Linking Fourier and PCA Methods for Image Look-up

Daniel Lichtblau

***Abstract*—We show a simple, yet effective, method for storing images, such that retrieval of nearby images is both fast and accurate. The main ingredients are discrete Fourier transforms to extract low frequency components, principal components analysis (PCA) for further compression, and storage in k-D trees. We illustrate the quality of results on the MNIST digit suite and also apply it to chromosome segments.**

***Index Terms*—Principal Components Analysis, image recognition, discrete Fourier transform, discrete sine transform, genome sequence visualization**

I. Introduction

IDENTIFICATION, classification and related processing of images is becoming ever more common, in internet services, face recognition, medical imaging, and elsewhere. In this note we describe a methodology for image identification that is based on low frequency elements compressed by a standard principal components method. We provide short code for this and illustrate the utility in image recognition and classification on the MNIST digit benchmark suite[8]. We then apply the methodology to recognition and visualization of gene sequences. While the ideas behind this dimension reduction method are not new, the simplicity and effectiveness of our approach is compelling.

The main ideas are simple. We start with a reference family of images. After possibly preprocessing with one or more filters (depending on application), we extract low frequency Fourier components using a form of the Discrete Fourier Transform (DFT). This serves two purposes. One is that it provides us with measurements of the images that are relatively unaffected by noise. Another is that it serves to reduce dimension, and this is important for our next steps. We now treat the discrete Fourier transformed images as vectors, that is, stack the individual rows of a given image to get one long vector. We next use principal components analysis (PCA). Specifically, we take the singular value decomposition of this matrix of DFT vectors. We use the largest singular values, and the corresponding left singular vectors, to obtain an array that effectively encodes the original matrix. The column dimension is equal to the number of singular values we have retained. We put this set of vectors into a k-D tree [1]. We use the right singular vectors to later preprocess input for purposes of look-up in the k-D tree.

Fourier and PCA methods have been used separately for purposes of image recognition, [2], [3], [4] being examples of the former and [5], [6] the latter. The methodology of classifying images based on a tandem of Fourier and PCA techniques is itself not entirely new. In addition to the present note it appears in [7], [8], for example, and likely has been independently developed in many other places. The novelty in our approach is the simplicity of the code combined with the quality of the results, computational efficiency, and applicability to genome sequence clustering and lookup.

Our primary goal is to illustrate a practical and simple method on a few different classes of examples. We describe the algorithms in section 2; full code is provided in an appendix. Section 3 shows how this method fares with recognition of the MNIST [9] handwritten digits suite. Section 4 shows an application to classification and visualization of genome fragments [10], including a clustering visualization by multidimensional scaling (MDS) [11]. We remark that this method might also be useful for purposes of compressive genomics, at least in cases where a lossy compression can be used [12]. We conclude with a brief summary followed by the code appendix. All was run in version 10 of Mathematica [13]. A variant of code in the appendix is used for handling image inputs to the **Nearest** function in Mathematica.

II. Description of the main algorithms

Our processing of image sets is split into two parts. One handles the dimension reduction and storing of a set of images, the other is for looking up images (which may come from a different set). The routine **nearestImages** is for this first part. It takes parameters that determine how many Fourier components to keep and how many singular values of a certain matrix to keep. It returns both a k-D tree and a matrix. We need this matrix in **processInput**, where we convert look-up images into a form suitable for use in searching the k-D tree for neighbors. Code for **nearestImages** and **processInput** is in the appendix.

First we provide a step-by-step a description of the underlying algorithm for **nearestImages**.

Algorithm **nearestImages**
Input:
1. A set of m images with all color channels having values in the range $[0,1]$.
2. A value n to determine how many Fourier components to retain.
3. A value k for how many of the largest singular values and corresponding vectors to retain.

