FacePerf: Benchmarks for Face Recognition Algorithms

David S. Bolme, Michelle Strout, and J. Ross Beveridge
Colorado State University
Fort Collins, Colorado 80523-1873
Email: {bolme,ross,mstrout}@cs.colostate.edu

***Abstract*—In this paper we present a collection of C and C++ biometric performance benchmark algorithms called FacePerf. The benchmark includes three different face recognition algorithms that are historically important to the face recognition community: Haar-based face detection, Principal Components Analysis, and Elastic Bunch Graph Matching. The algorithms are fast enough to be useful in realtime systems; however, improving performance would allow the algorithms to process more images or search larger face databases. Bottlenecks for each phase in the algorithms have been identified. A cosine approximation was able to reduce the execution time of the Elastic Bunch Graph Matching implementation by 32%.**

I. Introduction

Computer vision is a growing field in computer science that requires the use of many different computationally intensive algorithms. Recently, processor performance and digital camera improvements have enabled realtime face recognition systems. Performance bottlenecks do occur, so often these systems achieve realtime performance by sacrificing data throughput (dropping video frames), reducing the problem size, or reducing the accuracy of the image analysis. More efficient algorithm implementations could mitigate these issues and allow engineers to produce better realtime vision systems.

FacePerf is a collection of three face recognition algorithms that attempts to cover the major components of automatic face recognition systems. These specific algorithms were selected because they are important algorithms to the biometrics community, each one performs very different types of computations, and they have open source implementations. The test scripts included with FacePerf provide a standardized testing methodology by running the algorithms on well known face recognition datasets.

The first step of face recognition is face detection, which determines where in the image a face is located. Face detection algorithms typically work by scanning an image at different scales and looking for simple patterns that indicate the presence of a face. Once the face is located, a sub-image (or image chip) is produced at the appropriate location and scale such that the face appears centered and is presented at a uniform size. The second step is to compute a similarity score between the detected face and one or more face image chips stored in a database. If the goal of recognition is identification, then the identity of the person in the new image is assumed to be the same as the identity stored with the image in the database most similar to the new image.

TABLE I
Wall clock times for the three algorithms in the performance benchmark.

Phase	**Time.**
Face_Detection	5.9s
PCA_Train	220.4s
PCA_Test	47.0s
EBGM_Locate	177.2s
EBGM_Extract	148.2s
EBGM_Similarity	72.4s

FacePerf includes one face detection algorithm and two face identification algorithms. The OpenCV Haar-based Cascade Classifier (Haar Classifier)[1][2][1] is an implementation of a face detection algorithm that locates faces in images or video. This algorithm outputs a bounding box for every face detected in the imagery. The two identification algorithms are Principal Components Analysis (PCA) [3] and Elastic Bunch Graph Matching (EBGM) [4], which have both been implemented by the CSU Face Recognition Evaluation Project[2][5]. The algorithms produce similarity scores between the image chips provided by a face detector.

In this paper, we give a brief description of each algorithm included in the performance benchmark set. We describe how to obtain the source code and datasets needed to run the benchmarks. We discuss example runtimes and GNU Profiler (`gprof`) information for each benchmark. Finally, we present a test case that shows how automatic and manual optimizations were used to improve the performance of the EBGM similarity computation.

II. Algorithms

A. OpenCV Cascade Classifier

The Intel OpenCV cascade classifier is based on a face detection algorithm developed by Viola and Jones [1][2]. The algorithm scans an image and returns a set of locations that are believed to be faces. The algorithm uses an Ada-Boost based classifier that aggressively prunes the search space to quickly locate faces in the image. Figure 1 shows an example of the Haar Classifier applied to an image, where the squares indicate

[1] **http://www.intel.com/technology/computing/opencv**
[2] **http://www.cs.colostate.edu/evalfacerec**

