***Project Metrics***

Superior quality and flexibility for users compared to other applications out there

Better stability for machine learning kernel

Segmentation algorithm for better handling complex artworks

Low cost and high efficiency

Varies accelerations which can be applied to multi-platforms

Sight on mobile platform possibility

***Summary of Design***

There have been selective communities in public which have wide usage of image enhancement systems, for personal use, commercial use, or academic use. In developing and operating such a system, there are five key parameters upon which the design must focus on: (1) up-scaling enhancement quality, (2) efficiency, (3) system acceptance, (4) operating platform, (5) ease of training and implementation.

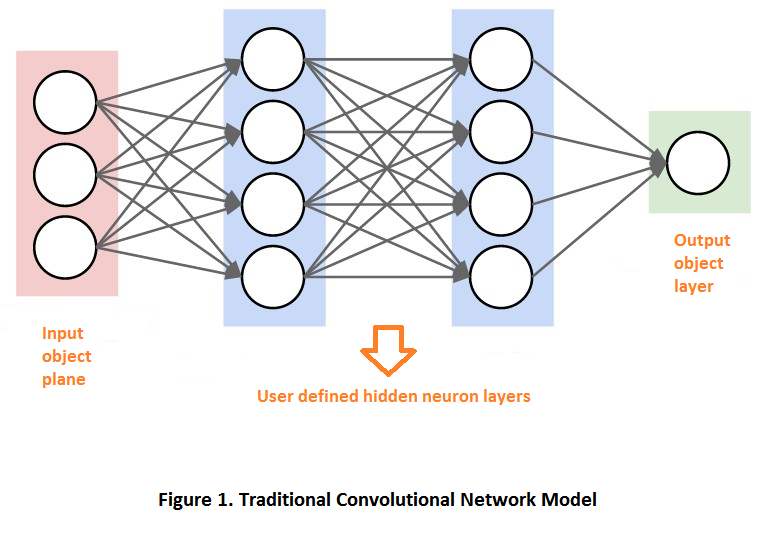
Through the use of extensive image up-scaling and enhancement quality testing, system design analysis, and operating platform evaluation, our team has improved and optimized some critical aspects of image up-scale and enhancement system by adapting existing algorithm and developing our unique logistics. Hereby, we are presenting the communities with a low cost and easy adaptive system that is close to being ready for commercial development and high level applications.

Our filter design system is based on technology of machine learning method for image high/super resolution. By mapping the images with low and high resolutions from end to end, a convolutional neural network can be achieved. It has the ability to provide intelligent filtering technology to the program that performs the convolution to transform the low-resolution input into a high-resolution output. It will also employ image segmentation to aide the filter in edge detection This system can finish the process in the following steps:

* A process that takes a large set of images to train the model for the filter and output the filters
* Denoise the image
* Segment the image into partitions
* Scale each segment using bicubic upscaling by the desired scale value and run the filter
* Merge all upscaled segments into one final output image

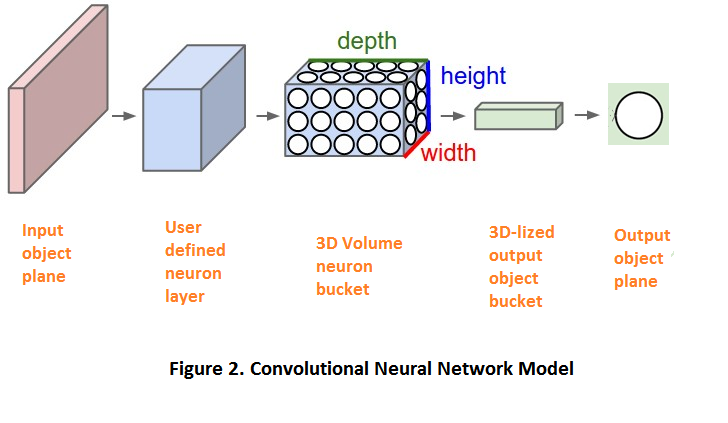
Building the CNN

In general, the convolutional neural network is like the traditional neural network. They partake the same features such as objects are made into neuron forms, and each neuron unit can take inputs, then performs dot product, and it does not require to be a linear procedure. The advantage of which is that neurons have learnable bias and weights, so that it makes it easy for users to model them. The layers can be formed by neurons and input can be decomposed into the mannequin. And so, as the result, a single differentiable score function can be held from the pixels of raw images. In the backdrop, a fully connected layer exists for the grading of the loss function to interpret the class grade. The simplified example is depicted in Figure 1.



Still, compared to the traditional neural network, the convolutional neural network explicitly takes images as inputs. Then the architecture is fully encoded for the properties of images. Therefore, due to the simplicity, it is easier to connect neurons and requires less parameters for the network implementation.