Steps:
1. Subtract the mean value from each image.
2. Take the discrete sine transform of the adjusted images.
3. Discard all but the nxn array of lowest frequency components.
4. Flatten each such array so we have an mxn^2 matrix of values.
5. Subtract the mean from each vector in this matrix.
6. Take the singular value decomposition of this matrix, finding only vectors and values corresponding to the k

Wolfram Research 100 Trade Center Dr., Champaign IL 61820, USA
Phone: (217) 398-0700, email: danl@wolfram.com

2470-881X/16 $31.00 © 2016 IEEE
DOI 10.1109/SYNASC.2016.22

largest singular values.
7. Take vector products of the left matrix with the diagonal of retained singular values.
8. For each (row) vector, compute the norm, normalize the vector itself, then augment with some function of that norm (we use the logarithm as this tends to give a good measure of the overall intensity scale).
9. Store the vectors thus computed in a k-D tree.
10. Return the tree and the set of right singular vectors (these are needed for preprocessing inputs prior to look-up).

Here is a description of the algorithm for **processInput**.

Algorithm **processInput**.
Input:
1. A set of m images with all color channels having values in the range $[0,1]$.
2. A value n to determine how many Fourier components to retain.
3. A set of right singular vectors needed to place the images into the same "space" as those in the look-up table.

Steps:
1. Subtract the mean value from each image.
2. Take the discrete sine transform of the adjusted images.
3. Discard all but the nxn array of lowest frequency components.
4. Flatten each such array so we have an mxn^2 matrix of values.
5. Subtract the mean from each vector in this matrix.
6. Multiply this matrix on the right by the right singular vectors from **nearestImages**.
7. For each (row) vector, compute the norm, normalize the vector itself, then augment with the logarithm of that norm.
8. Return this converted set of input images as a list of vectors.

Before proceeding with our examples we comment on the algorithms above. Most can be seen as straightforward dimension reduction. One step that might not be obvious is how we utilize the singular value decomposition (SVD) for image storage and later neighbor look-up. Recall that for a matrix M, the SVD gives a trio of matrices (U,W,V) where U and V are possibly truncations of orthogonal transform matrices and W is a diagonal matrix of singular values. For a "full" SVD no nonzero singular values are removed and we have the matrix identity $UWV^t = M$. With our truncating by retaining only k singular values, the identity becomes instead the best approximation, in a Euclidean norm measure, of M by any matrix of rank k. Thus we have $UW \approx MV$. In **nearestImages** we store UW in a k-D tree. Were we to look up any row vector in the original matrix M, we would first need to transform to MV. While our set of look-up images need not (and typically does not) coincide with the input set, we still need to perform this step in **processInput**, hence the need for step (6) in that algorithm.

III. Simple digit recognition

We now demonstrate a simple image recognizer based on finding neighbors and using a majority voting system to determine matches. We illustrate with the MNIST digit benchmark suite [9]. Code in the appendix shows how to obtain them programmatically. We proceed from there.

First we show a few of the images.

Fig. 1. First several digit images from MNIST benchmark set

Processing the 60000 training images and the test set of the remaining 10000 images runs in less than four seconds on a Ubuntu Linux desktop machine (see appendix).

For the recognition part of the task we use a simple "majority rules" schema. Specifically, we find the five images deemed to be closest, bin these by matched digit, and sort bins by size. The topmost wins (obviously there might be ties, but in such cases we don't have a strong guess regardless).

We run this simple recognizer on the set of 10000 trial images (short code for this is in the appendix). It takes around 13 seconds to do the k-D tree look-ups for all 10000 test images. The majority neighbor strategy finds correct matches on 9781 of these, for a recognition rate of 97.8 %. While clearly a solid percentage, this is by no means the best possible rate. Indeed, several classifiers exceed 99.5 % [8]. Nevertheless it is quite reasonable for a technique that is designed more to find nearby images than to explicitly recognize them. We remark that most of the best classifiers noted in [9] are based on sophisticated methodologies, using either or both of Support Vector Machines (SVMs) and convolutional neural nets.

