighbors in image B. Using scikit-learn to find nearest neighbors already results in a list of distances, alongside a list of indices, but because the patches have been reduced by PCA we cannot use these distances as a performance measure, if we want to compare it to PatchMatch.

Therefore, all distances have to be computed in the original dimension space. We do this by calling `numpyp.linalg.norm(numpy.array(patches_a_old, dtype=numpy.int32) - numpy.array(patches_b_old[indices, :], dtype=numpy.int32), axis=1)`. Note that we are performing this operation on `patches_a/b_old`. These are the original patches, before PCA was applied.

2.2.5 Exporting to MATLAB

In order to allow importing our ANNF field in MATLAB, we use the SciPy library to easily export Python array objects to .mat files. Calling `scipy.io .savemat(filename, "ann_field": self.ann_field)` achieves this. We use the exported ANNFs to reconstruct image A, given just image B and the ANNF from A to B, a functionality that comes with PatchMatch, which uses MATLAB.

2.3 Pseudocode

Algorithm 1: ANNF A → B via k-d trees + PCA

Data: RGB images A and B, patch size n
Result: ANNF A → B

```
1  patches_a_org = extract_patches(image A);
2  patches_b_org = extract_patches(image B);
3  subset = random_subset(patches_a_org);
4  PCA.fit(subset);
5  patches_a = PCA.apply(patches_a_old);
6  patches_b = PCA.apply(patches_b_old);
7  tree = build_kdtree(patches_b);
8  indices = [ ];
   // Find all nearest neighbors
9  for p in patches_a do
10     nearest = tree.search_nearest(p);
11     indices.append(nearest);
   // Use remainder to compute all the x-coordinates for the ANNF
12 x-coords = remainder(indices, (B.width - n + 1));
   // Use floor divide to compute all the y-coordinates for the ANNF
13 y-coords = floor_divide(indices, (B.width - n + 1));
   // Finally, compute distances in original dimension space
14 distances = compute_distance(patches_a_org, patches_b_org);
15 annf = [ ];
16 annf.append(x-coords);
17 annf.append(y-coords);
18 annf.append(distances);
19 return annf;
```

2.4 Additional speedups

There are numerous ways to improve on the current algorithm. Because these are out of the scope of this bachelor thesis these have not been implemented, but could be added as future extensions.

Backtracking One way to speed up a nearest neighbor search using k-d trees is to alter the backtracking behaviour when traversing the tree. In this way, the nearest neighbor search would stop early, instead of potentially traversing the entire tree during the backtracking phase. This speedup would of course come at the cost of some accuracy.

Buffer k-d trees A variant of the traditional k-d tree, called *buffer k-d tree*[3] can be implemented instead of a classical k-d tree used in this research. The buffer k-d tree harvests the power of GPUs to greatly accelerate the process of nearest neighbor searching. In contrast to changing the backtracking behaviour, using a buffer k-d tree would not lead to a decrease in accuracy.

Chapter 3

Results

In this chapter we describe some relevant parameters in this research and how they affect the algorithm's performance. We illustrate this by plotting L_2 distance between two images against the time taken to finish the ANNF computation for different parameter values. Furthermore, we measure performance in the form of ANNF visualizations and image reconstruction behaviour. We compare our findings with PatchMatch's performance.

All tests are done on a Windows 7 machine, an Intel Core i7-3820 CPU @ 3.6 GHz, with 8GB of RAM.

3.1 Data

For this research we are using the VidPairs[1] dataset. The set consists of 133 image pairs taken from movie trailers. The images are all 1920*1080 pixels, but for some tests we have reduced image size to 500*208 pixels. The two images in a pair are about 1-30 frames apart.