To void the overfitting effect that the traditional neural network can have, we deploy our neurons in a 3D volume bucket. On each layer, we have the size defined as the width and height, which is directly related to the input. We call the third dimension as depth, which is the depth of the activation volume instead of the depth of the neural network.



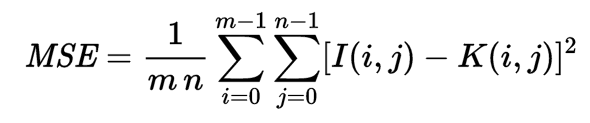
Next, if we consider the layers from the input to the output. Let’s say we have an image as input with the resolution of A by A, then let’s break down the parameters for each layer:

1. Input will be decomposed into RGB channels. Then the RAM usage would be [AxAx3].
2. The convolution layer will compute the number of neurons which are connected to the local input. If we decide to use number of B filters, the RAM allotment would be [AxAxB].
3. The next layer is defined as ReLu. It can perform elementwise activation functions, so that the max (0, x) has the threshold at 0. The size of the neuron volumes will keep at [AxAxB].
4. Then, the next polling layer will simply perform down sampling in the spatial dimensions.
5. Then the final fully-connected layer will compute the class grades, which is defined by users. If we categorize the image sets into C categories, we will have an output [1x1xC]. Per the nature of the convolutional network, each neuron cell in the final layer will be connected to all the coefficients in previous volumes.

Then, if we consider the convolutional layer, we know that it is defined as a set of learnable filters. It is the fundamentals of convolutional network firstly being applied in LeNet. It is the most computational heavy module for the machine learning part. Every single filter has a small planar size but a full depth matches with the 3D neuron block depth. Simply like human brains, certain layers will be activated if and only if similar features can be found in the object.

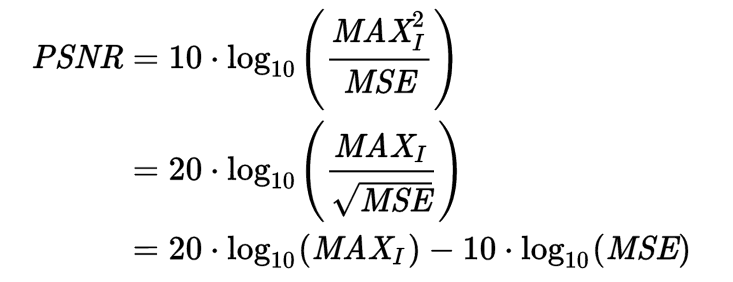
If we talk about the single image super resolution, there are 4 major categories: edge based modeling, prediction modeling, example patch modeling, and image statistical modeling. However, if we talk about the general performance, it seems that the example patch modeling can achieve the best overall quality. In addition, it can be derived into two distinct methods. The internal learning method was first proposed in 2009, and applied to the AlexNet in 2012. It utilizes the base similarity properties and generate the exemplar patch via the input image. The external one, which being used in this case, learns the mapping between low and high resolution image patches from the external libraries. With the help of nearest neighbor strategy, it makes the user easier to control the speed and the computation complexity, which later is being called the sparse coding formulation. There have been some applications for image restoration with the similar approaches, however, there is a key feature they did not focus on, which is denoising.

When we process the modeling for a specific patch set, an end to end mapping algorithm will be used with several network estimation parameters. These parameters are gained from minimizing the losses in between the related high resolution image and the reconstructed image. If we denote the reconstructed image as I, the high-resolution contradiction as k, the low-resolution image as i, the total pixel count is j, n is we can get the loss function by using the Mean Squared Error:



N is the number of images in the training patch.

Then, we can achieve a high Peak Signal to Noise Ratio. This is the standard metric function being widely used for image restoration quality evaluation.



In addition, *MAXI* is the maximum possible pixel value of the image.

Optimizing the CNN

Optimization of the CNN was done using the algorithm ADAM developed by Diederik P. Kingma and Jimmy Lei Ba described in their paper Adam: A Method for Stochastic Optimization. The motivation for choosing this algorithm is the low computational complexity of the method. As stated in the paper it only requires first order gradients on the input objective function which has the same computational complexity as evaluating the original function. Some other advantages are that the magnitudes of parameter updates are invariant to rescaling of the gradient, its stepsizes are bounded by the stepsize hyperparameter, and naturally performs a form of step size annealing. In any case this means the algorithm has a degree of automation and adaptivity that many other algorithms do not present.

We start with defining the pseudo code which defines how this functions. Adam uses the method of moments which is where unkown parameters are estimated using 0th, 1st, and higher order moments. Adam uses only the first two. For example in the case of a random vector, 1st order moment is the mean vector and second order is the covariance matrix, a measure of how related two variables are.