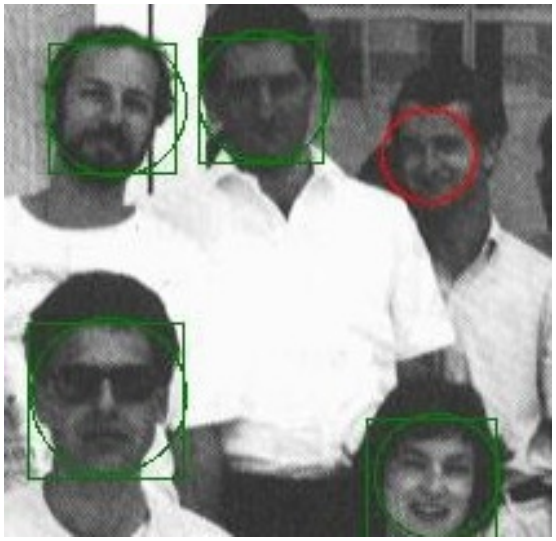

Fig. 1. This image illustrates the ouput of the Haar Classifier.

face detections and circles indicate ground truth. Circles with no corresponding square indicate false negatives (or undetected faces).

The accuracy of this algorithm can be evaluated using a standard configuration that ships with the OpenCV source code. The accuracy is tested by detecting a set of faces in the CMU Frontal Face Dataset[3], which is designed for testing face detectors. A script compares the detections to manually selected ground truth, and the results are reported as the ratio of true detections to the total number of faces.

Because the Haar Classifier is part of the OpenCV library, the source code should be well optimized. OpenCV is a popular open source image library that was originally created by Intel. The data structures and algorithms in OpenCV have been carefully tuned to run efficiently on modern hardware. It is evident from the code of the Haar Classifier that hand optimization was used to improve the performance of that algorithm.

B. CSU Principal Components Analysis

PCA [6][3] begins by representing face images as vectors where each element in the vector corresponds to a pixel value in the image (see Figure 2). Typically feature vectors contain at least 1024 pixels and often more. Computing a similarity score, such as correlation, between two vectors of this size is slow. The PCA process is therefore used to determine basis vectors for a subspace in which almost all common facial variation is expressed in a much smaller dimensionality. Both new images and images stored in a database of faces are represented in this more compact form. This step considerably reduces the amount of computation required to compare two images. Some researchers also conjecture that projecting imagery into the PCA subspace removes noise and leads to better identification; however, the evidence for this conjecture is not always compelling.

[3] **http://vasc.ri.cmu.edu/idb/html/face/frontal_images**

By its very nature, PCA does eliminate statistical covariance, at least with respect to a set of canonical training imagery. This has useful properties when measuring the similarity of the transformed feature vectors. It allows techniques such as whitening and L1 distance to be used to compute similarity. Whitening the data typically improves identification accuracy.

Most of the work performed in the PCA algorithm is computing matrix multiplications and solving eigenvector problems. This specific implementation of PCA was developed as a baseline for accuracy evaluations of face recognition algorithms and was never required to run in realtime. Simple hand-tuned optimizations have been applied to the matrix multiplication routines to reduce function calls and improve data locality. The eigen-solver is from the OpenCV library and should be fairly efficient. The performance of this algorithm would probably be improved by using linear algebra subroutines that are optimized for the underlying hardware.

C. CSU Elastic Bunch Graph Matching

The EBGM algorithm[4][7] identifies a person in a new image face by comparing it to other faces stored in a database. The algorithm extracts feature vectors (called Gabor jets) from interest points on the face and then matches those features to corresponding features from the other faces. The algorithm first has to extract a simplified representation of the face and then compare that representation to other images in the database.

The EBGM algorithm operates in three phases. First, important landmarks on the face are located by comparing Gabor jets extracted from the new image to Gabor jets taken from training imagery. Second, each face image is processed into a smaller description of that face called a FaceGraph. The last phase computes the similarity between many FaceGraphs by computing the similarity of the Gabor jet features. Figure 3 shows the landmark locations for three faces. The FaceGraphs with the highest similarity will hopefully belong to the same person. While all of these phases effect the performance of the algorithm, this example focuses on the performance of the similarity comparison computations.

Like PCA, the EBGM algorithm was also designed for accuracy and had no need to run in realtime. The execution of this algorithm is currently performed by running three separate executables on the data. The algorithm has some hand optimized code; however there is probably still room for improvement. We tried a number of simple hand optimizations intended to reduce the running time of the algorithm. The only modification that significantly improved the running time of the benchmark was the cosine approximation described in Section IV-C.