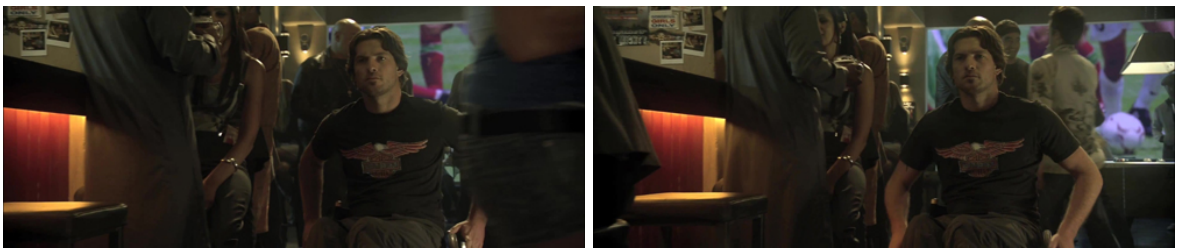

Figure 3.1: An image pair from the VidPairs dataset

3.2 Parameters

Patch size The size of the patch vectors. The size of a patch notably impacts algorithm performance as it grows. A pixel patch of 3×3 produces a $3 * 3^2 = 27$-d vector, but for a patch size of 8×8, this grows to a $3 * 8^2 = 192$-d vector, which dramatically increases runtime.

[1] `http://www.eng.tau.ac.il/~simonk/CSH/`

PCA reduction The number of dimensions that we reduce the patches to. The fewer dimensions we reduce to, the faster our algorithm performs. However, this leads to a loss of accuracy. The bigger the difference between the original dimensionality and the reduced dimensionality, the more information loss.

For example, starting with 27 dimensions and using PCA to reduce to 10 dimensions results in less accuracy loss compared to reducing to 10 dimensions from a 192-d vector.

PCA fitting The amount of patches that we use to fit the PCA model on. The trade-off between accuracy and runtime is not the same as in the PCA reduction step. Fitting to more patches does not necessarily mean a higher accuracy in the end. A good accuracy can be achieved by fitting the PCA model on a *random* subset of patches. However, fitting to fewer patches always leads to a shorter runtime, because less work has to be done.

3.3 Testing PCA reduction and patch size

In this series of tests we show how the average L_2 distances and runtime are affected by different PCA reductions and different patch sizes.

3.3.1 PCA reduction

Consider the plot in Figure 3.2. As can be seen from the figure, PatchMatch is quicker at reaching a reasonable accuracy (± 35). To reach similar accuracy, our approach is about 3 times slower.

Note that the image size is only 500*208 pixels for this test. The original image size in our data set is 1920*1080 pixels. Figure 3.3 shows performance for an image of full HD resolution. A similar result here; our approach is about 3-4 times slower to reach the same accuracy. PatchMatch reaches reasonable accuracy in a shorter time span. Whereas a 500*208 images takes about 1-2 seconds to reach a reasonable accuracy, 1920*1080 takes 20-30 seconds.

Surpassing PatchMatch

What is evident from the two figures presented in the previous section is that our algorithm benefits more in terms of accuracy with every PCA reduction step than PatchMatch benefits from every iteration. In fact, it appears as if PatchMatch benefits less with every iteration. What this implies is that our approach will eventually surpass PatchMatch in terms of accuracy, given enough time. To illustrate this, consider Figure 3.4

This test aims to show how PatchMatch behaves compared to our approach as we increase the number of iterations. It shows that PatchMatch's accuracy will at some point hardly benefit from increased iterations. As seen in Figure 3.3, L_2 distance reaches 9 after 8 iterations. At 60 iterations the distance has only increased ± 2, but it took 50 seconds longer.

Our approach surpasses PatchMatch in terms of accuracy, if given the same amount of time. However, recall that PatchMatch is designed to make ANNF computation viable in real-time applications, like image editing tools, where having to wait 50 seconds is most likely not desired.