Inputs: α (step size), β1β2 (exponential decay rates for moment estimation ranging from [0,1] ), (stochastic objective function), (initial parameter vector). As advised by the paper the input values for α=0.001, β1=0.9, β2=.999 and =10^-8 (for update last step)

Initialization: (1st moment vector), (2nd moment vector), (Initialize timestep)

Runtime:

While has not converged:

: Increment timestep

: Compute the gradients with respect to at the current timestep

: Update the first moment estimate

: Update the second moment estimate

: Compute bias corrected first moment estimate

: Compute bias corrected second raw moment estimate

: Use updated bias corrected raw moments to compute the new

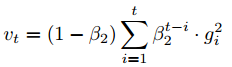
End while if converged

Return the estimated value(s) which we want to estimate

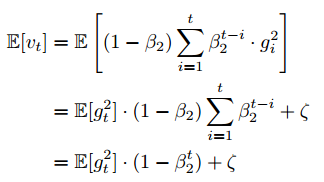
In general, we are trying to minimize a noisy objective function such that it matches our expected outoput (an image) with respect to the parameters. Moment estimates are used to compute a converging parameter using bias correction. Bias correction is required because moments are initialized to zero thus without the bias the estimate may or may not change at all.

Bias Rule

Given the gradient () of the stochastic objective f we can estimate the moment based upon all previous moments. This is done using the following summation formula. Where the beta represents the decay rate of the function and gradient squared is the elementwise square of the vector. (second moment shown here).



The expected value of the moment can be calculated in the same manner where ζ represents an offset value to account for the moving average of the second moment.



Looking at the final line of the equation we are able to see how we can calculate the expected moment and the scalar value multiplying the expected second order gradient to obtain our biased second order moment. This in turn is what is used to scale to obtain the correct value for the newly defined timestep scalar below.

Update Rule

Kingma and Ba have stated that the following conditions are a good choice of how to choose alpha as a step size. |∆t| ≤ α · (1 − β1)/ √( 1 − β2) in the case (1 − β1) > √ (1 − β2), and |∆t| ≤ α. Recall α is the step size and essentially the bound on how many iterations the algorithm will have to go through before we are able to reach an estimate of our parameter. Often we are able to choose α based upon some physical limitation in the model we are building. However, although inefficient often choosing a small enough step size (thus large amount of iterations) will still be able to generate the same result as long as the algorithm is run long enough. In Adam’s case the step size will be automatically scaled as we reach our expected parameter values in the asymptotic case. Thus, larger step size values will still be relatively effective. We are able to see this in the equation for the update step where α is scaled by the moments



Kingma and Ba “define” a signal to noise ratio which describes the condition in which we have converged on the expected parameter when the SNR reaches zero. This is easy to see because we can note that a value multiplied by zero is zero.

Image Segmentation

The algorithm used in this project is based on the paper "Efficient Graph-Based Image Segmentation" written by Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Originally, the segmentation algorithm used would be based on the paper “Normalized cuts and image segmentation.” by Shi, Jianbo, and Jitendra Malik. However after our implementation in MATLAB it was decided that although N-Cut is superior to the algorithm described by Felennswalb and Huttenlocher is much more efficient in calculation of the segments.

Felenswalb and Huttenlocher's algorithm uses a simple idea. The algorithm is based upon a decision factor. This is what is used to decide where a cut is made at a certain edge. Felenswalb and Huttenlocher defined several definitions which are used to define their decision predicate.

The decision predicate is as follows after creating a graph G=(V,E) where V represents a node and E represents an edge connecting two nodes. Here the graph is an undirected graph with vertices where V is the set of elements to be segmented. The edges are defined as where represent the two nodes in which the edge is connected. Finally, the weights on the edges are a measure of dissimilarity between neighboring nodes. This trait can be picked during the creation of the graph, however for this project we used the dissimilarity of pixels.

We want to create segmentations S (subset of V) made up of components C (subset of S) where each component is another subgraph, G’=(V,E’), of G. Because we are using dissimilarity of pixels as the weight, it is only natural that we conclude that low weight values mean that nodes are part of the same component. High weights mean that it is not part of this component.

Let us define the decision predicate. Felenswalb and Huttenlocher's algorithm uses three definitions to determine the predicate.

1. Internal Difference: Maximum weight edge in a component

1

1. Difference: Minimum weight edge connecting two components. If the two components are not connected this value is infinity.

1

1. Minimum internal Difference: min(minimum edge weight of component 1+ threshold, minimum edge weight of component 2+ threshold)

5

Note: Here tau is defined as τ(C) =k/|C| where k (controls sensitivity of segmentation) is a variable to be adjusted and |C| represents the cardinality of the component. Larger k will bias the algorithm to produce larger components.

We can explain the choices Felenswalb and Huttenlocher made as follows.

