Hochschule Bochum
Bochum University
of Applied Sciences
Campus **Velbert/Heiligenhaus**

Christof Kaufmann

Design of a Framework for Image Data Fusion Algorithms and Implementation of a Dictionary Learning Algorithm

Master Thesis

In the degree programme *Mechatronics and Information Technology*
In partial fulfillment of the requirements for the Degree of Master of Engineering

Advisor:	Prof. Dr. rer. nat. Marco Schmidt
Co-Advisor:	Prof. Dr. rer. nat. Jörg Frochte

Submitted: May 18, 2017

Declaration of Authenticity

I declare that all material presented in this paper is my own work or fully and specifically acknowledged wherever adapted from other sources.

I confirm that I have read and understood the valid examination regulations regarding non-attendance, withdrawal, cheating, infringement of regulations.

This work has not yet been submitted in equal or similar way for another degree or diploma at any university or other institute of tertiary education.

Heiligenhaus, April 19, 2020

(Christof Kaufmann)

Author and copyright

Christof Kaufmann
christof.kaufmann@hs-bochum.de
Version: April 19, 2020

Abstract

This work presents a software framework for satellite image data fusion algorithms and describes its goals and usage. Additionally to already existing algorithms, one new dictionary learning algorithm is extensively described, implemented and tested in the framework.

The *imagefusion* framework presented here serves as lightweight basis for the implementation of image fusion algorithms. It is designed to be simple and effective, extensible and efficient. The usage of C++11 allows to have a modern design combined with high computational performance and great availability and interoperability of third-party libraries for different purposes. The framework makes use of object orientated and generic programming in situations in which it is reasonable. The core of the framework is formed by the library `libimagefusion`. It contains the whole functionality that an implementation of an image fusion algorithms requires. The library is accompanied by the included parallelized image fusion algorithms – currently three – and utilities. The utilities are handy for common tasks like preparing images for input into an algorithm or comparing an output image with a reference image to optimize the algorithm further. There are also utilities to fuse images with the included algorithms. This makes it easy for users to try out an algorithm from the framework and compare it with other external algorithms. In general the entry threshold is kept low, so new users can quickly start to work and contribute or to develop new algorithms. A detailed API documentation helps new users and serves experienced users as reference.

The second part discusses, implements and extends a dictionary learning algorithm, named "SParse-representation-based SpatioTemporal reflectance Fusion Model" (SPSTFM). SPSTFM is a complex image fusion algorithm based on overcomplete dictionaries and sparse coding. The implementation in this frameworks provides lots of options to adapt the algorithm to the users needs and is equipped with good default options. Many of these options arose from ambiguities in the algorithm or because there was room for improvement to make optional extensions. A large part of this work tests different combinations of options to find the best default options and show their behaviour. Finally some real images allow for a comparison of SPSTFM with the other included algorithms.

Contents

1 Introduction

Image fusion is a very interesting technique to combine multiple images with the intent to increase the image quality or frequency. Satellite images of fine resolution from the Landsat satellite are roughly available every two weeks. By combining these with daily coarse resolution images image fusion allows to have daily fine-resolution images. This enables applications that would not be possible without image fusion. Especially crop monitoring and optimization relies on fine-resolution images with an almost daily frequency. However, the focus of this work is the image fusion itself rather than models that require image fusion. Chapter 2 gives a more detailed explanation about image fusion and also gives a general overview of related topics.

Using image fusion is quite hard because of practical problems. Firstly, satellite images are required. This is not a big problem, because these are freely available, though it is not trivial to find the right ones and to handle the formats. One real problem is that there is currently no free image fusion framework available. For some algorithms there are single reference implementations available, but these rely on special image formats or require a non-free (expensive) program to run, which also depend on the platform. For other algorithms there is not even a reference implementation available. These circumstances make it very hard to compare different algorithms with each other. But comparison is important to decide which algorithm is the best for one specific application. There are differences in quality, but also in computation time.

This work presents a free framework for image fusion algorithms. It is easily extensible, tries to be efficient and can serve as a basis to do research for new or modified algorithms. It is also easy to try out an included algorithm on an image set. In Chapter 3 the framework is extensively discussed. In addition the code is freely available with a detailed doxygen documentation. Chapter 4 goes into details about the main component: `libimagefusion`. This is the heart of the framework and certainly its most important component. It contains the core classes to implement image fusion algorithms. These perform input / output handling, image management and provide a common interface for image fusion algorithms.

Currently, three algorithms are already implemented: STARFM, ESTARFM and SPSTFM. The latter algorithm is also part of this work. It is a learning-based algorithm, which trains a double-dictionary, similarly as in super resolution applications. For SPSTFM there was no reference implementation available and it also requires sub-algorithms for implementation. The background, including equations, algorithms and options, is discussed in Chapter 5. There are also experiments included, which compare different options. The tests show also the typical performance of SPSTFM itself and in comparison to STARFM and ESTARFM.

Finally Chapter 6 draws a conclusion and gives an overview about further work. As often with software a lot of improvements are possible and thus it never seems to be finished.

2 Image Fusion Basics

This chapter gives an overview about the basic principles of topics related to image fusion. First, the purpose of data fusion in general is roughly described. Then, Section 2.2 describes how images can be represented in a computer. In Section 2.3 follow some properties of satellite images and how they can be structured for different image fusion algorithms. The algorithms that are mentioned in this work are also categorized briefly in the end.

2.1 Data Fusion

This work is about image fusion. Image fusion is an area in the discipline of data fusion. Data fusion describes the process of combining multiple data sources to form an artificial data source with increased accuracy. Often the data sources are sensors. Examples include voltmeter, distance sensors or image sensors. Sensor measurements have always parasitic errors, but can also have a bad resolution in time or quantity. Sometimes there might be no value at all for some samples. In the here considered application – satellite images – this could happen because of clouds. With sensor data fusion multiple sensors can be combined with an appropriate model to form a superior sensor. In this work only very general methods are considered, where no underlying physical model is applied, but a mathematical. Actually an algorithm itself can be considered as a model.

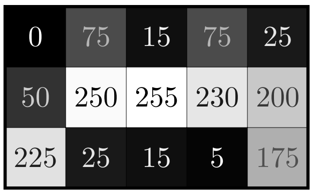
2.2 Digital Images

Image fusion is data fusion with image sensors, i. e. digital cameras. So the data to fuse are rasterized images. Rasterized images consist of a raster of pixels. The pixels are indexed by (x, y)-coordinates, where the top left pixel is at $(0, 0)$. Pixels have in general a rectangular shape and often the shape is quadratic. This work only considers quadratic pixels.

