DOI: 10.2352/ISSN.2470-1173.2016.18.DPMI-016

Use of Flawed and Ideal Image Pairs to Drive Filter Creation by Genetic Programming

Subash Marri Sridhar, Henry G. Dietz, Paul S. Eberhart; University of Kentucky; Lexington, Kentucky

Abstract

Traditional image enhancement techniques improve images by applying a series of filters, each of which repairs a specific type of flaw, but most modern digital cameras produce images with a variety of subtle interacting defects. Sequential repair is slow, and the interactions limit the effectiveness.

This paper describes a fundamentally different approach in which a single filter is created to repair the potentially myriad interacting defects associated with a particular camera configuration and set of exposure parameters. Genetic programming (GP) is used to automatically evolve a filter algorithm that will convert flawed images into images minimally differing at the pixel level from the corresponding provided ideal images. For example, the flawed images might be shot at a high ISO and the ideal ones might be the exact same static scenes, aligned at the pixel level, but shot at a low ISO using appropriately longer exposure times. Just as easily, the flawed images might be technically well-corrected, while the ideal ones were manually-edited to adjust and smooth skin tones, sharpen hair, enhance shadow regions, etc. The custom-coded parallel GP, its performance, and performance of the generated filters is discussed with an example.

Introduction

Image enhancement is the general process of taking an image and accentuating certain image characteristics, and suppressing others so that the resulting image is more suitable for subsequent analysis for a specific application[2]. Image restoration is a form of image enhancement in which the goal is repair of well-understood degradations by applying a type of "inverse" transformation. Common to virtually all forms of image enhancement in the literature is the design flow depicted in Figure 1. Indeed, the names of image enhancement techniques typically reflect this idea that each is correcting a specific, individual, type of flaw, e.g.:

- Contrast Stretching — stretching the contrast property of the image fixes flaws
- Edge Enhancement — enhancing the edges in the image makes the image more desirable
- Noise clipping — clipping of the pixel values reduces noise in the image
- Geometric distortion correction — Fixing geometric distortions in the image to make it desirable
- Color level histogram modelling — adjusting color levels in the image leads to a balanced and desirable image.

The greatest difficulty in using that process to create new image enhancement filters is in quantifying the criterion for enhancement (step 2 above). Hence, a large number of image enhancement techniques are empirical in nature and require human interactive procedures to obtain results that meet the requirements. This means the process is: time consuming, very subjective, requires expert understanding of image processing techniques and image properties, and generally involves humans in the evaluation process.

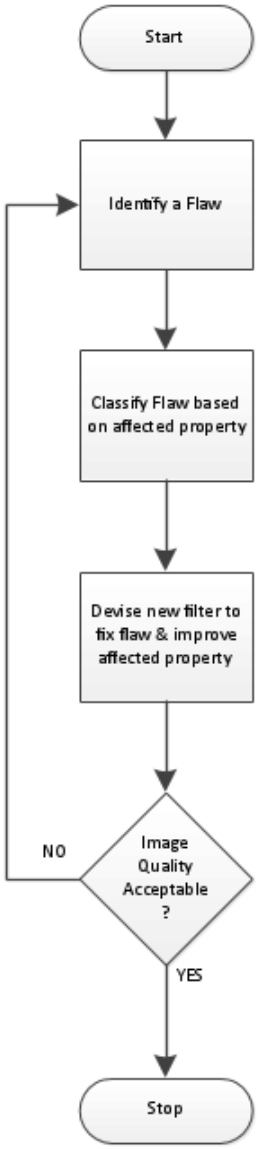


Figure 1: Traditional design flow for image improvement

DOI: 10.2352/ISSN.2470-1173.2016.18.DPMI-016

Filter design using an oracle

This paper explores a completely different approach. The approach here suggests that the image enhancement problem can be seen as simply finding a filter that, when applied to faulty image A, produces a close approximation to a "better" reference image B. In effect, this is using image B as an oracle: an example of how the perfectly-corrected image should appear.

The hope is that the filter that was found to remove whatever defects from A to replicate B should then be able to remove similar defects from any other given image. In essence, the A+B pairs constitute a training data set. However, it is reasonable to expect that a very small number of A+B image pairs should be needed for training, because each reasonably high-resolution image contains a multitude of regions, each of which constitutes an essentially independent training case.

Image defects that can be corrected by this approach can have complex causes, including limitations of sensors or environment, geometric distortions induced by optics, non-linearity induced by sensors, and quantization loss due to under-sampling. Some of the loss might be due to already-applied image enhancement techniques themselves, for example, intensive noise filtering or averaging can induce blurring of the image and cause loss in information. It is even possible to simply allow a skilled human to manually edit and modify image A by any means they wish – including painting and other non-algorithmic transformations – and then to use the result as an oracle for devising a way to automatically perform similar editing of other images.

Traditional filter design starts with examining image A and trying to guess at incremental improvements without having a well-defined ultimate goal. In contrast, the current work is simply given images A and B as initial conditions, and automatically develops a function that converts image A to image B. In this way, arbitrarily complex image enhancement problems can be handled without direct human effort in classifying or evaluating image defects. The evaluation of the resultant transfer function can be very mathematical and objective in nature, because in its simplest form the evaluation of the transfer function can be done by measuring the sum of pixel-level differences between B and the image produced by processing A. In other words, the approach in this paper has the following strengths:

1. The filter derivation is an objective process.
2. The procedure requires minimal human interaction (if any at all).
3. If there are humans involved, they are not required to have any image processing knowledge.
4. The process is fast and the approach will produce robust and stable transformation functions.

(a) Flawed high ISO image

(b) Perfect low ISO "oracle" image

Figure 2: Input Image Pair

Creating oracles

An obvious potential flaw of this approach is the need to create a perfect B image to serve as an oracle. However, it is actually very simple because B need not be a truly perfect capture. Neither is it required that it be obtained in the same way as A. It is sufficient for B to contain the same scene content at the pixel level and be relatively free of the particular soup of artifacts to be removed from A.

As a simple motivating example, consider the A+B image pair shown in Figure 2. In comparison to image B, A shows noise, disruptions of the tonal scale, loss of sharpness, etc. – a multitude of interacting defects coming from the use of a high ISO sensitivity to light for image A.