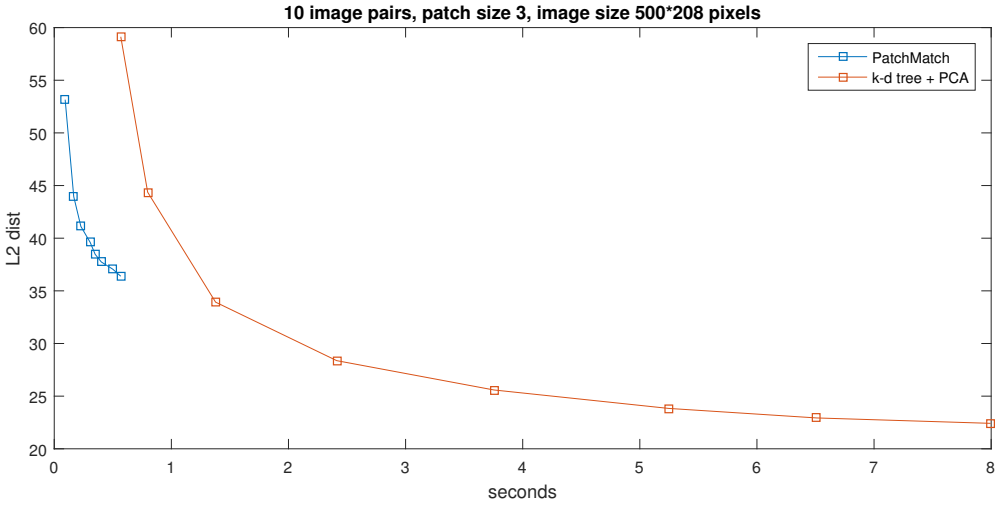

Figure 3.2: Runtime and L_2 distance averaged over 10 image pairs of size 500*208 pixels, patch size 3. Every marker PatchMatch's curve represents one iteration, starting from 1. Every marker on the k-d tree + PCA curve represents the number dimensions that the pixel patches were reduced to, starting from 2 and incrementing by 1 with every step.

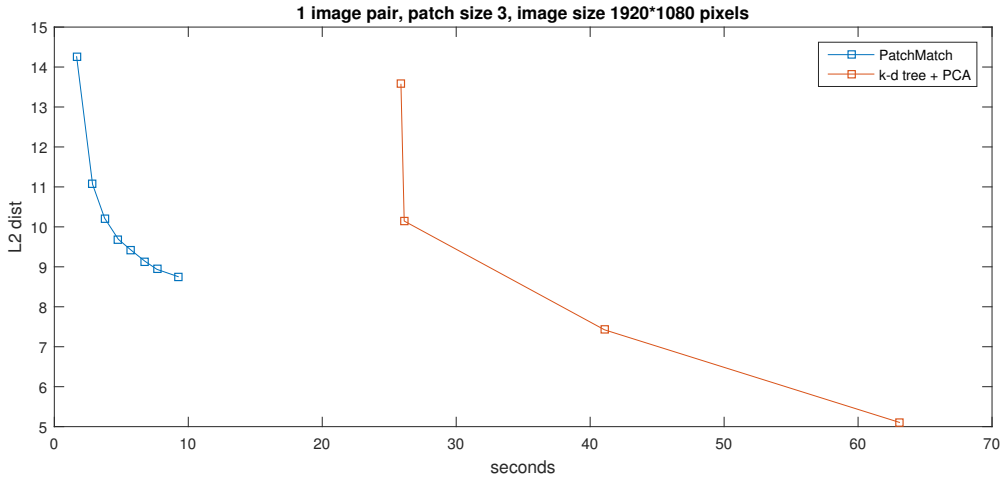

Figure 3.3: Runtime and L_2 distance measured for 1 image pair of size 1920*1080 pixels, patch size 3. PatchMatch ran for 8 iterations. Our approach with PCA to 2, 3, 4 and 5 dimensions.

Returning a reasonable result in a short amount of time is more important, which is exactly what PatchMatch does well, as we have seen so far.

3.3.2 Patch size

To illustrate the effects of an increase in patch size, consider the plot in Figure 3.5. What we can conclude from the difference in L_2 distance is that accuracy decreases as the patch size grows. For our approach this can be explained by the PCA reduction step.