Pixels hold scalar or vector values. Vector values mean that a each pixel contains multiple values from different frequency ranges or bands, often colours. A vector-valued image can be split into multiple scalar-valued images. These are called channels. Ordinary colour images can be represented with three channels; one for red, one for green and one for blue. This is also called RGB image. Generally, such images are called *multi-channel images* and the ones with scalar values *single-channel images*.

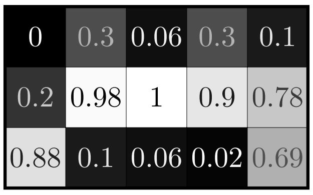
For technically the values are limited in range or precision. Because of efficiency the values are often integer numbers. Consumer applications and standards use mainly 8-bit unsigned values, i. e. $\{0, \dots, 255\}$. The satellite images considered in this work use 16-bit signed or unsigned type, but do not use the full available range.

A pixel value often represents the brightness of the corresponding band in the pixel area. So for a single-channel image that represents a grey scale image, pixels with a high value



0	75	15	75	25
50	250	255	230	200
225	25	15	5	175

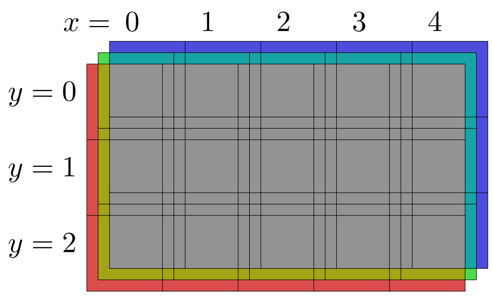
(a) 8-bit unsigned integer data range: 0 (black) to 255 (white).



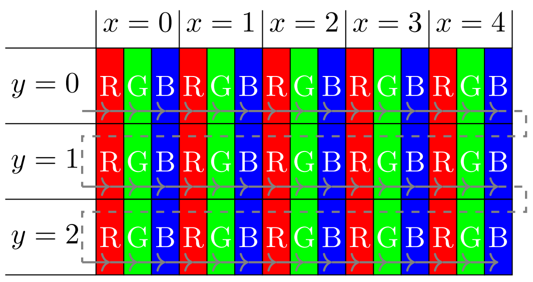
0	0.3	0.06	0.3	0.1
0.2	0.98	1	0.9	0.78
0.88	0.1	0.06	0.02	0.69

(b) Floating point data range: 0 (black) to 1 (white).

Figure 2.1: Gray scale images with given brightness values in different data types.



(a) A three channel rasterized image, where the three channels are interpreted as red channel, green channel and blue channel.


$x = 0$ $x = 1$ $x = 2$ $x = 3$ $x = 4$
$y = 0$ R G B R G B R G B R G B R G B
$y = 1$ R G B R G B R G B R G B R G B
$y = 2$ R G B R G B R G B R G B R G B

(b) Memory layout of an RGB image. From inside out: iterate first channels, then x, then y.

represent bright areas and pixels with a low value represent dark areas. Nevertheless, the value range depends on the data type used for representation. Figure 2.1 shows two example images, where similar brightnesses are given in different data types. While an 8-bit unsigned type like in Figure 2.1a requires 1 byte per pixel, a floating point image like in Figure 2.1b usually requires 4 bytes per pixel or for double precision 8 bytes per pixel. In addition computation with floating point types might require more time. Another possibility would be a fixed point type, which is basically an integer with a fixed divisor. However, this is more interesting for readability than for computation.

For RGB images each pixel has three elements and gives the brightness for each colour band. The channels are then blended to give the colour the pixel represents. This is illustrated in Figure 2.2a as three layers, which all have half brightness and thus add up to grey. With full brightness they would add up to white, with no brightness to black and with different brightness values in the channels they would become (non-grey) coloured. E. g. red and green without blue gives yellow etc., as can also be seen on the border of the figure.

To save a multi-channel image in a computer's memory, the order of the indices (x, y and channel) has to be determined. There might be small timing differences, depending on how the values are accesses but in general all orders are possible. Throughout this work a memory layout like done in the OpenCV library [Its17a] is assumed: first channels are grouped, then horizontal neighboring pixels and finally lines. This is clarified in Figure 2.2b. In reality memory has no two dimensional shape. It only has one dimension, which is indexed by the address. So the image is practically saved in one long line. The grey arrows in the figure

Table 2.1: Typical structure of available images.

date	0	1	⋯	15	16
resolution					
fine	🟦	—	⋯	—	🟦
coarse	🟦	🟦	⋯	🟦	🟦

(a) Structure of images for STARFM.

date	i	j
resolution		
fine	🟦	□
coarse	🟦	🟦

(b) Structure of images for ESTARFM or SPSTFM.

date	i	j	k
resolution			
fine	🟦	□	🟦
coarse	🟦	🟦	🟦

should indicate that.

2.3 Image Structure

All image fusion algorithms mentioned in this work, consider satellite images of two different satellites; MODIS and Landsat. On the one hand their images differ in spatial resolution, which means the pixel size (width or height) is different. For MODIS it can be $250 - 1000\,\mathrm{m}$ and for Landsat $10 - 30\,\mathrm{m}$. In this work $250\,\mathrm{m}$ in MODIS images and $30\,\mathrm{m}$ in Landsat images are assumed. These will be referred to by *coarse* (also: *low*) and *fine* (also *high*) resolution. On the other hand the temporal resolution is different. This means that the days between flyovers (and capturing an image) is different. MODIS captures one image per day and Landsat one per 16 days. Sometimes the time gap between Landsat images is less (e. g. 7 days), when the image region is overlapping.

So, when acquiring images over a 16 days period a typical situation of available images is depicted in Table 2.1. So on dates 0 and 16 images of the same geographic area are available in both resolutions. On the dates in between only the coarse resolution images are available. However, some applications require a fine spatial resolution and a temporal resolution of 1 day. For those the fine resolution images at dates $1 - 15$ are missing.