The quantum efficiency of a pixel generally is not adjustable; high ISOs are implemented using higher-gain amplification of the analog signal from each pixel. That unfortunately applies the amplification not only to the signals, but to noise as well. Even for a perfect camera, high ISO image quality would be compromised by the fact that high ISO images are given fewer photons

DOI: 10.2352/ISSN.2470-1173.2016.18.DPMI-016

to sample, making photon shot noise (statistical variation in the light source photon emission rate) more significant. In addition, most consumer cameras apply a variety of algorithms to try to reduce the appearance of noise (even to raw images), ranging from smoothing operations to desaturation of colors.

It is easy to understand how the A image in Figure 2 was produced. An old compact digital camera with particularly poor high ISO performance was used to make a properly-exposed photo of a test scene at ISO 400. Just how poor the quality of the captured image is can be clearly seen in the crops shown in Figure 3.

The more challenging task is creation of the B image to serve as an oracle. We have already suggested that the trick is to photograph the exact same scene, aligned at the pixel level, using a lower ISO setting – in this case, ISO 50. However, using the same shutter speed and aperture settings at ISO 50 would result in a severely underexposed image that would have far from ideal image quality. The answer is that the scene was captured scene using a longer exposure time, thus enabling the ISO 50 image to be relatively free of artifacts. Note that changing the aperture would not have the intended result, because it would alter depth of field.

Of course, giving the lower ISO capture a longer exposure time is cheating in the sense that the image is not truly captured under identical circumstances, but the point stands that this simple method trivially produces a near perfect reference image for improving the high ISO image. In general, it is perfectly acceptable to use such tricks to produce oracle images: the quality of the oracle is all that matters, not how it was produced.

Filter Performance Evaluation Metric

Having produced an oracle image B, it is still necessary to mechanically judge how close a processed version of A is to replicating image B. This is done by a simple entropy function described by the following equation:

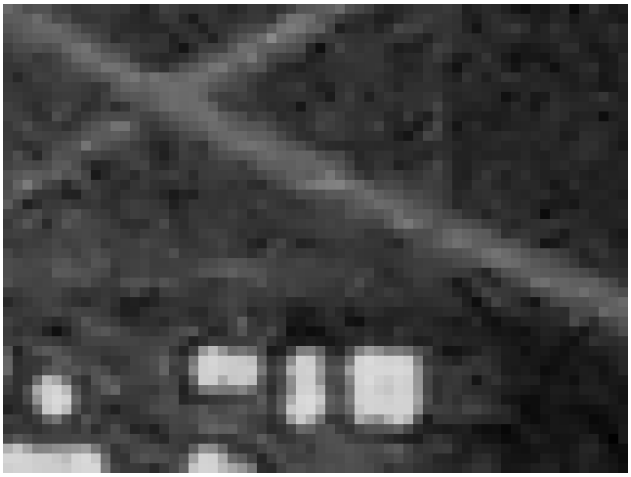
$$e = \sum_{\substack{0 \le x \le X \\ 0 \le y \le Y}} \{Filter[f(x,y)] - g(x,y)\}$$

This metric e is the accumulation of the absolute difference in the pixel values of the filtered image and the good quality lower ISO image. The lower the value of, e higher the rank/fitness value of the population member. Of course, other metric functions, for example comparing squares of differences, are also viable. Empirically, some will work better than others, but many formulations are viable choices.

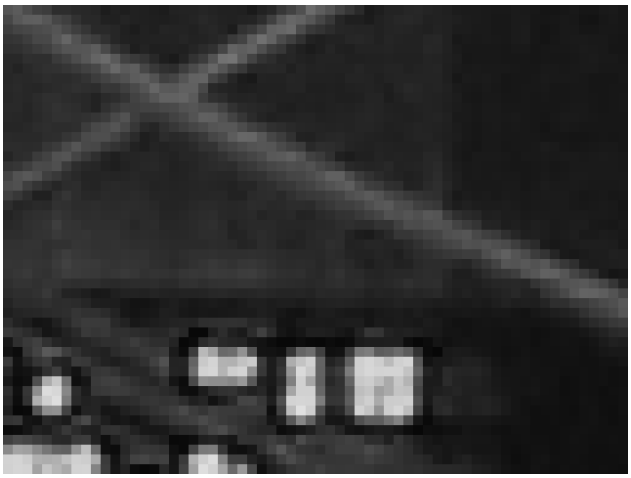
With the decisions made in the previous sections regarding image pair choices and specification of an evaluation metric, the filter development process is not specific to any single flaw type or artifact. The evaluation process allows the performance of the filter to be quantified for arbitrary behaviors. This means the filter generation process is completely objective and easily automated. There is no need for human interaction in the filter creation process nor for deep understanding of image processing techniques in the creator of the A+B image pairs.

Filter Building Blocks

There are various ways to automatically synthesize image filters using A+B image pairs and an evaluation metric. For example, conventional genetic algorithms or neural networks (including the latest "deep learning" methods) could be used. However, those techniques would require imposing a fixed structure on the filter design. The genetic programming approach suggested here does not imposing as many constraints on the form of the solution, which is likely to result in a more effective or less computationally-expensive filter. The process also will generally be able to work with a much smaller training data set.



(a) Crop of flawed high ISO image



(b) Crop of perfect low ISO "oracle" image

Figure 3: Crops of Input Image Pair

For genetic programming, composable primitive and basic building-block functions are specified, and the system uses them to build complex image transformations with an arbitrary structure and complexity. Although they could include well-known image-processing primitives, they also can be simple arithmetic operations on pixels. The primitive functions used in the example in this paper fall into two broad classes of image enhancement operations: point operations and spatial operations.

Point operations are zero memory operations where a given pixel's gray/color channel level $u \in [o,L]$ is mapped into a gray/-color channel level $v \in [o,L]$ according to the transformation $v = f(u)$.

1. The function f depends only on the pixel value.
2. It is independent of the spatial location (u,v)
3. The domain of f must match the range of the image.