1. Internal Difference: If the maximum weight edge of a component is connected we know that this component has relevance to be connected or segmented.
2. Difference: This is sort of like a “median” of the weights in a graph giving an idea of how the weights are biased in both components.
3. Minimum Internal Difference: This is the same as internal difference except it allows us to account for a threshold to control the “relevance” of a component and whether two components should be merged.

Finally, Felenswalb and Huttenlocher defined the decision predicate as follows.

6

The predicate above will check whether a boundary exists between these two components by checking if the difference between the components is large relative to the internal difference with at least one of the components. If it evaluates to true, it will merge; if false, it will not merge. This works because recall Int(C) will return the maximum weight of C and all weights greater than Int(C) will need to be merged. Thus, setting the condition to check for weights larger than Int(C) will produce a merge between both components.

When running the algorithm, we control the variable k which controls the sensitivity of the segmentation. A large k will produce a preference for larger components and a small k will produce a preference for smaller components. Small k will result in components with components that are long and thin, large k will discourage such segments.

**Bibliography**

P. Felzenszwalb, D. Huttenlocher  
International Journal of Computer Vision, Vol. 59, No. 2, September 2004

Kingma, Diederik P., and Jimmy Lei Ba.

"ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION." *ICLR* (2015): 1-15. 23 July 2015. Web.

**Technical Description**

Upscaling Block Diagram



Discussion of Merge

The process of merge took an extensive amount of time to make working. For some reason after denoising and upscaling there were many anomalies within the image datat. For instance a black pixel may have non zero RGB valuers or in YUV form it may have non 0 UV channels or very large Y values for 0 UV. We concluded that we cannot use ==0 as the threshold for computing whether a pixel is a part of a segment or not.

Originally each segment was passed as a map of RGB pixels to the merge portion of the code and we checked if the R,G,B channels had values <1. And if it did we would do nothing to the pixel. However this proved impossible as after denoising and upscaling some of the supposed black pixels weren’t truly black. Worst of all the values weren’t consistent sometimes they were (1,1,1) or (3,3,3) or (5,5,5) and we were not able to discern a pattern or upper bound on these values.

After that we decided to convert the image to a black and white format and then directly compare whether an image is “black” or not. The same problem occurred as above and some segments would end up overwriting parts of other segments due to blurring of pixels around the edges.

We tried the same thing going from RGB to YUV and performing the same computation. However we were faced with the same problem again with erratic Y values.

There are two things to consider about why the above three ways failed. Firstly, we were using OpenCV’s implementation of pixel conversion. We are not sure how it is converted and whether it will produce interpolation errors or computation errors. Secondly, the way we accessed the Y values of the YUV converted image were (probably) wrong. Thus, we were getting values as uint output instead of the float values the Y channel was supposed to be.

To resolve these issues, after much trial and error we discovered that the best way to correctly do this is to calculate the Y channel ourselves manually. This required scaling the output Y value by 255.0 and doing calculations in floating point to maintain as much accuracy as possible. Only with this way were we able to properly use a threshold to determine whether a pixel is black or not with a much smaller margin of error in the use of floating point.

Notable Equations Used in Specific Functions

***main.cpp***

* Calculating the number of iterations required for a non 2x scaling factor
  + Because the user may input a scaling factor which is not 2x (the model designed was to be used for 2x scaling) we need to calculate the number of iterations required to reach that value
    - where scale ratio is an input to the top level function

***convertThreading.cpp***

* Calculating the number of blocks to be used
  + Given the size of the block we need to calculate the number of iterations across the number of rows/columns it will be used across. This is done by calculating the two integers for splitColumns and SplitRows
    - and similarly for rows

Segmentation Block Diagram: **Base Code was From Professor’s Page. Removed uneeded parts and added functions to return the correct variable type and conversion from OpenCV to image class.**

****

Usage of Disjoint-Set Forests

Disjoint-set forests with union by rank and path compression. Let us define what these things mean. A disjoin-set data structure allows us to keep track of a set of elements made up of disjoint components. In our case, this would mean that each disjoint set would represent an individual segment. The two notable functions that this provides is the find and union (named join in the implementation). This allows us to determine if two representatives are part of the same root and to join two trees into one single tree building the segmentation. The two notable functions that this provides is the find and union (named join in the implementation). This allows us to determine if two representatives are part of the same root and to join two trees into one single tree building the segmentation.

In union-by-size, the determining factor for which set to join to the other is the number of elements in each set. Union by rank ensures that we can retain minimum depth for maximum efficiency. This is results in a complexity of ~O(log n). Rank is initially 0 when a one element set is formed. Then as we begin joining sets, sets of the same rank will be joined and then incrementing the rank variable of the new union set.