This is where image fusion comes into play. It aims to combine different images to generate (also: *fuse*, *predict*) the missing images (also: products). Different image fusion algorithms require a different set of input images. For example requires STARFM [GMSH06] three input images, namely fine and coarse resolution at the same date i and coarse resolution at date j to fuse a fine resolution image at date j. This is depicted in Table 2.2a, where i could be e. g. 0 and $j \in \{1, \dots, 15\}$. This also allows to predict the fine resolution image for the present day. Algorithms like ESTARFM [ZCG+10] or SPSTFM [HS12] require five input images. They require fine and coarse resolution at the dates i and k and coarse resolution at date j in between i and k to fuse a fine resolution image at date j. This is depicted in Table 2.2b, where e. g. i could be 0, k could be 16 and $j \in \{1, \dots, 15\}$. These algorithms only allow to fuse a fine resolution image for a past day, since an image pair from a later date k than the fusion date j is required.

The fused image is just a kind of extrapolation and thus only an approximation to a real (non-existing) image. Though, to measure the quality (or the exact error) of a fused image, a real image in fine resolution for the prediction date must be available. So for testing purposes one could choose $i = 0$, $j = 16$ and $k = 32$. Then the fused fine resolution image and the real fine resolution image can be compared to determine the error.

Image fusion algorithms can be categorized according to the way the algorithm works. Here three categories are mentioned:

- Reconstruction-based algorithms, like STARFM and ESTARFM. These kind of algorithms go through the fine resolution image and for each pixel they look at the time difference of nearby similar pixels in the coarse resolution images to predict the change of that pixel.

- Transformation-based algorithms, based e. g. on wavelet or tasseled cap transformations.

- Learning-based algorithms, like SPSTFM. These kind of algorithms try to learn or train – a series of maybe incomplete optimizations – an entity. This entity, which is in SPSTFM a dictionary-pair, is then used for the fusion.

This work will focus on the SPSTFM algorithm and use STARFM and ESTARFM only for comparison of the quality.

3 Image Fusion Framework

This chapter gives an overview about the imagefusion framework and describes some details. The first section describes the goals of the framework. Then, Section 3.2 gives an overview about the framework. It describes what is included in the framework and how it can be used. Next, Section 3.3 explains the utilities that come along with the library and what utilities are planned for the future.

3.1 Goals

There are several goals that led the design of the framework. One key goal is efficiency. Image fusion algorithms are often computational expensive and thus efficiency is very important. At the time of planning this framework, different image fusion algorithms were only available as reference implementations or as implementations in non-compiled languages. These were either not easy to use or inefficient. So with efficiency being a key-goal for the image fusion framework C++11 has been chosen as programming language of choice. C++11 allows a modern design and at the same time being more efficient than the predecessor C++03.

Another goal is generality. So the framework is suitable for different image fusion algorithms and also easily extendable for not yet included algorithms. This allows scientists to write new image fusion algorithms without requiring to compile the image fusion library. Also parallelization is supported in a general way, which means that a new image fusion algorithm can be parallelized, again without the requirement for a compilation of the library. For some algorithms parallelization works out of the box.

The previous goals also contain parts of the following goal: effectivity. This means the framework should make it easy for users to complete their tasks. To be effective on the one hand the library has a good interface, which tries not to restrict the user. On the other hand there are utilities to provide an easy way to fulfill common tasks or try out included algorithms.

The last goal is connected to all previous goals: reusability. For this framework, besides object oriented programming, reusability is also meant from a user's perspective. So the framework heavily relies on existing, efficient and effective libraries and tries to act as common harmonized interface. This allows seamless interoperability between the libraries, but without preventing the user from a direct access to these libraries.

3.2 Overview

The image fusion framework consists of several components. The core of the framework is the image fusion library `libimagefusion`. It contains everything that is required to write

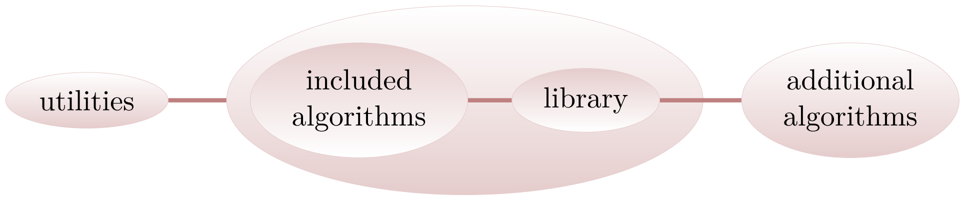


Figure 3.1: Overview of framework components

image fusion algorithms. `libimagefusion` is described in Chapter 4 in more detail.

As part of this thesis and another thesis [Kla16], some image fusion algorithms are already implemented within the library. These are

- STARFM,
- ESTARFM and
- SPSTFM.

The latter is part of this thesis and described extensively in Chapter 5. For each algorithm there exists a command line utility to fuse images, see Section 3.3. However, the algorithms can also be used in a program, e. g. to make a graphical user interface or a smartphone app. The framework allows users to implement additional algorithms in exactly the same way as the included ones. For this, the library has not to be recompiled, but only linked. This allows a direct start and makes contributing algorithm implementations easier for users. All the described components are depicted in Figure 3.1. With its simple but effective design the framework should grow to a collection of image fusion algorithms.

One large component that is not shown in the overview, but present in every component, is the documentation. The library comes with an extensive API documentation and tutorials written as doxygen comments [vH16]. Documentation is considered as very important for the success of this framework. The utilities come with a usage documentation, which is generally accessible with the option `--help`.

All of that comes along with this work as a git repository. This makes it easy for users to contribute. The repository also contains a readme-file, which contains the dependent libraries and has instructions of how to compile the framework and documentation. As build manager CMake is used. It can even generate Linux packages and Windows installers. In general the code base should have low threshold for new users. The readme is therefore written in kind of tutorial style.

3.3 Utilities

The utilities are a collection of command line tools. Their purpose is to fulfill common tasks, such as using an image fusion algorithm, preparing images for that or comparing results. All utilities have a common style for their option interface, which is described in Section 4.3. This includes that all options that are possible on command line, can be put into a file and this can be used with `--option-file=<file>`.

For each of the included implementations of STARFM, ESTARFM and SPSTFM a utility will be provided to fuse images with it. These utilities are not described here, since they are very similar in usage, but have different algorithm specific options. The `--help` options describe all possible options. Listing 4.1 gives an example how to specify date and resolution tag for an image.