DOI: 10.2352/ISSN.2470-1173.2016.18.DPMI-016

Spatial operations are image enhancement operations that work on a neighborhood of pixels. Often, the image is convolved with a finite impulse response filter called a *spatial mask*. A spatial domain process can be expressed by the following equation:

$$g(x,y) = T[f(x,y)]$$

where $f(x,y)$ is the input image, $g(x,y)$ is the output image, and T is an operator on f defined over a neighborhood of point (x,y). The following building blocks were chosen as the raw material for the example in this paper:

1. Weighted Sum Operation — point operation. This building block is defined by

   $$out = in_1(w) + in + 2(1-w)$$

   where $0 < w < 1$
2. Difference operation — point operation. This primitive function is defined by

   $$out = Abs(in_1 - in_2)$$

3. Multiply operation — point operation. This primitive function is defined by the following equation

   $$out = (in_1 \div 255) \times (in_2 \div 255)] \times 255$$

   .
4. Median operation — spatial operation. This function is a basic median filter that operates on the neighborhood of the current pixel. The window of the neighborhood is variable to the median filter function.

   $$out = Median(f(x,y), w))$$

   where $w = windowsize$ and $f(x,y) = current\,pixel$
5. Average operation – spatial operation. This primitive operation basically averages the given set of inputs.

   $$out = \frac{1}{n} \times \sum_{i=1}^{n} x_i$$

6. Interpolate operation - spatial operation. This primitive function is a basic interpolation function. The window and type of interpolation is chosen at random. The window type used for the interpolation can also be called a spatial mask filter and is best represented with figures, the interpolation masks shown in figure 4 were chosen.
7. Pixel — point operation. This is the simplest primitive of all, the current pixel value.

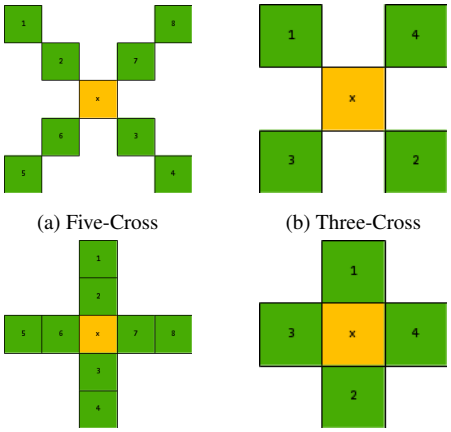
   $$out = in$$

8. Random pixel — point operation. This operation allows the genetic program to assign another pixels value to the current pixel. This primitive also allows the algorithm to randomize the choice of the source pixel within a 5x5 window around the current pixel.

   $$out = Random(in)$$

9. Constant operation — point operation. This operation allows the genetic program to add offsets to pixels or assign constant values to pixel values.

   $$out = c$$



(a) Five-Cross (b) Three-Cross

(c) Five-Plus (d) Three-Plus

Figure 4: Interpolation masks

Genetic Programming

Genetic programming[3] is a method by which computer programs can be created to solve arbitrary problems. The method constrains neither the form nor the complexity of the generated program, but is essentially a type of genetic algorithm.

Genetic algorithms are search algorithms that use the principles of natural selection to converge on the search result. Genetic algorithms are composed of the following three components:

1. A string based structure used to represent the search results for each generation. This also known as the test *candidate or population member*.
2. An *evaluation and selection algorithm*. Scores or fitness metrics are quantified and assigned for each member of the population. The population of search results is then pruned using a selection algorithm to allow the fittest members of the population to survive while the rest do not pass on to the next generation.
3. A *reproduction algorithm* which creates population members for the next generation, the methodology used for reproduction follows one of the following three approaches:
   (a) *Crossover breeding*, two of the surviving members of the population are mated or spliced together to create a new member of the population.
   (b) *Mutation breeding*, a single surviving member of the population is modified and features are altered (added or deleted) to create a new member of the population.
   (c) *Brand new member creation*, new population members are created and introduced into the population.

The above components are employed for population size p and for n number of generations. The values for p and n are chosen by the user depending on the complexity of the search space. Genetic algorithm efficiently exploits historical information to speculate and create new search patterns/points with expected improved performance. This generational method coupled with the passing on of positive "traits" from generation to generation is the foundation of genetic search algorithms and the back bone of the artificial intelligence philosophy – Genetic Programming.

DOI: 10.2352/ISSN.2470-1173.2016.18.DPMI-016

Table 1: Image Filter Building Blocks

Building Block	Maximum inputs (kids)	Building Block Attribute	Symbol
Weighted Sum	2	weight	SUM
Difference	2	none	DIFF
Multiply	2	none	MUL
Median	0	window size - 1/2/3	MED
Average	25	none	AVG
Interpolate	0	Interpolate type - Five Cross/Three Cross/Five Plus/Three Plus	INT
Pixel	0	none	PIX
Random Pixel	0	X offset and Y offset	RAND
Constant	0	Constant Value 0 – 255	CONST

Traditional genetic algorithms are search algorithms that are primarily used for curve fitting. In contrast, genetic programming aims at complete autonomy of the computer in developing a programmatic solution for the problem at hand. Genetic programs use genetic algorithm techniques to learn and solve problems autonomously. In essence, genetic programs are Computer programs creating specialized computer programs to solve a problem in a broadly defined space. The computer is in full control and is responsible for:

1. Using fundamental building blocks to build complex functions/equations to solve the problem at hand.
2. Evaluate the complex functions/equations it generates in a quantifiable manner in order to rank them and weed out solutions that are not favorable
3. Use the best solutions generated as the baseline to generate more solutions, thereby generating a better class of solutions with improved performance than its predecessors.
4. Finally after several iterations of the above process present the user with the solution in a way such that it can be reused for solving problems from a similar problem space in the future.

Clearly, using Genetic programming as the entity/user to generate the image enhancement filter we are *completely removing the need for human interaction in the filter creation process*. Thus achieving goal number 2 we set out to meet – *The procedure involves minimal human interaction (if any at all)*. The only human interaction needed in this approach would be to provide the image sets and of course to develop the genetic algorithm itself. The genetic algorithm itself need only be implemented once, and can be reused for any arbitrary image transformation process based on A+B image pairs.

The fundamental requirements behind genetic programming based image filter generation are:

1. Provide the genetic program with the tools to build image enhancers.
2. Provide the genetic program with benchmark images which are used to evaluate the image enhancers.
3. Provide the genetic program with an evaluation metric to generate fitness numbers for the image enhances.