Now what is a disjoint-set forest? Instead of implementing a disjoint-set data structure using either an array or vector, in which a representative could be any element inside the set, a disjoint set forest uses a tree structure where the representative is the root of the tree. This alters the find and so that when called it reaturs root of each tree instead. In the case of find called on a node, it will follow parent nodes until it reaches the root. This means that we do not need to keep track of the actual set in which an element belongs to as we can easily search through a tree to determine if two elements are in the same set. In addition to this, when using path compression when find is called on any node it will attach itself and all its parent nodes directly to the root hence lowering the height of the tree making subsequent finds much faster.

Felenswalb and Huttenlocher’s algorithm has complexity O(m α(m)) Where α(m) is the inverse Ackerman function and m represents the number of vertices in the graph. Because there are at most three edges connected to each vertex (one for each RGB component) all operations can be done in constant time. Therefore the only limiting factor is the rank (depth) which is bounded by O(m α(m)).

Notable Equations Used in Specific Functions

s***egment-image.h***

* The diff function measures the dissimilarity between two pixels in this case the function has been written such that it will calculate the square root of the square sum of all 3 RGB channels. This will be stored as the weight

***filter.h***

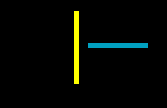
* As mentioned above the Gaussian filter is used to smooth the RGB channels as it has denoising properties which will help in in edge detection. The Gaussian function is defined as follows.
  + then the filter kernel is created by iterating through width of the filter which was defined to be 4.0

**Results**

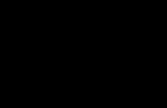
Segementation Results

The hardware used to run the segementation is a i7-5600U Processor (due to the lack of a dGPU in this device). Using a more powerful processor will improve the runtime of the algorithm. However, as we will see the runtime of the segmentation algorithm (CPU implementation) is insignificant to the runtime of the upscaling therefore there is not much to be improved from a hardware point of view. Note: Other than the first test image, it appears that a sigma of 0.8 (for smoothing seems to work best.

Here is the source image of an image to segment. It is composed of 3 colors, a black background and 2 colored bars.



. These calculations were run with the input arguments k=30, min\_size=10, sigma=0.1 and had a run time of .153 seconds for 3 segments.



We still have several things to address for our project. Most of the issues have to do with run time and efficiency of the application of both segmentation and upscaling together. The current form of the implementation requires upscaling every segment individually and then merging them together. Given larger images especially for the CPU based implementation, for 1080p->2160p upscaling the runtime is already 44minutes for one segment. This means if we were to have a lot of segments, the of the city scene, this means 205\*44=9020 minutes approximately 150 hours. This is absolutely unacceptable. If we were to incorporate segmentation again it would have to be inside the upscaling loop itself or somehow built into the models. In hindsight, we can completely disregard this problem when using a DGPU to do the processing as the runtime for one segment will take only several seconds meaning any image will take only several minutes to scale by 2x. However, this will completely nullify any notion of a low power (no DGPU) or mobile implementation (mobile) using image segmentation. Another way we could achieve a much better run time is to use an algorithm developed by someone else for super resolution. For example, we could use [Milanfar](https://users.soe.ucsc.edu/~milanfar/software/superresolution.html)’s algorithm.

The current denoising model developed for this project is somewhat peculiar. It introduces noise to a completely black image. Although the output appears to the human eye to be completely the same, the segmentation algorithm appears to be very sensitive to the noise (albeit visually appealing) added by the denoising process. It would be good to redevelop the model to add the exception for the case of RGB values of (0,0,0) to do nothing. This has led to the use of the original file to produce segments due to the inconsistencies produced by the model.

Currently, the current segmentation code uses a custom RGB class. Although simpler than the OpenCV matrix representation, it would be easier to read the code and easier to adapt the code if everything used was of type OpenCV. In addition to this, the segmentation, although completely dependent on the RGB class (can adapt the diff condition to fit YUV), can be implemented into the YUV format. This will make it unnecessary to use cv::cvtColor to convert between YUV and RGB. This will make it unnecessary to use cv::cvtColor to convert between YUV and RGB.

For improvements to the CNN, a better optimizing algorithm can be implemented in the future (when one is developed) to further reduce the required memory for computation. Currently when running the model, it takes several days and a server with 128Gb of RAM to finish. Using a rig with any lower amount of RAM will cause the memory to overflow. Although probably infeasible at the moment there is a possiblility of moving the core algorithm (without segmentation).

As we know that, Android has native support for OpenCL, which will open the possibility to the device’s native GPU to implement the algorithm on mobile devices. This will significantly reduce the runtime as we have seen above due to the parallelization capabilities of GPUs. As the chips are getting faster and faster, it is possible to run the implementation in real-time, especially for 480P->720P, 720P->1080P. This makes it possible to make the most use out of the available cable bandwidth. This is very desirable to for streaming services (albeit at the dismay of the user) as they are able to stream compressed video and upscale it on the users’ end claiming “true HD resolution”.