Instead two other utilities are described in more detail. *Image geo crop* is presented in the next section and *image compare* in Section 3.3.2. What they have in common is that both require images as inputs. These can be specified by their filename, e. g. for image geo crop:

```
$ imggeocrop landsat1.tif modis1.tif
$ imggeocrop -i landsat1.tif -i modis1.tif
$ imggeocrop --img=landsat1.tif --img=modis1.tif
```

Instead of just giving the plain file names of the images as arguments, one can also use the `-i <file>` or `--img=<file>` option, as can be seen above. It is also possible to use just a single channel from a multi-channel image as input or to mix them up. This can be done with nested options or sub-options. Nesting can be done with `'...'`, `"..."` or (not as first level on bash) `(...)` (see also Section 4.3). To read in for example only channels 0 and 2, one can specified these with the nested option `-l <num-list>` or `--layers=<num-list>`. But then also the `-f <file>` or `--file=<file>` option has to be used to specify the file name:

```
$ imgcompare --img='-f landsat1.tif -l (0 2)' --img='-f modis1.tif ↲
    -l (0 2)'
$ imgcompare --img='--file=landsat1.tif --layers=(0 2)' ↲
    --img='--file=modis1.tif --layers=(0 2)'
```

It would also be possible two use twice the same channel, e. g. `-l (0 0)`, which would also work with a single-channel image. Now, if only a part of an image should be used, that can be specified as well. For this the `-c <rectangle>` or `--crop=<rectangle>` sub-option accepts `-x <num>`, `-y <num>`, `-w <num>` (or `--width=<num>`) and `-h <num>` (or `--height=<num>`) sub-sub-options. These are always in pixel coordinates. For example

```
$ imggeocrop --img='--file=img1.tif --crop=(-x 20 -y 20 -w 400 -h ↲
    400)' --img=img2.tif
```

Would use only the 400×400 image region, starting at $(20, 20)$ for `img1.tif`. Note, all of these options work (currently) for all utilities.

3.3.1 Image geo crop

Image geo crop (`imggeocrop`) is a utility to crop images to the same geographical extends and scale the resolution appropriately. So for example using

```
$ imggeocrop landsat1.tif modis1.tif
```

will rescale the coarse resolution image (independent from the order of the input images) to the fine resolution with bilinear interpolation. Furthermore it computes the the common geographical region of both images. The images are then cropped – if appropriate. Cropping is done with subpixel accuracy, again with bilinear interpolation. The cropped image(s) are output with a new file name. By default a prefix `cropped_` will be used. The naming scheme can be changed with the options `--prefix=<file-prefix>` and `--postfix=<file-postfix>`. Afterwards the images will have exactly the same geographical extents and also the same image size. Note that images that do not need to be cropped will not be copied.

Nevertheless there are some requirements for the images. The images must have the same geographical coordinate system. The utility is not yet able to reproject the coordinate system. Though this is a planned feature. So currently the example above requires a reprojection with an external tool, since coordinate systems are different for Landsat and MODIS images. GDAL [GDA16] provides utilities to reproject images. The images must also have a valid coordinate system at all. This requirement might be relaxed, too, in a future release.

3.3.2 Image compare

Image compare (`imgcompare`) is a utility mainly to measure the quality of fused images, but it can also extract properties or make plots. Usually it compares two single-channel images by various error means (mean absolute difference, root square mean error, etc.):

```
$ imgcompare img1.tif img2.tif
```

It is also possible to use a single input image:

```
$ imgcompare img1.tif
```

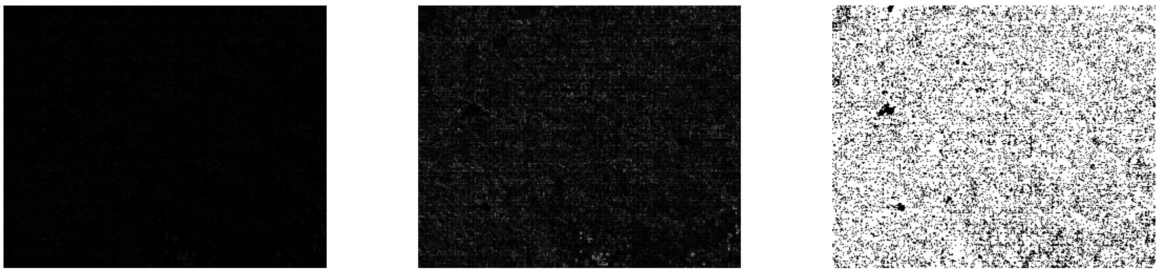
However, not all options make sense with a single image and thus are not possible.

Generally, when using two input images, they should have the same size. If they differ in size, image compare uses a correlation to find the best match of the images and crops the larger image at the found location to the size of the small image. It also prints the found crop window as information to the user. However, this is only a guess and it might take a lot of time. If the user knows the exact crop location, he/she can specify it with sub-option `--crop`, as described in Section 3.3. Cropping both images is of course possible to compare only a specified region. The same holds when using a single input image.

For comparing images and see local differences, image compare can output three kinds of absolute difference images:

- A plain difference image $I_D = |A - B|$ with the same data date as the source images. This is useful for further analysis, since the result is left unchanged, but hence maybe inappropriate for visual inspection. The option for this is `--diff=<out-file>`. An example is shown in Figure 3.2a.

- A scaled difference image with maximized contrast. This is $I_S = \frac{I_D - \min I_D}{\max I_D - \min I_D} \cdot m$, where m is the maximum of the image data range. For integer data types m is the greatest possible value (e. g. $m = 255$ for 8 bit unsigned integers) and for floating point data types is $m = 1$. The data type is the same as the data type of the source images. This kind of difference image can be used to easily visualize the local quality of the image though it actually does not contain more information than the non-scaled version. The option for this is `--diff-scaled=<out-file>`. An example is shown in Figure 3.2b, which is just Figure 3.2a with maximized contrast.



(a) Plain absolute difference image.

(b) Absolute difference image scaled for maximum contrast.

(c) Binary difference image.

Figure 3.2: Examples for several kinds of difference images.