Genetic Program Internals

The field of genetic programming has grown and today there are several models and approaches to the design of genetic algorithm. For the purposes of this paper, two are relevant; a traditional simple genetic program model, and a more sophisticated island model based genetic program. We will first discuss the simple genetic program model to generate filters, note some deficiencies, then modify the design to an island model genetic program.

Traditional Genetic Program algorithm

A traditional Genetic programming model is the oldest and most widely used model. This model is at the core of pretty much all of the Genetic Programming models in use today. This model involves the following steps [3]:

1. Create a Population of Members
2. Evaluate Population Members and assign fitness scores to members and sort members based on Fitness scores.
3. Save off a select few members with high fitness levels
4. Perform Crossover to Replace a few of the remaining members
5. Perform Mutation to Replace a few more of the remaining members
6. Replace the remaining unhealthy members (members with low fitness scores) with brand new members.
7. Repeat $2 \rightarrow 6$ for multiple generations

The last generation contains the population which contains the best members evolved by the genetic program.

The following sections detail the *Genome, Population Member Creating Algorithm*, *Evaluation Algorithm*, *Crossover Algorithm* and the *Mutation Algorithm* used in this paper procedure/-experiment.

Genome Structure

The genome is represented using an inverted tree structure, as shown in figure 5. Each node in the tree is a primitive operation and the nodes feeding into it are the inputs (or children) of that primitive operation. The primitive operations or building blocks that will be used in this genetic program are described in and Table 1.

Population Member Creating Algorithm

The genetic program is initially seeded with a population of size n. After some experimentation, it was found that a population size of $n = 100$ was a good choice. Population sizes larger than

DOI: 10.2352/ISSN.2470-1173.2016.18.DPMI-016

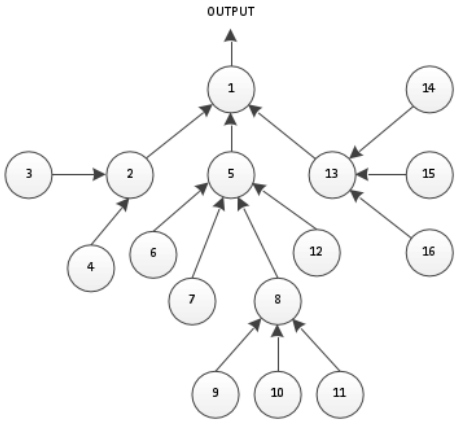


Figure 5: Genome Structure

100 took far too long to be evaluated and didn't add to the diversity of the population. At the same time population sizes smaller than 100 didn't provide the genetic diversity needed for healthy genetic evolution. Each population member is represented by a an inverted tree consisting of `Max_Nodes`.

Each node is a randomly selected primitive operation to create an inverted tree structure. Each Genome is generated by selecting an initial primitive operation as the root node and then by selecting random operations (child nodes) for each of the of inputs of the root node. The child nodes of the root node are then taken as root nodes and further child nodes are attached to their inputs recursively until `Max_Nodes` have been added to the tree. `Max_Nodes` was set to a value of 64 in the example experiment. During the beginning phase of the project `Max_Nodes` was set to 128, however the algorithms tended to run for extremely long periods of time, sometimes days and the performance of the generated filters were only marginally better than the performance of filters with 64 nodes. Hence 64 nodes were deemed large enough to generate a complex image filter from the chosen primitive building blocks.

Population Evaluation Algorithm

During the course of every generation, each population member (image filter) undergoes an evaluation process and a score is generated for each image filter. This score is then used to rank the population members in order from best performing to least performing members. The population members are evaluated by:

1. Running the flawed image through the population member and generating a filtered image.
2. Performing a pixel by pixel difference of the filtered image and the ideal image.
3. Integrating/accumulating the difference (or error) for each pixel.
4. Assigning the accumulated result as the score for the given population member (image filter).

A lower score indicates that the filtered image generated by the population member is closer to the quality of the ideal image. Using this metric, the population is ranked from low $\rightarrow$ high scores.

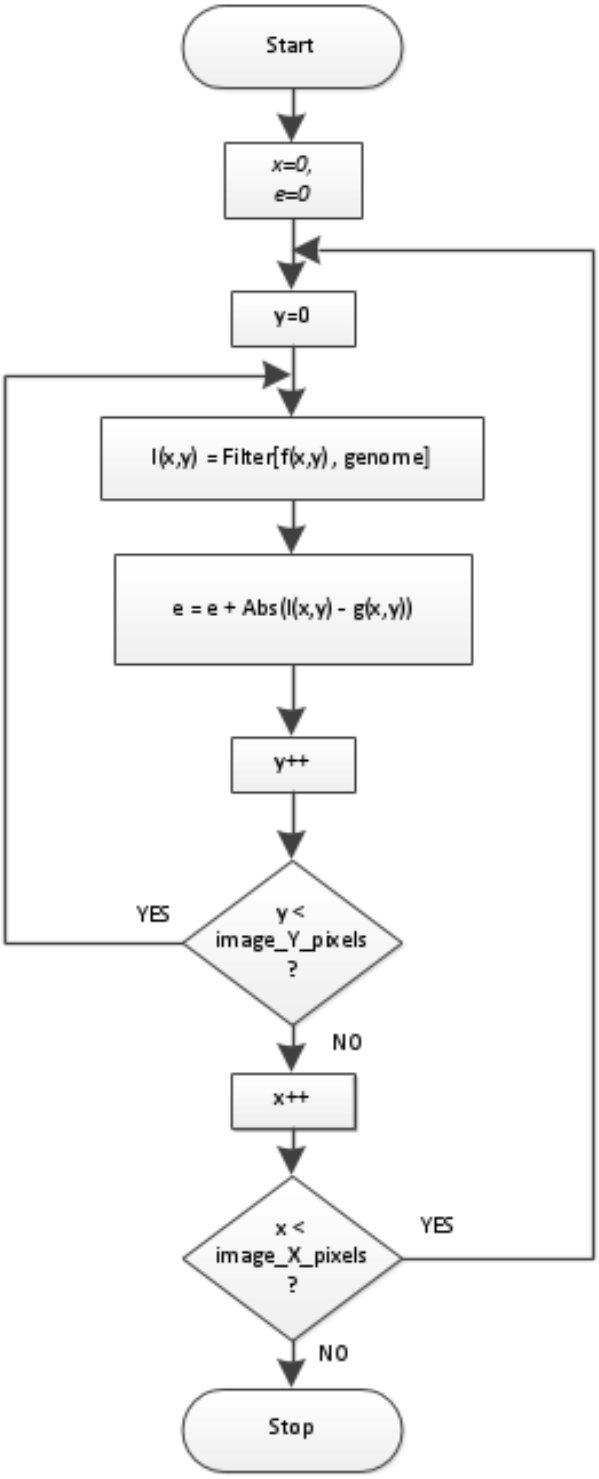


Figure 6: Evaluation Algorithm

Lower ranked members (members with high score values) of the population will be removed and replaced in the *Crossover* and *Mutation* phase of the population's lifetime. For this experiment, during the lifetime of every population:

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.18.DPMI-016

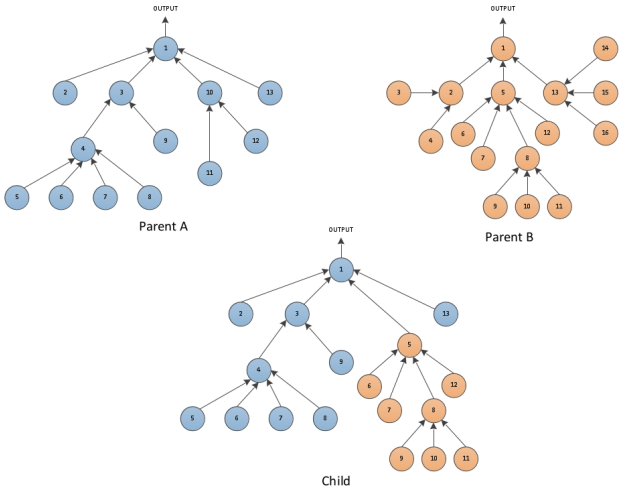


Figure 7: Visual representation of the crossover algorithm

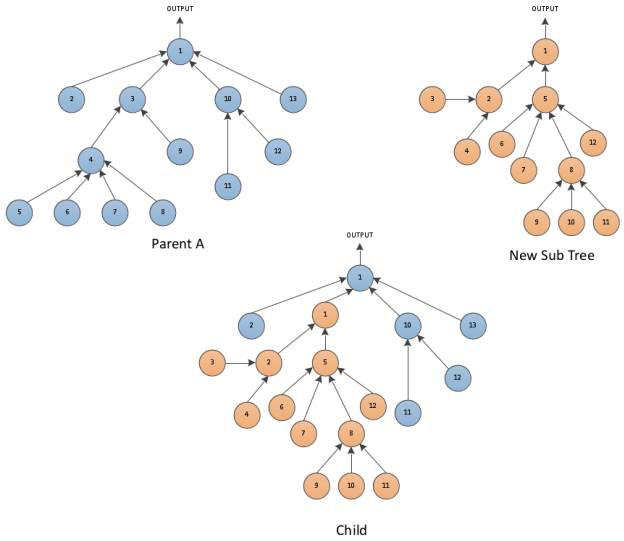


Figure 8: Mutation Visual Representation

- Top 25% of the population survived.
- Next 25% of the population was replaced by performing *Crossover*.
- Next 25% of the population was replaced by performing *Mutation*.
- Remaining 25% of the population was replaced with brand new members.

Crossover Algorithm

Crossover in genetic programming terminology is akin to sexual reproduction in the animal kingdom. New members of the population are created by taking features and characteristics from well-performing members of the population. In this experiment population members are represented by trees, so crossover reproduction will be performed as depicted in Figure 7, with the following steps:

1. Randomly select 2 high ranked surviving members of the population to represent *Parent A* and *Parent B*.
2. Randomly select a sub node of *Parent A* tree and randomly select a sub node of *Parent B* tree.
3. Using the selected sub nodes as the splicing points, create a new tree by splicing together top half of *Parent A* and the bottom half of *Parent B*.
4. If the resultant new tree has more than `Max_Nodes` repeat steps 3 & 4
5. Replace one of the weaker members of the population with the newly created member.

Mutation Algorithm

Mutation is performed by taking a well-performing member of the population and slightly modifying the genetic code/structure of the member to create a new population member, as shown in Figure 8. In this experiment, mutation is performed with the following steps:

1. Randomly select a high ranked surviving members of the population to represent the *Parent*.
2. Randomly select a sub node of the *Parent* tree.
3. Using the *Population member creating algorithm* create a new subtree.
4. Using the selected sub node as the splicing point, create a new tree by splicing together top half of *Parent* and the subtree newly created in step 3.
5. If the resultant new tree has more than `Max_Nodes` repeat steps 3 & 4
6. Replace one of the weaker members of the population with the newly created member.

The traditional simple genetic programming model described section is a good start, but it has the drawback of being very slow and not very scalable when it comes to improvements and does not allow for population diversity. It also does not accurately model the real-world genetic evaluation process. More importantly, while the fundamental idea of using Genetic Programming for filter generation meets our requirement of developing filters that are robust and stable, it is fairly time consuming. Evolving filters over several generations means running the Genetic Program for extended periods of time and this defeats one of our initial requirements of making the process fast. The subsequent sections are dedicated to solving this final piece of the puzzle. In the followings sections we will be exploring:

- Island mode Genetic Programming
- Using Supercomputers to improve performance numbers and reduce run time.
- Using TCC and its run time compilation library libtcc to further improve run time performance and tremendously increase the speed of the algorithm.

Island Model Based Genetic Program

The Island model takes a more advanced approach to evolution. It adds the following 2 main features to traditional Genetic Programming:

1. Multiple Independent Simultaneously Evolving populations.
2. Periodic Migration of population members between islands.

DOI: 10.2352/ISSN.2470-1173.2016.18.DPMI-016

As a result this approach comes closer to the evolution observed in nature where populations usually evolve in groups/-colonies or Islands and migrate over to other colonies and continue to evolve in the new colony. This approach allows for the following benefits[4]:

- Each population can possibly bring in a new flavor of solutions suited to solving specific issues in the problem space
- Migration allows for populations that have evolved to different local peaks to take features from one another and continue evolving towards the true peak
- Running several populations in parallel can significantly increase the number of members evolving at a time leading to more possible solutions.
- The continuous migration of population members allows for good genetic material to be spread across other populations and this can result in a healthier and diverse genetic code.
- Parallelizing also significantly increases the speed of the Genetic Program.