III. Source Code and Datasets

A. Source Code

The source code for these performance benchmarks can be obtained from the following website:

http://www.cs.colostate.edu/~vision/faceperf

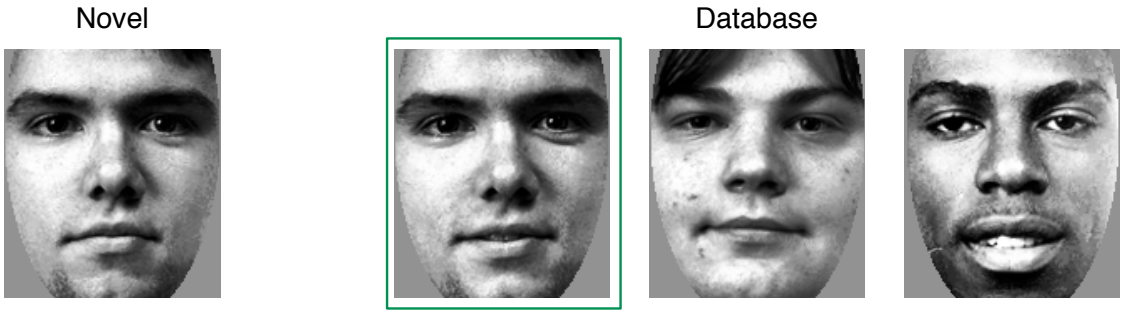

Fig. 2. These are examples of normalized images used for PCA identification. Each 150X130 image is flattened into a feature vector of length 19500. The PCA basis vectors are used to project this vector from $\Re^{19500}$ to $\Re^{300}$. Finally, the faces are compared to a database of known subjects, and the most similar face is selected.

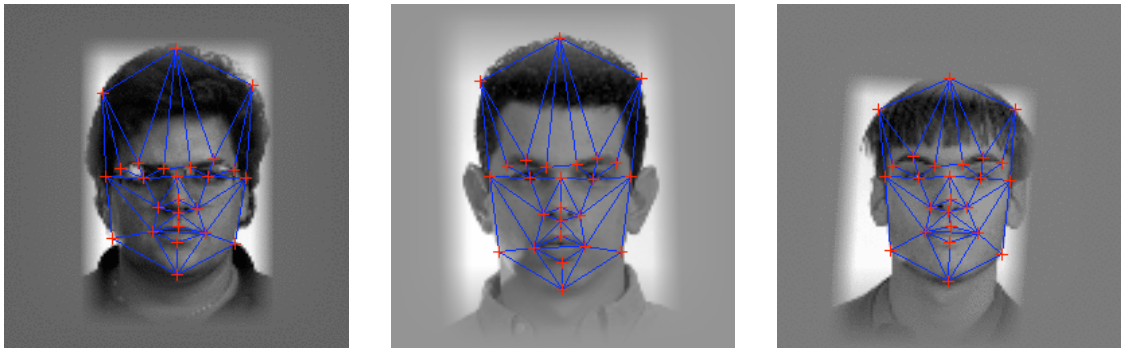

Fig. 3. Landmarks used for the EBGM algorithm.

The distribution contains source code for the CSU Face Identification Evaluation System version 5.1 and OpenCV version 1.0.0. This collection of source code is intended for performance benchmarking only and has been modified for this purpose. For biometric and computer vision research, we would suggest you obtain original copies of the source codes from their official websites. Links to these original sites are available through our site above.

The performance benchmark was originally developed for systems running gcc 4 and the GNU make utilities. It is known to compile and run using gcc on Mac OS X, linux, Solaris, and cygwin. The CSU code is written in C, and the OpenCV code is written in C++. The following supporting utilities are required to convert imagery and perform analyses: python[4], ImageMagick[5], and bash[6].

B. Datasets

The distribution contains a small face dataset (csuScrapShots) compiled by us from CSU yearbook photos dating prior to 1927. This dataset can be distributed freely because the copyright for the imagery has expired. It is included as a way to test the performance benchmarks and verify that the system was downloaded and compiled correctly. This dataset is of much lower quality than is typical of face recognition datasets, and it is unclear how this will effect the performance of the Face Detection and EBGM algorithms. It is provided solely to test that the code executes without errors; it should never be used to assess face detection or recognition performance.

