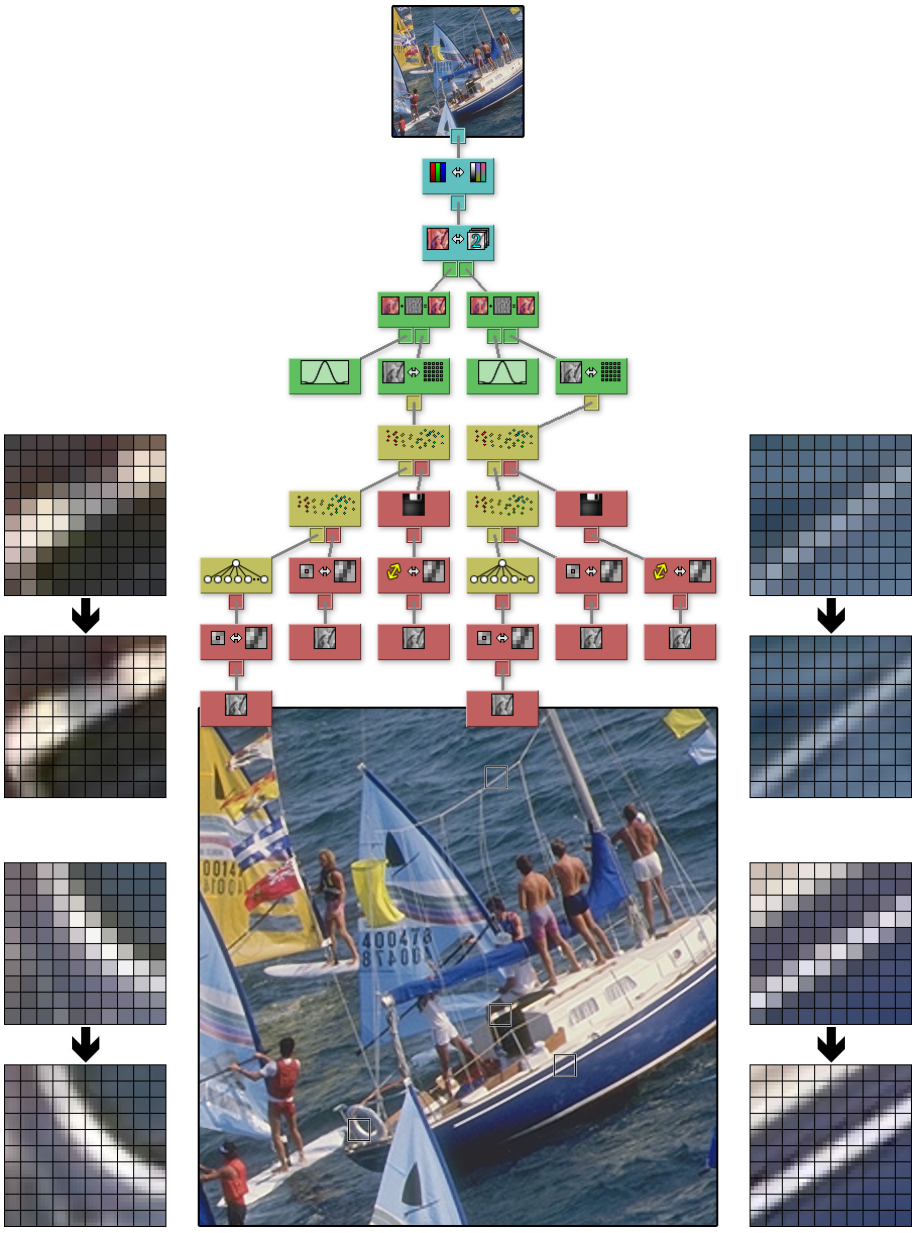
A Modular Approach to Image Super-Resolution Algorithms

J.D. van Ouwerkerk
Department of Media & Knowledge Engineering
Delft University of Technology
The Netherlands
E-mail: jos@jvojava.com



Abstract

This research designs and implements a tool that allows the combination of state of the art super-resolution algorithms in a flexible way to facilitate the search for high quality single frame super-resolution algorithms. The ability to combine algorithmic parts in a flexible way decreases the time needed to design new super-resolution algorithms.

Current state of the art super-resolution algorithms are researched and broken down into elemental parts. A modular approach is designed guided by these elemental parts and a proof of concept toolkit is implemented to perform testing. It offers the ability to design super-resolution algorithms without requiring any programming or scripting. Promising test results of a new super-resolution approach based on directional information are included.

Acknowledgements

I would like to send special thanks to my parents for supporting me in the many ways they did throughout my years of education. I would furthermore like to thank Léon Rothkrantz for his guidance and insight during my graduation period. Finally, I would like to thank C.B. Atkins, S. Battiato, F.M. Candocia, C. Staelin and F. Stanco for providing me with test material during my research phase.

Personal information

Name:	Joseph Daniel
Surname:	van Ouwerkerk
Study number:	9636365
University:	TU Delft, Delft, the Netherlands
Faculty:	Computer engineering
Department:	Media and knowledge engineering
Date:	July 6th, 2006

Graduate committee

- Dr. Drs. L.J.M. Rothkrantz
- C.A.P.G. van der Mast
- Ir. F. Ververs

Contents

1 Introduction

1.1 Background

Super-resolution is the process of generating a raster image with a higher resolution than its source. The source can consist of one or more raster images or frames. This thesis focuses on single-frame super-resolution, meaning that the source is a single raster image. Single-frame super-resolution is also known as image scaling, interpolation, zooming and enlargement.

Resampling of images to change size, resolution or orientation is common in all sorts of devices, like computers, television sets, mobile phones and digital camera's. Performing this resampling by using kernels is easy and fast, but not always optimal in terms of quality. Specific algorithms have been designed for scaling up, scaling down and rotating images that deliver higher quality results. This thesis focuses on the process of scaling up or super-resolution, which in contrast to scaling down or rotating deals with the problem of increasing the amount of pixels or, more general, data.

Super-resolution is among others applied to enlarge digital (consumer) photographs, increase printer resolution, convert PAL or NTSC resolution video to HDTV resolution video, improve the presentation of images and enhance photographs taken with low resolution equipment like mobile phones. Like resampling in general, super-resolution is a common process that is performed frequently by devices like digital camera's and computers.

The classical way of obtaining super-resolved images is by using kernel functions. The implementation of this approach is easy, but the results aren't always as good as one envisions. A kernel function is used to obtain a continuous approximation of the original signal from the known sampling points. This continuous function is then sampled at new positions to obtain a higher resolution image. Some common kernel functions and an example continuous approximation are shown in Figure 1.

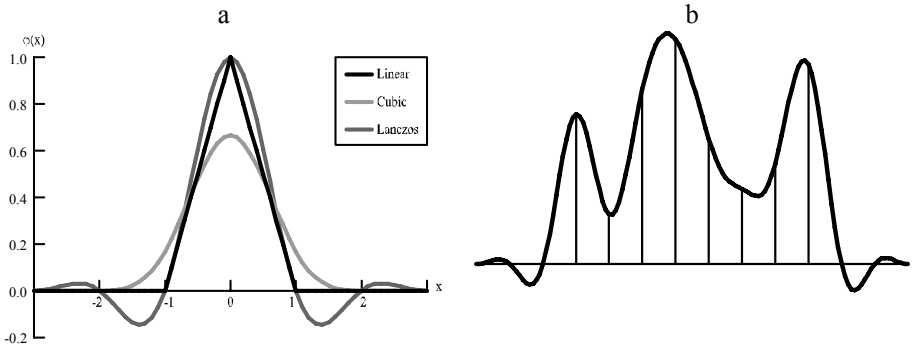


Figure 1 – kernel based resampling (a) three common kernels; (b) continuous approximation using a Lanczos kernel

Kernels used for super-resolution are typically linearly separable so the process can be split in two consecutive steps, horizontal and vertical, to reduce computation. 2-dimensional interpolation using a linear kernel is also known as bilinear interpolation and 2-dimensional interpolation using a cubic kernel is also known as bicubic interpolation. A super-resolution example based on kernels is shown in Figure 2. The scale factor is 4×4 and a grid is used as overlay showing which sixteen high resolution pixels come in the place of one low resolution pixel.

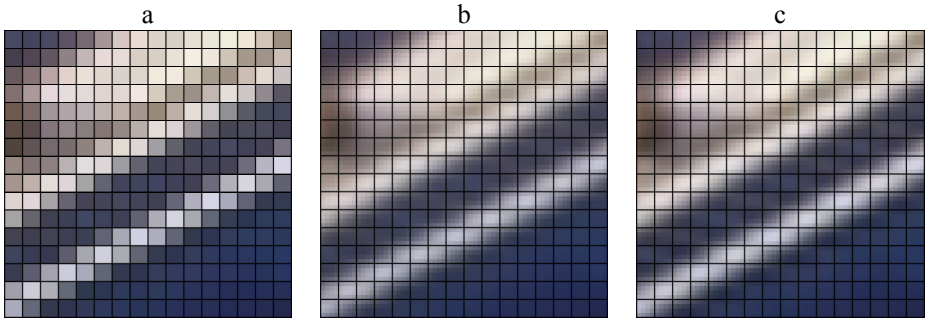


Figure 2 – kernel super-resolution results (a) original; (b) using a linear kernel; (c) using a cubic Catmull-Rom kernel

Some shortcomings of the kernel-based approach are apparent and can be attributed to the underlying model of the interpolated signal. They are listed in section 2. These shortcomings drive the study of improved super-resolution algorithms that address some or all of these problems trying to increase the quality of the super-resolved images.

The approaches taken in the past few years differ from each other greatly. Some use explicit models to represent natural images, while others use machine learning to obtain the relation between a source image and its super-resolved counterpart. Some use the concept of pixel classes to perform class specific interpolation for each pixel, while others use a single interpolator for every pixel in the image. Some use iterative belief propagation to gradually improve the super-resolved image, while others perform super-resolution in a single pass.

What they do have in common is that they take advantage of the strong relation between neighbouring pixels to estimate values of high resolution pixels. They also have in common that they treat the image as a 2-dimensional signal instead of separating the image into a range of 1-dimensional signals to take advantage of the 2-dimensional patterns occurring in natural images.

While approaches to super-resolution taken in the past few years sometimes differ greatly, some highly similar components are used in multiple algorithms. Most implementations are ad hoc, while some components from one algorithm could have been reused in others. This started the idea of a tool that allows the user to build a super-resolution algorithm from basic components. These components are mainly based on recent approaches to super-resolution.

1.2 Purpose

This research attempts to design and implement a tool that allows the combination of state of the art super-resolution algorithms in a flexible way to facilitate the

search for high quality single frame super-resolution algorithms. The ability to combine algorithmic parts in a flexible way should decrease the time needed to design new super-resolution algorithms.

To maintain flexibility, not all known state of the art super-resolution algorithms need to be included in the design of the prototype tool. It should however be attempted to incorporate as many algorithms as possible to improve the quality that can be achieved. The design should be flexible enough to allow future addition of other techniques. Operation speed should be considered, but it is not a key design factor.

The implementation of the tool should offer the user a way to design super-resolution algorithms interactively, without using any programming or scripting. Usage of the mouse should be available wherever this improves interactivity over the keyboard. The main purpose of the usage of colours in the interface should be to improve the recognition of interface elements; not to create a visual style. Since the training or super-resolution application process may take a considerable amount of time, feedback on the progress should be incorporated. Since training is often needed for super-resolution algorithms, it should be easy to select images for the purpose of training and to apply the training. The tool should allow for saving and loading super-resolution designs in order to improve the reuse of previously built designs.

1.3 Overview

To create a tool that allows the construction of super-resolution algorithms from basic components, some steps need to be taken. The first step is to research current state of the art algorithms and the techniques they use. The results of this research can be in found in section 2. The second step is focussed on breaking down the known algorithms into elemental parts and the design of the modular approach. It is described in section 3. Implementing a proof of concept tool is the third step and implementation details are presented in section 4. The proof of concept modular tool is used to explore new possibilities in super-resolution and the results of this exploration are presented in section 5. Section 6 concludes this thesis.

2 Previous Research

The shortcomings in commonly used kernel-based super-resolution drive the study of improved super-resolution algorithms of higher quality. In the past years a wide range of very different approaches has been taken to improve super-resolution.

This section compares approaches to high quality super-resolution by looking at theoretical backgrounds and practical results. Strengths and weaknesses are listed with the intent to spot chances for combination or improvement of techniques, thereby forming a base for future improved super-resolution algorithms.

The classical way of obtaining super-resolved images is by using kernels as described in subsection 2.1.1. The implementation of this approach is easy, but the results aren't always as good as one envisions. Some shortcomings of the kernel-based approach are apparent and can be attributed to the underlying model of the interpolated signal. These three shortcomings are identified next and will have a central role in the subjective evaluation.

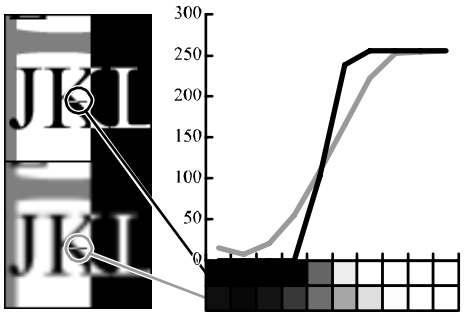


Figure 3 – original and result after decimation and bilinear super-resolution by one octave (factor of two)

A common problem with kernel super-resolution is the blurring of sharp edges. Kernel filters typically perform very well in smooth areas, but not in edge areas. Figure 3 shows a graphical image at the top left and at the bottom left the same image after decimation and super-resolution by one octave (a factor of two) using a linear kernel. The intensities of a ten pixel part are displayed in the graph on the right of the image. The darker line represents the original image, while the lighter line represents the super-resolved version. It is clear that the originally steep edge has become less steep in the super-resolved image, which is visible as edge blurring. The second problem is the introduction of blocking artifacts in diagonal edges or lines as shown in Figure 4.

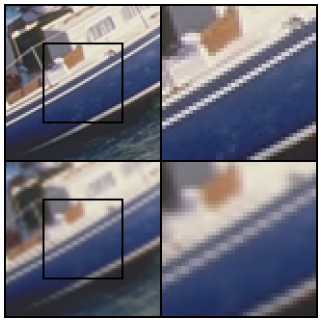


Figure 4 – original and result after decimation and bilinear super-resolution by one octave

The diagonal line in this image not only looks blurry, but a staircase pattern is also emerging in the bilinearly super-resolved image. This staircase pattern or blocking artifacts is caused by the horizontal and vertical orientation of the resampling kernels. Kernel super-resolution is unable to recognize or follow diagonal lines, which causes blocking.

The third problem is the inability to generate high frequency components or fine detail, as illustrated in Figure 5. This is needed to make the super-resolved image look more plausible. The original and super-resolved image are taken through low pass and high pass filters to show the higher deterioration in the high pass signal compared to the low pass signal.

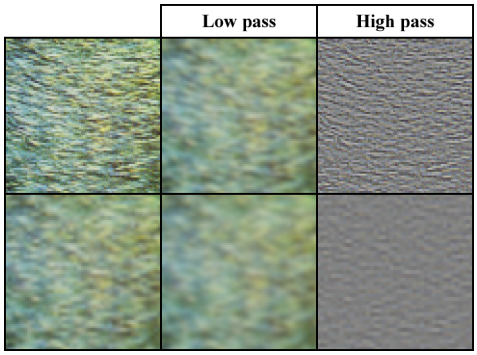


Figure 5 – original and result after decimation and bilinear super-resolution by one octave

These shortcomings have driven the study of improved super-resolution algorithms that address some or all of these problems trying to increase the quality of the super-resolved images. The approaches taken in the past few years sometimes differ from each other greatly. Some use explicit models to represent natural images, while others use machine learning to obtain the relation between a source image and its super-resolved counterpart. Some use the concept of pixel classes to perform class specific super-resolution for each pixel, while others use a single algorithm for every pixel in the image. Some use iterative belief propagation to gradually improve the super-resolved image, while others perform super-resolution in a single pass.

What they do have in common is that they take advantage of the strong relation between neighbouring pixels to estimate values of missing pixels. They also have in common that they treat the image as a 2-dimensional signal instead of separating the

image into a range of 1-dimensional signals to take advantage of the 2-dimensional patterns occurring in natural images.

In the past few years a whole range of super-resolution techniques and algorithms were introduced. This section aims to shine some light on these techniques and algorithms by describing them and comparing them by looking at theoretical backgrounds and practical results. The theoretical view aims at determining how each algorithm reduces blurring and blocking and increases detail. The test results are used to determine how effective blurring and blocking is reduced and detail is increased. Strengths and weaknesses of each algorithm are listed with intend to form a base for future improved super-resolution algorithms.

2.1 Algorithms

William T. Freeman et al. approached super-resolution from a low level vision learning perspective. An approach to low level vision tasks using belief propagation is presented in [14]. The scene underlying the supplied image data is estimated using a Markov network. To make the estimation feasible, both the image data and the scene are separated into patches. This approach to low level vision is specifically applied to super-resolution in [13]. It uses the high frequency part of the low resolution image as ground truth and the high frequency part of the high resolution image as scene to be estimated. A variant of this algorithm incorporating the distribution of pixel intensity derivatives is presented in [32]. A faster version of the algorithm that only uses a single pass is introduced in [15]. The belief propagation technique is described in subsection 2.1.7.

C.B. Atkins, C.A. Bouman and J.P. Allebach approached super-resolution using pixel classification. Pixel classification aims to sort pixels into classes like horizontal edges and smooth areas. A tree-based classification approach to super-resolution is introduced in [1]. This algorithm builds a decision tree with a linear interpolator at each leaf of the tree. Each non-leaf node in the tree represents a binary choice. In [2] a similar algorithm is introduced, which assigns a pixel to one or more classes in a single step instead of using a set of binary choices. The algorithm introduced in [2] is described in subsection 2.1.2 and included in the test results.

S. Battiato, G. Gallo and F. Stanco have published papers on several super-resolution algorithms. A rule based super-resolution approach called LAZA is described in [4]. In LAZA, the authors use simple rules and configurable thresholds to detect edges and update the interpolation process accordingly. In [6] the same authors introduce an algorithm that incorporates anisotropic diffusion (SIAD) to sharpen edges. The SIAD algorithm is described further in subsection 2.1.5, while the LAZA algorithm is described in subsection 2.1.8. Both LAZA and SIAD are included in the test results. In [5] S. Battiato et al. compare several super-resolution algorithms including LAZA and SIAD using the PSNR measure described in subsection 2.2.2.1.

D.D. Muresan and T.W. Parks published several papers [21, 22, 23, 24, 25] on super-resolution based on the optimal recovery principle. The authors model the image as belonging to a certain ellipsoidal signal class. Together with K. Kinebuchi a wavelet-based algorithm using hidden Markov trees was introduced in [17]. It uses lower frequency wavelet coefficients to predict the highest frequency coefficients.

By applying an inverse wavelet transform after prediction, a one octave super-resolved image results.

D. Su and P.J. Willis present super-resolution by triangulation on pixel level in [31]. X. Yu, B.S. Morse and T.W. Sederberg present super-resolution by Data-Dependant Triangulation (DDT) in [35]. These methods use linear interpolation that is not generally aligned with the coordinate axes to reduce visible artifacts caused by this alignment. It is shown by the last authors that results can be further improved by improving the algorithm that searches for the optimal triangulation of the source image.

Another approach that is explicitly directed at maintaining sharp edges is the use of Subpixel edge localization by K. Jensen and D. Anastassiou in [16]. This approach detects the most prominent edge in the local window with subpixel precision and uses the resulting edge template to obtain sharper edges in super-resolved images.

Other techniques include training a neural network for interpolation using a spatial error measure as proposed by C. Staelin et al. in [29] and described in subsection 2.1.3. X. Li and M.T. Orchard propose New Edge-Directed Interpolation (NEDI) in [20], which makes use of the geometric duality between the covariance in the low and high resolution images. This algorithm has been extended by D.D. Muresan and T.W. Parks as noted in [24] and is described in subsection 2.1.7. X. Xu et al. propose super-resolution using a combination of wavelet and fractal image models in [34].

There are also a range of commercial products available that rely on an algorithm more advanced than kernel-based resampling. I would like to mention PhotoZoom Professional by BenVista [36], Imagener by Kneson Software [37], Qimage by Digital Domain [38], Pictura by Digital Multi-Media Design [39] and SmartScale by Extensis [40]. The Pictura software makes use of a modified version of the algorithm presented in [25] by D.D. Muresan and T.W. Parks. The PhotoZoom Professional software was previously known as S-Spline by Shortcut, but has gone through some changes in name and company.