Supercomputers for Genetic Programming

To fully realize the potential of the Island model, it would make sense if each individual Island GP was running on its own dedicated Processor with its own dedicated memory to work with. The design of the island model naturally fits in the architecture of cluster supercomputers[4]. There are several benefits of using supercomputers to realize Island models:

- Dedicated processors means that each Island can fully utilize the potential and power of a whole CPU. Since each machine is running only a single Genetic Program, the HW doesn't have to be very sophisticated and can be a generation older, which yields to cheaper HW.
- Isolated Islands with dedicated CPUs allows GPs to evolve at their own pace. When it is time for migration the Island absorbs a new member and moves on again continuing at its own pace.
- The whole is much greater than the sum of the parts; Research has shown that Genetic Programs solves a problem faster with semi-isolated island models [4]. Now parallel genetic programming often delivers a superlinear speed-up in terms of the computational power needed for yielding a solution.

Island Model Overview

The Island Model Genetic Algorithm developed for this experiment, summarized in Figure 9, uses the following parameters:

- Each Genetic Program uses a Population size of $n = 100$. The reason for choosing this value has already been explained in previous sections.
- the Generation limit for the islands was chosen to be 100 generations. Through the course of developing this experiment it was determined that a population size of 100 evolved marginally beyond 100 generations.
- Migration between islands was timed to occur every 1 hour. This gave the islands sufficient time to evolve before migrating members. Shorter time durations resulted in islands that had gone through one 1 or 2 generations.

Migration and the Transit Authority

Migration is the process of exchanging genetic material between Islands of Genetic Programs. In this project, migration is controlled and timed by the central node, called the transit authority. When it is time for migration each node or island picks its best member for migration and sends it over to the transit authority. It also lets the transit authority know if it has gone through all its generations and is done. The transit authority will not send any more immigrants to an island which has marked itself complete. The transit authority collects the generation state and travelers from each island. It then sends travelers (now immigrants) to adjacent islands in a circular fashion, such that the traveler from $Island[n]$ is sent to $Island[n+1]$. If the island is the last in the node list, then its traveler wraps around and immigrates to the first island $Island[0]$. Finally the transit authority updates its list of currently active islands and sends them a time marker for indicating when the next migration will take place. After this the transit authority also saves the list of travelers to a file for user. This process repeats every migration cycle until all islands have finished evolving.

Direct evaluation with TCC & LibTCC

As with all algorithms, genetic programs have their own performance bottlenecks. After analyzing the genetic programming algorithm used in this project, it was clear that the genetic program was spending a majority of its execution time in the population member evaluation stage. This was due to the fact that for every pixel the GP was traversing an entire tree of nodes. This meant traversing and processing as many as 64 nodes per operation. Tree traversal is one of the most inefficient and power hungry operations. To improve the performance of the GP, we we look to a little known but very stable and impressive project called the *Tiny C Compiler* or TCC [1].

TCC was originally developed by Fabrice Bellard, and is maintained by a community of volunteers. TCC is extremely small, yet it compiles C into native x86 machine code much faster than most C compilers. Better still, TCC can also be invoked as the `libtcc` library to dynamically generate code into memory that can then be called as a function by the currently-running program.

The fast compile speed combined with `libtcc` allow additional speedup of several fold by evaluating population members using direct execution of compiled code, rather than interpretation. The inverted tree structure of the genome is converted into C code which then the GP can dynamically compile and use on the fly with the help of `libtcc`. Figures **??** and **??** show the changes to the Genetic Program that were made in order to fully utilize libtcc.

Results

Filters 1 and 2 were trained from the Low ISO Image and High ISO image shown at the beginning of this paper and figures 12 and 13 show the result of applying the generated filters to the high ISO image.