IV. Genome signatures

An important problem in genomics is that of classifying sequences according to possible species from which they arise. A related problem is to visualize families of gene sequences in such a way that related sequences appear near to one another and, to the extent possible, clustered away from sequences in unrelated species. We will illustrate our approach using the examples from [10].

We begin, as is done in [10], with a set of six chromosomes represented by strings of nucleotide characters. The six chromosomes come from H. sapiens, S. cerevisiae, A. thalania, P. falciparum, E. coli, and P. furiosus respectively. We discard all characters that do not correspond to nucleotides (that is, unrecognized sections), and split what remains into chunks of non-overlapping sequences comprised of 150,000 nucleotides each. This gives 508 sequences in total. We then use the Jeffrey's "Chaos Game Representation" (CGR) [14], [10] to create an image for each sequence. The idea behind this is simple. One

labels a 1-by-1 square with a distinct nucleotide in each corner. Beginning in the center, place a dot halfway from there to the corner corresponding to the first nucleotide in the sequence. From that point, place a dot halfway toward the corner corresponding to the next nucleotide, and continue in this manner until the string is exhausted. Given a positive integer k, if we form pixels at granularity of 2^k then Jeffrey shows that each dot corresponds to a specific character string of length k [5]. Moreover this gives rise to a different computational stratagem, the "Frequency CGR" (FCGR) as employed in [10] (our code is a variation of that used in the supplemental material to [10]). This amounts to tallying occurrences of each k-mer, using these counts to darken corresponding pixels. We do a nonlinear rescaling in order to get an average value that is not too close to white or black.

We show images corresponding to the first sequence in each of the six genomes. Visual differences are fairly clear between all pairs except perhaps the top right and bottom left pictures, corresponding respectively to A. thalania and P. falciparum.

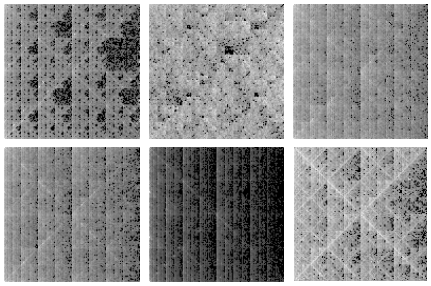

Fig. 2. Initial genome sequences CGR images

We see strong similarity between the images above and corresponding ones below, which come from the final sequences in their respective genomes.

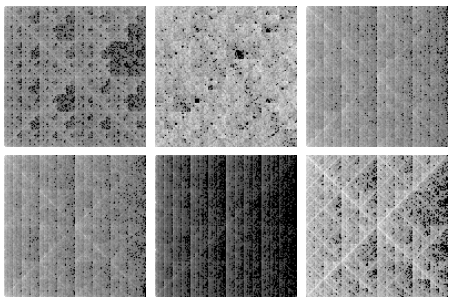

Fig. 3. Final genome sequences CGR images

Our settings for the image preprocessing are different from previous examples. We now retain a 20-by-20 subgrid of Fourier frequencies (larger than in prior examples) but use only 8 singular values. The look-up is now extremely efficient. We found the ten nearest neighbors to each genomic sequence FCGR. In the case of five species, all ten neighbors of every sequence FCGR came from FCGR from the same chromosome. In the one exceptional case, A. thalania (Plantae kingdom), the first eight neighbors all arose from FGCRs in the same chromosome, with the ninth and/or tenth closest neighboring FGCRs coming instead from P. falciparum (Protista kingdom). As there were only 10 sequences from the A. thalania chromosome, there could be at most nine neighbors for each arising on that same chromosome. Moreover in only two of the ten cases was the ninth neighbor from P. falciparum, so it is clear that this recognition method was quite successful even in distinguishing the one pair of chromosomes for which a visual discrimination seems difficult. Similar results are seen when we look at the 15 nearest neighbors for each FCGR.

We can also use the vectors produced in our preprocessing for purposes of visualizing sequences from each chromosome. We use six colors to distinguish segments from the six chromosomes. The plot below corresponds to vectors truncated to their first three components.