**Software Documentation**

***UPSCALING***

*Classes/Structs by File:*

***modelCraft.hpp/cpp***

Model Class:

* nInputPlanes: This number is the number of input planes returned from the json file
* nOutputPlanes: This number is the number of output planes returned from the json file
* weights: This is of type std::vector<cv::Mat> which contains weights per pixel
* biases: This is of type std::vector<double> from bias data taken from the json file
* kernelSize: This is the size of the kernel used to denoise

modelUtility Class:

* static modelUtility\* instance: This is the instance of the modelUtility class
* int nJob: This is the number of threads to be used to do computation
* cv::Size blockSplittionSize: This is of type cv::Size and represents the size of each block used to denoise sequentially

*Functions by File*:

***main.cpp***

**int main(int argc, char\*\* argv)**

-Input Args:

* argc: Number of input arguments
* \*\*argv: A double pointer to the number of arguments

-Usage:

This is the top level function which calls all the denoising, scaling, and merge components of the project. Not inputting any arguments will prompt the user to input the correct input arguments such as the location of the input file or the number of threads to use during computation.

***modelCraft.hpp/cpp***

modelUtility()

Public:

**static bool generateModelFromJSON(const std::string &fileName, std::vector<std::unique\_ptr<Model> > &models)**

-Input Args:

* &filename: This is the input filename in our case this would be the noise model or the scaling model.
* &models: This is a vector of unique pointers for the model

-Usage:

Call this function when you load the json file used to store the model. This will return true if the model was successfully loaded and false if it wasn’t. It will then store the model in a std::unique\_ptr which will be passed to the function which denoises.

**static modelUtility& getInstance()**

-Input Args:

* None

-Usage:

This function will return the pointer to an instance of the modelUtilityThis function will return the pointer to an instance of the modelUtility variable. If one does not exist it will dynamically allocate one for usage.

**bool setNumberOfJobs(int setNJob)**

-Input Args:

* setNJob: This is the number of number of threads set by the user to be used by the function.

-Usage:

This function returns true if it was able to set the number of threads. It will return false if the number of threads entered is <1. (aka 0 or negative)

**int getNumberOfJobs()**

-Input Args:

* None

-Usage:

Returns the number of threads chosen by the user.

**bool setBlockSize(cv::Size size)**

-Input Args:

* size: This is of type cv::Size which is a type which represents the width and height dimensions of a matrix

-Usage:

This function returns true if it was able to set the private variable blockSplitonSize and returns false if the width and height of the input size was less than zero.

**bool setBlockSizeExp2Square(int exp)**

-Input Args:

* exp: This is the exponent.

-Usage:

This function computes 2^exp and scales the blockSplitonSize width and height to be 2^exp.

**cv::Size getBlockSize()**

-Input Args:

* None

-Usage:

This function returns the size of a block of type cv::Size which contains the width and height of a block.

Private:

**modelUtility()**

-Input Args:

* None

-Usage:

This is the modelUtility constructor and will initialize default number of jobs if not specified and block size if not altered by other functions.

Model CLASS

Private:

**bool loadModelFromJSONObject(jsonjumper::object& jsonObj)**

-Input Args:

* jsonObj: This is the loaded json object from the models

-Usage:

This function will load the json model and return true if possible and return false if it can’t.

**bool filterWorker(std::vector<cv::Mat> &inputPlanes, std::vector<cv::Mat> &weightMatrices, std::vector<cv::Mat> &outputPlanes, unsigned int beginningIndex, unsigned int nWorks)**

-Input Args:

* &inputPlanes: This is the vector of input planes
* &weightMatrices: This is the vector of weight matrices aka the kernel
* &outputPlanes: This is the vector of output planes
* beginningIndex, nWorks: These are two indices for the filter loop

-Usage:

This is the actual function which performs the filtering given the proper json input.

Public:

**Model(jsonjumper::object &jsonObj)**

-Input Args:

* &jsonObj: This is the input jsonObj after it was loaded

-Usage:

This is the Model class constructor.

**~Model()**

-Input Args:

* None

-Usage:

This is the Model class destructor.

**int getNInputPlanes()**

-Input Args:

* None

-Usage:

This returns the number of inputPlanes.

**int getNOutputPlanes()**

-Input Args:

* None

-Usage:

This returns the number of outputPlanes This returns the number of outputPlanes.

**bool filter(std::vector<cv::Mat> &inputPlanes,std::vector<cv::Mat> &outputPlanes std::vector<cv::Mat> &outputPlanes)**

-Input Args:

* &inputPlanes: This is the vector of matrix of input planes
* &outputPlanes: This is the vector matrix of output planes

-Usage:

This is the top level filter function which calls the filterWorker function. The filter function will also use multithreading to speed up the computation.