DOI: 10.2352/ISSN.2470-1173.2016.18.DPMI-016

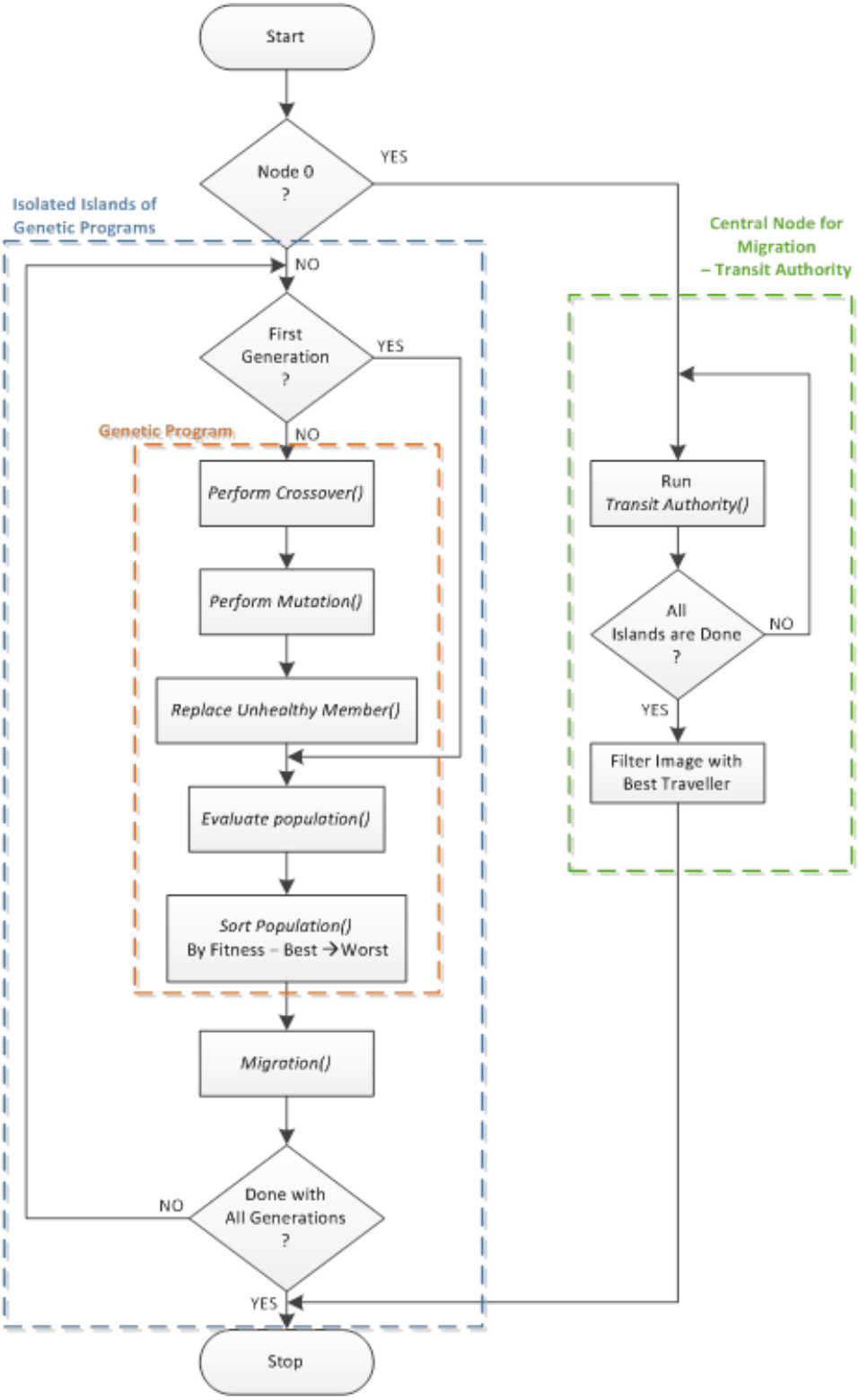


Figure 9: Island Model Algorithm

DOI: 10.2352/ISSN.2470-1173.2016.18.DPMI-016

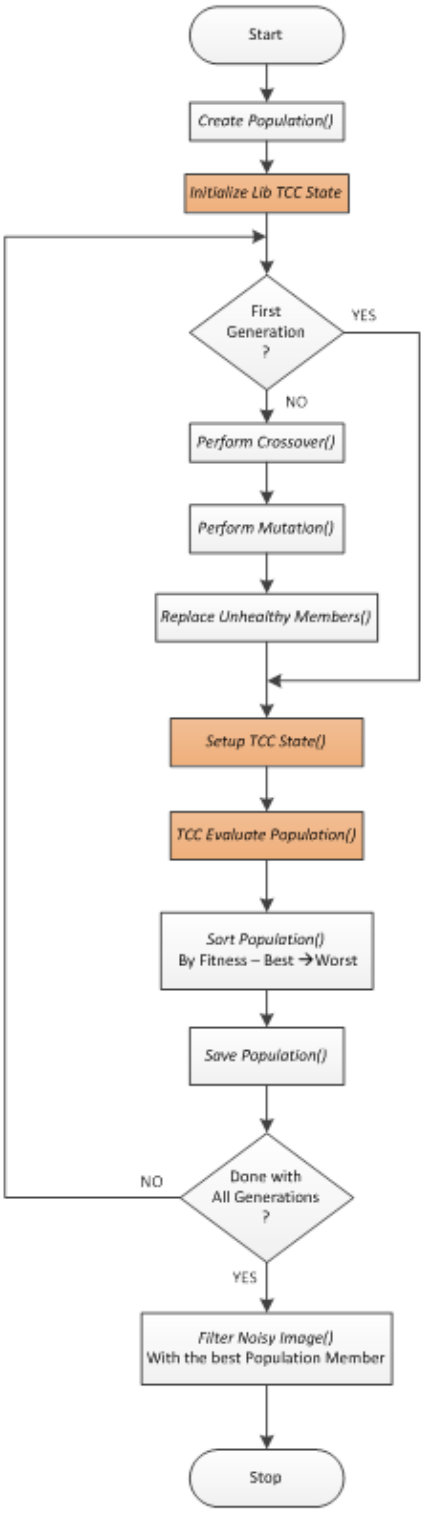


Figure 10: LibTCC Based Genetic Program

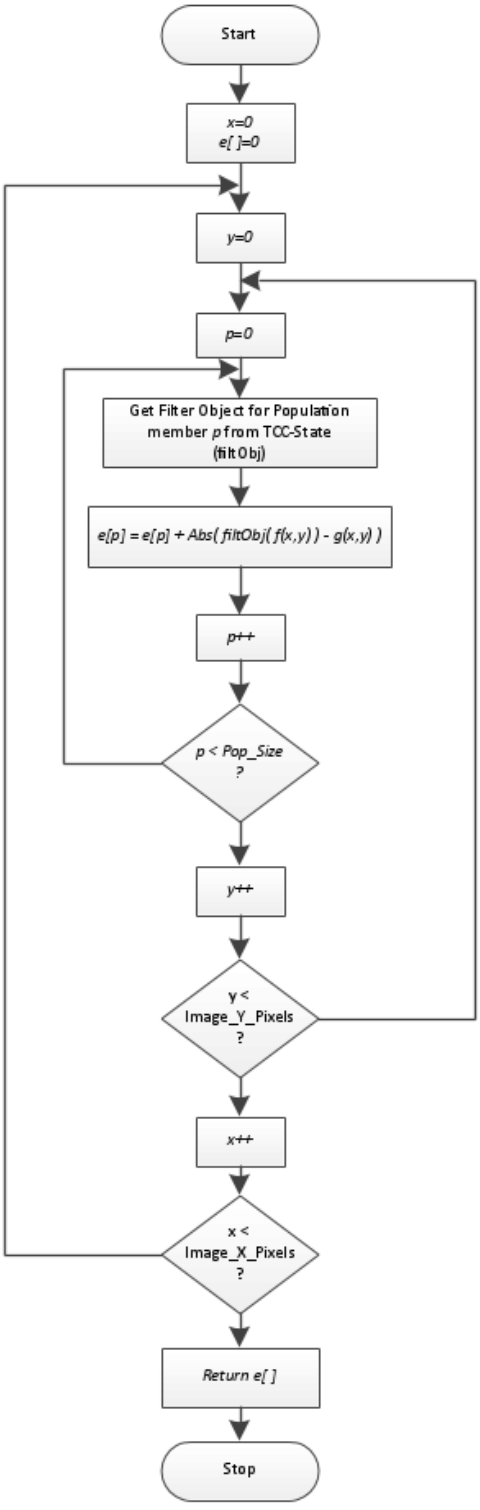


Figure 11: Population Evaluation Using LibTCC

DOI: 10.2352/ISSN.2470-1173.2016.18.DPMI-016

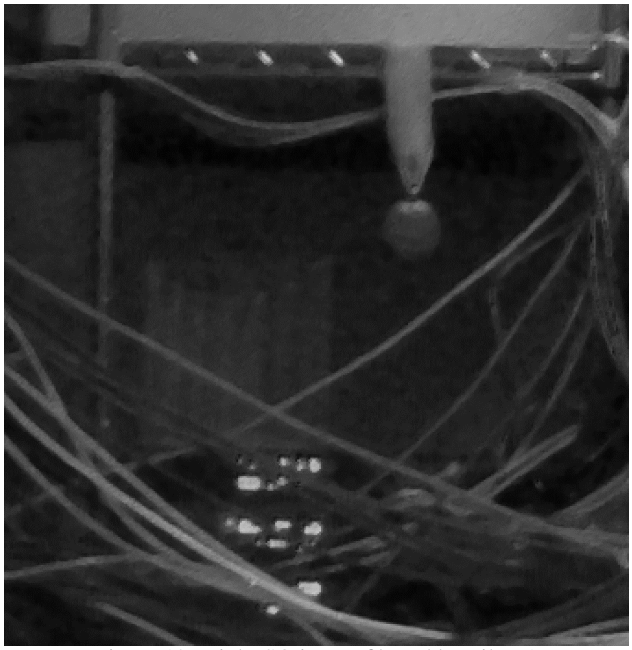


Figure 12: High ISO image filtered by Filter 1

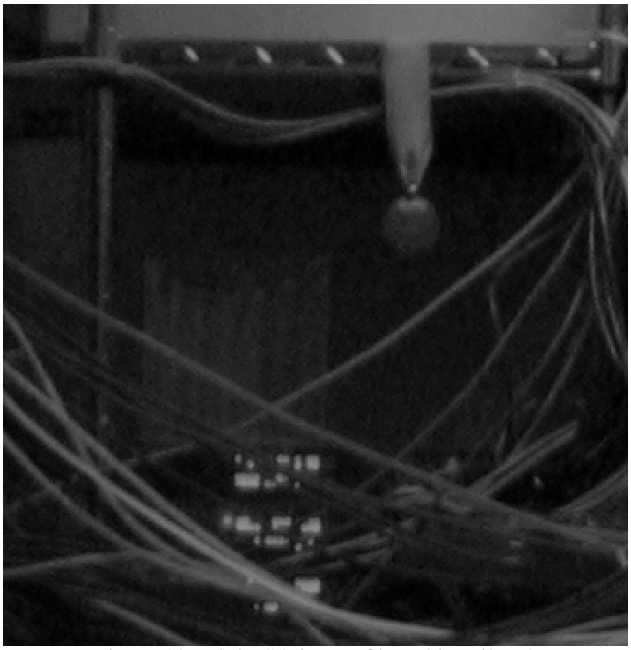


Figure 13: High ISO image filtered by Filter 2

Listing 1: Filter 1

```
AVG(13,ADD(MUL(PIX,PIX),MED(3)),
RPIX(-2,1),
DIFF(DIFF(INTERPOL(1),INTERPOL(0)),INTERPOL
    ↪ (2)),
MUL(INTERPOL(1),MUL(AVG(6, MUL(ADD(ADD(
    ↪ INTERPOL(2),CONST(219)),RPIX(-2 ,2))
    ↪ ,MUL(PIX,MUL(INTERPOL(2),RPIX(-1,2))
    ↪ )),MED(1),MED(2),MED(3),INTERPOL(3),
    ↪ PIX),CONST(192))),
MUL(MUL(PIX,RPIX(-2,2)),DIFF(ADD(ADD(
    ↪ INTERPOL(0),DIFF(RPIX(0,-1),PIX)),
    ↪ AVG(6, MED(2),ADD(DIFF(MED(1),
    ↪ INTERPOL(1)),INTERPOL(0)),CONST(153)
    ↪ ,RPIX(1,0),CONST(21),RPIX(1,-1))),
    ↪ RPIX(2,0)))
PIX,RPIX(1,2),RPIX(2,-1),RPIX(0,-1),RPIX
    ↪ (0,1),INTERPOL(0),MED(1),CONST(30)),
```

Listing 2: Filter 2

```
ADD(MUL(CONST(125),INTERPOL(2)),RPIX(0,2))
```

Conclusion

This paper has outlined a system that automatically can develop filter algorithms to perform complex image transformations. The primary input to the system is one or more A+B pairs of pixel-aligned original and oracle images. From that, the system uses island-model parallel genetic programming, accelerated by direct execution of TCC-compiled code (rather than interpretation) for fitness evaluation, to create optimized filter programs that can convert image A into a good approximation to image B.