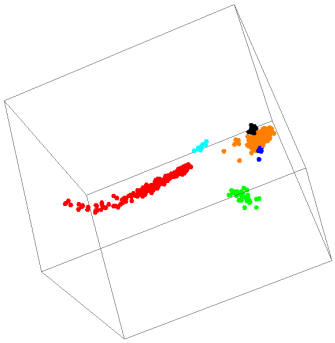

Fig. 4. Clustering by largest three singular values

We seem to get better separation of the clusters when we instead use the second through fourth components of those vectors.

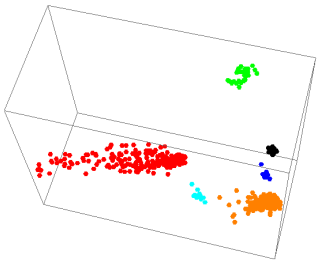

Fig. 5. Clustering by second, third, and fourth largest singular values

We can obtain a different visualization with a simple application of Multidimensional Scaling (MDS) [11]. We use the simplest form of this, based on PCA applied to a distance matrix formed from the processed image vectors. This gives rise to the plot below.

Again we observe that segments from the same chromosome tend to be near to one another, and, for the most part, remain isolated from those on different chromosomes. The picture above is roughly comparable in quality to the better ones produced in

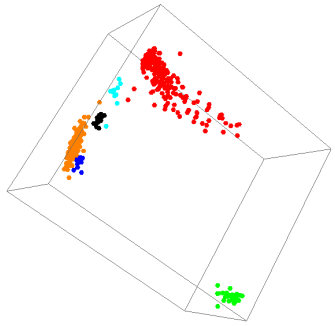

Fig. 6. Clustering by multidimensional scaling

page 11 of [10] using more expensive computations with various distance metrics.

V. Summary and further directions

We have presented a simple, yet effective method for processing image libraries so that, for any input image, similar ones might be located. The method is fully automated and, in particular, there is no need for tuning parameters, nor for finding landmarks or other off-line processing of test images. We explained the main principles behind the inner workings and illustrated on several families of examples, including face, digit, and gene sequence recognition. Full code was provided for implementation and all examples.

It would of course be useful to lower the failure rate for the recognition tasks. For the case of faces, for example, the code we show is not sufficiently adept at handling shadows. In addition to problem specific processing (e.g. removal of shadows), there might be better general preprocessing methods. One possible area to explore would be to use wavelets instead of Fourier-type transforms. Some modest experimentation has given results comparable to what we showed in terms of quality, albeit two to three times slower in processing. But this is by no means a definitive finding, and it is quite plausible that certain wavelets, alone or in conjunction with Fourier methods, might give notable improvement for image recognition at tolerable cost in speed.

Another weakness is that we seem to require around 30 singular values in order to attain the success levels shown in all examples other than the gene signatures. This number then becomes the dimension used in the storage structure. It is at the outer edge of what can be handled effectively using k-D trees. Thus it would be useful to find better preprocessing to allow a reduction of this parameter, say to 20 or smaller. Here is one brief example. For the MNIST test, if we keep 18 singular values then the look-up time is less than half what we showed earlier, but the recognition rate drops from almost 98% to around 97%.

Acknowledgements

I thank Rallis Karamichalis for sharing code for constructing images from genomes, and for explaining some of the finer points lurking behind such representations. I thank the anonymous referees for their many useful comments and questions. I also thank a referee of an earlier version of this paper for pointing out some typographical errors. The idea for application to genome lookup came about following an excellent talk by Lila Kari at ACMES 2016 (London Ontario).