Two better datasets can be obtained for free from other sources. The first is a standard dataset for evaluating face detection algorithms made available through Carnegie Mellon University (CMU):

http://vasc.ri.cmu.edu/idb/html/face/frontal_images

The second dataset is the FERET database, which is a standard dataset used to evaluate face recognition algorithms. This dataset can be requested through the National Institute of Standards and Technology: Information Technology Laboratory.

http://www.itl.nist.gov/iad/humanid/feret/

IV. Results

By analyzing profiles of the three algorithms, we determined the performance bottlenecks for each algorithm implementation. A summary of the results are shown in Table II. We profiled the three algorithms with gprof. All of the timings reported in this paper are based off of the CMU and FERET

[4]**http://www.python.org**
[5]**http://www.imagemagick.org/script/index.php**
[6]**http://www.gnu.org/software/bash**

TABLE II
THIS TABLE SHOWS A SUMMARY OF THE BOTTLENECKS IN THE PROGRAMS, THE TIME SPENT IN THOSE FUNCTIONS, AND THE NUMBER OF CALLS TO THAT FUNCTION.

Algorithm	Bottleneck	Function	BN Time
Face Detection	Cascade Classifier	`cvRunHaarClassifierCascade`	88.0%
PCA Training	Matrix Multiply	`multiplyMatrix & transposeMultiplyMatrixL`	41% & 52%
PCA Test	Matrix Multiply	`transposeMultiplyMatrixL`	90%
EBGM Locate	Convolution	`convolvePoint`	93%
EBGM Extract	Convolution	`convolvePoint`	96%
EBGM Similarity	Similarity	`DEPredictiveIter & cos`	63% & 30%

datasets. Unless otherwise indicated, the code was compiled with the GNU Compiler Collection (GCC) -O3 settings and run on an Apple MacBookPro 2Gz Intel Core Duo with 2GB 667Mhz DDR2 SDRAM of memory.

A. Haar Classifier Profile

The performance bottleneck for the Haar Classifier implementation is the `cvRunHaarClassifierCascade` function, found in `opencv-1.0.0/cv/src/cvhaar.cpp`. In this function, a series of tests are perform for detecting a face within a particular window. An explanation of this bottleneck is found in Figure 4.

B. PCA Profile

Matrix multiply and a variant on matrix multiply are the bottlenecks when training the PCA algorithm and computing similarities. The `transposeMultiplyMatrixL` function takes two matrices A and B as input and computes $A^T B$ as output. Neither `transposeMultiplyMatrixL` nor `multiplyMatrix` uses any blocking or unrolling. The implementation for both functions can be found in `src/csuCommonMatrix.c`.

C. EBGM Performance

In the EBGM algorithm, a similarity computation function called `DEPredictiveIter` was identified as the performance bottleneck when classifying faces. Performance was tuned using various optimization settings for the GCC compiler, using scalar replacement for a set of reused memory references, and substituting an approximation for an expensive cosine operation. It was found that the GCC -O3 optimization setting and cosine approximation had the largest effects on performance.

Execution timings are reported as the lowest user time measured from three runs of the program. The manually optimized code was compiled with the GCC -O3 automatic optimizations. A summary of the results can be found in Table III.

The simplest method for improving the performance of an application is to turn on automatic compiler optimizations. This work tests three different optimization settings for GCC : `-O0`, `-O3`, and Profile Guided(PG)[7].

The profile guided optimizations uses runtime profiling information to better determine which paths are more frequently traveled and perform better optimizations along those frequently traveled paths. To produce the profile information, the program was compiled and linked with the -fprofile-generate flag. The program was then run on a subset of the problem to produce profile information. Finally, the program was recompiled using the profiling information and the runtimes were recorded. This method only produced a modest improvement in the total runtime.

The `gprof` profiling tool showed that much of the computation time of the `DEPredictiveIter` function was spent computing a cosine. Improved performance is observed using either of two cosine approximations (see Figure 5).

The first approximation (`COS_A`) computes the 4th order Taylor expansion of the cosine function for both the top lobe and the bottom lobe. The approximation is similar to the full cosine function except for a slight discontinuity where $\cos(x) = 0$. The face identification accuracy of the algorithm was indistinguishable from the same algorithm using the `libm` implementation of cosine.

The second approximation (`COS_B`) exploits the fact that the algorithm is specifically concerned with the positive values of the cosine function when x is close to zero. This approximation is a second order Taylor expansion for just the top lobe. The minimum value for the cosine function is clamped at zero. The algorithm accuracy for this approximation is slightly better than the original algorithm.