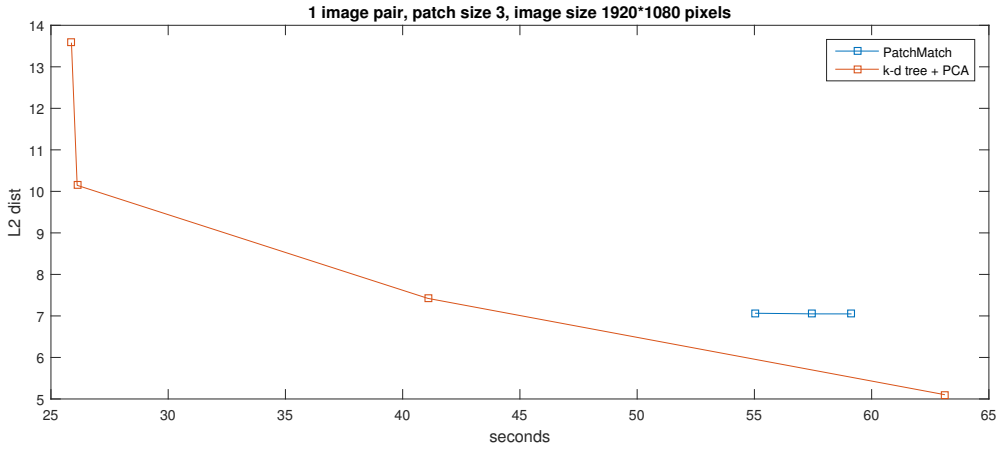

Figure 3.4: Runtime and L_2 distance measured for 1 image pair of size 1920*1080 pixels, patch size 8. PatchMatch ran for 58, 59 and 60 iterations. Our approach with PCA to 2, 3, 4 and 5 dimensions. The image pair is the same as in Figure 3.3.

When patch size is 8, a pixel patch is a 192-d vector. Reducing from 192-d to 5-d leads to much more information loss than reducing from 27-d to 5-d, as is the case when patch size is 3. Our previous observation still holds: PatchMatch is roughly 3 times faster.

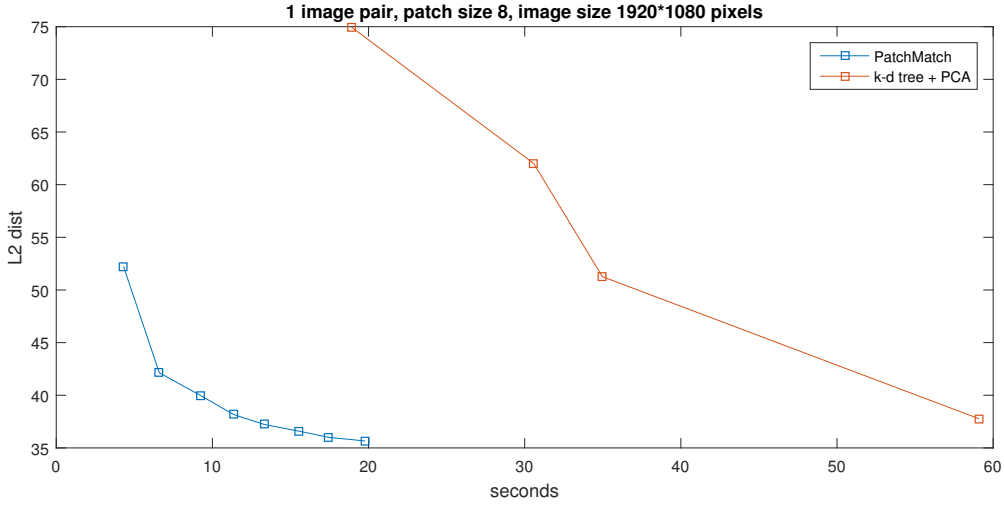

Figure 3.5: Runtime and L_2 distance measured for 1 image pair of size 1920*1080 pixels, patch size 8. PatchMatch ran for 8 iterations. Our approach with PCA to 2, 3, 4 and 5 dimensions. Note that this test was run on the same image pair as in Figure 3.3 above.

3.3.3 Testing PCA fitting

This test aims to show that it is not necessary to use the entire image when fitting a PCA model to the patches. Using a random subset of patches suffices. The plot shown in Figure 3.6 confirms our insight that it is not necessary to fit PCA on the entire set of patches to get a good PCA model. Fitting on more data takes increasingly longer, but the accuracy does not