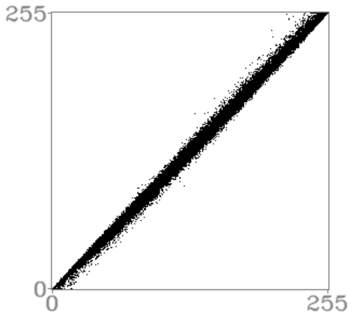


Figure 3.3: Scatter plot with quite low deviations.

- A binary difference image. It will output an 8 bit unsigned integer image I_B with

$$I_B(x,y) = \begin{cases} 255 & \text{if } I_D(x,y) \neq 0 \\ 0 & \text{if } I_D(x,y) = 0 \end{cases} \tag{3.1}$$

  for all (x, y). This is useful to see directly where differences are and where not. The option for this is `--diff-bin=<out-file>`. An example is shown in Figure 3.2c.

In addition to the difference images, image compare can also output a scatter plot with the option `--out-scatter=<out-file>`. A scatter plot visualizes the relation between input images A and B, i. e. $A \times B$ for all coordinates. So the values of A are on the horizontal axis and the values of B of the respective same locations on the vertical axis. To make it more clear, let $a := A(x, y)$ and $b := B(x, y)$. Then the scatter plot will set a marker at (a, b). This implies that if the images are the same, the scatter plot will only have markers in the diagonal. The further away the markers are from the diagonal the more different are the images. One example plot is shown in Figure 3.3. In this figure it is easy to see that the images are very similar, since all scatter dots are near the diagonal. Obviously this makes only sense for two input images.

Also very helpful for a single input image is a histogram plot. A histogram maps a set of intervals (called *bins*) to the number of pixels having a value in the corresponding intervals.

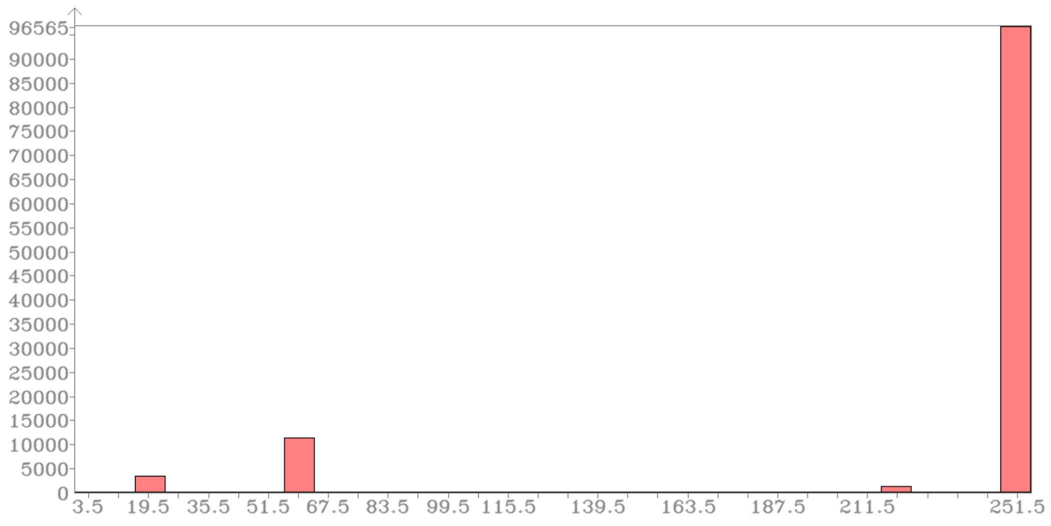


Figure 3.4: Histogram of Figure 5.2a, plotted with the image compare utility.

With a histogram plot it is easy to see how the values in an image are distributed. Image compare has four options to plot histograms:

- `--out-hist-first=<out-file>` will plot the histogram of the first input file. This options works also if there is only a single input image.
- `--out-hist-second=<out-file>` will plot the histogram of the second input file.
- `--out-hist-both=<out-file>` will plot the histograms of both input files into one plot.
- `--out-hist-diff=<out-file>` will plot the histogram of the plain difference image. When comparing two images, this allows to visualize the error distribution.

Note, all of the above `<out-file>` parameters can be image file names (e. g. `hist.png`) or CSV (character separated value) file names with the extension `.csv` or `.txt` (e. g. `hist.csv`). The latter is interesting if the plot should be done with an external tool, which is used for the histogram plots in Section 5.6. To show the image output at least once, Figure 3.4 is the histogram plot of Figure 5.2a.

There are also some options to modify the output of the plots. The number of histogram bins can be changed with `--hist-bins=<num>`. By default this is set to 32. A logarithmic scale for the histogram count is available with `--hist-log`. By default a linear scale is used. When using an image output file the size can be changed with `--hist-size=<size>`, e. g. `--hist-size=500x1000`. This specifies the histogram plot size in pixel (without axis and ticks etc.).

Apart from that it is also possible to switch on grids with `-g` or `--enable-grids` and legends with `-l` or `--enable-legends`. These options apply to the scatter plot as well as to the histogram plot.

Sometimes an image contains invalid values. For satellite images there might be clouds or there can be a border without image content (from image rotation). To restrict the comparing only to the valid values, image compare allows to specify masks. The most simple form is to use a mask file, which is an 8-bit unsigned single-channel image. Such a file can be given with the option `-m <img>` or `--mask-img=<img>`. Also multiple mask files can be specified to restrict the valid locations further. Then image compare considers only locations where all masks are valid. This might be useful if both images come with a mask, like:

```
$ imgcompare img1.tif img2.tif -m mask1.tif -m mask2.tif
```

If no mask image is available, there is another way to specify a mask. Often invalid pixels are marked with special values, like -9999. These or any other known invalid values can be excluded with the option `--mask-invalid-ranges=<range-list>`. This will create a mask on-the-fly. For example to exclude all negative values, the user can specify:

```
$ imgcompare img1.tif img2.tif --mask-invalid-ranges=(-infinity,0)
```

However, the same can be achieved by specifying only the non-negative numbers as valid range with `--mask-valid-ranges=<range-list>`:

```
$ imgcompare img1.tif img2.tif --mask-valid-ranges=[0,infinity)
```

In this example a square bracket has been used for the lower bound to include it in the range. So $[a, b]$ is the range from a to b including a and b and (a, b) excluding a and b. Multiple ranges can be specified and both options can be combined. So to allow all non-negative numbers, except 42 and everything greater than 23456, the following can be used:

```
$ imgcompare img1.tif img2.tif --mask-valid-ranges=[0,inf) ↲
     --mask-invalid-ranges='[42,42] (23456,inf)'
```

If additionally one or more mask files are used, the invalid and valid (!) ranges only restrict it further. So using `-m mask.tif --mask-valid-ranges=(-inf,inf)` will have the same effect as only using the mask file.