2.1.1 Kernel resize

The most common way of achieving super-resolution is using a base function or interpolation kernel (KERN). This algorithm uses the base function to approximate the continuous function underlying the discrete samples that make up the image. The continuous function is approximated by combining instances of the base or kernel function φ multiplied by the known discrete samples f_i for all pixel locations i in Z and is a linear operation [12] as shown in equation (1). Note that the equation in (1) is for the one-dimensional case.

$$f(x) = \sum_{i \in Z} f_i \cdot \varphi(x - i) \tag{1}$$

An example approximation using a Lanczos kernel with radius 3 is shown in Figure 6. The thin vertical lines denote the known discrete samples and the thick line denotes the continuous approximation.

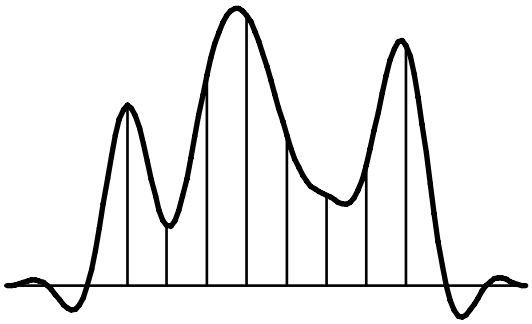


Figure 6 – continuous approximation using a Lanczos kernel

Typical choices of base functions include linear, cubic spline and Lanczos which are shown in Figure 7.

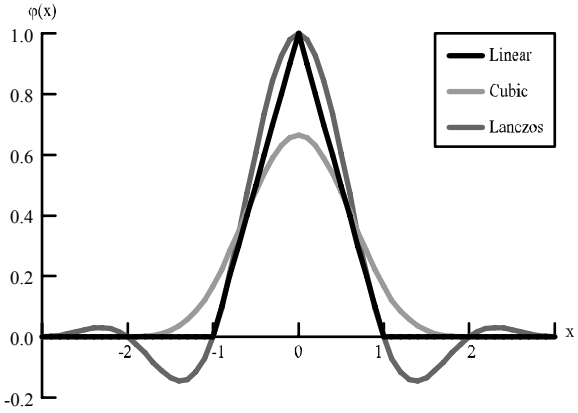


Figure 7 – typical super-resolution kernel functions

Since the used base functions are continuously valued functions, there is no limit on the scaling factor. Kernels used for super-resolution are typically linearly separable so the process can be split in two consecutive steps, horizontal and vertical, to reduce computation. This reduces the process to a single dimension instead of treating the data two dimensional.

Interpolation using a linear kernel is also known as bilinear interpolation and interpolation using a cubic kernel is also known as bicubic interpolation. Equation (2) denotes the one-dimensional Lanczos base function with radius r.

$$\varphi(x) = \begin{cases} \dfrac{r \cdot \sin(x\pi) \cdot \sin(x\pi / r)}{x^2 \pi^2} & ,|x| < r \\ 0 & ,|x| \geq r \end{cases} \tag{2}$$

2.1.2 Resolution Synthesis

Resolution Synthesis (RS) super-resolution as described in [2] is based on the assumption that there are different classes of pixels, like classes of pixels on object

edges with different orientations and the class of pixels in flat areas, that each require specific treatment when enlarged. Each of these classes can benefit from a dedicated super-resolution scheme to better preserve edges and generate details. The class of pixels on horizontal edges for example need to be treated in a way that doesn't blur the present horizontal edge. Clustering, also known as unsupervised learning, groups similar inputs into a number of classes or clusters.

In RS each pixel can be assigned to multiple classes with different degrees. A pixel can for example be classified to be 60% horizontal edge and 40% smooth. For each pixel and class the chance of membership is estimated using a Gaussian distribution. The centre of the Gaussian distribution is positioned at the centre of the pixel class. The chances are later normalised for each pixel to sum up to one.

The algorithm uses a specific linear filter for each class. A filter window containing the local neighbourhood of a low resolution pixel is constructed. This filter window is used as input for the linear filter to form the high resolution pixels.

A projection operator constructs a classification window from the filter window. This classification window is typically smaller and has the centre pixel subtracted. It is then scaled to accentuate edges. The classification window is used to determinate the degree of membership to each pixel class. The degree of membership to a class equals the chance of the low resolution pixel belonging to this class based on Gaussian distributions. These chances act as weights for the results of the linear filters for each class. The weighted average of the filter results is the final result.

Let n be the number of inputs into the linear filter and c the number of classes. Let I be the vector of size n containing the input values from the filter window, C the matrix of size $n \times c$ containing the filter coefficients for all classes and M the vector of size c containing the memberships to each class. The output value o is calculated using equation (3).

$$o = I^T \cdot (C \cdot M) \tag{3}$$

The RS algorithm is limited to an integer scaling factor. It attempts to use classification into edge and non-edge pixels with different orientations to limit edge blurring and blocking effects.

The implementation of RS in [2] uses eight difference values between the centre pixel and its direct neighbours for classification. The input of the linear filter is a 5×5 pixel window around the low resolution pixel. The number of classes used is 100. It is advised to use at least 100 000 training examples.

2.1.3 Spatial neural Network

Neural Network image scaling using Spatial Errors (NNSE) as described in [29] uses a single generalized feed forward neural network as interpolator.

The input of the neural network consists of the pixels in the local window around the source pixel and the output consists of the pixels needed for the super-resolved image. The neural network can be trained by using a decimated image as input and the corresponding original image as target.

The value of the centre pixel in the input window is subtracted from the inputs and is later added to the outputs. This input normalisation reduces the difference between cases to improve the training process, while staying general. The contrast in the input window is also normalised to further improve training.

While kernel resize is always linear, a multilayer neural network can be trained to recognize nonlinear relations between input and output. It is therefore able to better preserve edges and enhance detail than linear interpolators.

In the generalized feed forward network used, the input of each hidden node consists of the input nodes and all previous hidden nodes. The input for the output nodes consists of all hidden. The first input node is a bias node with a constant value of one.

The training error introduced into the neural network is a spatial error measure instead of the basic squared error. The spatial error measure combines the squared errors in a 3×3 local window. Let the errors in the 3×3 window be stacked column by column in vector V of size 9. A 9×9 matrix A is designed to accentuate errors on edges and lines. The spatial error e is calculated using equation (4).

$$e = V^T \cdot A \cdot V \tag{4}$$

Matrix A is composed of the nine 3×3 kernels shown in Figure 8. The first kernel is an average kernel, the next four are edge detection kernels and the last four are line detection kernels. Each kernel in Figure 8 is normalised and weighted before forming a row in matrix A.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$
$$\begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix} \begin{bmatrix} -1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix}$$
$$\begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix} \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix} \begin{bmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix}$$

Figure 8 – nine 3×3 spatial error kernels

The implementation of NNSE in [29] uses a feed forward network with 30 hidden nodes, a tanh activation function and a 5×5 pixel input window. The centre pixel is subtracted from the window and the contrast in the window is scaled to accentuate edges.

2.1.4 Local correlation

Local Correlation super-resolution (LCSR) as described in [9] is a two step algorithm. The first step is a classification step and the second step consists of assigning a Local Associative Memory (LAM) to each class of pixels. Each LAM is trained using a single step or gradient descent depending on the number of layers in the LAM.

LCSR is very similar to RS as described in 2.1.2. The main difference is that in LCSR each pixel belongs to a single class during super-resolution, while in RS each pixel can belong to multiple classes. LCSR uses a local input window like RS, which is used as input to the LAM and the classification.

The training exists of two steps. In the first step pixel classification is performed using a Kohonen Self-Organising Map (SOM) using the local window as input. The

second step consists of training each LAM for the pixels that fall into the corresponding class.

The implementation of LCSR in [9] uses a 3×3 or 5×5 input window and up to 30 classes. The mean of the input window is normalised to zero.

2.1.5 Anisotropic diffusion

Smart Interpolation by Anisotropic Diffusion (SIAD) as described in [6] uses anisotropic diffusion to sharpen edges and generate plausible detail. Anisotropic diffusion lets pixel intensity values diffuse over neighbours. The diffusion at a point is inversely proportional to the local contrast to enhance edges. Anisotropic diffusion is able to estimate a piecewise smooth image from a noisy source as noted in [8].

To reduce artefacts and aliasing caused by the anisotropic diffusion, the image is reduced in size afterwards. This requires the image to be enlarged beyond the needed size before the anisotropic diffusion step.

Super-resolution using SIAD is a three step process. The first step consists of enlarging the image beyond the required resolution using kernel resize. The enlargement factor used in [6] is eight. This will create a large, but blurry image. The second step consists of performing the anisotropic diffusion. This will sharpen the blurry edges. The third and final step consists of reducing the image by averaging pixels. This is done with a factor four to obtain an image super-resolved by one octave.

2.1.6 Sparse derivative prior

Super-resolution exploiting the Sparse Derivative Prior (SDPSR) as described in [32] uses the distribution of the derivative of natural images as model. This distribution of this derivative or differences between neighbouring pixels is sharply peaked at zero for natural images and is modelled with a generalized Laplace distribution shown in equation (5). The graph of this function for the range –100 to 100 and $s = 20$ for different values of α is shown in Figure 9.

$$p(x) = \exp(-|x|^{\alpha} / s) \tag{5}$$

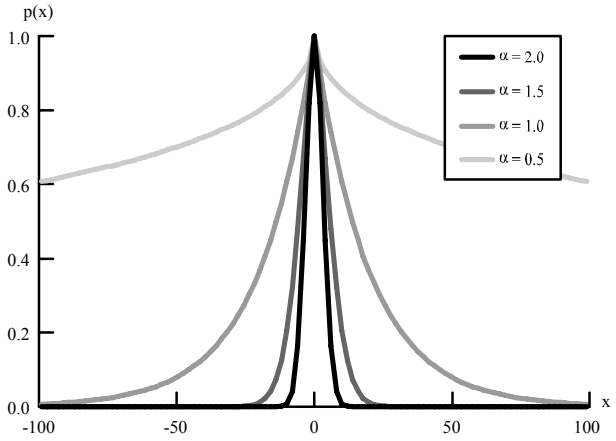


Figure 9 – generalized Laplace distribution for different values of α

Every low resolution neighbourhood is interpolated by the same set of interpolators to form a set of high resolution patches for each location. The selection of patches is updated iteratively maximizing the chance of the selection using Belief Propagation (BP). The choice of a patch is influenced by the low resolution image and the choice of neighbouring patches, creating a wider spatial dependency of high resolution pixels with each iteration. The BP in SDPSR uses two types of constraints, a generalized Laplace distribution for differences between neighbouring pixels and a Gaussian distribution for the differences between the low resolution pixel and the corresponding high resolution patch. While it is not guaranteed that belief propagation containing loops will converge, it normally does when applied to natural images. Satisfactory results have been obtained in any case.

The implementation of SDPSR in [32] trains a set of interpolators by clustering low resolution pixel neighbourhoods into 16 classes and calculating the RMS-optimal linear interpolator for each class. During super-resolution each of these interpolators is applied to each low resolution pixel and the optimal one is selected by BP afterwards. About 500 000 training vector pairs were used for training. The BP needed five iterations in the experiments done in [32].

2.1.7 Edge-Directed

New Edge-Directed Interpolation (NEDI) as described in [20] uses the duality between the low resolution and high resolution covariance for super-resolution. The covariance between neighbouring pixels in a local window around the low resolution source is used to estimate the covariance between neighbouring pixels in the high resolution target. An example covariance problem is represented in Figure 10.

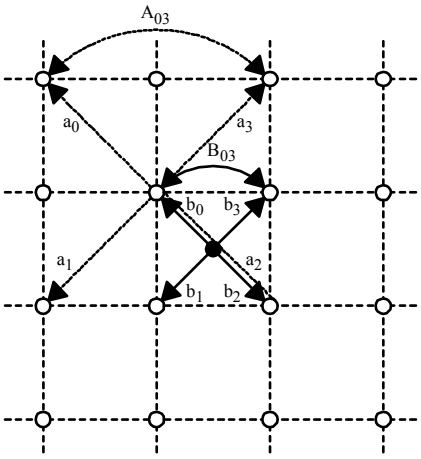


Figure 10 – local covariance

The covariance of the b_0 relation in Figure is estimated by the covariance of neighbouring a_0 relations in the local window. The open circles in the figure represent the low resolution pixels and the closed circle represents a high resolution pixel to be estimated.

The covariance used is that between pixels and their four diagonal neighbours. The covariance between low resolution pixels and their four diagonals in a m×m local window is calculated. This covariance determines the optimal way of blending the four diagonals into the centre pixel. This optimal value in the low resolution window is used to blend a new pixel in the super-resolution image. By using the local covariance, the interpolation can adhere to arbitrarily oriented edges to reduce edge blurring and blocking.

Let I be a vector of size four containing the diagonal neighbours of the target pixel o, X a vector of size m^2 containing the pixels in the $m \times m$ window and C a $4 \times m^2$ matrix containing the diagonal neighbours of the pixels in X. The equation for enlargement using this method is shown in (6).

$$o = \left(C^T \cdot C\right)^{-1} \left(C^T \cdot X\right) \cdot I^T \tag{6}$$

The NEDI algorithm uses two passes to determine all high resolution pixels. The first pass uses the diagonal neighbours to interpolate the high resolution pixels with both coordinates odd and the second pass uses the horizontal and vertical neighbours to interpolate the rest of the high resolution pixels as illustrated in Figure 11.

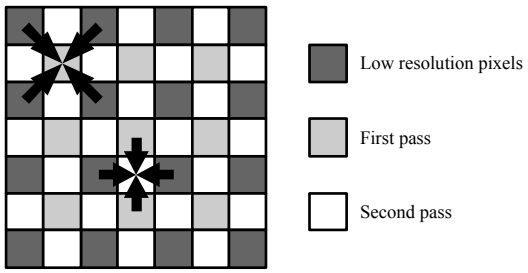


Figure 11 – two steps in NEDI

2.1.8 Locally-Adaptive

The Locally-Adaptive Zooming Algorithm (LAZA) introduced in [4] uses a set of simple rules to take information about discontinuities or sharp luminance variations into account while doubling the horizontal and vertical resolution of the input image. The base scaling factor of the LAZA algorithm is thereby 2×2, but by repeating the process, any power of two scaling factor can be obtained.

The algorithm is performed in four steps. All steps except the last one can leave pixel values undefined. In the case that they are left undefined, they will be set in a later step. The first step simply spreads out the pixel values to locations with two odd coordinates as is shown in the two leftmost steps of Figure (b). The second and third steps set the values of undefined pixels that are detected to have a "simple" spatial dependence with the neighbouring pixels. The membership to a spatial dependency is determined by a simple rule that uses surrounding pixel values and two threshold values. The second step recognizes five spatial dependencies, while the third step recognizes four spatial dependencies. After the first three steps, some pixels might still be undefined. All undefined pixels will therefore be set in the fourth and final step. In this final step, four surrounding pixels of the undefined pixel are combined to form the value of this pixel. To preserve more detail, the four pixel values are not simply averaged to form the new value. The value spectrum is instead divided into a number of bins. The four values of the surrounding pixels are placed into these bins and the values of the bins with at least one pixel value in them are averaged. The median of the values in the bin is usually taken as the value of the bin.

2.1.9 Overview

There are some things that all or most algorithms have in common. One important one is that they always use a local low resolution window as input. This local window often has a size of 3×3 or 5×5. The centre pixel is often subtracted from all values in the window (offset) and in some cases the contrast in the window is scaled to accentuate edges (scale). The used normalizations are listed in Table 1.

Most algorithms are limited to an integer scaling factor; unlike kernel resize which has no limit on the scaling factor. NEDI and LAZA are limited to a scaling factor of two. A limit on the scaling factor does not mean that other scaling factors can not be obtained. An algorithm can be repeated until the resulting image is at least as large as the required size. The resulting image can then be scaled down to

the required size using kernel resize for example. Limits on scaling factors are listed in Table 1.

Each algorithm either divides a low resolution pixel into several high resolution pixels or generates extra pixels between low resolution pixels. The difference in these two approaches is shown in Figure 12 for super-resolution by one octave.

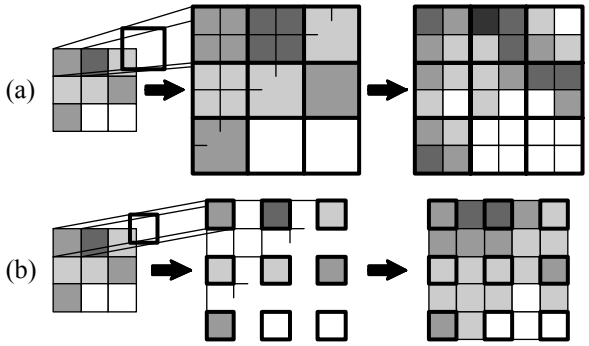


Figure 12 – (a) even parity; (b) odd parity

Figure (a) demonstrates super-resolution with even parity enlarging an n×m image to a 2n×2m image. Every low resolution pixel is divided into four high resolution pixels. Figure (b) demonstrates super-resolution with odd parity enlarging a n×m image to a (2n - 1) × (2m - 1) image. The low resolution pixels are first moved apart and pixels are added in between them to form the high resolution image. The term super-resolution parity is chosen, because it coincides with the parity of the number of high resolution pixels after super-resolution by one octave.

Four algorithms have to be trained by 'showing' example images. These four are RS, NNSE, LCSR and SDPSR. Training is used both supervised to train interpolators and unsupervised to perform clustering.

Table 1 – algorithm overview

Paragraph	*Code*	*Normalisation*	*Scale factor*	*Parity*	*Trained*
2.1	KERN	None	Any	Any	No
2.2	RS	Offset	Integer	Even	Yes
2.3	NNSE	Offset+scale	Integer	Even	Yes
2.4	LCSR	Offset	Integer	Even	Yes
2.5	SIAD	None	Integer	Any	No
2.6	SDPSR	None	Integer	Odd	Yes
2.7	NEDI	None	2	Odd	No
2.8	LAZA	None	2	Odd	No

2.2 Test setup

Both objective and subjective tests are performed. Both tests make use of the same set of test images. The objective test will make use of several objective measures, while the subjective test is aimed at performance in edge blurring, edge blocking and generation of detail.

Error measures are used to objectively compare a super-resolution image to the original one as shown in Figure 13. The original image is first decimated by factor f and then super-resolved with factor f. The original and super-resolved images are compared using an error measure.

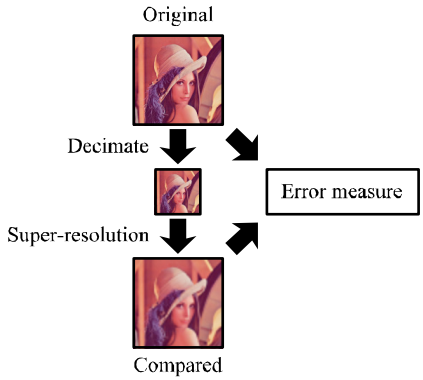


Figure 13 – error measure layout

The decimation used in this section consists of applying a block or zero order filter followed by a down sampling step. For even decimation by one octave, this comes down to taking the average of four high resolution pixels for each low resolution pixel.

2.2.1 Test images

The test set consists of the seven images with an original resolution of 512×512 shown in Figure 14. The images have been selected to test on edge blurring, edge blocking and generation of detail.

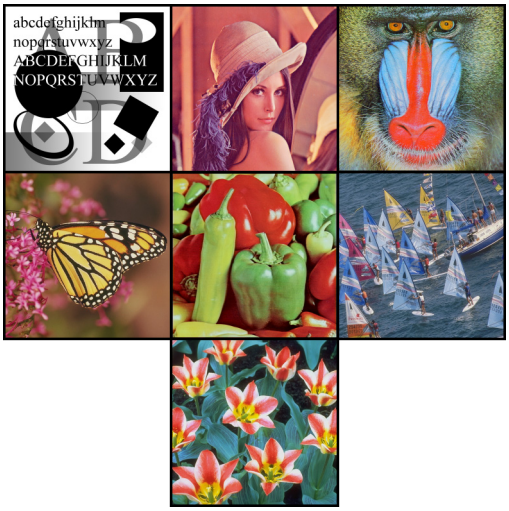


Figure 14 – test set images

The names of the images in Figure from left to right and top to bottom are graphic, lena, mandrill, monarch, peppers, sail and tulips. The graphic image has been made for this test. The other images are royalty free and have been used in other computer graphics performance tests. Some, like lena and peppers, are widely used for performance tests in computer graphics.