VI. Code appendix

The basic code in entirety is below. The only other processing is to get all images in a given set into the same dimensions, using the Wolfram Language function **ImageConform**. The function **nearestImages** takes an image set and produces a look-up function (internally based on a k-D tree) and also a conversion matrix arising from a singular values decomposition. This latter is from the PCA part of the computation, and is what reduces dimension to the point where a k-D tree becomes an effective storage and look-up mechanism. Preprocessing to get low frequency Fourier components is useful for removing detail and also for compressing to an extent so that the SVD computation is fast. Look-up is preprocessed by the function **processInput**. This does the same Fourier and PCA compression, so that images are in the same format suitable for individual look-up. Certain parameters are set at the beginning. The settings shown below are based on some amount of testing, and seem to work well. Specifically we use the fourth type of discrete sine transform, and we retain the upper left 10-by-10 matrix of components. These get flattened into vectors of length 100 and are further reduced to length 31 by a singular values decomposition.

```
keep = 31;
dn = 10;
dst = 4;
Clear[nearestImages, processInput];

nearestImages[ilist_, vals_, dn_, dnum_, keep_]:=
 Module[
   {idata, images = ilist, dcts, top,
    topvecs, uu, ww, vv, udotv, norms},
   idata = Map[ImageData, images];
  dcts = Map[
    FourierDST[# - Mean[Flatten[#]], dnum]&, idata];
   top = dcts[[All, 1;;dn, 1;;dn]];
   topvecs = Map[Flatten, top];
   topvecs = Map[# - Mean[#]&, topvecs];
   {uu, ww, vv} =
    SingularValueDecomposition[topvecs, keep];
   udotv = uu.ww;
   norms = Map[Sqrt[#.#]&, udotv];
   udotv = udotv/norms;
   udotv = Join[udotv, Transpose[{Log[norms]}], 2];
   {Nearest[udotv → vals, Method → "KDTree"], vv}]

processInput[ilist_, vv_, dn_, dnum_]:=
 Module[
   {idata, images = ilist, dcts, top,
    topvecs, tdotv, norms},
   idata = Map[ImageData, images];
  dcts = Map[
    FourierDST[# - Mean[Flatten[#]], dnum]&, idata];
```

```
  top = dcts[[All, 1;;dn, 1;;dn]];
  topvecs = Map[Flatten, top];
  topvecs = Map[# - Mean[#]&, topvecs];
  tdotv = topvecs.vv;
  norms = Map[Sqrt[#.#]&, tdotv];
  tdotv = tdotv/norms;
  tdotv = Join[tdotv, Transpose[{Log[norms]}], 2];
 tdotv]
```

The MNIST suite [9] of 70000 digit images can be imported as below. The step of creating explicit images is not really needed since it is easy to modify the processing code above to handle the raw byte arrays.

```
trainingBytes = Import[
 "http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz"
 ,"Byte"];
trainingImages = Map[Image[Partition[#, 28]]&,
  Partition[Drop[trainingBytes, 16], 28^2]];
testBytes = Import[
 "http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz"
 ,"Byte"];
testImages = Map[Image[Partition[#, 28]]&,
  Partition[Drop[testBytes, 16], 28^2]];
trainingLabels = Drop[Import[
 "http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz"
 ,"Byte"], 8];
testLabels = Drop[Import[
 "http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz"
 ,"Byte"], 8];
```

These images are fairly crude (28 by 28 pixels) so we sharpen them as a preprocessing step.

```
func = Sharpen[#, 12]&;
trainingImages = Map[func, trainingImages];
  testImages = Map[func, testImages];
```

We process in the usual way.

```
Timing[{nf, vv} =
  nearestImages[trainingImages,
   trainingLabels, dn, dst, keep];
testvecs =
 processInput[testImages, vv, dn, dst];]
```

Code for guessing from the test set is based on majority rule when comparing to the training set.

```
guesses[nf_, tvecs_, n_]:=Module[
  {nbrs, counts},
  nbrs = Map[nf[#, n]&, tvecs];
  counts = Map[Tally, nbrs];
  counts =
   Map[Reverse, Map[SortBy[#[[2]]&], counts]];
 counts[[All, 1, 1]]]

correct[guess_, actual_]/;
   Length[guess] == Length[actual]:=
 Count[guess - actual, 0]
correct[__]:=$Failed

Timing[
 guessed = guesses[nf, testvecs, 3];]
correct[guessed, testLabels]
```