Image compare always shows some statistics, like mean error and standard deviation, minimum error and maximum error. For a single input file the mean and standard deviation are also shown, but there is no error measure. However, the minimum and maximum values are shown, which can also be of interest.

4 libimagefusion

This chapter describes the core library of this framework `libimagefusion` and gives an overview of its most important classes. The library tries to keep things simple, but powerful. To be efficient and effective, it is written in C++11. Being a compiled language C++ allows fast execution times. It is also a widely used language so there is a large choice of available (external) libraries. Section 4.1 will explain the design and give rationales for fundamentals. It presents the core classes that are essential for every image fusion algorithm. Then, in Section 4.2, a brief introduction about how geo information can be used with `libimagefusion`. Finally, the build-in support for option parsing as required in every utility is presented in Section 4.3.

4.1 Image Fusion

This section describes some significant core classes, which are required to write an image fusion algorithm. They are depicted with their relationships in Figure 4.1. Before going into details, a short overview explains the purpose about each core class:

`Image` represents an image. It can be used for image processing (also with OpenCV) and reading and writing image files.

`MultiResImages` is used as structured image storage. It holds all the input images that an image fusion algorithm requires. The images are marked by date and resolution tag.

`DataFusor` is the base class for all image fusion algorithm implementations. This provides a common interface to set up and run an algorithm.

`Options` is the base class for all algorithm option classes. It serves as interface for `DataFusor` and also holds the prediction area option.

`Parallelizer` can be used as a meta data fusor to run a `DataFusor` in parallel. For that it runs multiple instances of an underlying `DataFusor` to fuse separate parts of an `Image`.

Next, the classes are discussed in more detail.

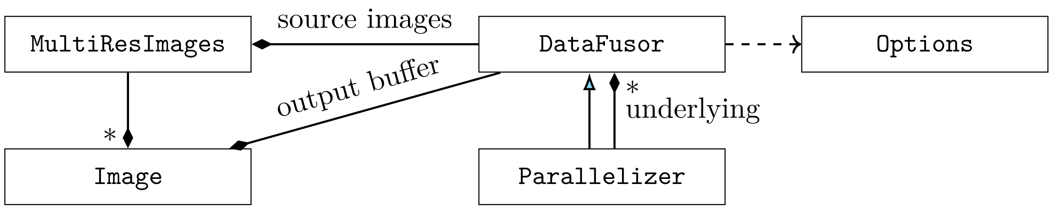


Figure 4.1: Important classes in `libimagefusion` and their relation.

4.1.1 Image

The `Image` class represents images. It should be easy to use, but still not restrict the user. Besides a copy and a move constructor, it has constructors to make empty images (default constructor), images with specified size and type and for images read from a file:

```cpp
Image nothing;
Image graystripe{1000, 20, Type::uint8x1};
Image img{"image.tif"};
```

So `Image` is not a generic type. An `Image` object can hold an image of an arbitrary type, because it uses a *dynamic type system* inherited from OpenCV. This has the advantage of being able to read an image file with an unknown type (unknown at compile time). The type can be requested at runtime with `basetype()` and `channels()` or `type()`. `channels()` returns the number of channels, so for an RGB image the number of channels is 3 and for a grey scale image it is 1. `basetype()` returns the base data type independent of the number of channels, like e. g. `Type::uint8` for an 8-bit unsigned integer image, which can contain values from 0 to 255, `Type::int16` for a 16-bit signed integer image, which can contain values from -32768 to 32767 or `Type::float32` for a floating point image with single precision. Mask images are also supported and have always the base type `Type::uint8`. `type()` returns a full type, which not only contains the base type, but also the number of channels. One example would be `Type::uint8x3` for an 8-bit unsigned integer image with 3 channels. Although type information is not required explicitly when reading in an image file, it is still required at compile time when accessing the pixel values of an image. So to get or set for example the pixel value at $x = 10$ and $y = 15$ in channel 1 (first channel is 0) of an 8-bit unsigned integer image `img.at<uint8_t>(10, 15, 1)` can be used. Here it is assumed that `img` has got an 8-bit unsigned integer type, has at least a size of 11×16 and at least two channels. A program must assure that this is the case. There are no internal checks of this for performance reasons. For masks `mask.getBoolAt(10, 15, 1)` or `mask.setBoolAt(10, 15, 1, true)` can be used, where the latter sets the pixel value to 255.

So in principle, it is not easy to access a pixel of an image, whose type is unknown at compile time, e. g. like `img` above. To simplify that the library provides a mechanism to call a functor with the correct compile time type information. This can be accessed via `CallBaseTypeFunctor::run(Functor{img}, img.basetype())`. There is also a type traits class, called `DataType` to get the C++ data type corresponding to the `Type`. Just to give an example functor that returns as double the pixel value at $x = 10$ and $y = 15$ in channel 1 of an image with arbitrary base type (but appropriate size and number of channels):

```cpp
struct Functor {
    // reference on image, initialized at construction
    Image& img;

    // t is the real image base type
    template<Type t>
    double operator()() const {
        // get corresponding C++ data type
        using basetype = typename DataType<t>::base_type;

        // retrieve pixel value at x = 10, y = 15, c = 1
        basetype value = img.at<basetype>(10, 15, 1);

        // return pixel value in double type
        return static_cast<double>(value);
    }
}
```

Note, this example seems to be rather long for this simple task, but almost the same code can be used for much more complicated tasks. So this is quite all that has to be done to retrieve values from images with a base type, that are unknown at compile time. For details on this mechanism and more advanced usage patterns, as how to use it with a specified set of allowed types (to give good error messages), the reader is referred to the API documentation.