The graphic and monarch images exhibit strong and sharp edges in varying directions, making them prime images to test edge blurring and edge blocking effects. The mandrill image contains a lot of detail in the hairs, making it a prime image to test detail generation. A sharp diagonal white line is visible on the side of the boat in the right of the sail image. This makes the super-resolved sail image prone to display edge blocking.

The first derivative of an image gives some insight in the amount of detail present in the image. Part of the histograms of the horizontal first derivative of the seven test images are shown in Figure 15. The images are first converted to greyscale with equation (7). The greyscale intensity value y is assumed to be a linear combination of the red, green and blue intensity values r, g and b.

$$y = 0.299r + 0.587g + 0.114b \tag{7}$$

Then each difference d in greyscale intensities y of two horizontally neighbouring pixels is calculated. These differences are grouped into bins that are 0.02 wide. The counts of elements in each bin are shown in Figure after normalisation for each image.

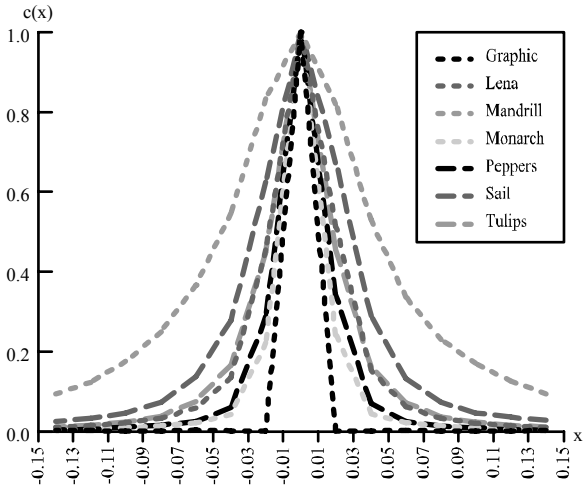


Figure 15 – horizontal first derivative histogram

The derivative histograms are sparse as noted and used explicitly for super-resolution in [32]. It can be seen that the mandrill image has the least sparse derivative histogram, since it contains the most detailed texture. The graphic image has the sparsest derivative histogram, since it doesn't contain any texturing.

2.2.2 Error measures

2.2.2.1 Peak signal to Noise Ratio

The Peak Signal to Noise Ratio (PSNR) as used in various image quality assessments is very common and based on the Mean Squared Error (MSE).

The MSE is simply the mean of the squared differences for every channel for every pixel. Letting x_{ic} denote the value of pixel i in channel c of the original image, y_{ic} the value of pixel i in channel c of the compared image, n the number of pixels and m the number of channels, the MSE can be obtained using equation (8).

$$MSE = \frac{1}{nm}\sum_{i=1}^{n}\sum_{c=1}^{m}(x_{ic} - y_{ic})^2 \tag{8}$$

The PSNR can be obtained from the MSE and the maximum signal value s using equation (9).

$$PSNR = 10\log_{10}(s^2 / MSE) \tag{9}$$

The PSNR is expressed in decibels (dB) and a higher value corresponds to a lower error and thus a higher quality.

2.2.2.2 Structural Similarity

The mean square based error measures as described in subsection 2.2.2.1 are most widely used, but don't have a high correlation with the visual degradation in quality as noted by several authors [3, 22, 29, 32]. In [33] Z. Wang et al. describe and test the Mean Structural SIMilarity (MSSIM) error measure. They conclude it to have a higher correlation with the visual degradation than the Mean Squared Error (MSE), while not being a very complex measure. The Structural SIMilarity (SSIM) error measure calculates the similarity in a local window by combining differences in average and variation and correlation. It is described in detail in [33].

The SIMM measure starts with two sets of n intensity values from linked windows in the original and compared image. The averages μ_x and μ_y are calculated using equation (10).

$$\mu_x = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{10}$$

Next, the variances σ_x and σ_y of the two sets of intensity values are calculated using equation (11).

$$\sigma_x = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu_x)^2} \tag{11}$$

Finally the correlation between the two sets of intensity values, σ_{xy}, is calculated using equation (12).

$$\sigma_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu_x)(y_i - \mu_y) \tag{12}$$

The averages, variances and correlation are used to obtain the SSIM index with equation (13).

$$SSIM_{xy} = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{13}$$

The (small) constants c_1 and c_2 guarantee that the denominator can't be equal to zero. The SSIM quality measure used in this section has a window size of 8×8 and parameters c_1 and c_2 set to 0.0001 and 0.0009 respectively.

The local similarity measures are averaged over all possible window offsets and all channels to obtain the Mean Structural SIMilarity (MSSIM), a similarity measure for the whole image. The value of the MSSIM lies between 0 and 1 and a higher value denotes a higher structural similarity and thus a higher quality.

2.2.2.3 Edge stability

A range of error measures has been tested in [3]. Edge stability is mentioned as being the most sensitive to the whole set of distortions and being sensitive to a blurring distortion. Since the most common distortion in super-resolution images is blurring, edge stability is chosen here as an appropriate error measure.

The edge stability error measure uses five Canny edge detectors [10] with different blur deviations to obtain an ordered set of five edge maps. The Canny edge detector used starts with a Gaussian blur with the specified deviation. The horizontal and vertical Sobel kernels as shown in Figure 16 are applied to the blurred image.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 16 – Sobel kernels

The two kernel filtered images are combined to form an edge intensity and direction map. This map is thinned and a threshold set to 0.9 times the minimum value plus 0.1 times the maximum value is applied. The edge maps are combined to a consecutive edge map by counting the maximum number of consecutive occurrences of an edge at each pixel in the ordered set of edge maps as visualized in Figure 17. Note that all edge maps in Figure 17 are inverted and normalised for visibility.

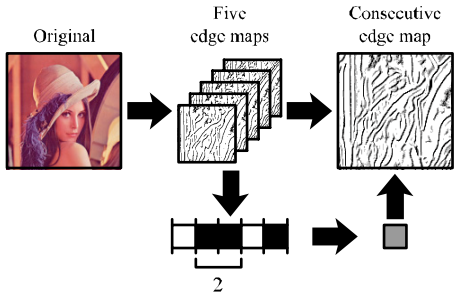


Figure 17 – consecutive edge map

The resulting consecutive edge map contains numbers ranging from 0 to 5. The edge stability mean squared error (ESMSE) between the reference and compared image is calculated with equation (14), where *ex* is the original consecutive edge map, *ey* is the compared consecutive edge map and *n* is equal to the number of edges that are detected in at least one of these two consecutive edge maps.

$$ESMSE = \frac{1}{n}\sum_{i=1}^{n}(ex_i - ey_i)^2 \tag{14}$$

The edge stability (ESMSE) quality measure used in this section uses blur deviations 1.19, 1.44, 1.68, 2.0 and 2.38. Lower values of the ESMSE denote higher edge stability and thus a higher quality.

2.3 Results

Several super-resolution algorithms are compared using PSNR, MSSIM and ESMSE error measures, which are described in the previous section. Because of the differences in super-resolution with even and odd parity, the objective results are split into these two parities. For reference linear kernel resize (LIN) and Lanczos kernel resize with radius 3 (LANC) are included in both the even and odd parity tests. A toolkit implemented for this research was used to among others calculate objective quality measures and perform kernel super-resolution.

Super-resolution results from the RS and NNSE algorithms have been kindly provided by C.B. Atkins and C. Staelin respectively. Matlab code for the LCSR and SIAD algorithms has been kindly provided by F.M. Candocia and F. Stanco respectively. The SIAD algorithm is performed using the toolkit mentioned earlier for resizing and the provided Matlab code for anisotropic diffusion. The SIAD algorithm is included in the test using both a linear kernel (SIADLIN) and a Lanczos kernel (SIADLANC) for initial enlargement. The NEDI algorithm has been implemented in the toolkit and is applied using an 8×8 window. The LAZA algorithm was performed using an online applet mentioned by S. Battiato.

PSNR results for even super-resolution are shown in Table 2. MSSIM results for even super-resolution are shown in Table 3. ESMSE results for even super-resolution are shown in Table 4. The results for odd super-resolution are shown in Table 5, Table 6 and Table 7. The best result in each row is marked with bold face.

Table 2 – PSNR results for even super-resolution in dB, higher is better

128 to 256	*LIN*	*LANC*	*RS*	*NNSE*	*LCSR*	*SIADLIN*	*SIADLANC*
Graphic	19.67	21.29	22.77	**23.23**	21.91	20.08	22.18
Lena	29.81	31.26	**32.39**	31.93	31.93	30.47	31.68
Mandrill	23.81	24.29	**24.49**	24.29	24.41	23.89	24.33
Monarch	25.45	27.28	**28.81**	28.20	28.24	26.11	28.17
Peppers	28.75	30.27	**31.03**	30.14	30.14	29.29	30.53
Sail	25.64	26.52	**27.20**	27.12	26.98	25.92	26.82
Tulips	26.79	28.63	**30.29**	28.87	29.63	27.73	29.50
128 to 512							
Graphic	17.94	18.98	**20.19**	19.43	19.55	18.07	19.44
Lena	27.86	28.70	**29.57**	28.85	29.08	27.98	28.92
Mandrill	20.40	20.60	**20.71**	20.58	20.63	20.41	20.63
Monarch	23.91	25.11	**26.41**	25.22	25.90	24.08	25.63
Peppers	25.31	26.02	**26.26**	25.72	25.66	25.37	26.15
Sail	23.54	24.03	**24.63**	24.08	24.31	23.58	24.16
Tulips	25.43	26.69	**28.19**	26.14	27.56	25.67	27.24
256 to 512							
Graphic	22.48	24.15	26.69	**27.38**	25.77	23.25	25.61
Lena	31.89	33.13	**34.13**	33.76	33.67	32.33	33.20
Mandrill	22.63	23.24	**23.47**	23.38	23.37	22.70	23.31
Monarch	28.75	30.75	**33.22**	32.89	32.45	29.85	31.83
Peppers	28.44	**29.38**	29.07	28.66	28.41	28.56	29.18
Sail	27.20	28.61	29.67	**29.76**	29.25	27.62	29.04
Tulips	30.77	33.11	**35.53**	34.10	34.40	31.95	33.85

Table 3 – MSSIM results for even super-resolution, higher is better

128 to 256	*LIN*	*LANC*	*RS*	*NNSE*	*LCSR*	*SIADLIN*	*SIADLANC*
Graphic	0.850	0.891	0.929	**0.939**	0.920	0.870	0.916
Lena	0.904	0.930	**0.940**	0.932	0.937	0.912	0.935
Mandrill	0.701	0.756	**0.779**	0.768	0.772	0.709	0.768
Monarch	0.920	0.947	**0.960**	0.953	0.956	0.930	0.955
Peppers	0.926	0.946	**0.954**	0.941	0.951	0.933	0.950
Sail	0.780	0.828	0.854	**0.854**	0.845	0.791	0.843
Tulips	0.889	0.921	**0.935**	0.912	0.932	0.903	0.930
128 to 512							
Graphic	0.775	0.800	**0.864**	0.798	0.854	0.784	0.823
Lena	0.778	0.805	**0.821**	0.805	0.810	0.781	0.810
Mandrill	0.459	0.502	**0.536**	0.519	0.522	0.460	0.509
Monarch	0.848	0.873	**0.896**	0.865	0.889	0.852	0.882
Peppers	0.838	0.859	**0.873**	0.840	0.864	0.840	0.864
Sail	0.586	0.633	**0.679**	0.653	0.657	0.590	0.642
Tulips	0.779	0.812	**0.843**	0.785	0.831	0.784	0.822
256 to 512							
Graphic	0.916	0.934	0.963	**0.969**	0.960	0.932	0.953
Lena	0.879	0.900	**0.907**	0.904	0.906	0.884	0.902
Mandrill	0.696	0.758	**0.784**	0.778	0.777	0.704	0.773
Monarch	0.940	0.957	**0.967**	0.964	0.964	0.948	0.962
Peppers	0.926	0.940	**0.945**	0.939	0.943	0.929	0.941
Sail	0.820	0.874	0.899	**0.903**	0.888	0.834	0.888
Tulips	0.923	0.952	**0.962**	0.951	0.957	0.933	0.957

Table 4 – ESMSE results for even super-resolution, lower is better

128 to 256	*LIN*	*LANC*	*RS*	*NNSE*	*LCSR*	*SIADLIN*	*SIADLANC*
Graphic	1.331	1.504	1.226	**1.171**	1.189	1.276	1.617
Lena	1.776	1.408	**1.321**	1.331	1.485	1.769	1.377
Mandrill	2.147	1.816	**1.791**	1.795	1.921	2.125	1.860
Monarch	2.299	1.926	**1.845**	1.866	1.952	2.346	1.948
Peppers	1.807	1.391	**1.327**	1.362	1.466	1.753	1.432
Sail	1.898	1.584	1.514	**1.465**	1.621	1.882	1.626
Tulips	1.631	1.198	**1.146**	1.167	1.272	1.552	1.225
128 to 512							
Graphic	3.309	4.689	**2.998**	4.175	3.098	3.176	4.519
Lena	5.480	4.908	4.718	4.840	**4.706**	5.461	4.915
Mandrill	6.609	6.538	6.301	6.472	**6.278**	6.563	6.503
Monarch	5.448	4.850	**4.518**	4.783	4.606	5.408	4.854
Peppers	5.531	5.081	4.905	5.057	**4.864**	5.522	5.079
Sail	6.211	6.230	**5.776**	6.136	5.808	6.172	6.166
Tulips	5.994	5.664	**5.198**	5.627	5.286	5.927	5.590

256 to 512							
Graphic	1.059	1.367	1.085	0.981	**0.974**	1.045	1.360
Lena	1.971	1.679	**1.632**	1.645	1.728	1.964	1.686
Mandrill	2.067	1.750	**1.705**	1.723	1.813	2.097	1.777
Monarch	2.298	1.952	**1.884**	1.894	1.964	2.442	1.977
Peppers	1.942	1.575	**1.524**	1.525	1.641	1.959	1.616
Sail	1.656	1.237	1.151	**1.128**	1.282	1.613	1.277
Tulips	1.505	1.074	0.973	**0.963**	1.151	1.472	1.104

Table 5 – PSNR results for odd super-resolution, higher is better

128 to 255	*LIN*	*LANC*	*NEDI*	*LAZA*
Graphic	21.57	**22.92**	21.23	20.92
Lena	31.67	**32.87**	32.45	30.68
Mandrill	26.13	**26.47**	26.16	24.97
Monarch	27.13	**28.74**	27.18	26.27
Peppers	30.47	**31.53**	30.88	29.05
Sail	27.47	**28.18**	27.67	26.63
Tulips	28.50	**30.07**	29.05	26.82
128 to 509				
Graphic	18.08	**19.00**	17.83	17.59
Lena	27.86	**28.66**	28.44	27.02
Mandrill	20.56	**20.76**	20.60	19.95
Monarch	23.82	**24.95**	24.03	23.06
Peppers	27.46	**28.27**	27.90	26.11
Sail	23.47	**23.95**	23.67	22.84
Tulips	25.43	**26.61**	26.00	23.87
256 to 511				
Graphic	22.53	**23.59**	22.41	22.24
Lena	31.81	**32.59**	32.25	31.08
Mandrill	22.81	**23.21**	22.90	22.17
Monarch	28.60	**29.94**	29.10	28.35
Peppers	32.02	**32.67**	32.30	31.14
Sail	27.14	**28.04**	27.56	26.69
Tulips	30.71	**32.16**	31.57	29.62

Table 6 – MSSIM results for odd super-resolution, higher is better

128 to 255	*LIN*	*LANC*	*NEDI*	*LAZA*
Graphic	0.898	**0.915**	0.884	0.894
Lena	0.936	**0.947**	0.939	0.915
Mandrill	0.793	**0.818**	0.791	0.730
Monarch	0.944	**0.958**	0.943	0.923
Peppers	0.950	**0.959**	0.951	0.921
Sail	0.846	**0.870**	0.853	0.816
Tulips	0.926	**0.942**	0.928	0.878

128 to 509				
Graphic	0.780	**0.801**	0.761	0.780
Lena	0.777	**0.802**	0.788	0.749
Mandrill	0.459	**0.498**	0.463	0.390
Monarch	0.846	**0.870**	0.848	0.814
Peppers	0.845	**0.865**	0.851	0.802
Sail	0.582	**0.625**	0.596	0.542
Tulips	0.779	**0.811**	0.787	0.717
256 to 511				
Graphic	0.917	**0.926**	0.908	0.918
Lena	0.879	**0.889**	0.879	0.861
Mandrill	0.697	**0.732**	0.699	0.643
Monarch	0.938	**0.948**	0.941	0.930
Peppers	0.931	**0.938**	0.930	0.913
Sail	0.818	**0.852**	0.833	0.799
Tulips	0.920	**0.938**	0.926	0.894

Table 7 – ESMSE results for odd super-resolution, lower is better

128 to 255	*LIN*	*LANC*	*NEDI*	*LAZA*
Graphic	**1.032**	1.402	1.454	1.222
Lena	1.513	**1.312**	1.735	2.096
Mandrill	1.751	**1.604**	1.926	2.816
Monarch	2.035	**1.816**	2.416	2.946
Peppers	1.430	**1.225**	1.783	2.179
Sail	1.521	**1.338**	1.799	2.391
Tulips	1.273	**1.053**	1.565	2.277
128 to 509				
Graphic	3.258	4.602	4.552	**3.256**
Lena	5.476	**4.806**	5.365	5.661
Mandrill	6.443	6.366	**6.289**	6.999
Monarch	5.386	**4.771**	5.493	5.494
Peppers	5.339	**4.839**	5.462	5.601
Sail	6.135	6.128	**5.959**	6.642
Tulips	5.830	**5.490**	5.572	5.966
256 to 511				
Graphic	**1.030**	1.316	1.196	1.107
Lena	1.706	**1.445**	1.899	2.250
Mandrill	1.553	**1.242**	1.871	2.663
Monarch	2.074	**1.814**	2.432	2.880
Peppers	1.690	**1.433**	2.091	2.323
Sail	1.440	**1.101**	1.700	2.244
Tulips	1.402	**1.082**	1.717	2.098

If we sum the above results over all seven images and all three resolutions, we get a general result for each algorithm and error measure. If we order this list from best result down, we get the list shown in Table 8 for the even tests and the list shown in Table 9 for the odd tests.

Table 8 – even combined results

#	*PSNR*	*MSSIM*	*ESMSE*
1	RS	RS	RS
2	NNSE	LCSR	LCSR
3	LCSR	SIADLANC	NNSE
4	SIADLANC	NNSE	LANC
5	LANC	LANC	SIADLANC
6	SIADLIN	SIADLIN	SIADLIN
7	LIN	LIN	LIN

Table 9 – odd combined results

#	*PSNR*	*MSSIM*	*ESMSE*
1	LANC	LANC	LANC
2	NEDI	NEDI	LIN
3	LIN	LIN	NEDI
4	LAZA	LAZA	LAZA

In these tests the RS algorithm performs best when tested with PSNR, MSSIM and ESMSE, but, since many algorithms require training, a bigger test set might increase performance of several algorithms.

For a subjective comparison of the results, an 80×80 pixel part of the lena image after super-resolution from 256 to 512 is shown in Figure 18. The top left image is the original and the name of the corresponding algorithm is above the other images.