{13.5626, Null}

9781

Here is the preprocessing for a data set of comprised of six genomes, taken from the supplementary material of [10].

```
srules = {Y → T, Except[Characters["ACGTY"]] → ""};
ncraw[21] = Import[
 "https://github.com/rallis/intraSupplemental$\
_$Material/blob/master/code/alltogether/six$\
_$kingdoms/fasta/NC$\_$000021.fasta?raw=true"
 ];
nc[21] = StringReplace[ncraw[21][[1]], srules];
ncraw[913] = Import[
 "https://github.com/rallis/intraSupplemental$\
_$Material/blob/master/code/alltogether/six$\
_$kingdoms/fasta/NC$\_$000913.fasta?raw=true"
 ];
nc[913] = StringReplace[ncraw[913][[1]], srules];

ncraw[1136] = Import[
 "https://github.com/rallis/intraSupplemental$\
_$Material/blob/master/code/alltogether/six$\
_$kingdoms/fasta/NC$\_$001136.fasta?raw=true"
 ];
nc[1136] = StringReplace[ncraw[1136][[1]], srules];

ncraw[3070] = Import[
 "https://github.com/rallis/intraSupplemental$\
_$Material/blob/master/code/alltogether/six$\
_$kingdoms/fasta/NC$\_$003070.fasta?raw=true",
 ];
nc[3070] = StringReplace[ncraw[3070][[1]], srules];

ncraw[4317] = Import[
 "https://github.com/rallis/intraSupplemental$\
_$Material/blob/master/code/alltogether/six$\
_$kingdoms/fasta/NC$\_$004317.fasta?raw=true"
 ];
nc[4317] = StringReplace[ncraw[4317][[1]], srules];

ncraw[18092] = Import[
 "https://github.com/rallis/intraSupplemental$\
_$Material/blob/master/code/alltogether/six$\
_$kingdoms/fasta/NC$\_$018092.fasta?raw=true"
 ];
nc[18092] = StringReplace[ncraw[18092][[1]], srules];

ncvals = {21, 913, 1136, 3070, 4317, 18092};
seglen = 150000;
chars = {A, T, G, C};
```

```
dim = 7;
```

FCGR code for producing images of dimension $2^7 x 2^7$ from each sequence of length 150,000.

```
makePositionsC = Compile[{{shft, _Integer, 2}, {k, _Integer}},
  Module[{posns},
   posns = FoldList[Mod[2 * #1 + #2, 2^k]&, Reverse@shft];
   Most[Reverse[
     Map[{2^k, 1} + {-1, 1} * Reverse[#]&, posns]]]],
  RuntimeOptions → "Speed", CompilationTarget → C];

replace = Dispatch[Thread[
  chars->{{0,0},{0,1},{1,1},{1,0}}]];

FCGR[chars_, k_]:=Module[
  {shifts, posns, newposns},
  shifts = chars/.replace;
  newposns = Round[makePositionsC[shifts, k]];
 Normal[SparseArray[
   Apply[Rule, Tally[newposns], {1}], {2^k, 2^k}]]]

AbsoluteTiming[Do[
  images[j] = Map[FCGR[#, dim]&,
    Partition[Characters[nc[j]], seglen]];
  images2[j] = Map[(#/N[Max[#]])^(1/6)&, images[j]];
 , {j, ncvals}]]
{91.2833, Null}

allimages = Apply[Join, Map[images2, ncvals]];
speciesvals = Flatten[Table[
  ConstantArray[j, Length[images2[j]]], {j, ncvals}]];
actualImages = Map[Image, allimages];
```

We use this to create relatively short vectors testvecsY from our FCGR images, and also to create a lookup function nfY suitable for finding neighbors.

```
AbsoluteTiming[{nfY, vvY} =
  nearestImages[actualImages, speciesvals, 20, 4, 8];
testvecsY = processInput[actualImages, vvY, 20, 4];]
{0.469278, Null}
```