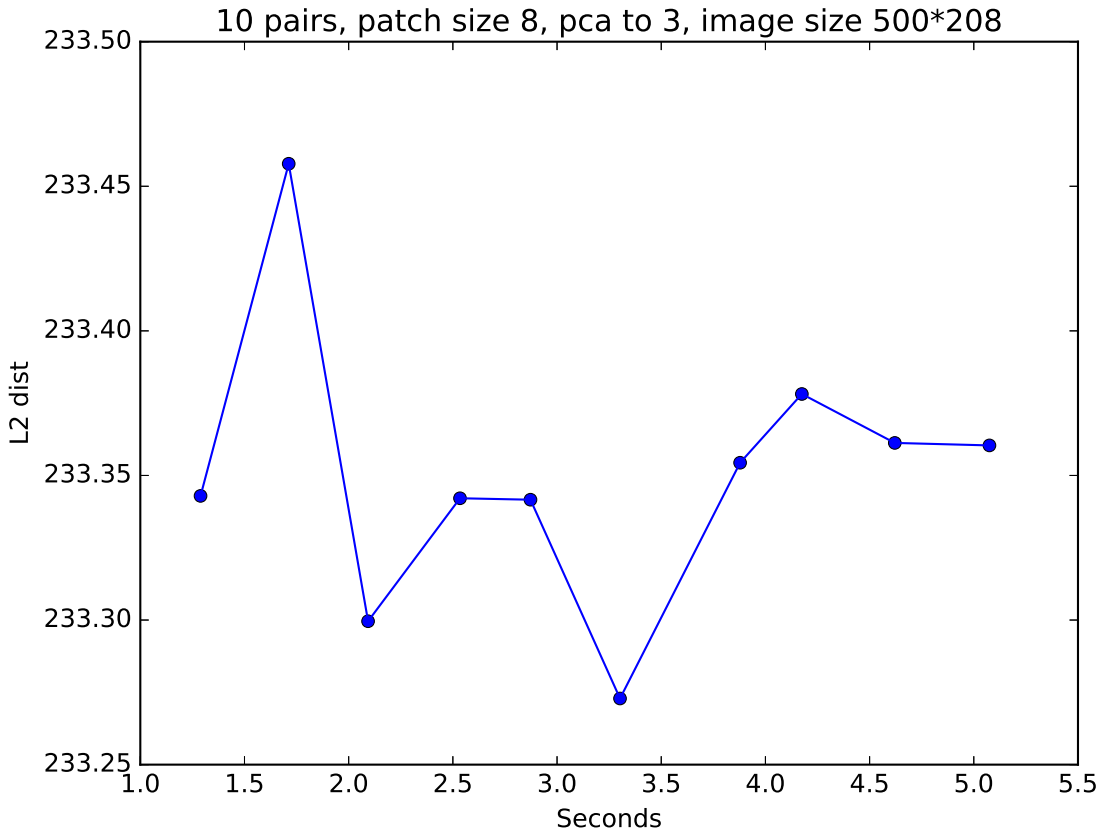

Figure 3.6: The plot above shows performance of our approach averaged over 10 pairs where every marker denotes the percentage of data used to fit the PCA model on. Starting at 10% and incrementing by 10% with every step.

necessarily improve. In fact, in some cases it yields slightly worse accuracy compared to a PCA model fitted on a random subset.

In conclusion, fitting on 10% of patches is sufficient to bring the final L_2 distance within a range of ± 1 of the accuracy achieved by a 100% fit, at a fraction of the time and memory cost. This makes PCA viable in ANNF computation, which was discouraged in [5] due to time cost and instead a Walsh-Hadamard transform [6] was used

3.4 ANNF visualized

Visualizing the ANNF is done by plotting the different layers of the field. In the figures below we show the second (y-coordinates) and third (L_2 distances) layer of the field for a given image pair. In Figure 3.7 we plot the y-coordinates in gray scale and the L_2 distances in hue.

In this plot, PatchMatch reaches about 35 L_2 distance (in 0.5 seconds) and our approach reaches approximately 20 L_2 (in 10 seconds). A lower L_2 distance results in a detailed field which shows more subtleties, compared to a high L_2 distance field, which is still quite rough.

Figure 3.7: ANNF visualization for the image pair in shown at the top. Left shows the y-coordinates. Right shows the L_2 distances. Image size 500*208 pixels, patch size is 3 and PCA was used to reduce to 10 dimensions. Runtime ± 10 seconds. Bottom row shows the y-coordinates of the field that PatchMatch created after 5 iterations. Runtime ± 0.5 seconds.

To show a more detailed view of the ANNF, consider the plots in Figure 3.8. This shows an ANNF for a high resolution image, which allows for a more detailed visualization.

3.4.1 Ground truth comparison

We show a visual comparison of a *ground truth* matched field and our field from Figure 3.7. Ground truth means an *exact* nearest neighbor matching, so PCA was not applied before finding the nearest neighbors, resulting in the most accurate NNF that can be created between two images. Figure 3.9 illustrates this further.