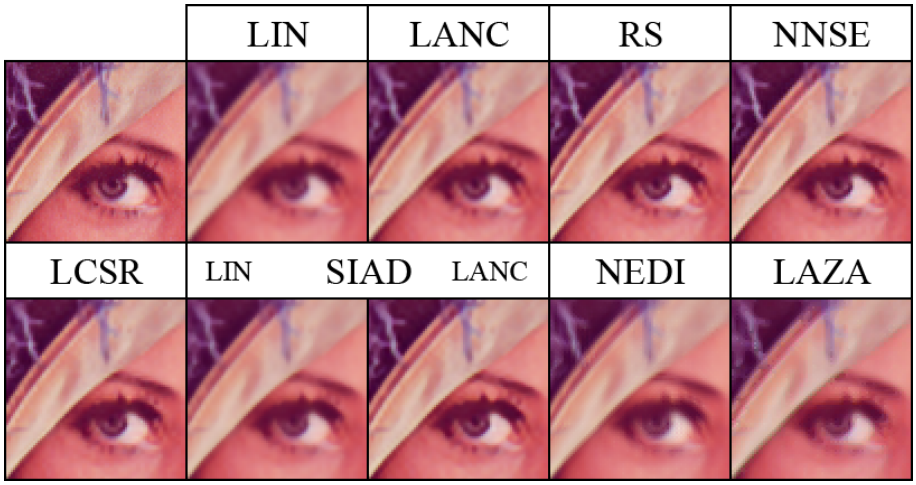


Figure 18 – detail of lena after super-resolution from 256 by one octave

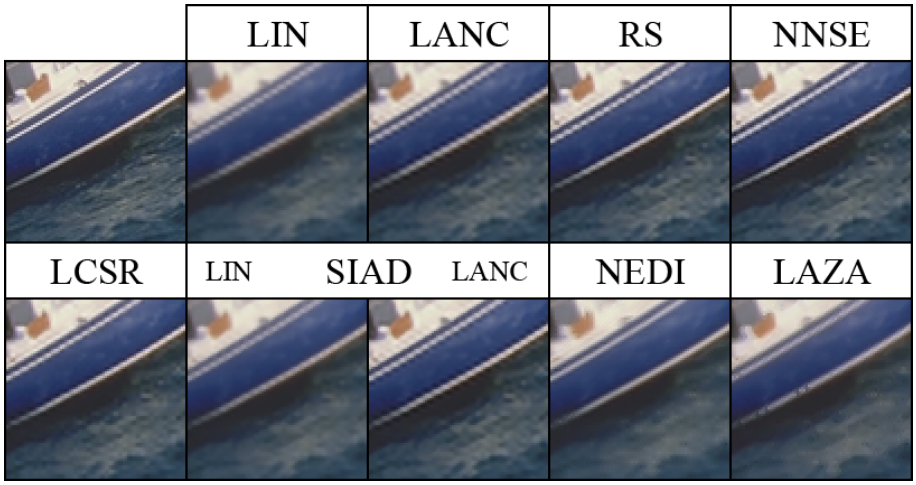


Figure 19 – detail of sail after super-resolution from 256 by one octave

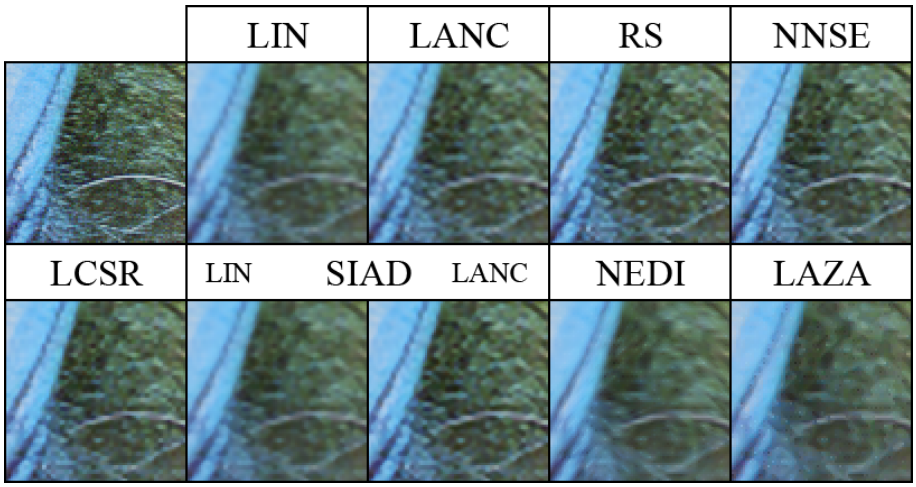


Figure 20 – detail of mandrill after super-resolution from 256 by one octave

Looking at the amount blur introduced, it can be seen that RS, NNSE, LCSR and SIADLANC algorithms introduce the least amount of blur. This can for example be seen in the rim of the hat and the eye. The RS algorithm has generated a single brighter pixel as a reflection detail in the eye, while other techniques did not.

An 80×80 pixel part of the sail image enlarged from 256 to 512 is shown in Figure 19. The diagonal line on the boat in the centre of the detail shows various levels of blocking artefacts over the enlargement. Only the NEDI and LAZA algorithms show no blocking artefacts at all at the expense of being more blurry than other techniques. The NNSE algorithm hardly shows any blocking artefacts, while belonging to the group of algorithms showing the least blurry super-resolution image.

An 80×80 pixel part of the mandrill image enlarged from 256 to 512 is shown in Figure 20. It can be seen that all algorithms have problems generating the fine detail in the hairs of the mandrill. The amount of blur introduced is least using the RS and NNSE algorithms and slightly more in the LCSR and SIADLANC algorithms.

2.4 Conclusions

Looking at the amount of edge blurring in the super-resolution results first, it seems that some algorithms do a very good job at preserving the sharpness of edges; most notably the RS and NNSE algorithms, but also the LCSR and SIADLANC algorithms. It seems that this most well known issue of bilinear super-resolution can be solved in a number of different ways. When it comes to lines instead of edges, the results are less good in general. So, as one might expect, it seems that lines or double edges pose more difficulty than single edges. Since lines are rarer than edges in natural images, an improved training set might improve this for the trained algorithms.

Looking at blocking artifacts on edges, the conclusion is again that some algorithms do a very good job at reducing blocking artifacts; most notably the NNSE, NEDI and LAZA algorithms, but also the RS and LCSR algorithms. Lines again offer a bigger difficulty than edges.

After looking at blurring and blocking we come to the generation of detail. Looking at the results, this seems to be the most difficult part of the three. None of the algorithms can generate the full amount of detail present in the original image. This is very noticeable in the super-resolution results of the mandrill image, since it contains a lot of fine detail. The RS algorithm generates a single brighter pixel as reflection in the eye on the lena image, but in general, both the RS and NNSE algorithms generate the most amount of detail. The LCSR and SIADLANC algorithms are just below that.

An overview of the subjective strengths and weaknesses is presented in Table 10 using a four step scale consisting of - -, -, + and ++ from low to high.

Table 10 – strength and weaknesses

Algorithm	*Blurring*	*Blocking*	*Detail*
LIN	- -	-	- -
LANC	-	- -	-
RS	++	+	++
NNSE	++	++	++
LCSR	+	+	+
SIADLIN	- -	-	-
SIADLANC	+	-	+
NEDI	- -	++	-
LAZA	- -	++	-

The algorithms using classification (RS and LCSR) show promising results. Classifying pixels and only applying an interpolator to the tasks it is good at, seems to be a successful way of improving overall quality. It might be an appropriate way of keeping the interpolator complexity and the overall complexity of the algorithm as low as possible, while being able to get complex results.

Incorporating a more visually correct error measure than MSE like in the NNSE algorithm also seems to improve quality. An interpolator trained in this way can not only obtain a more visually pleasing super-resolution image, but also one that has a good MSE quality.

Super-resolution images obtained by the NEDI algorithm are quite blurry, but do provide a visually appealing image without blocking effects. The resulting images can generally be improved by applying a sharpening filter. The ability of

covariance-based enlargement to remove blocking artifacts might make it useful as an input for other methods. A neural interpolator might for example benefit from local covariance information.

Not all trained algorithms were trained using the same test set. It has been noted in several papers though that typical constructs in one image can be used as training material so to recognize similar constructs in other images. While the content of images in different training sets may differ greatly, they will probably account to similar training results, as long as both training sets consist of a fairly large set of natural images. Taking this into account and perceiving the training set as part of the super-resolution algorithm, the comparisons made above are legitimate. This of course does not mean that a difference in training set could not influence the quality of the super-resolution algorithm.

When looking to the general results in the tests, it seems that handling blurring and blocking for a two by two enlargement is becoming more and more feasible, but that the generation of detail is still a difficult point.

3 Modular Approach

When looking at the different super-resolution algorithms, some similarities between them can be seen. Some elemental parts are used by several algorithms. Clustering is for example used by resolution synthesis [2], super-resolution based on location correlations [9] and super-resolution by exploiting the sparse derivative prior [32]. LAM's are also used by several algorithms [1, 2, 9, 32]. The trained part of a super-resolution algorithm is often presented the local window with the centre subtracted as input and the details as output. These details are formed by subtracting a kernel resized version of the low resolution image from the target high resolution image. In this way, the average value of the pixels is no longer included in the training process, targeting it to details only. This concept is used in a wide range of super-resolution algorithms [1, 2, 9, 15, 29, 32, 34].

The use of similar parts in different algorithms led to the search for a more modular way of constructing super-resolution algorithms by the author. Instead of viewing a super-resolution algorithm as a whole, it is viewed as the structured collection of elemental parts. Several advantages come to mind with this approach to super-resolution algorithms. Reusability is of course a major advantage. New parts based on existing super-resolution algorithms or algorithms from other fields like texture classification can be implemented, while all existing parts can be reused. This allows rapid construction of new algorithms by implementing a single new part. Various combinations of multiple super-resolution algorithms can be made once parts of these algorithms are implemented. This allows for a search for an improved algorithm composed of parts from other state of the art super-resolution algorithms. A final advantage is that it opens the door to a "super-resolution-algorithm" algorithm that searches the space of possible super-resolution algorithms to find the optimal one based on some quality measure. This task has been a manual one up to now, where every part of the structure and every parameter of an algorithm had to be optimized by hand. While it is very hard to include any significant prior knowledge in an automated search for the most optimal super-resolution algorithm, it might be worthwhile to test this concept. It could automate the search for a high quality super-resolution algorithm at the expense of large computational resources.

The modular approach allows basic components to be combined into a super-resolution algorithm. These basic components are based on techniques used in current super-resolution algorithms. The algorithm can be trained on a set of images if needed and then be used to super-resolve images. The basic components will be called nodes in the rest of the thesis and the constructed algorithm the structure. The nodes can be connected to form any valid tree structure. Not all nodes perform the same base function, so not all nodes all interchangeable. There are a number of base node types that each performs a specific function. All nodes belonging to the same node type are interchangeable in the structure. A node can require one or more child nodes to be attached. These child nodes then perform the function specified by their node type in service of the parent node.

3.1 Base Node Types

The design of the modular approach deals with two important goals. The first goal is to incorporate as many techniques that are currently used as possible. The second goal is to make the modular design of super-resolution algorithms as flexible as possible. A choice has to be made about the number of base node types. For a fixed amount of nodes, having less node types results in a higher amount of nodes belonging to the same node type. Each node of a certain type can be used wherever a node of that type is required. Having less node types therefore results in a wider range of nodes that can be selected for a given location and a more flexible construction process. To increase the amount of possible techniques however, it might be needed to increase the number of base nodes. In this way, the two goals interfere with each other. It is decided to incorporate four base types of node in the design. The reason for this will be explained in the next few paragraphs.

Since the modular structure as a whole should be able to super-resolve colour images, a base node type is required that does exactly that. This node type can create a super-resolved version of a given image and is named the image node. The root node of a structure should always be of the image node type, so the structure is able to super-resolve colour images. This immediately allows any super-resolution algorithm to be included in the modular tool as an image node. While this of course doesn't allow for a flexible design of super-resolution algorithms, it at least enables any super-resolution algorithm to be included.

Most super-resolution techniques are designed for greyscale images and can be extended to colour images by applying the same algorithm for each colour plane in the image. Raster images on computers are most often represented in the RGB colour space, consisting of a red, green and blue colour plane. By applying a greyscale technique to each of the three planes, a colour image can be super-resolved. Since the human eye is more sensitive for changes in the luminance than in the chrominance (colour), the YCrCb colour space is development. In this colour space the Y plane represents the luminance or greyscale value, while the Cr and Cb planes represent the chrominance. Some super-resolution algorithms perform high quality super-resolution on the Y channel, while a computationally less expensive technique is used for the Cr and Cb channels. To include the option of using different colour spaces and to easily apply a greyscale technique to colour images, a node type is added that can super-resolve a single image plane. This node is called the plane node. Specific image nodes can be used to distribute the work over one or more plane nodes. The plane node base type allows for any greyscale super-resolution algorithm to be incorporated into the modular tool and to be used for colour images.

Clustering or unsupervised learning is often used in super-resolution. In this technique, every pixel is first classified and then super-resolved according to its class. The classification is done by finding clusters in the information about a pixel. This information can for example consist of the difference with the neighbouring pixels. To enable this selection by pixel, a node type is added that can perform super-resolution on a single pixel in a plane. This node is named the pixel node. It works slightly different than the image and plane nodes. During super-resolution it is first given the whole low resolution plane. It can then be asked to return a super-resolved version or patch of the low resolution pixels at the given location. For each low resolution pixel, a number of high resolution pixels are returned by the pixel node based on the scaling factors. The scaling factors of the super-resolution can

therefore only be integer. Multiple patch variants can be returned by a single pixel node to enable usage of Belief Propagation (BP). The BP plane node can then later determine which patch is used in the super-resolved image.

The clustering node needs some sort of information related to a pixel to base the clustering on. A Linear Associative Memory (LAM) node performs a linear transformation on for example the pixel values in a local window to obtain the high resolution pixels. These pixel values in a local window are similar to the information used by the clustering node in that they are related to a certain centre pixel. This introduces the fourth and final base node type, the information node. The information node can give specific information related to a pixel. The length of the information or the number of values given is determined by the type of information node. A basic 3×3 local window would for example return the values of the nine pixels in that window and would have a length of nine. Like pixel nodes, information nodes first receive the whole plane as information source. They can then be asked to return relevant information for the given coordinates.

Other node types that were considered, but not included are the kernel node and the error node. The kernel node represents a 1-dimensional kernel function that can be used by the kernel resize node for example. Since the kernel node would only be used by the kernel resize node, it seemed a better choice to make the kernel selection a parameter of the kernel resize node than a separate node type. The error node represents an error function between an original and compared image. It could for example be used for training neural networks. Since the error is a single value related to a location on the images, it could also be implemented as an information node and the error node type is therefore not included.

Since some nodes need training before they can function, it is sometimes needed that a structure is trained. Even though a node might not need training, it should know how to distribute the training work among its child nodes. Training of a node and thereby the whole structure can and often does consist of multiple passes. All training images are shown to the structure in every training pass. All nodes should be able to determine whether they or any of their child nodes requires another training pass and should be able to redirect that training pass to the correct child node if needed.

3.2 Resolution Synthesis

The way in which current super-resolution algorithms can be split into common parts may be best explained by using an example. In this example the resolution synthesis algorithm [2] will be split up. This algorithm has been chosen as an example for three main reasons. The first is that the resolution synthesis algorithm is easily split into its elemental parts, making it a clear example. The second is that the resolution synthesis algorithm includes a range of frequently used parts, showing the potential of reusability. The third reason is that resolution synthesis has shown some high quality results in super-resolution, forming a good basis for exploration of higher quality algorithms.

Resolution synthesis is applied to separate image planes instead of a whole image at once. For a full color image it is applied three times; one time for each of the red, green and blue image planes. The algorithm can therefore be split up into a part that divides an image into separate planes and a part that applies resolution

synthesis to the separate planes. This first part is named plane multiplex, since it combines super-resolved planes to a super-resolved image during super-resolution.

Instead of directly super-resolving the plane by resolution synthesis, it is first super-resolved using a kernel and the details are generated using the trained part. This leads to another split, where one part super-resolves the base plane, another part super-resolves the detail plane and a third part combines the base and detail planes to the super-resolved plane. The structure including the plane multiplex and the base and detail approach is shown in Figure 21. The part performing kernel based super-resolution is simply called kernel resize and the part combining the base and detail planes is called base and detail.

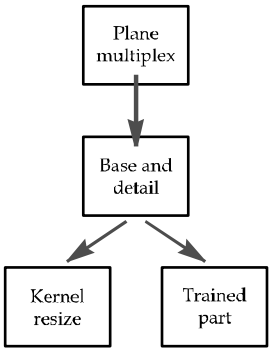


Figure 21 – using base and detail in a modular approach

The trained part of resolution synthesis consists of a clustering part and a LAM for each cluster. The clustering part uses a 3×3 local window as information, while the LAM's use a 5×5 local window as information. The center value is subtracted from the other values in the local window in both cases. All information is of course obtained from the low resolution image plane, since this is the only information present during actual super-resolution. Since the clustering and LAM parts of resolution synthesis operate on separate pixels, we also need a part that splits a plane into separate pixels. Putting this all together, we come to the final super-resolution structure as shown in Figure 22.

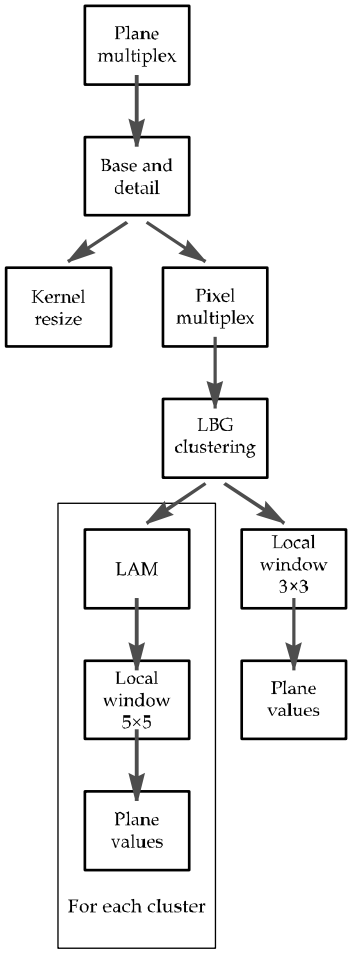


Figure 22 – full modular structure for resolution synthesis

The super-resolution structure in Figure 22 is constructed from ten node instances of eight different nodes. Details on the combination of the two local window nodes into a single node will follow later.

3.3 Workflow

The modular super-resolution approach is a three step process. The first step consists of constructing the super-resolution algorithm by adding and connecting nodes. This untrained structure can then be saved. An untrained structure can of course also be loaded. The second step consists of training. To train the super-resolution structure a set of training images should be selected and the scaling factors should be given. As long as any of the nodes still requires training, another training pass is added. Once the training is done, the super-resolution structure can be saved including the data gathered during training. Again, trained structures can of course also be loaded. The third step consists of the actual use of the trained super-resolution structure. By simply selecting the image to super-resolve, super-resolution is applied using the

scaling factors the structure is trained for. A schematic representation of the workflow described above is shown in Figure 23.

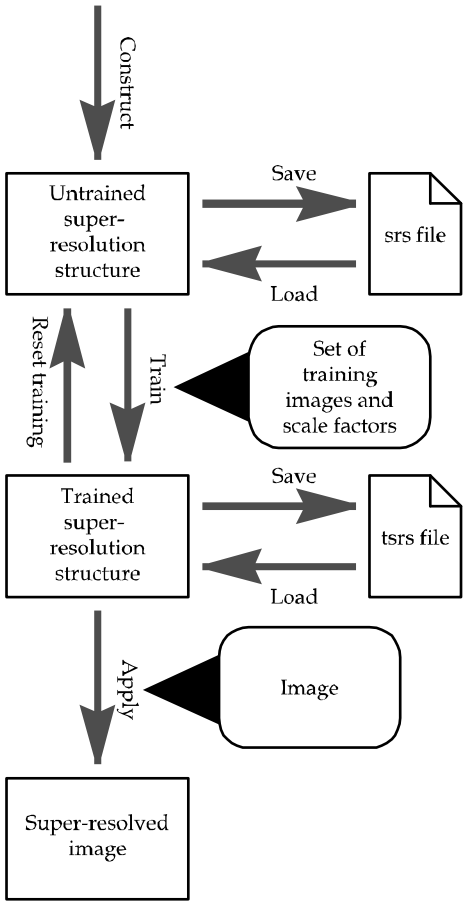


Figure 23 – modular super-resolution workflow

Once a structure is trained, all parameters that would influence the training results are locked. To regain access to them, the structure needs to be reset to an untrained structure. After adjustments of the parameters the structure can be trained again.

3.4 Core Class Overview

An UML diagram of the com.jvojava.sr package is shown in Figure 24. This package contains the main classes for modular super-resolution. All actual implementations of nodes are contained in the imagenode, planenode, pixelnode and informationnode subpackages.

A simple event system enables listeners to receive information on the status of the training or super-resolution process and can been found in the SREvent, SRNotifier and SRListener classes. An instance of SREvent contains information on the status of the process. These status messages can contain the percentage done for

a given node, the current training image and the current training pass. It is send to all registered implementers of the SRListener interface by the SRNotifier class. This last class provides methods for registration and notification of listeners and is used by the SRNode and SRTree classes.