V. SUMMARY

FacePerf is a collection of three open source face recognition algorithms. These algorithms span two major components of face recognition systems: face detection and face identification. Scripts have been included with FacePerf to run the benchmark on small freely downloaded datasets. Scripts that run the benchmarks on the standardized FERET dataset are also included. The source code combined with the scripts provides

[7] `-O3 -fprofile-use -falign-loops-max-skip=15 -falign-jumps-max-skip=15 -falign-loops=16 -falign-jumps=16 -falign-functions=16 -malign-natural -ffast-math -funroll-loops -ftree-loop-linear -fsched-interblock -fgcse-sm`

TABLE III
THIS SHOWS THE MINIMUM RUNTIME FOR THE EBGM ALGORITHM USING VARIOUS AUTOMATIC AND AND MANUAL OPTIMIZATION TECHNIQUES. THE NORMALIZED TIME IS BASED ON THE GCC -O3 SETTING. THE MANUALLY OPTIMIZED TESTS WERE ALSO RUN USING GCC -O3 OPTIMIZATION.

Optimization	**Execution Time (sec)**	**Normalized Time**
GCC -O0	159.975	2.205
GCC -O3	72.539	**1.000**
GCC Profile Guided	78.969	1.088
Manual Cosine Replacement A (`COS_A`)	50.175	0.692
Manual Cosine Replacement B (`COS_B`)	49.198	0.678
Manual Scaler Replacement	73.184	1.008

a standardized methodology for performance evaluation of the face recognition algorithms.

This paper describes the three algorithms that make up the FacePerf benchmarks. Bottlenecks have been identified using `gprof` and we have shown one method for improving the performance of the EBGM Similarity computation.

These algorithms perform different computations and provide a variety of different performance problems. The algorithms are taken from two well established sources. The first is OpenCV which is a popular computer vision library. The second is the Evaluation of Face Recognition Project which is a baseline for comparing the accuracy of identification algorithms.

REFERENCES

[1] P. Viola and M. J. Jones, "Robust real-time face detection," *Int. J. Comput. Vision*, vol. 57, no. 2, pp. 137–154, 2004.

[2] R. Lienhart and J. Maydt, "An extended set of haar-like features for rapid object detection," in *Proceedings of the 2002 International Conference on Image Processing* (P. of the 2002 International Conference on Image Processing, ed.), vol. 1, pp. 900–903, 2002.

[3] M. A. Turk and A. P. Pentland, "Face recognition using eigenfaces," in *Proc. of IEEE Conference on Computer Vision and Pattern Recognition*, pp. 586 – 591, June 1991.

[4] L. Wiskott, J.-M. Fellous, N. Kruger, and C. von der Malsburg, "Face recognition by elastic bunch graph matching," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, pp. 775–779, July 1997.

[5] D. S. Bolme, J. R. Beveridge, M. L. Teixeira, and B. A. Draper, "The CSU face identification evaluation system: Its purpose, features and structure," in *Proc. 3rd International Conf. on Computer Vision Systems*, (Graz, Austria), Apr. 2003.

[6] M. Kirby and L. Sirovich, "Application of the karhunen-loeve procedure for the characterization of human faces," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 12, pp. 103 – 107, January 1990.

[7] D. S. Bolme, "Elastic bunch graph matching," Master's thesis, Colorado State University, May 2003.