While the above methods and approach provided good results, there is still room for improvement both in the approaches used to evolve filters as well as computing performance. In particular, more sophisticated use of the island model seems promising. For example, individual islands could be dedicated to specific sections of the images and/or specific color channels in order to increase speed of convergence without sacrificing diversity across islands. It should also be possible to parallelize evaluations within individual islands so that the evaluations are sped-up both by parallelizing execution over different portions of an image and by reducing the amount of image data each processor must access. Of course, it is also easy and effective to incorporate stronger primitives. Obvious possible additions include logarithm, threshold, Gaussian blur, and tone mapping.

References

[1] Fabrice Bellard et al., Tiny C Compiler, *http://www.tinycc.org/*, (2013).

[2] Rafael C. Gonzalez and Richard E. Woods. Digital Image Processing (3rd Edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, (2006).

[3] John R. Koza, Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, Norwell, MA, USA, (2003).

[4] John R. Koza, Asynchronous "Island" Approach to Parallelization of Genetic Programming, *http://www.genetic-programming.com/parallel.html*, (1999).

Author Biography

Subash Marri Sridhar is a Staff Engineer at Qualcom primarily involved in indoor positioning and range measurement technology. The work reported here is very closely related to research he did as part of his MS degree in Electrical Engineering from the University of Kentucky, completed in 2015. The co-authors, Henry Dietz and Paul Eberhart, are respectively his MS advisor and a fellow graduate student also working in the Aggregate.Org research group