The SRTree class is able to represent a super-resolution structure. It contains methods to train the structure, apply super-resolution, save and load a structure and check whether a structure is fully connected. The nodes in the structure are all instances of the SRNode class, which contains all basic functionalities that node implementations need. It is able to keep track of connections and parameters and handles propagation of common method calls to the child nodes.

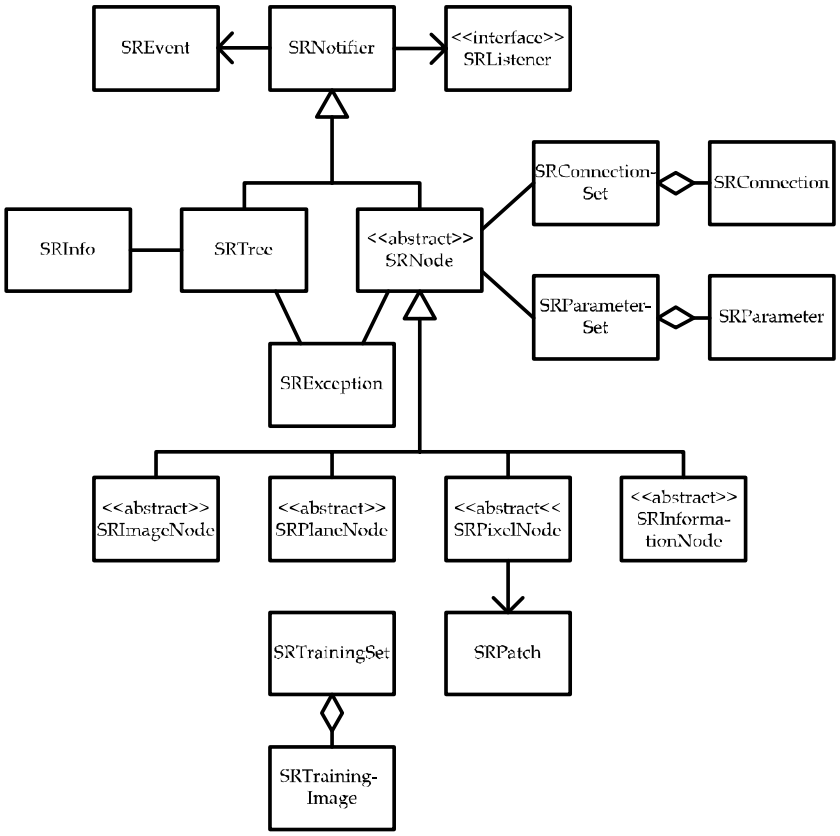


Figure 24 – com.jvojava.sr package UML

Each instance of SRNode has its own instance of SRConnectionSet. This instance is used to manage the connections to child nodes for a SRNode instance. Each connection is of course represented by an instance of SRConnection. The same system goes for the node parameters. Each parameter is represented by an instance of SRParameter and the whole set of parameters of a node is managed by a SRParameterSet instance.

The SRNode class is extended by four other classes, representing the four base types of nodes. These four abstract classes are the ones that actual node implementations extend. The SRImageNode class for example provides the interface to super-resolution and training on full images. The implementations are left to the image nodes that extend the SRImageNode class. All image node implementations

that extend the SRImageNode class are located in the imagenode subpackage. The abstract SRPlaneNode, SRPixelNode and SRInformationNode classes follow a similar approach.

The SRTrainingImage class represents a training image. It contains both a high and low resolution image, so training can be performed using this information. The low resolution image can be either a separate image or can be obtained by the SRTrainingImage class using the high resolution image and a decimation filter. The SRTrainingSet represents a set of SRTrainingImage instance. It is used as a full set of training images for training. The SRPatch class represents one or more high resolution variants for a single pixel. With a scale factor of 2×2 it contains one or more sets of four pixels to be used in the high resolution image.

3.5 Package Overview

The modular super-resolution tool consists of eleven packages of which seven are main super-resolution packages. Three packages are general imaging packages that offer floating point image representations, decimation filters and 1-dimensional kernel functions. One package is a math package that offers matrix calculations for the LAM node. The core super-resolution package com.jvojava.sr contains all base classes for modular super-resolution. All actual node implementations are located in the imagenode, planenode, pixelnode and informationnode subpackages. The bp subpackage contains classes to perform Belief Propagation (BP). The ui subpackage finally contains all User Interface (UI) classes needed to display and use the modular system interactively. An overview of the eleven packages and their relation is shown in Figure 25.

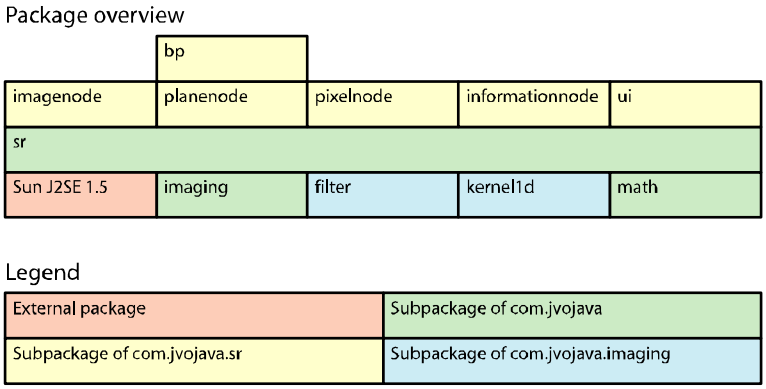


Figure 25 – package overview

As shown in the overview in Figure 25, the core super-resolution package makes use of the standard J2SE 1.5 packages and four imaging and math packages. The subpackages of com.jvojava.sr are constructed on top of the classes provided by com.jvojava.sr. The com.jvojava.sr.bp package offers some functionality to plane nodes that use Belief Propagation.

Table 11 – modular super-resolution tool packages

com.jvojava.imaging
Contains classes for floating point representation of images and image planes used during the whole super-resolution process.
com.jvojava.imaging.filter
Contains image decimation filters that are used to obtain low resolution versions of the training images.
com.jvojava.imaging.kernel1d
Contains a range of 1-dimensional kernels for use in kernel-based image operations.
com.jvojava.math
Contains classes to perform matrix calculations. These are used by the LAM node to obtain a RMS-optimal solution to the overdetermined system of linear equations.
com.jvojava.sr
Contains the base super-resolution classes to represent and work with modular super-resolution structures. These include the four base types of nodes, the parameter and connection system and the super-resolution event system.
com.jvojava.sr.bp
Contains helper classes to perform belief propagation using a line by line approach. The belief propagation main classes can be found in the com.jvojava.sr.planenode package.
com.jvojava.sr.imagenode
Contains all image node classes.
com.jvojava.sr.informationnode
Contains all information node classes.
com.jvojava.sr.pixelnode
Contains all pixel node classes.
com.jvojava.sr.planenode
Contains all plane node classes.
com.jvojava.sr.ui
Contains all super-resolution related user interface classes. These include graphical representations of nodes and connections and the required windows and panels.

3.6 Functional nodes

With the nodes described up to now it is of course possible to construct the resolution synthesis algorithm. It is also possible to construct an algorithm with two clustering steps instead of one and it is possible to vary parameters like the local window size or number of clusters. To increase the number of possible super-resolution structures, more nodes are needed. An overview of functional nodes available in the modular super-resolution toolkit is shown in Table 12.

Instead of only being able to super-resolve in RGB color space, it might be worthwhile to super-resolve in YCrCb color space. Since the human eye is more sensitive to differences in luminance than in chrominance, an algorithm could be constructed that performs high end super-resolution in the luminance channel and less complex super-resolution in the chrominance channels. This reduces the required processing, while maintaining high visual quality. To this end an image

node that performs conversion from RGB color space to YCrCb color space and an image node that distributes the first plane in an image to one plane node and the other planes to another plane node are added. These two nodes combined enable construction of an algorithm that performs super-resolution in YCrCb color space while have different algorithms for the Y plane and the two other planes.

In super-resolution exploiting the sparse derivative prior [32] Belief Propagation (BP) is used. In this algorithm, like the resolution synthesis algorithm, a clustering step is followed by a set of LAM's for each cluster. This time, every pixel is super-resolved by each LAM and BP is used to find the combined set of patches that is likely to form the super-resolved image based on prior knowledge of natural images. To include the possibility to use BP in the modular tool, two important things are needed. Besides the pixel multiplex node which combines patches into a super-resolved image, a node which combines multiple variants of patches into a super-resolved image based on belief propagation is needed. Also, the LBG clustering node should be able to return the results of all its children, instead of just returning the result of the child belonging to the cluster a pixel falls in. To achieve this, a belief propagation node is added and a parameter named result is added to the LBG clustering node. This result parameter can be set to either best or all, returning the results of a single child or all children respectively.

A second node that performs clustering is also added to the tool. This pixel node clusters based on a 2D Kohonen Self Organizing Map (SOM) [18]. Besides kernel based super-resolution, nearest neighbor super-resolution is another fast and low quality algorithm that is frequently used. Since the nearest neighbor algorithm can operate on separate pixels, it is added as a pixel node to the tool, instead of a plane node like kernel resize.

Since a clustering step in super-resolution might benefit from experiences in texture classification [7, 26] a textural information node has been added based on the directional texture classification property in [26]. It returns the local direction as percentages of eight directions. It classifies the direction in each pixel in the local window as being one of eight directions using the horizontal and vertical Sobel filter as in [26]. This node is named the local direction node. Because information for clustering should be normalized, a normalize information node that normalizes the information of its child to have zero mean and unit variance is added. Some utility information nodes that for example return the absolute value of the information provided by its child are also included.

Table 12 – functional nodes in the modular approach

RGB to YCrCb	**Image node**
No parameters	▪ Image node child: Color space converted
This node converts an image from RGB color space to YCrCb color space and back. Its sole child node should handle the super-resolution of the color space converted image. The transformation between these two color spaces is a linear one.	

Plane multiplex 1	**Image node**
No parameters	▪ Plane node child: All planes
This image node separates work on an image into work on separate planes. It needs a single plane node as child and redirects all planes in the incoming images to this child.	
Plane multiplex 2	**Image node**
No parameters	▪ Plane node child: First plane ▪ Plane node child: Other planes
This image node separates work on an image into work on separate planes. This plane multiplex requires two plane nodes as children and redirects the first plane to the first child and all other planes to the second child.	
Base and detail	**Plane node**
No parameters	▪ Plane node child: Base ▪ Plane node child: Detail
This plane node splits the work to be done into two other plane nodes. The first one applies the base enlargement and the second one will add the detail. During super-resolution the results of the two child plane nodes are simply added. During training, the base child is trained first. After the base child is trained, the detail child is trained to return the difference between the actual high resolution plane and the super-resolved plane by the base child.	
Belief propagation	**Plane node**
▪ Floating point parameter: Reconstruction standard deviation ▪ Floating point parameter: Natural image prior standard deviation ▪ Floating point parameter: Natural image prior	▪ Pixel node child: Patch variants
This plane node is a bridge to pixel nodes that can handle and combine multiple patches for each super-resolved pixel. This node applies iterative belief propagation to select the locally optimal patch for each pixel. It uses two equations to determine the likelihood of a patch being the best one. The first equation compares a patch to the source pixel to ensure that the average intensity of the patch corresponds to the intensity of the source pixel. The second equation compares neighbouring pixels in neighbouring patches to ensure that the patches fit together based on a natural image prior.	
Kernel resize	**Plane node**
▪ List parameter: Kernel (Linear, Cubic spline, Catmull-Rom, Blackman sinc 3, Blackman sinc 5)	*No children*
This plane node applies simple kernel based resizing. It has no children and it needs no training. It has one parameter determining the type of kernel used.	
Pixel multiplex	**Plane node**
No parameters	▪ Pixel node child: All pixels
This plane node separates work on a plane into work on separate pixels. This node needs a single pixel node as child and redirects all pixels in the incoming planes to this child.	

Base and detail	**Pixel node**
No parameters	▪ Pixel node child: Base ▪ Pixel node child: Detail
This pixel node splits the work to be done into two other pixel nodes. The first one applies the base enlargement and the second one will add the detail. This node functions very similar to the plane node version above, but on single pixels instead of full planes.	
LAM	**Pixel node**
No parameters	▪ Information node child: Information
This node incorporates a linear associative memory that trains itself in one pass. It needs a single information node as child to supply information. It will calculate the root mean square optimal linear transformation from the supplied information to the high resolution pixels during training. During super-resolution this linear transformation is applied to the information to obtain the high resolution pixels.	
LBG clustering	**Pixel node**
▪ List parameter: Result (Best, All) ▪ Integer parameter: Cluster count ▪ Integer parameter: Number of passes	▪ Pixel node child: Classified pixels ▪ Information node child: Information
Node to perform clustering based on the Linde-Buzo-Gray (LBG) algorithm. This iterative Vector Quantization (VQ) algorithm can be used to find a given number of clusters in the information vectors given by the child information node. Each cluster will use its own copy of the pixel node child for super-resolution.	
Nearest resize	**Pixel node**
No parameters	*No children*
This method performs a simple nearest neighbour resize on the given pixels. It has no children and is always fully trained, so it needs no training. It simply duplicates the low resolution pixel to form the high resolution pixels.	
Pixel demultiplex	**Pixel node**
No parameters	▪ Plane node child: Source plane
This pixel node combines work on separate pixels to work on a whole plane during both training and application. This node needs a single plane node as child and redirects a whole plane to this child if a single pixel request comes in.	
Result multiplex	**Pixel node**
No parameters	▪ Pixel node child: First results ▪ Pixel node child: Second results
This node combines the Super-Resolution patches from its two pixel node children. It can be used to return patch variants of multiple algorithms to a belief propagation node.	
SOM2D clustering	**Pixel node**
▪ List parameter: Result (Best, All) ▪ Integer parameter: Size per dimension ▪ Integer parameter: Number of passes	▪ Pixel node child: Classified pixels ▪ Information node child: Information
Node to perform clustering based on the 2-dimensional version of the Kohonen Self-Organizing Map (SOM). Each cluster will use its own copy of the pixel node child for super-resolution.	

Absolute	**Information node**
No parameters	▪ Information node child: Information
This information node simply takes the values from its child information node and makes them absolute.	
Cross multiply	**Information node**
No parameters	▪ Information node child: First information ▪ Information node child: Second information
This information node takes the information from two child information nodes and multiplies every information element of the first node with every information element of the second node. If the first child returns n information elements and the second child returns m information elements, then this node will return n×m information elements.	
Information multiplex	**Information node**
No parameters	▪ Information node child: First information ▪ Information node child: Second information
This information node takes the information from two child information nodes and combines them by appending them to each other. This node can be used to combine multiple sources or types of information into a single information element.	
Local direction	**Information node**
▪ Integer parameter: Radius	▪ Information node child: Information
This information node calculates the distribution of the directions of edges in the local window. It classifies each edge direction determined by two Sobel filters in one of eight directions. It will return the percentages for each of the eight direction bins. There is one parameter in this node, the window radius. With a radius of five, the local window will be 11×11 pixels in size. 9×9 = 81 directions can be obtained from this area.	
Local window	**Information node**
▪ Integer parameter: Radius ▪ List parameter: Processing (No processing, Subtract center) ▪ Integer parameter: Distance multiplier	▪ Information node child: Information
This information node takes information from its child node in the local window around the target pixel. This node has several parameters. The most basic setting is the radius setting. With the radius set to one, the width and height of the local window will be three. The second setting is the processing setting. With no processing, all values in the local window are just returned. With processing set to subtract center, the center value will be subtracted from all values. Since the center value will then always be equal to zero, the center value is not returned. The last parameter of this node is the distance multiplier. The standard value of this parameter is one. The coordinates relative to the center pixel are multiplied by this value. The top left pixel in a radius one window normally has relative coordinates (-1, -1). When the distance multiplier is set to three, the pixel with relative coordinates (-3, -3) is used as top left pixel in a radius one local window.	

Normalize	**Information node**
No parameters	▪ Information node child: Information
This information node normalizes the information from the child information node to zero mean and unit variance. It uses two training passes during training. In the first pass the mean value is calculated and in the second pass the variance is calculated. This information is then used to return information with zero mean and unit variance.	
Plane values	**Information node**
No parameters	*No children*
This is the most basic information node. It simply returns the value of the current plane at the given location. It has no children and no parameters.	
Static value	**Information node**
▪ Floating point parameter: Value	*No children*
This information node represents a static value that can be set by its only parameter. It has no children.	

With these nodes at hand a wide range of super-resolution structures can be built and tested.

4 Implementation

The modular super-resolution tool is implemented in the Java programming language. This language is chosen, because the author has affinity with the language and because it would require no additional packages besides J2SE 1.5 [42] to implement the tool. The development environment used is Eclipse 2.1 [41].

4.1 User Interface

The target audience of the modular super-resolution tool consists of people with programming experience and experience in creating super-resolved images. Since the people in this target audience have a wide experience in using computers and software, the most important element of the user interface is an adequate feedback of what the software is doing. Experienced users often prefer using shortcut keys instead of moving the mouse. With these aspects of the target audience in mind, the interface of the modular super-resolution tool is implemented to allow fast use and return visual feedback of what the software is doing. No specific metaphor is used, since the target audience does not require this for efficient usage.
The screenshot in Figure 26 shows the different parts that are visible in the modular tool main window. The name of the software is displayed at the top in the title bar along with the status. The menu bar with all commands needed to operate the software is located right below the title bar. Five menus are present in the menu bar and most frequently used commands have shortcut keys to execute the commands without going into the menu. The node work space fills the rest of the windows, unless the preview pane is present. The preview is only visible during training. The title bar of the window will also display a small piece of status information. During training it will display the current training pass. During application of super-resolution it will display that the software is busy with the application process.

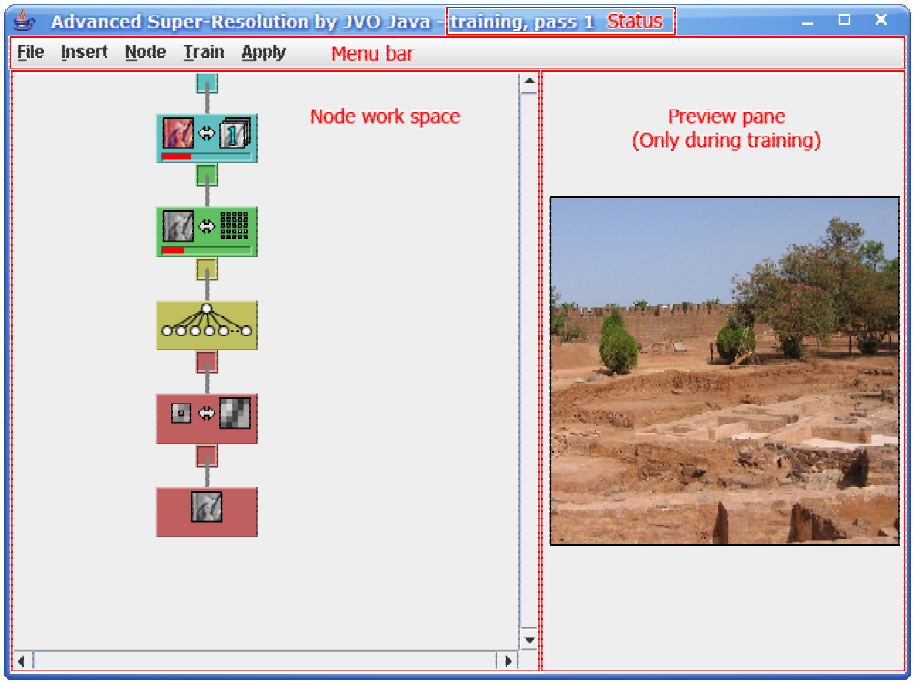


Figure 26 – modular tool screenshot

The node work space is the main working area. It fills as much of the main window as possible and is the location where super-resolution structures are built. By using common scroll bars, the actual size of the node work space is allowed to be bigger than the displayed part. The preview pane shows the current training image during training to give a sense of progress within the current training pass. At other times, the preview pane is not visible to make more room for the node work space.

The node work space contains the nodes and connections that together form the super-resolution structure. Nodes are displayed as rectangular objects with a color according to their base type. Image nodes are colored cyan, plane nodes are green, pixel nodes are yellow and information nodes are red. Each node also shows an icon as an indication of the node's functionality. The required children of nodes are displayed as square connectors below the parent node. The colors of these connectors coincide with the colors of the node types. The work space itself contains a image node connector at the top, which needs to be connected to the root node. Connections between nodes are displayed as grey lines from the connector of the parent node to the child node.