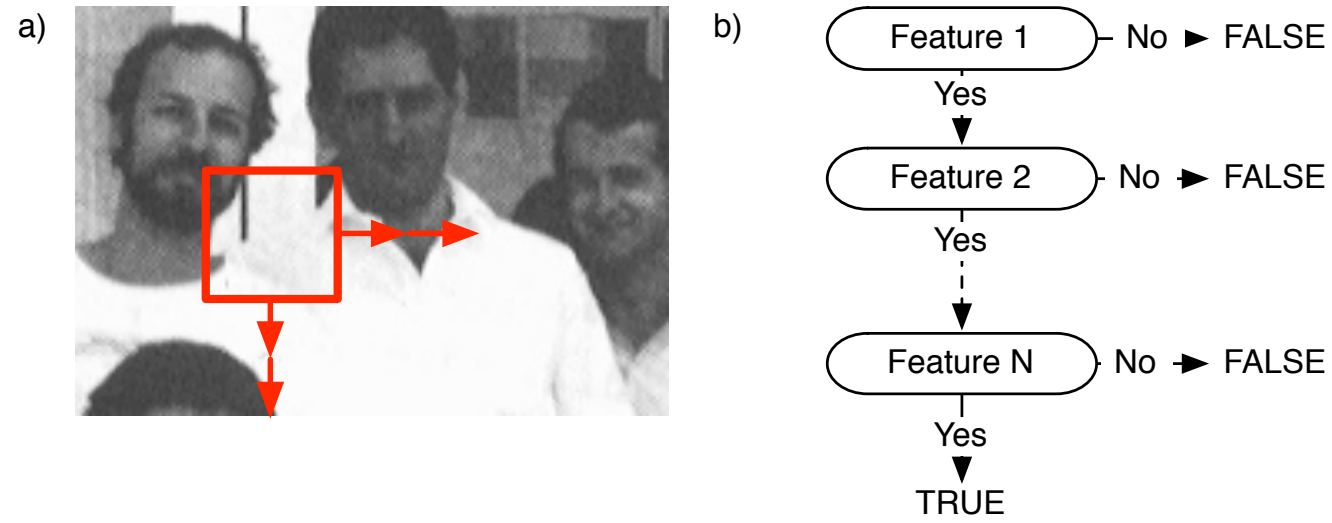

Fig. 4. a) The Haar Classifier scans an image by positioning small windows at multiple locations and scales. b) At each location, a decision tree is evaluated based on simple Haar-based features computed in the sub-window. The most discriminating features are considered first; so if a sub-window is not a face, it is rejected after considering only a small fraction of the features. The large number of sub-windows needed to cover all pixels in many scales is the primary reason for the bottleneck.

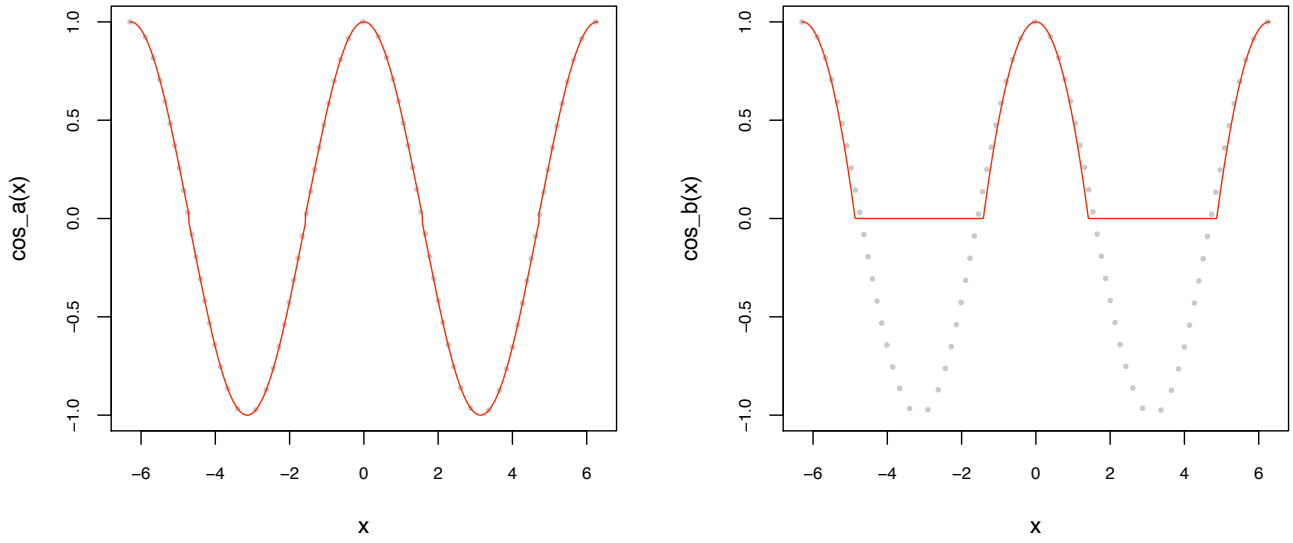

Fig. 5. These are plots of the two cosine approximations that were tested with the EBGM algorithm. The dotted gray line is the true cosine function. The red line is the cosine approximation used in the experiments.