Now find the 10 nearest neighbors to each genome sequence.

```
nearSigs = Map[{First[#], Rest[#]}&,
 Map[nfY[#, 11]&, testvecsY]]
```

Form 3D graphics with chromosome sequences coded by color.

```
svcolors = speciesvals/.
  Thread[ncvals → {Red, Green, Blue, Orange, Black, Cyan}];
pts123 =
  Transpose[{svcolors, Map[Point, testvecsY[[All, 1;;3]], {1}]}];

pts234 =
  Transpose[{svcolors, Map[Point, testvecsY[[All, 2;;4]], {1}]}];
```

Use MDS to find and plot 3D coordinates from the 8 dimensional vectors.

```
diffs = Table[
  vecj - veck, {vecj, testvecsY}, {veck, testvecsY}];
dist2mat = -Map[#.#&, diffs, {2}]/2;
len = Length[testvecsY];
onevec = ConstantArray[{1}, len];
hmat = IdentityMatrix[len] - onevec.Transpose[onevec]/len;
bmat = hmat.dist2mat.hmat;
{uu, ww, vv} = SingularValueDecomposition[bmat, 3];
newvals2 = uu.Sqrt[ww];
newptsb = Transpose[{svcolors, Map[Point, newvals2, {1}]}];
```

REFERENCES

[1] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[2] S. Hui and S. H. Zak, "Discrete fourier transform based pattern classifiers," *Bulletin of the Polish Academy of Sciences: Technical Sciences*, vol. 62, no. 1, pp. 15–22, 2014.

[3] J. H. Lai, P. C. Yuen, and G. C. Feng, "Face recognition using holistic fourier invariant features," *Pattern Recognition*, vol. 34, pp. 95–109, 2001.

[4] H. Spies and I. Ricketts, "Face recognition in fourier space," in *Vision Interface 2000*, 2000, pp. 38–44.

[5] G. Dashore and V. C. Raj, "An efficient method for face recognition using principal component analysis (pca)," *International Journal of Advanced Technology and Engineering Research (IJATER)*, vol. 2, no. 2, pp. 23–29, 2012.

[6] J. Yang, D. Zhang, A. F. Frangi, and J.-Y. Yang, "Two-dimensional pca: a new approach to appearance-based face representation and recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 1, pp. 131–137, 2004.

[7] J. Ashok and E. G. Rajan, "Principal component analysis based image recognition," *International Journal of Computer Science and Information Technologies*, vol. 1, no. 2, pp. 44–50, 2010.

[8] D. Zhang, D. Ding, J. Li, and Q. Liu, "A pca-based face recognition method by applying fast fourier transform in preprocessing," in *3rd International Conference on Multimedia Technology (ICMT 2013)*, 2013, pp. 1155–1162.

[9] Y. LeCun, C. Cortes, and C. J. C. Burges. (1998) The mnist database. http://yann.lecun.com/exdb/mnist/.

[10] H. J. Jeffrey, "Chaos game representation of gene structure," *Nucleic Acids Research*, vol. 18, no. 8, pp. 2163–2170, 1990.

[11] A. Buja, D. F. Swayne, M. L. Littman, N. Dean, H. Hofmann, and L. Chen, "Data visualization with multidimensional scaling," *Journal of Computational and Graphical Statistics*, vol. 17, no. 2, pp. 444–472, 2008.

[12] P.-R. Loh, M. Baym, and B. Berger, "Compressive genomics," *Nature Biotechnology*, vol. 30, no. 7, pp. 627–630, 2012.

[13] W. Research. (2015) Mathematica 10.2. http://www.wolfram.com. Champaign, IL, USA.

[14] R. Karamichalis, L. Kari, S. Konstantinidis, and S. Kopecki, "An investigation into inter- and intragenomic variations of graphic genomic signatures," *BMC Bioinformatics*, vol. 16, no. 246, pp. 1–22, 2015.