Internally, the `Image` class uses OpenCV's `Mat` [Its17b] for most of the functionality, including the dynamic type system. OpenCV is also responsible for the memory management, which does not only consist of allocating and deallocating memory, but also reference counting. So OpenCV allows to make shared copies of images, which share the image memory. This means that changing a pixel is reflected in all shared copies. Using the same memory means also that making a shared copy is a very cheap operation. Still the view on the image can be different across shared copies. So it is possible to make a cropped shared copy. This can be accessed by using `Image small = img.sharedCopy(r);`, where `r` is a `Rectangle` to specify the top left corner and size of the crop window. Note that in OpenCV a shared copy is made by the usual copy constructor and copy assignment, which is a very unusual behaviour in C++. This design flaw undermines const-correctness in the sense that a const `Mat` can be modified via a shared copy. This has mostly been fixed in `Image`. It will make a deep copy (also: clone) when copying it, which means that the new image will have its own memory and hence be independent from the original. `Image` has an interface designed in the spirit of C++11 and thus implements also move constructor and assignment, which is also helpful for efficiency.

The `Image` class even goes beyond plain const-correctness by returning a `ConstImage` object in specific situations (analogue to the `const_iterator` concept). A `ConstImage` object only allows read access on its pixels. This makes it suitable for input images.

The most operations are defined in appropriate methods and can be used in a Java style, like

```
Image sum = img1.add(img2);
```

In case this would result in an overflow the result is limited to the data range. So if both images would be for example 8-bit unsigned integer and both had somewhere value 150 the resulting value would be 255. If saturation is not desired, the data type of the result can be changed like:

```
Image sum = img1.add(img2, Type::uint16x3);
```

Many often required operations are defined like this. Though, more specific operations available in OpenCV, might not be directly available in `Image`. However, to not restrict the user, the underlying OpenCV `Mat` can be accessed with `cvMat()`. With this function, everything from OpenCV is available in `libimagefusion`. Remark: This again opens the possibility to modify a ConstImage, but only if the user really wants to do that on purpose, similarly as with a const_cast. Const-correctness does not easily break by accident here.

For one thing OpenCV is not used in Image and this reading and writing images from and to files, respectively. The library that handles this is GDAL [GDA16]. GDAL provides good image format drivers, especially for TIFF. Also, it can read specific parts or channels of images. Image provides a constructor to read only a rectangular part of an image, specified channels or to read it flipped.

4.1.2 Source Image Container

The MultiResImages class is used to store the source images for every image fusion algorithm. In image fusion algorithms the source images are required in a specific structure. This can be thought of a table with the date against the resolution, like illustrated in Table 2.1. So typically there are one or more fine resolution images at specific dates and coarse resolution images available on every considered date. MultiResImages is such a structured storage for these images. Note that DataFusor takes a shared pointer on MultiResImages. So the DataFusor can be the owner, but is not required to. This protects e. g. a MultiResImages object from destruction in a situation where a function creates a MultiResImages and a DataFusor object and returns only the DataFusor object. A small example illustrates this, where SomeFusor is an image fusion algorithm implementation and SomeOptions the corresponding options class:

```cpp
SomeFusor getFusor() {
    std::string fine_tag   = "fine";
    std::string coarse_tag = "coarse";

    auto imgs = std::make_shared<MultiResImages>();
    imgs->set(Image("coarse1.png"), 1, coarse_tag);
    imgs->set(Image("coarse2.png"), 2, coarse_tag);
    imgs->set(Image("coarse3.png"), 3, coarse_tag);
    imgs->set(Image("fine1.png"),   1, fine_tag);
    imgs->set(Image("fine3.png"),   3, fine_tag);

    SomeOptions o;
    o.setCoarseTag(coarse_tag);
    o.setFineTag(fine_tag);
    o.setDate1(1);
    o.setDate3(3);

    SomeFusor f;
    f.setSrcImages(imgs);
    f.processOptions(o);
    return f;
}
```

Although imgs gets destroyed at the end of getFusor(), this is not harmful, since it is only one pointer on the underlying MultiResImages object. Another pointer is saved in f, which is why the underlying object does not die with imgs.

Retrieving an image from a MultiResImages object is quite easy:

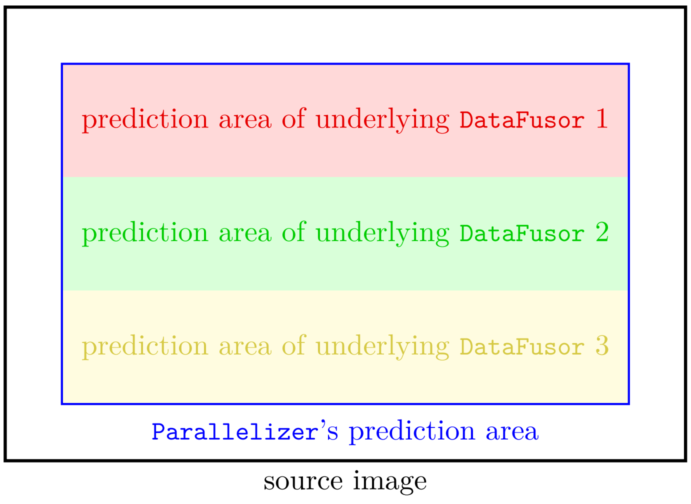

prediction area of underlying DataFusor 1
prediction area of underlying DataFusor 2
prediction area of underlying DataFusor 3
Parallelizer's prediction area
source image

Figure 4.2: Stripes that are predicted in parallel by the three underlying data fusors of `Parallelizer`.

```
ConstImage const& fine1 = imgs->get("fine", 1);
```

This returns a const reference on an `Image` and it is assigned to a const reference on a `ConstImage` to make it more clear that it can be read only.

4.1.3 Fusion Algorithms

Then, there is also a base class for all image fusion algorithms, called `DataFusor`. Basically, it receives a `MultiResImages` object (shared pointer) containing the source images and an `Options` object, as shown in the example of the previous section. The fusion can be started easily. After fusion the fused `Image` can be retrieved and for example written to a file.

```
f.predict(2);
ConstImage const& out = f.getOutputImage();
out.write("fine2_pred.png");
```

The base class for options (`Options`) has a prediction area options. This specifies a rectangle in pixel coordinates, where the prediction should be made. It is set by `o.setPredictionArea(r)`, where `r` is a `Rectangle`. It often makes sense to restrict the fusion to an inner region, since many fusion algorithms require a window or similar around the fused pixels. So, using a larger image as actually desired and restricting the prediction area might help to keep the quality homogeneous, also near the prediction border.