Selected nodes display a black outline as shown in Figure 27(c). A node can be selected by simply clicking it once with the left mouse button. It is also possible to do a block selection as displayed in Figure 27. A block selection can be started by clicking and holding the left mouse button anywhere besides on nodes, connectors or connections. All nodes that are fully inside the grey rectangular shape created by dragging the mouse will be selected once the left mouse button is released. Keyboard modifiers can be used to increase or decrease the current selection. If the CTRL button is pressed while making a selection, it is added to the current selection. If the ALT button is pressed while making a selection, it is subtracted from the current selection.

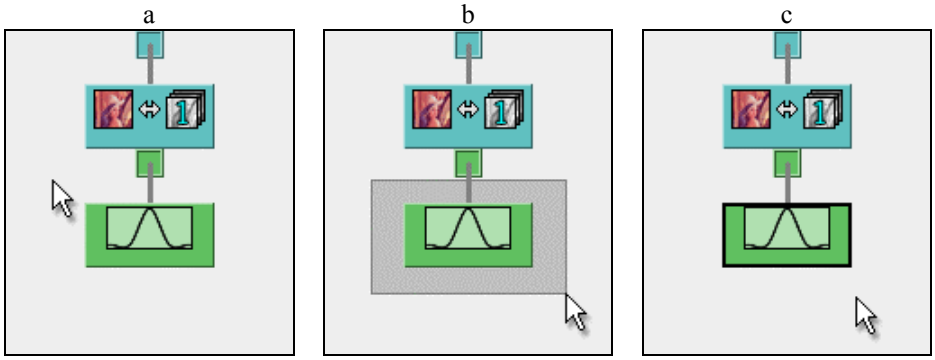


Figure 27 – block select screenshots (a) click and hold on background; (b) drag to create block; (c) release to apply selection

Nodes can be moved by starting a left mouse button drag on a node. All selected nodes will move along with this dragging motion. A move cursor will be shown when hovering over a node.

Nodes can be connected by dragging the mouse from a connector to a node. If this node is of the same type as the connector, a connection will be made. A hand cursor will be shown when hovering over a connector as shown in Figure 28. Once the left mouse button is down, one end of the line will be positioned in the centre of the connector and the other end of the line will be dragged with the mouse cursor. Once the mouse button is released, the connection will be made if possible.

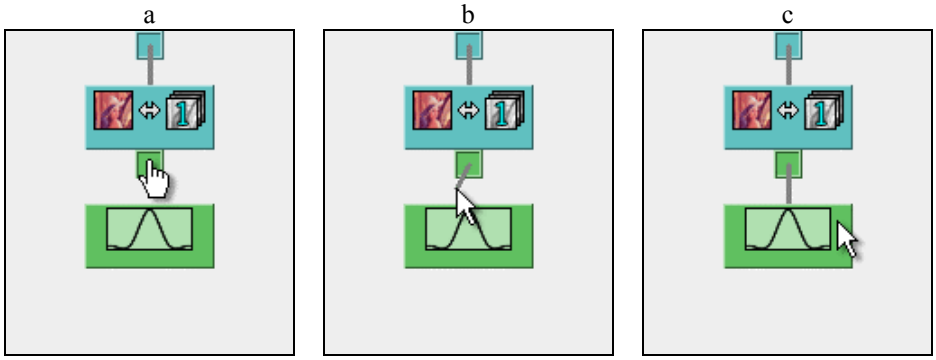


Figure 28 – connection screenshots (a) click and hold on connector; (b) drag line end; (c) release on node to create connection

Since nodes can be dragged by using the mouse, it is not possible to create a connection by dragging from the node to the connector. Since the structure is tree shaped every child can have only one parent. Every connector can also only have one child. Any existing connection that conflicts with these two rules is removed when creating a new connection. A connection can therefore be removed by simply creating a new connection from the same connector.

During training and application of super-resolution some nodes show a progress bar with the progress of that node on the current image. Together with the preview of the current training image in the preview pane and the training pass in the title bar this forms a full visual progress indication. The progress bar can have one of three

colours, cyan, red or white. The cyan colour denotes application, the red colour denotes training and the white colour denotes other tasks. Buffering nodes that buffer results by saving and loading to temporary files use red for saving and cyan for loading.

The node properties window allows to change the name of a node and to adjust its parameters. It can contain the information for one or more nodes, depending on how many nodes are selected. The node properties window can be shown by simply double clicking a node or by using the menu.

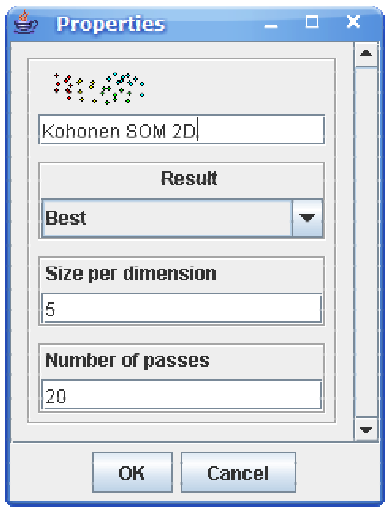


Figure 29 – properties window screenshot

For every node, the property window displays the icon of the node, the name of the node and the parameters of the node. The name and parameters can be changed using this window. If the structure is trained, some parameters might be locked and can't be changed in the properties window. This is made visual by disabling the input elements of locked parameters. An example properties window is shown in Figure 29. There are two buttons on the bottom of the properties window. The "OK" button will apply the changes, while the "Cancel" button will discard the changes made to the names and/or parameters.

The training window allows the selection of training images to be used during the training of a super-resolution structure. The scale factor to be trained on should also be set in the training window. The scale factor consists of two numbers, denoting the horizontal and vertical increase in resolution. Since these factors will often influence the training process, they should be given before the training starts and can only be changed after a training reset. The training window shows the scale factor at the top, the current list of images on the left, a preview of the selected image on the right and four buttons on the bottom. The "Add" button brings up a file selection window, which allows selection of one or more images for addition to the list of current training images. The "Remove" button simply removes the selected images from the list. The "Train" button starts the training process, while the "Cancel" button discards the training window without any further action. Once an image is selected in the list of current images on the left of the window, a quick preview of this image will be shown on the right of the window. The training window is shown in Figure 30.

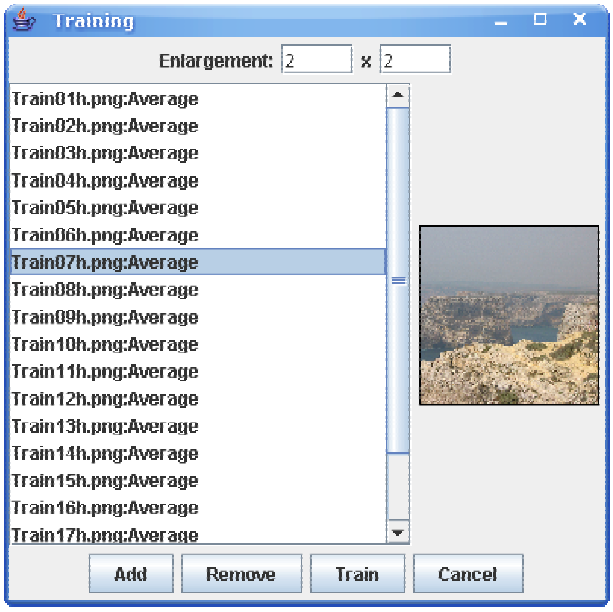


Figure 30 – training window screenshot

4.2 Node Methods

The SRNode class is the parent class for all nodes that can be used in the modular super-resolution tool. Table 13 shows the nine most important methods from the SRNode class. These methods mainly handle the training process. The application of super-resolution is handled by methods that are specific to the type of node. The FloatImage and FloatPlane classes can be found in the com.jvojava.imaging package and represent an image and an image plane respectively in floating point format.

Table 13 – important methods in SRNode

Method	Description
void setLow(FloatPlane)	Sets the low resolution FloatPlane. Subclasses should extend this method if they want to store the plane.
void setHigh(FloatPlane)	Sets the high resolution FloatPlane. Subclasses should extend this method if they want to store the plane.
boolean isFullyConnected()	Checks whether this node and all nodes below it are connected. This means that all sets of connections should exist and all the connections in it should have a child node connected to it.
boolean isFullyTrained()	Returns whether this node and all nodes below it are trained.

void trainReset()	Resets training for this node and all nodes beneath it. Results of Super-Resolution training should be discarded by all nodes and all nodes requiring training should return false when isTrained is called after a call to this method.
void trainStart(SRInfo)	This method is called when training is started. All nodes should perform their one time setup for training in this method.
void trainStartPass(SRInfo)	This method is called when a training pass is started. All nodes should perform their setup for a training pass in this method.
void trainEndPass(SRInfo)	This method is called when a training pass has ended. All nodes should perform their clean up after a training pass in this method.
void trainEnd(SRInfo)	This method is called when training has ended. All nodes should perform their one time clean up after training in this method.

Before training can commence, the structure should be fully connected; all children required by all nodes should be present. The isFullyConnected method is used to check this. If the structure is connected, it can be trained. This starts with a call to the trainStart method and ends with a call to the trainEnd method. The SRInfo instance, which is used as a parameter in these training methods, contains the target scale factor. All nodes should do their initial setup for training in the body of the trainStart method and their final cleanup after training in the body of the trainEnd method. As many training passes as needed are performed between the calls of these two methods. A call to the isFullyTrained method determines whether the structure needs another training pass. Each training pass is started with a call to trainStartPass and ended with a call to trainEndPass. All nodes should do their training pass setup in the body of the trainStartPass method and their training pass cleanup in the body of the trainEndPass method. The pseudo code for the train method in SRTree is shown in Code block 1.

```
train(SRTrainingSet set, SRInfo info, SRImageNode root) {
    if (!root.isFullyConnected()) throw SRException
    root.trainStart(info)
    while (!root.isFullyTrained()) {
        root.trainStartPass(info)
        for each image in set {
            root.train(image.low, image.high, info)
        }
        root.trainEndPass(info)
    }
    root.trainEnd(info)
}
```

Code block 1 – SRTree train method pseudo code

The train method from SRImageNode is called for each image in the set of training images in each pass as can be seen in the pseudo code for the train method in SRTree. The two most important methods in the SRImageNode class, including this train method, are shown in Table 14.

Table 14 – important methods in SRImageNode

Method	Description
void train(FloatImage, FloatImage, SRInfo)	Abstract method that when implemented should perform training on the given low and high resolution versions of an image.
void apply(FloatImage, FloatImage, SRInfo)	Abstract method that when implemented should perform Super-Resolution on the given low resolution image and store the result in the given high resolution buffer.

The SRImageNode class contains a specific method for performing training on the given pair of low and high resolution images and one for super-resolving the given image and storing the result in the given high resolution buffer. The SRInfo instance containing the scale factors is also included as a parameter in both methods. Both methods are abstract and should be implemented by the classes extending the SRImageNode class.
The SRPlaneNode class contains similar methods as the SRImageNode class, but now targeted at separate planes instead of images. The two most important methods in SRPlaneNode are shown in Table 15.

Table 15 – important methods in SRPlaneNode

Method	Description
void train(FloatPlane, FloatPlane, SRInfo)	Abstract method that when implemented should perform training on the given low and high resolution versions of a plane.
void apply(FloatPlane, FloatPlane, SRInfo)	Abstract method that when implemented should perform Super-Resolution on the given low resolution plane and store the result in the given high resolution buffer.

The SRPixelNode class is slightly different from the SRImageNode and SRPlaneNode classes in its method organization. To reduce overhead, the SRPixelNode receives the low and high resolution planes by use of the setLow and setHigh methods in the SRNode class. The train and apply methods in SRPixelNode have a x and y coordinate as parameter, but lack the low and high resolution planes as parameters. The two most important methods in the SRPixelNode class are shown in Table 16.

Table 16 – important methods in SRPixelNode

Method	Description
void train(int, int, SRInfo)	Abstract method that when implemented should perform training at the given location in the low and high resolution versions of the current plane.

void apply(int, int, SRPatch, SRInfo)	Abstract method that when implemented should perform Super-Resolution at the given location in the current low resolution plane and store the result(s) in the given high resolution buffer.

The SRPatch instance used as parameter in the apply method of the SRPixelNode is used as a buffer to hold one or more high resolution versions of the super-resolved pixel. The SRInformationNode class works similar to the SRPixelNode class in that it also receives the low and high resolution planes by use of the setLow and setHigh methods in the SRNode class. Since the length or dimension of the information is variable, a method is included to retrieve this dimension of the information. The three most important methods in the SRInformationNode class are shown in Table 17.

Table 17 – important methods in SRInformationNode

Method	Description
void train(int, int, SRInfo)	Abstract method that when implemented should perform training at the given location.
int getLength()	Abstract method that when implemented should return the dimension of the information supplied by this node.
void getInformation(int, int, float[])	Abstract method that when implemented should supply the information at the given coordinate and store it in the given float array.

4.3 Plane Multiplex Image Node

In this subsection an example image node will be described in some more detail. One of the bridges between image and plane nodes, SRPlaneMux2, is chosen as example here. This image node distributes the work to be performed on an image over two plane nodes; the first plane in an image is handed to the first child and the other planes are handed to the second child. This node can for example be used in combination with a conversion from RGB color space to YCrCb color space. Once the image is in YCrCb color space, this node distributes the work on the Y plane (luminance) to the first child and the work on the Cr and Cb planes (chrominance) to the second child. This behaviour is reflected in the pseudo code for the train and apply methods as listed in Code block 2.

```
SRPlaneNode child1, child2

train(FloatImage low, FloatImage high, SRInfo info) {
    int plane = low.getPlaneCount()
    if ((plane > 0) && (!child1.isFullyTrained())) {
        child1.train(low[0], high[0], info)
    }
    for (int i = 1;i < plane;i++) {
        if (!child2.isFullyTrained()) {
            child2.train(low[i], high[i], info)
        }
    }
}

apply(FloatImage low, FloatImage high, SRInfo info) {
    int plane = low.getPlaneCount()
    if (plane > 0) {
        child1.apply(low[0], high[0], info)
    }
    for (int i = 1;i < plane;i++) {
        child2.apply(low[i], high[i], info)
    }
}
```

Code block 2 – SRPlaneMux2 pseudo code

The planes in an image are accessed by simple indexing in the pseudo code above. If the first plane in the image named low is needed, it is simply referenced to as low[0]. The child1 and child2 variables refer to the two plane node children of the node. A graphical illustration of the process is shown in Figure 31 for training and application in RGB colour space and application in YCrCb colour space.

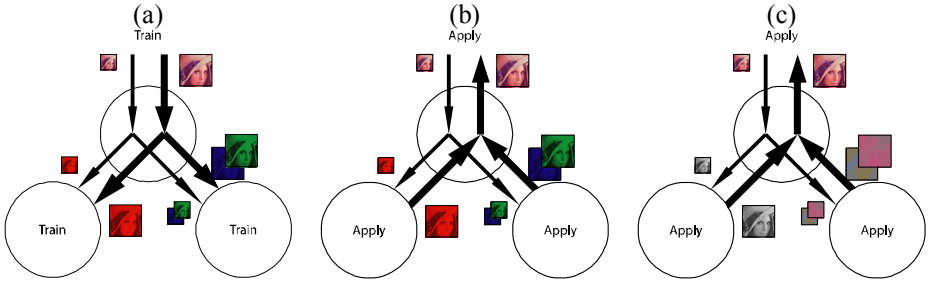


Figure 31 – SRPlaneMux2 flowchart (a) training in RGB colour space; (b) applying in RGB colour space; (c) applying in YCrCb colour space

The three circles represent the plane multiplex node, the node responsible for the first plane and the node responsible for the remaining planes. The thin arrows represent the flow of low resolution images and image planes, while the thicker arrows represent the flow of high resolution images and image planes.

Beside the node used in this example, there are some other bridge nodes. There are for example also plane multiplex nodes that have one or three plane node children. The one with one child simply redirects all planes in an image to this child. The one with three children redirects the first plane to the first child, the second plane to the second child and the other planes to the third child. A similar bridge is

available from plane nodes to pixel nodes in the form of the SRPixelMux class. This node has a single pixel node as child and redirects all work to this child.

4.4 Base and Detail Plane Node

This subsection contains some details on an example plane node. Many trained super-resolution algorithms first use a simple kernel resize for a base image and use the trained part to generate the missing details. By adding the details to the kernel resized image, the final super-resolved image is created. The trained part needs to generate the details in the image only. These details can be obtained for training by subtracting the kernel resized image from the original image. Flowcharts for the case of training the detail generation and the case of applying super-resolution are displayed in Figure 32. The square marked "L" represents the low resolution image, while the square marked "H" represents the high resolution image. The high resolution intermediate kernel and detail results are shown as squares marked "HB" and "HD" respectively. The circle marked "-" represents a subtraction operator, while the circle marked "+" represents an addition operator. The kernel resize and detail generation operators are shown as circles marked "Kernel" and "Detail" respectively.

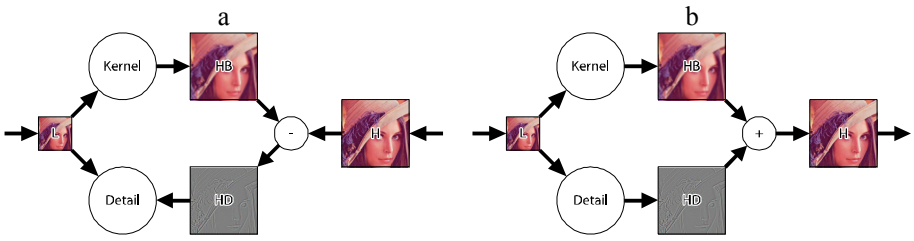


Figure 32 – kernel and detail flowcharts (a) training on details; (b) applying super-resolution

In the modular tool, the base and detail node is implemented in a more general way and as a plane node. The base side doesn't have to be a kernel resize, but can be any plane node. The same of course goes for the detail side. The base and detail node requires two child plane nodes; one to supply the base image and one to supply the detail image. The base and detail node will first train the first child node (on the base side) if needed and then the second child node (on the detail side) if needed. The base and detail node also executes the subtraction and addition operators in Figure 32. The flowcharts for training and application of the base and detail node are shown in Figure 33. The three circles represent the base and detail node, the node responsible for the base image and the node responsible for the detail image. All three nodes are of the plane node type. The thin arrows represent the flow of low resolution image planes, while the thicker arrows represent the flow of high resolution image planes. In training mode, a plane node will receive both the low and high resolution plane. In application mode, a plane node will receive the low resolution image and return the high resolution plane.

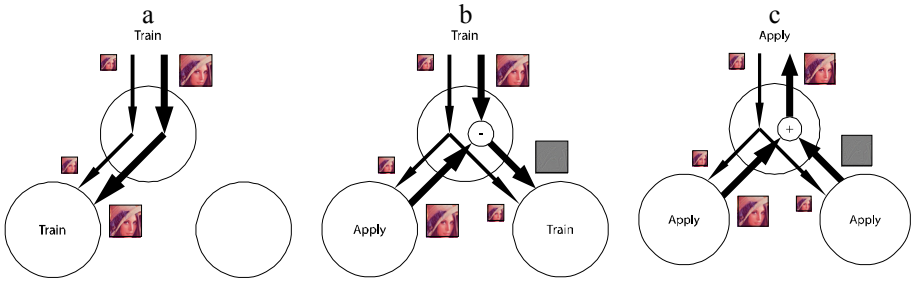


Figure 33 – SRBaseDetailPlane flowchart (a) training the first child; (b) training the second child; (c) applying super-resolution

The methods in SRBaseDetailPlane that reflect the behaviour shown in Figure 33 are isFullyTrained, trainStartPass, train, trainEndPass and apply. The pseudo code for these methods is listed in Code block 3.

```
SRPlaneNode base, detail

boolean isFullyTrained() {
    if (!base.isFullyTrained()) return false
    return detail.isFullyTrained()
}

trainStartPass(SRInfo info) {
    if (!base.isFullyTrained()) {
        base.trainStartPass(info)
    } else {
        detail.trainStartPass(info)
    }
}

train(FloatPlane low, FloatPlane high, SRInfo info) {
    if (!base.isFullyTrained()) {
        base.train(low, high, info)
    } else {
        FloatPlane highDetail
        base.apply(low, highDetail, info)
        subtract high from highDetail
        detail.train(low, highDetail, info)
    }
}

trainEndPass(SRInfo info) {
    if (!base.isFullyTrained()) {
        base.trainEndPass(info)
    } else {
        detail.trainEndPass(info)
    }
}
```

```
apply(FloatPlane low, FloatPlane high, SRInfo info) {
    FloatPlane highDetail
    base.apply(low, high, info)
    detail.apply(low, highDetail, info)
    add highDetail to high
}
```

Code block 3 – SRBaseDetailPlane pseudo code

The base and detail variables used in Code block 3 refer to the first and second plane node child. For flexibility, there is also a pixel node version of the base and detail node. This node works similar, but on pixels instead of planes.