***convertThreading.hpp/cpp***

**bool convertWithModels(cv::Mat &inputPlanes, cv::Mat &outputPlanes, std::vector<std::unique\_ptr<Model> > &models, bool blockSplittion = true)**

-Input Args:

* &inputPlanes: This is of type cv::Mat and contains all the input planes
* &outputPlanes: This of type cv::Mat and contains all the output planes
* &models: This is a vector of unique\_ptrs to models.
* blockSpliton: This is set on by default to perform denoising on each block individually.

-Usage:

This is the top level function which calls the filter function, convertWithModelsFunda, and convertWithModelsBlockSplit. If the size of the output image is small enough and does not require splitting, convertWithModels will call convertwithModelsFuna once. If the output image is large enough that we need to denoise multiple blocks, call convertWithModelsBlockSplit. This function will return true if the denoising was successful and false if it was not.

**static bool convertWithModelsFunda(cv::Mat &inputPlane, cv::Mat &outputPlane, std::vector<std::unique\_ptr<Model> > &models)**

-Input Args:

* &inputPlane: This is of type cv::Mat and contains one input plane
* &outputPlane: This is of type cv::Mat and contains one output plane
* &models: This is a vector of unique\_ptrs to models

-Usage:

This is the function which loops through unique\_ptr and calls the filter function to denoise each plane. The number of iterations that converWithModelsFunda goes through is dependent on the the size of the model vector. This function will return true if the denoising was successful and false if it wasn’t.

**static bool convertWithModelsBlockSplit(cv::Mat &inputPlane, cv::Mat &outputPlane, std::vector<std::unique\_ptr<Model> > &models)**

-Input Args:

* &inputPlane: This is of type cv::Mat and contains one input plane
* &outputPlane: This is of type cv::Mat and contains one output plane
* &models: This is a vector of unique\_ptrs to models

-Usage:

This function is called if the output image is large enough to require block splitting. The number of iterations is calculated based on the output image size. It will loop through each column and each row once.

# rows: ceil(outputSize.width / (blockSize.width - 2 \* # Model)

# cols: ceil(outputSize.height / (blockSize.height - 2 \* # Model)

***SEGEMENTATION***

*Classes/Structs by File:*

***segment-graph.h***

edge struct:

Members:

* w: w is the weight on the edge
* a, b: a and b are representing the two nodes connected to this edge

***image.h***

image class:

Members:

* Public:
  + T \*data: 1D array to the RGB pixel
  + T \*\*access: 2D array to access a specific pixel
* Private:
  + w, h: Width and height of image

***misc.h***

RGB class:

Members:

* r, g, b: These are unsigned chars (0-255) representing the RGB values of a pixel

***disjoint-set.h***

uni\_elt struct:

Members:

* rank: In union by rank, rank represents the depth of a tree.
* p: This is the pixel number. Recall that pixels are stored in a 1D array. This is the index of the array
* size: Size of the disjoint-set forest (tree)

universe class:

Members:

* Public:
  + None
* Private:
  + \*elts: This is a pointer to all the leaves of the tree.
  + num: This is the number of nodes we have yet to join.

*Functions by File:*

***segment.h***

map<int, vector<pair<int, int> > > test(float sigma, float k, int min\_size, char \*file\_name, char \*out\_name, cv::Mat denoised)

-Input Args:

* Sigma: Argument used by smoothing function
* k: Tolerance value (for size of component)
* min\_size: Minimum size of a segment
* file\_name: Name of input file
* out\_name: Name of output file
* denoised: Denoised image input

-Usage:

This function will convert the input OpenCV YUV image into a simple RGB class so that the segmentation algorithm can be used without dealing with OpenCV matrices. Afterwards it will call the segment\_image function which will perform the segmentation.

-Return Value:

This function returns map<int, vector<pair<int, int> > > seg\_x\_y. This is a map with key: segment number, a vector<pair<int, int>> as the vector of x,y coordinates each segment contains.

***segment-image.h***

Defines:

* #define THRESHOLD(size, c) (c/size)

**map<int, vector<pair<int, int> > > split(image<rgb> \*orig, int \*\*indices, int num\_edge, int num\_seg, int width, int height)**

-Input Args:

* \*orig: This is a pointer to the original image input
* \*\*indices: This is a double pointer to the indices of the image
* num\_edges: This is the number of edges calculated by segment\_image
* num\_seg: This is the number of segments returned by segment\_graph
* width: This is the width of the image
* height: This is the height of the image

-Usage:

This function will create a map of vector pairs which will represent the (x,y) coordinates of each image. Call this function only after the image has already been segmented

**static inline float diff(image<float> \*r, image<float> \*g, image<float> \*b, int x1, int y1, int x2, int y2)**

-Input Args:

* \*r, \*g, \*b: These are pointers to an image containing only their respective RGB component values.
* x1, y1, x2, y2: These are the (x,y) coordinates of two pixels which you wish to calculate the dissimilarity