The left image shows y-coordinates from a ground truth matching for the pair in Figure 3.7. The middle image shows a zoomed in region of this field. The right image is a zoomed in region of the y-coordinate plot of Figure 3.7, put here for visual comparison which shows minimal

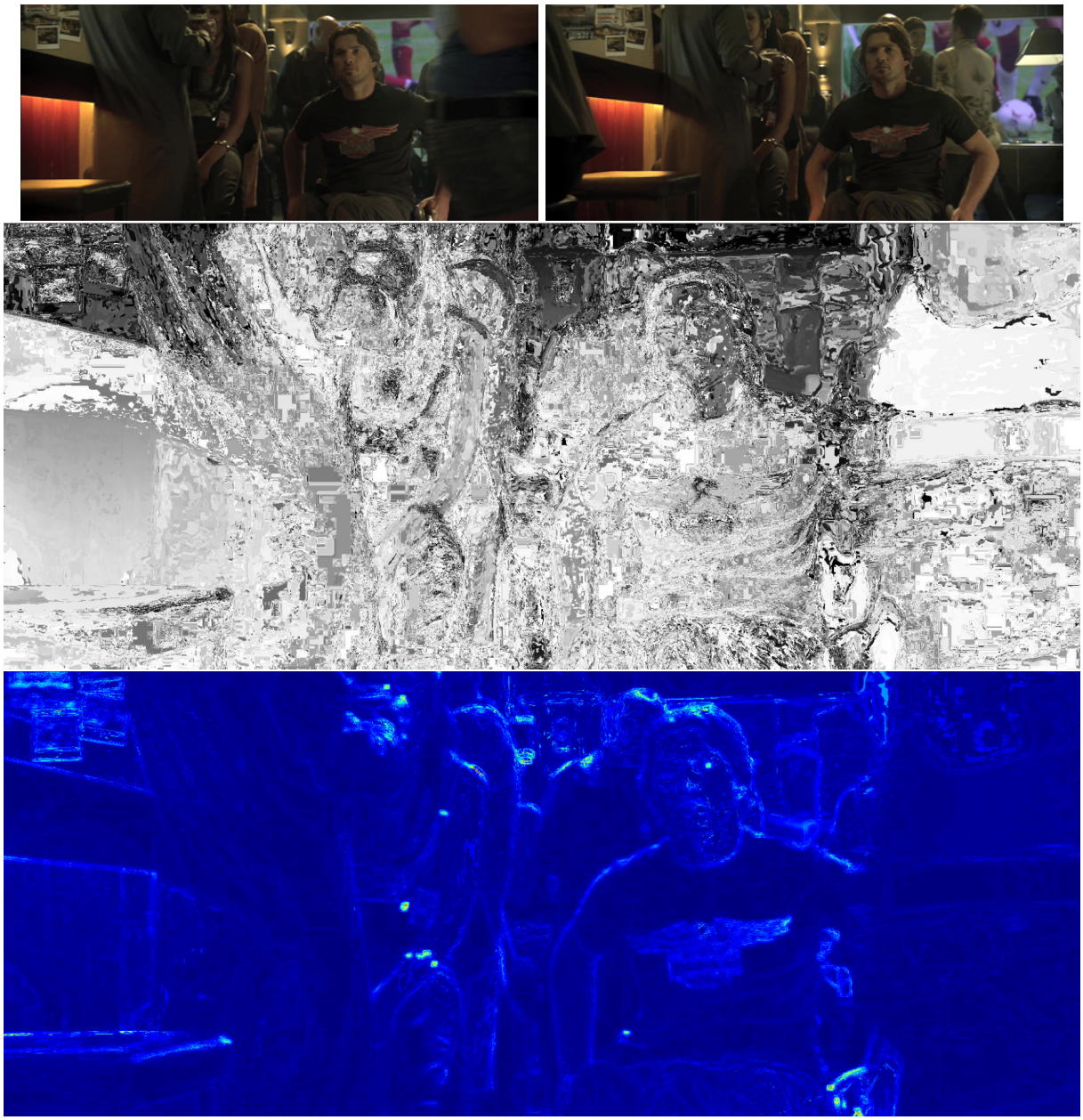

Figure 3.8: ANNF visualization for different image pair. Size 1920*1080 pixels, patch size 8, reduced to 5 dimensions with PCA.

difference. This shows that using PCA to reduce from a 27-d vector to 10-d vector yields a field that is similar to the ground truth.