4.5 2D SOM Pixel Node

This subsection shines some more light on an example pixel node. The 2D Self Organising Map (SOM) node is a pixel node that performs clustering on the inputs it gets from its child information node. This clustering is based on the SOM architecture by Kohonen [18] and is also known as unsupervised learning. A 2-dimensional SOM consists of a 2-dimensional regular grid which is normally initialized with random values. After training, the SOM offers a mapping from input information to a cluster number. The number of clusters is determined up front by the dimensions of the SOM. The training process is designed to find clusters in the information. An example of this is illustrated in Figure 34 for 2-dimensional input information. The white circles represent the cluster centres and the black lines between them their relation in a regular grid. The red areas in the background image denote the densest information, while the blue areas denote the least dense information.

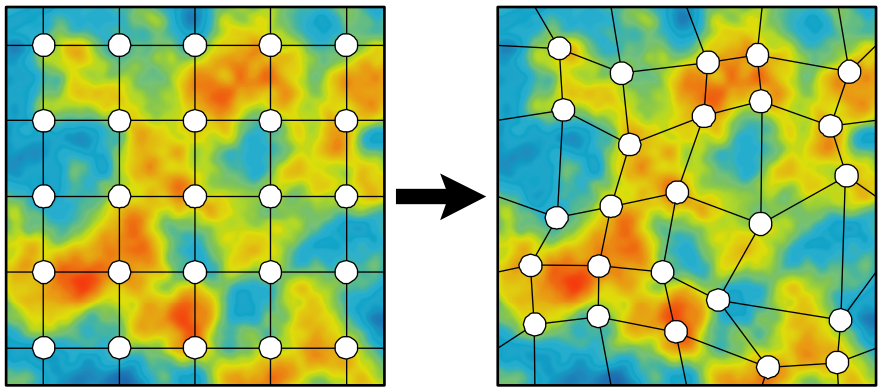


Figure 34 – 2D SOM illustration

The cluster centres will move toward the denser information areas during training, which results in higher densities of clusters near higher densities of information. The pseudo code for the important methods in the 2D SOM node is shown in Code block 4.

```
float DECAY_CONSTANT = 10
SRInformationNode information
SRPixelNode[] child
int clusterCount, pass, passCount, size
float radius, rate
float[][] cluster

int nearestCluster(float[] vector) {
    int result = 0
    float min = #inf
    for (int i = 0;i < clusterCount;i++) {
        float current = distance(cluster[i], vector)
        if (current < min) {
            min = current
            result = i
        }
    }
    return result
}

float distance(float[] vector1, float[] vector2) {
    float result = 0
    for (int i = 0;i < vector1.length;i++) {
        result += pow(vector1[i] - vector2[i], 2)
    }
    return sqrt(result)
}

boolean isFullyTrained() {
    if (!information.isFullyTrained()) return false
    if (!trainedSelf) return false
    for (int i = 0;i < clusterCount;i++) {
        if (!child[i].isFullyTrained()) return false
    }
    return true
}

trainStartPass(SRInfo info) {
    if (!information.isFullyTrained()) {
        information.trainStartPass(info)
    } else if (!trainedSelf) {
        if (pass == 0) {
            int length = information.getLength()
            cluster = new float[clusterCount][length]
            for (int i = 0;i < clusterCount;i++) {
                for (int j = 0;j < cluster[i].length;j++) {
                    cluster[i][j] = (random() - 0.5) / 10
                }
            }
        }
        float decay = exp(-pass / DECAY_CONSTANT)
        radius = size * decay / 2
        rate = decay / 10
    } else {
```

```
        for (int i = 0;i < clusterCount;i++) {
            if (!child[i].isFullyTrained()) {
                child[i].trainStartPass(info)
            }
        }
    }
}

trainEndPass(SRInfo info) {
    if (!information.isFullyTrained()) {
        information.trainEndPass(info)
    } else if (!trainedSelf) {
        if (pass >= passCount) trainedSelf = true
        pass++
    } else {
        for (int i = 0;i < clusterCount;i++) {
            if (!child[i].isFullyTrained()) {
                child[i].trainEndPass(info)
            }
        }
    }
}

train(int x, int y, SRInfo info) {
    if (!information.isFullyTrained()) {
        information.train(x, y, info)
    } else if (!trainedSelf) {
        float[] info = information.getInformation(x, y)
        train(info)
    } else {
        float[] info = information.getInformation(x, y)
        int nearest = nearestCluster(info)
        child[nearest].train(x, y, info)
    }
}

train(float[] vector) {
    int nearest = nearestCluster(vector)
    int wx = nearest / size
    int wy = nearest % size
    for (int i = 0;i < clusterCount;i++) {
        int dx = i / size - wx
        int dy = i % size - wy
        float d = sqrt(dx * dx + dy * dy)
        if (d < radius) {
            float effect = rate * effect(d)
            for (int j = 0;j < vector.length;j++) {
                float motion = vector[j] - cluster[i][j]
                cluster[i][j] += effect * motion
            }
        }
    }
}

float effect(float d) {
    return exp(-d * d / (2 * radius * radius))
}
```

```
apply(int x, int y, SRPatch patch, SRInfo info) {
    float[] info = information.getInformation(x, y)
    int nearest = nearestCluster(info)
    child[nearest].apply(x, y, patch, info)
}
```

Code block 4 – SRSOM2D pseudo code

The 2D SOM node allows finding groups in any type of information supplied by information nodes. It can then be used to train and apply specific super-resolution for each group.

4.6 LAM Pixel Node

This subsection contains an explanation of an example pixel node. The Linear Associative Memory (LAM) node is a pixel node that performs a linear transform on the inputs it gets from its child information node. The results of the transform are the values of the high resolution pixels. The transform essentially consists of a matrix, that when multiplied with the information vector from the child node results in a vector of high resolution values. The formula is shown in equation (15) with A being the transformation matrix, x the vector of inputs from the child information node and b the vector containing the high resolution pixels.

$$A \cdot x = b \tag{15}$$

Training is required to obtain the values in matrix A. A single training pass is required to calculate these values in a RMS optimal way. The pseudo code for the important methods of the SRLAM class is listed in Code block 5.

```
SRInformationNode information
int hor, ver, len
float[][] matrix
float[][][] factor

trainStartPass(SRInfo info) {
    if (!information.isFullyTrained()) {
        information.trainStartPass(info)
    } else {
        hor = info.getHor()
        ver = info.getVer()
        len = information.getLength()
        matrix = new float[len + hor * ver][len]
    }
}
```

```
train(int x, int y, SRInfo info) {
    if (!information.isFullyTrained()) {
        information.train(x, y, info)
    } else {
        float[] input = information.getInformation(x, y);
        for (int i = 0;i < len;i++) {
            for (int j = 0;j < len;j++) {
                matrix[i][j] += input[i] * input[j]
            }
        }
        for (int dx = 0;dx < hor;dx++) {
            for (int dy = 0;dy < ver;dy++) {
                int hx = hor * x + dx
                int hy = ver * y + dy
                double output = high.getValue(hx, hy)
                for (int i = 0;i < len;i++) {
                    int index = len + dx + hor * dy
                    matrix[index][i] += input[i] * output
                }
            }
        }
    }
}

trainEndPass(SRInfo info) {
    if (!information.isFullyTrained()) {
        information.trainEndPass(info)
    } else {
        factor = new float[hor][ver][len]
        matrix = MatrixSolver.solve(matrix)
        for (int dx = 0;dx < hor;dx++) {
            for (int dy = 0;dy < ver;dy++) {
                int index = dx + hor * dy
                for (int i = 0;i < len;i++) {
                    factor[dx][dy][i] = matrix[index][i]
                }
            }
        }
    }
}

apply(int x, int y, SRPatch patch, SRInfo info) {
    patch.removeAll()
    patch.add()
    float[] input = information.getInformation(x, y)
    for (int dx = 0;dx < hor;dx++) {
        for (int dy = 0;dy < ver;dy++) {
            float output = 0
            for (int i = 0;i < len;i++) {
                output += input[i] * factor[dx][dy][i]
            }
            patch.setValue(dx, dy, output)
        }
    }
}
```

Code block 5 – SRLAM pseudo code

The hor, ver and len variables are used to store the horizontal scale factor, vertical scale factor and length of information respectively. The matrix array is used to store temporary results during training. The factor array is used to store the final values for the linear transform. Since there are usually more training samples than inputs from the child information node, the matrix is usually overdetermined. To solve equation (15) for A in a RMS optimal way, equation (16) is used.

$$X^T \cdot X \cdot A^T = X^T \cdot B \tag{16}$$

Matrix X in this equation contains the inputs from the child node for each training sample in each row. Matrix B contains the target high resolution values for each training sample in each row. The transpose of X multiplied with X only has a number of columns and rows equal to the number of inputs from the child node. The transpose of X multiplied with B only has a number of columns equal to the number of high resolution values and a number of rows equal to the number of inputs from the child node. These premultiplied values are stored in the matrix array, instead of the matrices X and B to reduce the required memory.

4.7 Directional Information Node

An information node is used as example in this subsection. The directional information node returns the edge directions in a local window represented as eight percentages. It calculates its information as follows. Both a horizontal and a vertical Sobel filter is applied to a local window centred on the target low resolution pixel. Each of the Sobel filters consists of a 2-dimensional kernel with a size of 3×3. They detect horizontal and vertical edges and the results of the two filters are combined forming the local edge direction. Each of these directions is classified into one of eight bins of directions. These eight bins span half a circle; directions falling in the other half are rotated by π. The percentages of directions falling in each of the eight bins are used as the directional information. Figure 35 shows a graphical representation of obtaining directional information.

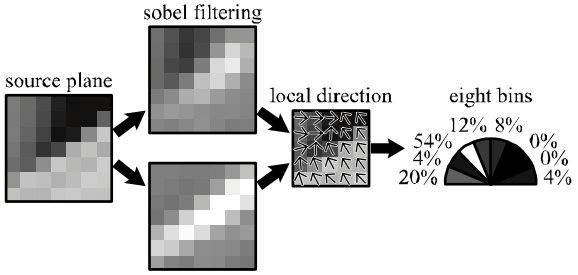


Figure 35 – directional information flowchart

The pseudo code for the important methods of the SRLocalDirection class is listed in Code block 6.

```
SRInformationNode child
int radius, dia, length
float[][][] buffer
```

```
int getLength() {
    return 8 * child.getLength()
}

getInformation(int x, int y, float[] information) {
    length = child.getLength()
    if (buffer == null) buffer = new float[dia][dia][length]
    for (int dx = -radius;dx <= radius;dx++) {
        for (int dy = -radius;dy <= radius;dy++) {
            int ix = x + dx
            int iy = y + dy
            int bx = dx + radius
            int by = dy + radius
            child.getInformation(ix, iy, buffer[bx][by])
        }
    }
    int pos = 0
    for (int i = 0;i < length;i++) {
        int[] bin = new int[8]
        for (int dx = 1;dx < dia - 1;dx++) {
            for (int dy = 1;dy < dia - 1;dy++) {
                float hor = buffer[dx][dy] ** SobelHor
                float ver = buffer[dx][dy] ** SobelVer
                float angle = atan2(hor, ver)
                int n = floor(angle / PI * 8)
                while (n < 0) n += 8
                while (n >= 8) n -= 8
                bin[n]++
            }
        }
        float div = dia * dia
        for (int j = 0;j < 8;j++) {
            information[pos++] = bin[j] / div
        }
    }
}

train(int x, int y, SRInfo info) {
    for (int dx = -radius;dx <= radius;dx++) {
        for (int dy = -radius;dy <= radius;dy++) {
            child.train(x + dx, y + dy, info)
        }
    }
}
```

Code block 6 – SRLocalDirection pseudo code

The child variable used in Code block 6 refers to the information node child which supplies the information that is converted to local directions. The radius, dia and length variables contain the radius of the local information, the diameter of the local information and the length of the child node information respectively. The buffer variable is used as a buffer for the information of the child node in the local area.

5 Test Results

A whole range of tests are performed using the modular super-resolution tool. While trying out different structures, it seemed that clustering based on local direction resulted in some good results. This component is therefore included in this approach.

The tested structure is designed to apply super-resolution in YCrCb colour space. The conversions from RGB to YCrCb and YCrCb to RGB are shown in Figure 36 and Figure 37 respectively and are implemented in the RGB to YCrCb image node. One part of the structure is trained for super-resolving the Y channel (luminance) and another part is trained for super-resolving both the Cr and the Cb channels (chrominance).

$$\begin{bmatrix} Y \\ Cr \\ Cb \end{bmatrix} = \begin{bmatrix} 0.256789 & 0.504129 & 0.097906 \\ 0.439215 & -0.367789 & -0.071426 \\ -0.148223 & -0.290992 & 0.439215 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0.0625 \\ 0.5 \\ 0.5 \end{bmatrix}$$

Figure 36 – conversion from RGB to YCrCb

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.164383 & 1.596027 & 0 \\ 1.164383 & -0.812969 & -0.391762 \\ 1.164383 & 0 & 2.017230 \end{bmatrix} \cdot \begin{bmatrix} Y-0.0625 \\ Cr-0.5 \\ Cb-0.5 \end{bmatrix}$$

Figure 37 – conversion from YCrCb to RGB

A base and detail step is added to both the luminance and the chrominance side of the structure. This often improves results, because the trained part will only be responsible for the details in the image and the training process is more general. The cubic spline kernel used for the basic enlargement is a standard cubic spline with the formula shown in Figure 38.

$$f(x) = \begin{cases} |x|^3/2 - |x|^2 + 2/3 & |x| < 1 \\ -|x|^3/6 + |x|^2 - 2|x| + 4/3 & 1 \le |x| < 2 \\ 0 & |x| \ge 2 \end{cases}$$

Figure 38 – cubic spline function

A range of clustering steps can be found in the structure next. Both the luminance and the chrominance side first perform a clustering step based on directional information and then a clustering step based on the differences with the neighbouring pixels. All clustering is based on a 2D Self-Organising Map (SOM). The sizes of the SOM vary between 2×2 and 4×4, resulting in 4 to 16 clusters in each step.

Each of the resulting clusters is finally linked to a LAM that performs a linear transform on the 5×5 local window around the low resolution pixel to obtain the high resolution pixels. This local window has its centre subtracted, to result in the 24 difference with neighbouring pixels.

A screenshot of the hybrid structure used for the results in this section is shown in Figure 39. Text is added to the screenshot to supply more information on the structure.

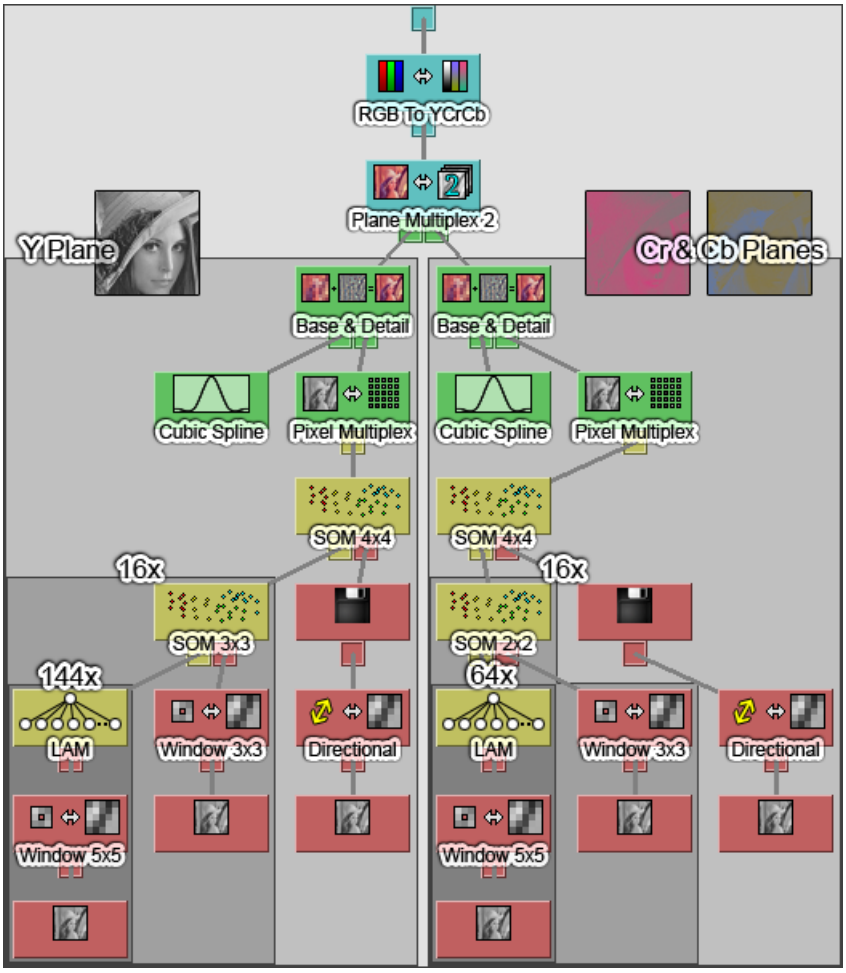


Figure 39 – hybrid structure screenshot with information overlay

The hybrid structure is trained using a training set consisting of 52 images with a size of 768×768 pixels. With the scale factor set to 2×2 this results in 7 667 712 training vectors in each channel. By creating three rotated and flipped versions of each training image, the structures were also trained with scale factor 4×4 and 7 667 712 training vectors in each channel. None of the images used for the results in this section are part of the training set. Three example training images and their flipped and rotated versions for the 4×4 scale factor are shown in Figure 40.

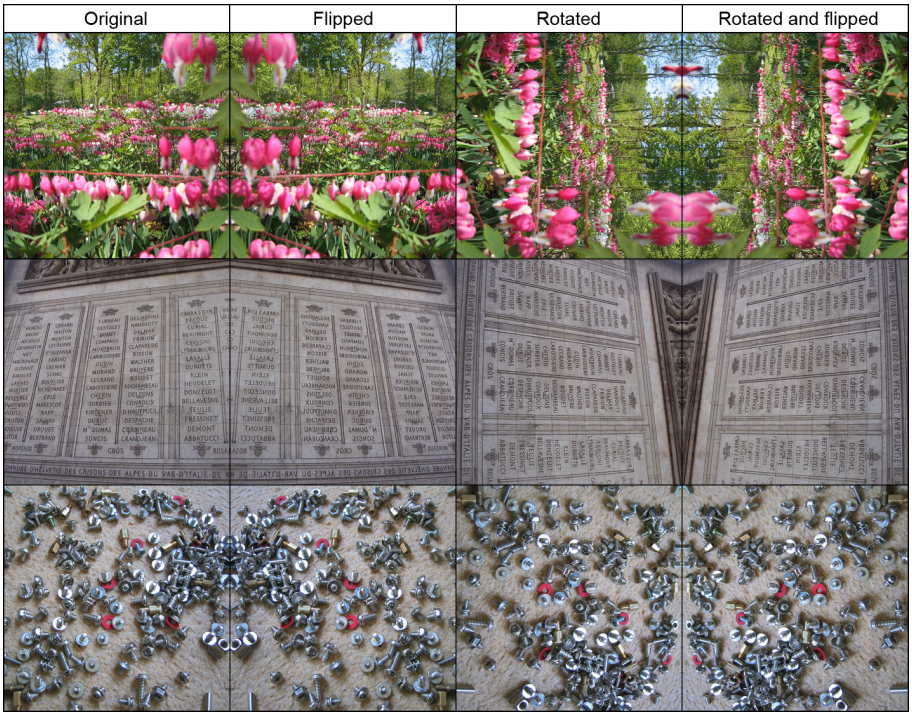


Figure 40 – example training set images

The training set images have been selected on amount of detail. Since sharp lines and other high frequency components are often most difficult to obtain in super-resolved images, the training set was selected to at least contain some of these high frequency components.