-Usage:

This function will calculate the dissimilarity of two pixels which is in turn the weight of each edge. Used inside segment\_image in the process of creating the graph

**map<int, vector<pair<int, int> > > segment\_image(image<rgb> \*im, float sigma, float c, int min\_size, int \*num\_ccs)**

-Input Args:

* \*im: The input image in the format of the image class defined in image.h
* sigma: The argument used to smooth the RGB channels
* c: Renamed to c this is the threshold argument used to control the sensitivity of the segmentation
* min\_size: The minimum size of each segment
* \*num\_ccs: A counter for the number of segments

***segment-graph.h***

**bool operator<(const edge &a, const edge &b)**

-Input Args:

* a, b: These are two edges passed by reference.

-Usage:

This is the operator overload for calculating whether an edge is “less” than another. Essentially this compares the weights of either edge and returns true if edge a<edge b.

**universe \*segment\_graph(int num\_vertices, int num\_edges, edge \*edges, float c)**

-Input Args:

* num\_vertices: Total number of nodes after creating graph
* num\_edges: Total number of edges after creating graph
* \*edges: This is a pointer to a dynamically allocated set of edges
* c: Renamed to c this is the threshold argument used to control the sensitivity of the segmentation

-Usage:

This function will segment the graph into separate sub graphs and return a variable of type disjoint-set forest universe class defined in disjoint-set.h.

***filter.h***

Defines:

* #define WIDTH 4.0
* #define MAKE\_FILTER(name, fun)
  + This creates the Gaussian filter

**static image<float> \*smooth(image<uchar> \*src, float sigma)**

-Input Args:

* \*src: This is a pointer to a specific single component RGB picture
* sigma: This is the input argument which adjusts the smoothing of the Gaussian filter (this was also used in Ncut)

-Usage:

Felenswalb and Huttenlocher's algorithm uses Gaussian filters (Gaussian blur) to smooth RGB components because they have a property of improving edge detection.Felenswalb and Huttenlocher's algorithm uses Gaussian filters (Gaussian blur) to smooth RGB components because they have a property of improving edge detection.

***convolve.h***

**static void convolve\_even(image<float> \*src, image<float> \*dst, std::vector<float> &mask)**

-Input Args:

* \*src: Source input image
* \*dst: Output image
* &mask: Mask is the approximation of the Gaussian filter defined in filter.h

-Usage

This is used to convolve the image with the specified filter. In this case this is the Gaussian filter used to help edge detection.

***image.h***

**image<T>::image(const int width, const int height, const bool init)**

-Input Args:

* width, height: These are the width and height of the image.
* init: Useless bool value which defaults to 1.

-Usage:

This is constructor of the image class. In our case we initialize it with type RGB defined in misc.h

**image<T>::~image()**

-Input Args:

* None

-Usage:

This is the destructor for the image class

**image<T> \*image<T>::copy() const**

-Input Args:

* None

-Usage:

This is the copy constructor for the image class.

***misc.h***

**inline bool operator==(const rgb &a, const rgb &b)**

-Input Args:

* &a, &b: Two RGB pixels passed by reference

-Usage:

This returns true if all 3 components of the pixel are equal.

**inline T abs(const T &x)**

-Input Args:

* &x: An input of type T passed by reference

-Usage:

Returns 1 if x is > 0 and -1 otherwise.

**inline T square(const T &x)**

-Input Args:

* &x: An input of type T passed by reference

-Usage:

Computes the square of the value of x and returns it.

**inline T bound(const T &x, const T &min, const T &max)**

-Input Args:

* &x, &min, &max: 3 variables of the same type. X is the value to be checked, min is the minimum value, max is the maximum value

-Usage:

Returns true if x is in the range of min and max and false otherwise.

***disjoint-set.h***

universe::universe(int elements)

-Input Args:

* elements: This is the number of nodes in an image.

-Usage:

This is the disjoint-set forest constructor.

universe::~universe()

-Input Args:

* None:

-Usage:

This is the disjoint-set forest destructor.

int universe::find(int x)

-Input Args:

* x: This is the index of a pixel in the 1D array.

-Usage:

This find function will find the root of the disjoint set tree. And then use path compression to assign the roots of all its predecessors.

void universe::join(int x, int y)

-Input Args:

* x, y: These are indexes of two pixels

-Usage:

This function will join the trees of each component. To determine which tree to attach to the root of the either, check the rank of both pixels. If the rank of x=y then connect y to x. Then we increment the rank of the new joined tree by one. Note: To determine which tree a pixel depends on, the “p” variable of the uni\_elts struct is assigned the same value of the root. This is a design choice to further improve the time complexity as now find returns in constant time. Allowing us to determine the tree in which a node belongs in quickly.