3.5 Image reconstruction

Given an image B and an (A)NNF A → B, image A can be reconstructed by means of a voting mechanism, which checks for every patch a pixel is included in, in order to determine its final

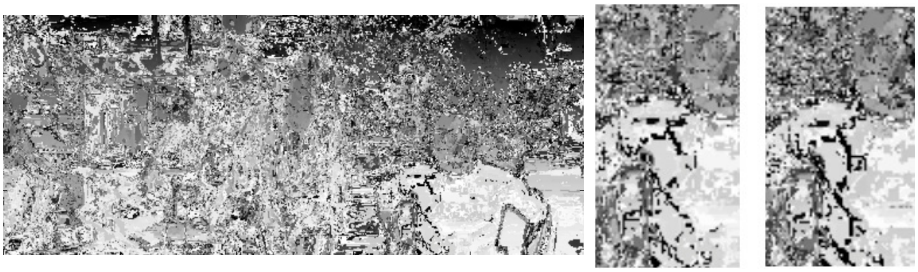

Figure 3.9: ANNF ground truth comparison.

value. This reconstruction method can be used for image editing purposes. It is included with PatchMatch.

Feeding our ANNF into this algorithm allows us to gauge reconstruction performance visually. Consider Figure 3.10. This plot shows reconstructions created for the image pair from Figure 3.7. What can be seen is that PatchMatch is able to create a more accurate reconstruction in 0.7 seconds, which is in line with Figure 3.2. Our reconstruction seems to generate the structures properly, but the colors are not quite close to the original.

A reconstruction detail can be found in Figure 3.11. This plot shows that our algorithm will create a slightly better reconstruction if given more time. Our algorithm surpasses PatchMatch's performance when both are given 8500ms, which is in line with Figure 3.4, where we show that our algorithm benefits from increased runtime, whereas PatchMatch will stop benefiting as much from a certain point onwards.

Figure 3.10: Reconstruction for image size 500*208 pixels, patch size 3. **Top**: Original image, **middle**: Our reconstruction after 0.7 sec runtime (PCA to 3 dimensions), **bottom**: PatchMatch reconstruction after 0.7 sec runtime (10 iterations).

Figure 3.11: Detail of reconstruction for image size 500*208 pixels, patch size 3. **Left**: original detail. **Middle**: Our approach after 8.5 seconds (PCA to 10 dimensions). **Right**: PatchMatch reconstruction after 8.5 seconds (120 iterations).

Chapter 4

Conclusions

We have presented an algorithm for Approximate Nearest Neighbor Field computation. This algorithm is based on a k-d tree approach to perform nearest neighbor search, combined with PCA for approximation. We have tested our approach and compared the results with a state-of-the-art method called PatchMatch.

In our tests we have found that our approach works reasonably well, but is not as fast as PatchMatch. In general, our approach is 3 to 4 times slower to achieve the same accuracy. However, if given enough time, our approach surpasses PatchMatch by not reducing as much in the PCA step.

In terms of reconstruction behaviour, we have found that PatchMatch is more suited to this task, due to its ability to generate reasonable results in a short amount of time. Our approach needs more time to reach the same quality of reconstruction.

Furthermore, it has become clear that fitting the PCA on the entirety of the image is not necessary to get a quality fit. Using a random subset of roughly 10% of all patches is sufficient.

In the future, our algorithm could be extended in numerous ways to improve performance. For example, backtracking behaviour in the k-d tree could be altered or a much more efficient version of a k-d tree, called buffer k-d tree, could be used.

Bibliography

[1] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan Goldman. Patchmatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics-TOG*, 28(3):24, 2009.

[2] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. A non-local algorithm for image denoising. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 60–65. IEEE, 2005.

[3] Fabian Gieseke, Justin Heinermann, Cosmin Oancea, and Christian Igel. Buffer kd trees: processing massive nearest neighbor queries on gpus. In *Proceedings of The 31st International Conference on Machine Learning*, pages 172–180, 2014.

[4] Daniel Glasner, Shai Bagon, and Michal Irani. Super-resolution from a single image. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 349–356. IEEE, 2009.

[5] Kaiming He and Jian Sun. Computing nearest-neighbor fields via propagation-assisted kd-trees. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 111–118. IEEE, 2012.

[6] Yacov Hel-Or and Hagit Hel-Or. Real-time pattern matching using projection kernels. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(9):1430–1445, 2005.