A super-resolution example on an image called sail is shown in Figure 41. The used scale factor is 4×4, so the super-resolved image contains sixteen times the pixels the original image contains. The example is made using the super-resolution tool presented in this thesis and an original image of size 512×512 that is shown in part (a). Part of the super-resolved image with a resolution of 2048×2048 is shown in part (b). Enlarged versions of the same part of the original and super-resolved image are shown in parts (c) and (d) of Figure 41 respectively. A grid is used as overlay to show which sixteen high resolution pixels come from one low resolution pixel.

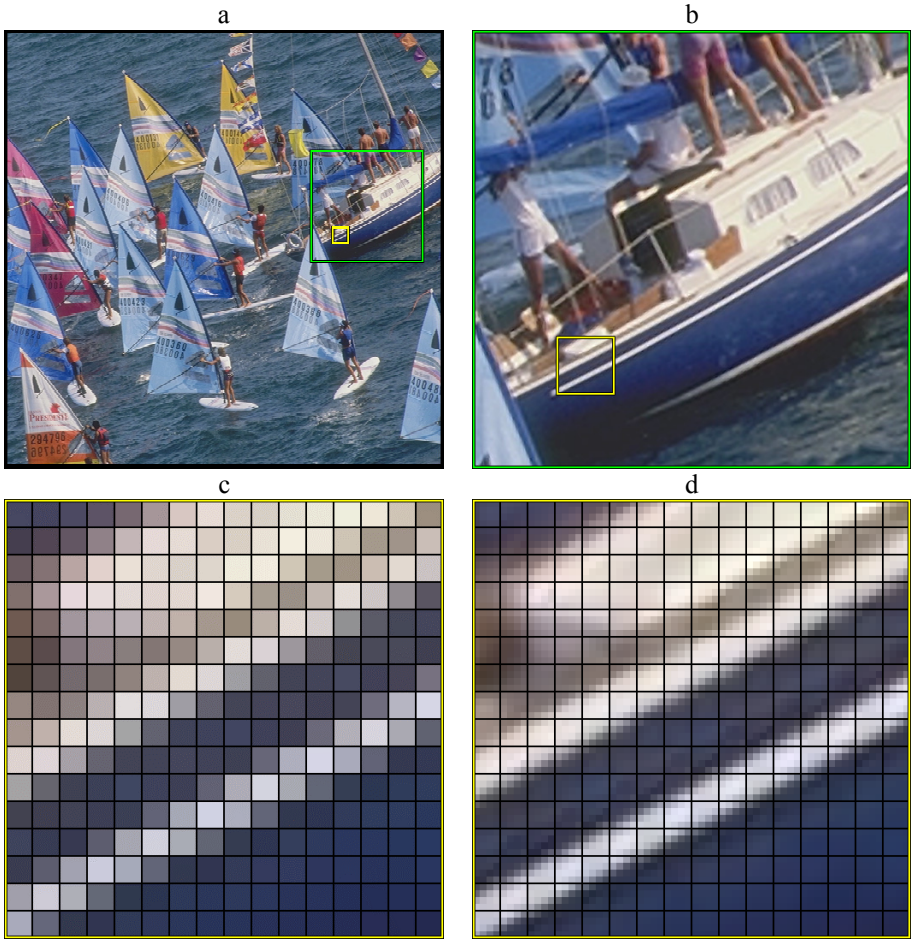


Figure 41 – modular super-resolution sail results (a) original image; (b) part of the super-resolved image; (c) enlarged part of the original image; (d) enlarged part of the super-resolved image

It can be seen that the diagonal line is kept fairly sharp and doesn't exhibit a staircase pattern. This can be explained by the super-resolution structure used. Since the area the line is part of has a strong diagonal orientation, all pixels in this area will be classified as having this orientation by the directional information and the attached SOM. The second SOM will classify the edges of the line and the selected LAM will interpolate the pixels as being on an edge with the orientation specified by the first SOM. This combination of the two clustering steps and a set of LAM's is specifically designed to detect both local orientation and edge pixels, which results in the sharpness and the lack of staircase patterns in the diagonal line.

Another example, using the monarch image, can be seen in Figure 42. The four sub images in the figure are obtained in the same way as in Figure 41. A third example with the same set of sub images can be seen in Figure 43. This third example is based on the mandrill image.

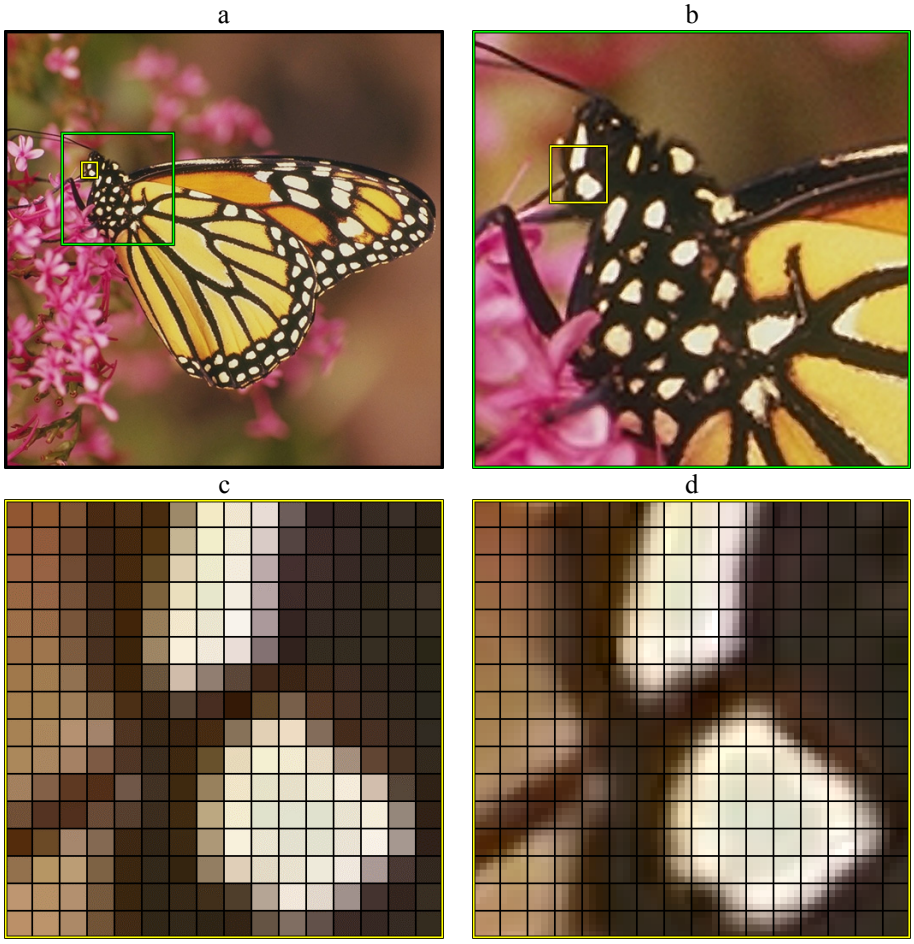


Figure 42 – modular super-resolution monarch results (a) original image; (b) part of the super-resolved image; (c) enlarged part of the original image; (d) enlarged part of the super-resolved image

The enlarged monarch image also shows some sharp edges. They are in some cases not as sharp as in the sail image, because the edge orientation can't be determined very precisely in the arbitrary shapes of the white spots of the butterfly.

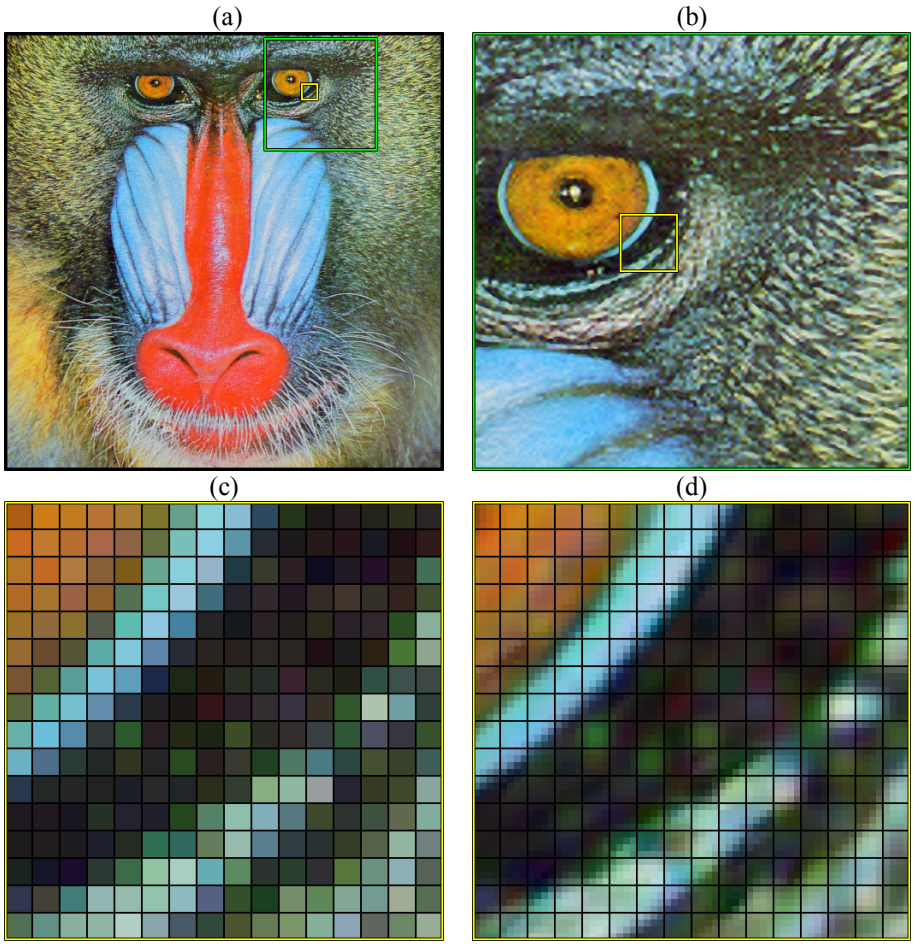


Figure 43 – modular super-resolution mandrill results (a) original image; (b) part of the super-resolved image; (c) enlarged part of the original image; (d) enlarged part of the super-resolved image

The enlarged version of the mandrill image shows sharper lines in combination with more smooth colour and intensity variations. It shows that the chosen super-resolution structure will not treat every transition as an edge. Some are interpolated similar to kernel interpolation, while others are treated as edges and interpolated accordingly. The blue rim around the mandrill's eye is for example kept sharp, while the colour variations in Figure 43(c) and (d) are interpolated more smoothly.

6 Conclusions

To improve the search for high quality single frame super-resolution algorithms, this research aimed to design and implement a tool that allows rapid combination of algorithmic parts of known state of the art super-resolution algorithms. Research into state of the art algorithms revealed some high quality approaches to image super-resolution and common parts between these approaches. It became clear that influences from several fields of computing, like signal processing, computer graphics and artificial intelligence, had found their way into super-resolution algorithms. Though several approaches to super-resolution share common parts, little to no reuse of these parts can be found among the researched approaches.

The design phase of this thesis results in a break-down into four base types of algorithmic parts. The amount of four base types and the ability to build simple bridges between these types accounts for a wide range of possible structures. This made sure that the tool allowed flexible construction. It is further found that most state of the art super-resolution algorithms can be broken down into an organised set of algorithmic parts to form a modular structure. Because of the presence of a base type that super-resolves full images, the image node, any full colour super-resolution algorithm can be converted into a modular structure with a single node. By use of specific nodes any greyscale super-resolution algorithm can also be converted into a modular structure. Though some approaches, like combining a base image and a detail image, required more thought, it was possible to include them into the modular design.

A whole range of different super-resolution structures has been constructed using the modular toolkit. The ease of construction and testing reduced the time needed to compare different approaches and inspired the search for good results using more unusual techniques. It finally resulted in the use of directional information in the clustering step. A technique from the field of texture classification was implemented as an information node and combined with an existing approach. This shows the modular super-resolution tool allows for flexible combination of existing super-resolution algorithms with new algorithmic parts. Easy exploration of the wide range of possible structures improves the search for high quality results. Since the window size parameter of the directional information could be easily adjusted in the toolkit, it was easy to find the optimal setting for this parameter.

The results of the tests show that clustering based on directional information performs well. This seems plausible, since pixels were clustered based on local direction. This means that each interpolator is trained for super-resolution of pixels with specific configurations of edge directions in the local area instead of pixels with specific relations to their neighbours. In general, it seems logical to select an interpolator based on the properties of the texture a low resolution pixel belongs to. The tests in this research show that at least texture classification based on local directions leads to some very convincing super-resolution results.

No interface metaphors were used, because of the small, highly educated target audience of the prototype tool. The combination of colours, shapes and icons in the main interaction window enables rapid recognition of components in exchange for a small learning time.

Though the modular super-resolution toolkit is easier to use than a programming language, the intended audience for the tool consists of people that are familiar with

state of the art super-resolution techniques. Experience in this field is needed to build structures that are sensible in terms of quality and performance. The current implementation is a proof of concept that allows flexible construction of high quality super-resolution algorithms using a graphical interface. It provides access to components like a Self Organising Map (SOM) without requiring any programming. Performance in terms of computing time and memory use were not major design objectives and are open for improvement in the future. Removing the constraint of a tree-based shape in the structure can for example reduce both the amount of required computation and the amount of required memory for some structures. A binary decision tree can reduce the computation in cluster lookups.

Another improvement of the modular approach would be the inclusion of additional algorithmic parts. Neural networks and data-dependent triangulation are example candidates that can be included as pixel node and plane node respectively.

Bibliography

[1] C.B. Atkins, C.A. Bouman, J.P. Allebach, Tree-Based Resolution Synthesis, Proceedings of IEEE ICIP, Apr 1999, pp. 405-410.

[2] C.B. Atkins, C.A. Bouman, J.P. Allebach, Optimal image scaling using pixel classification, Proceedings of International Conference on Image Processing, 2001, pp. 864-867.

[3] İ. Avcıbaş, B. Sankur, K. Sayood, Statistical evaluation of image quality measures, Journal of Electronic Imaging, Vol. 11, No. 2, Apr 2002, pp. 206-223.

[4] S. Battiato, G. Gallo, F. Stanco, A Locally-Adaptive Zooming Algorithm for Digital Images, Image Vision and Computing Journal, Elsevier Science. Inc., Vol. 20, Issue 11, Sep 2002, pp. 805-812.

[5] S. Battiato, G. Gallo, M. Mancuso, G. Messina, F. Stanco, Analysis and characterization of super-resolution reconstruction methods, Proceedings of SPIE Electronic Imaging 2003, Jan 2003.

[6] S. Battiato, G. Gallo, F. Stanco, Smart Interpolation by Anisotropic Diffusion, Proceedings of 12th International Conference on Image Analysis and Processing, 2003, pp. 572-577.

[7] S. Battiato, G. Gallo, S. Nicotra, Perceptive Visual Texture Classification and Retrieval, Proceedings of ICIAP 2003, pp. 524-530.

[8] M.J. Black, G. Sapiro, D.H. Marimont, D. Heeger, Robust Anisotropic Diffusion, IEEE Transactions on Image Processing, Vol. 7, No. 3, Mar 1998, pp. 421-432.

[9] F.M. Candocia, J.C. Principe, Superresolution of Images Based on Local Correlations, IEEE Transactions on Neural Networks, Vol. 10, No. 2, Mar 1999, pp. 372-380.

[10] J.F. Canny, A computational approach to edge detection, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 8, No. 6, 31717, pp. 679-698.

[11] H. Chang, D.Y. Yeung, Y. Xiong, Super-Resolution Through Neighbour Embedding, Proceedings on IEEE CVPR, Jun/Jul 2004, pp. 275-282.

[12] N.A. Dodgson, Quadratic Interpolation for Image Resampling, IEEE Transactions on Image Processing, Vol. 6, No. 9, Sep 1997, pp. 1322-1326.

[13] W.T. Freeman, E.C. Pasztor, Markov Networks for Super-Resolution, Proceedings of 34th Annual Conference on Information Sciences and Systems, Mar 2000.

[14] W.T. Freeman, E.C. Pasztor, O.T. Carmichael, Learning Low-Level Vision, International Journal of Computer Vision, Vol. 40, No. 1, Oct 2000, pp. 25-47.

[15] W.T. Freeman, T.R. Jones, E.C. Pasztor, Example-Based Super-Resolution, IEEE Computer Graphics and Applications, Vol. 22, No. 2, Mar/Apr 2002, pp. 56-65.

[16] K. Jensen, D. Anastassiou, Subpixel Edge Localization and the Interpolation of Still Images, IEEE Transactions on Image Processing, Vol. 4, No. 3, Mar 1995, pp. 285-295.

[17] K. Kinebuchi, D.D. Muresan, T.W. Parks, Image Interpolation Using Wavelet-Based Hidden Markov Trees, IEEE ICASSP, 2001.

[18] T. Kohonen, Self-Organizing Maps, Springer Series in Information Sciences, Vol. 30, Springer, Berlin, Heidelberg, New York, 1995.
[19] O. Kurşun, S. Joshi, O. Favorov, Single-Frame Super-Resolution by Inference from Learned Features, Istanbul University - Journal of Electrical & Electronics Engineering No. 2.
[20] X. Li, M.T. Orchard, New Edge-Directed Interpolation, IEEE Transactions on Image Processing, Vol. 10, No. 10, Oct 2001, pp. 1521-1527.
[21] D.D. Muresan, T.W. Parks, Prediction of Image Detail, Proceedings of IEEE ICIP, Sep 2000.
[22] D.D. Muresan, T.W. Parks, Image Interpolation Using Adaptive Linear Functions and Domains, IEEE 2001 Western New York Image Processing Workshop, 2001.
[23] D.D. Muresan, T.W. Parks, Optimal Recovery Approach to Image Interpolation, Proceedings of IEEE ICIP, 2001.
[24] D.D. Muresan, T.W. Parks, Adaptive, Optimal-Recovery Image Interpolation, IEEE International Conference on Acoustics, Speech, and Signal Processing, Vol. 3, 2001, pp. 1949-1952.
[25] D.D. Muresan, T.W. Parks, Adaptively Quadratic (AQua) Image Interpolation, IEEE Transactions on Image Processing, Vol. 13, No. 5, May 2004, pp. 690-698.
[26] S. Nicotra, Organizing Texture in a Perceptual Space, Eurographics Italian Chapter, Milan, Italy, July 11-12- 2002.
[27] K. Palaniappan, J. Uhlmann, D. Li, Extensor-Based Image Interpolation, Proceedings of IEEE ICIP, Sep 2003, pp. 945-948.
[28] D. Ryder, O.V. Favorov, The new associationism: A neural explanation for the predictive powers of cerebral cortex, Brain and Mind 2, pp. 161-194.
[29] C. Staelin, D. Greig, M. Fischer, R. Maurer, Neural Network Image Scaling Using Spatial Errors, HP Laboratories Israel, Oct 2003.
[30] A.J. Storkey, Dynamic Structure Super-Resolution, Advances in Neural Information Processing Systems 15, 2003, pp. 1295-1302.
[31] D. Su, P. Willis, Image Interpolation by Pixel Level Data-Dependent Triangulation, Computer Graphics Forum, Vol. 23, No. 2, 2004.
[32] M.F. Tappen, B.C. Russell, W.T. Freeman, Exploiting the Sparse Derivative Prior for Super-Resolution and Image Demosaicing, 2003.
[33] Z. Wang, A.C. Bovik, H.R. Sheikh, E.P. Simoncelli, Image Quality Assessment: From Error Visibility to Structural Similarity, IEEE Transactions on Image Processing, Vol. 13, No. 4, Apr 2004, pp. 600-612.
[34] X. Xu, L. Ma, S.H. Soon, C.K.Y. Tony, Image Interpolation Based on the Wavelet and Fractal, International Journal of Information Technology, Vol. 7, No. 2, Nov 2001.
[35] X. Yu, B.S. Morse, T.W. Sederberg, Image Reconstruction Using Data-Dependent Triangulation, IEEE Computer Graphics and Application, Vol. 21, No. 3, May/Jun 2001, pp. 62-68.
[36] http://www.benvista.com
[37] http://www.kneson.com
[38] http://www.ddisoftware.com
[39] http://www.dmmd.net
[40] http://www.extensis.com
[41] http://www.eclipse.org
[42] http://java.sun.com