Next, to accelerate there is a `Parallelizer` class, which is a special kind of (meta) data fusor. It will internally use multiple instances of a `DataFusor` class to fuse an image in parallel. For that the output image is cropped into horizontal stripes and each fusor instance is restricted with the prediction area to predict that specific stripe. This is depicted in Figure 4.2. Rather simple image fusion algorithms from the class of reconstruction based algorithms usually work out of the box with `Parallelizer`. Here is an example with STARFM:

```
StarfmOptions opt;
... // (set options as usual)

ParallelizerOptions<StarfmOptions> p_opt;
p_opt.setPredictionArea(opt.getPredictionArea());
p_opt.setAlgOptions(opt);

Parallelizer<StarfmFusor> p;
p.setSrcImages(imgs);
p.processOptions(p_opt);
p.predict(2);
```

So, just the `ParallelizerOptions` are extra and the `StarfmFusor` object is replaced by a `Parallelizer<StarfmFusor>`.

However, not every algorithm is as easily parallelizable as STARFM. SPSTFM for example is parallelized in the implementation by the internally used matrix library and would otherwise be more difficult to parallelize. This is because for the learning process information from the whole image is required. Hence, with the internal parallelization, for an `SpstfmFusor` the `Parallelizer` class cannot be used (and trying that gives a descriptive error message). Generally an algorithm could provide a specialization of `Parallelizer` for a specific data fusor class to provide a custom parallelized implementation. The SPSTFM implementation is always parallelized, though.

4.2 Geo Information

Rather unrelated to the is the `GeoInfo` class. It can read geo and meta information from images and has methods to modify these information. To add these information to an image, the image has to exist as file already, which is then just updated with the information. This allows very flexible handling of geo and meta information without the requirement to read or write a whole image. An example illustrates this:

```
// get prediction area from the fusor options
Rectangle r = opt.getPredictionArea();

// read in geo info from fine1.png
GeoInfo gi{"fine1.png"};

// shift top left corner and change size
gi.geotransformTranslateImage(r.x, r.y);
gi.setSize(r.size());

// write geo info to predicted image fine2_pred.png
gi.addTo("fine2_pred.png");
```

4.3 Utility Support

To support the existing command line utilities and also the production of new utilities, the imagefusion framework has got a command line option parser. Option parsing is a task that every command line utility has to do and thus every utility profits from this. Also, for the utilities of a framework, it is beneficial if they behave in a consistent style. Thus using the same option parser for all utilities not only saves work and code, it improves also the user experience and makes the code easier maintainable.

The requirements for an option parser in imagefusion are as follows. It should...

- ...use a common syntax for command line options.
- ...be easy to use.
- ...only require a small amount of code in the utility.
- ...support a pretty printed yet easy to make help.
- ...be extensible by a utility to parse custom data types.
- ...integrate into the framework.
- ...not limit a utility in the sense that it cannot handle complex options.
- ...be able to handle files with options.
- ...give good error messages to the user in case of a bad input.
- ...be able to parse various data types, including the ones from `libimagefusion` (like `Rectangle`).

For the first six points the *Lean Mean C++ Option Parser* [Ben12] seemed to suffice. Especially, since it is a single-file header-only library it is easy to integrate. It uses a commonly known syntax; double dash long options, e. g. `--long-opt` and single dash short options, which can be grouped, e. g. `-abc` to use options `a`, `b` and `c`. They can also take an argument (configured in the utility code), either separated, like `-o arg` and `--long-opt ↲ arg`, or for long options with an equal sign, like `--long-opt=arg`. The Lean Mean C++ Option Parser is also easy to use and requires only a small amount of code to specify the options and the help text. The help text can be written in a tabular form and be printed with alignment also for broken lines. However, this was originally only supported for the last column and has been extended in imagefusion for arbitrary columns to support sub-tables without regressing the original behaviour. The option parser can also be extended by someone who makes a utility to parse and check for custom data types.

Non the less the remaining requirements were originally not existing in the way the imagefusion framework required it. So, to allow arbitrary nested options, a tokenizer has been added. It allows for an option argument to be an option string again, which separates tokens by spaces, but respects quotings with double quotes `"..."`, single quotes `'...'` and parentheses `(...)`. The latter does not work as first nesting level on bash, but in option files (see below). For example a user might be required to give an image as argument for a utility. In image fusion often a date and a resolution has to be specified. The extended option parser allows just that. An image option can have the format

Listing 4.1: Image argument with date and resolution tag.

```
--image='--file=landsat-file-name.tif --date=42 --tag=fine'
```

So the argument of `image` consists itself of options. Also their arguments could again consist of options and so on. So very complex options are possible, but more important not required. This makes it still easy to use, but does not limit the options for complex situations.

There is also a build-in pseudo option `--option-file`, which accepts a file as argument. This is by default enabled, which on the one hand means that a utility does not have to write any code for this and on the other that it can be disabled, if a utility does not want to support this. An option file can contain any valid option just as on command line. This is by design, because the file content is tokenized and these tokens replace the `--option-file` option before the actual option parsing begins. Line comments beginning with a hash `#` are also supported. This allows to write well documented option files. All whitespace is handled the same, including newlines. This allows to structure the option file to make it more readable.

Especially because complex options are possible, good error messages are important in case of typos and faulty user inputs. This is done via exceptions and a utility is not required to catch them, because the messages are printed by default. However, it can decide to handle them. The error messages tell precisely which argument was considered to be wrong for which option and in case of nested options they tell recursively the options where it is located. This works also for all imagefusion data types like `Rectangle`. When the argument is checked for correctness, it is tried to be parsed with the corresponding parsing function. These parsing functions can not only be used by the framework to check for correctness, but also by the utility developer to gather the option arguments in the right type.

Generally, when parsing options with the option parser, the options are stored in two ways:

- In the order the options were specified on command line.
- In a grouped fashion and each group contains options in the order the options of this group were specified on command line.

Figure 4.3 presents an example for command line input and the two storages. The groups are the usual way to handle options and often one is even only interested in the presence or the last argument. However, the most interesting scenarios are easily implementable by a utility:

- Test for presence of an option / print the help:

```cpp
if (!options[Opt::HELP].empty()) {
    printUsage(usage);
    std::exit(0);
}
```

- Get the argument of the last option of a kind / parse an argument:

```cpp
if (!options[Opt::NUM].empty()) {
    std::string& arg = options[Opt::NUM].back().arg;
    int n = Parse::Int(arg);
    ...
}
```

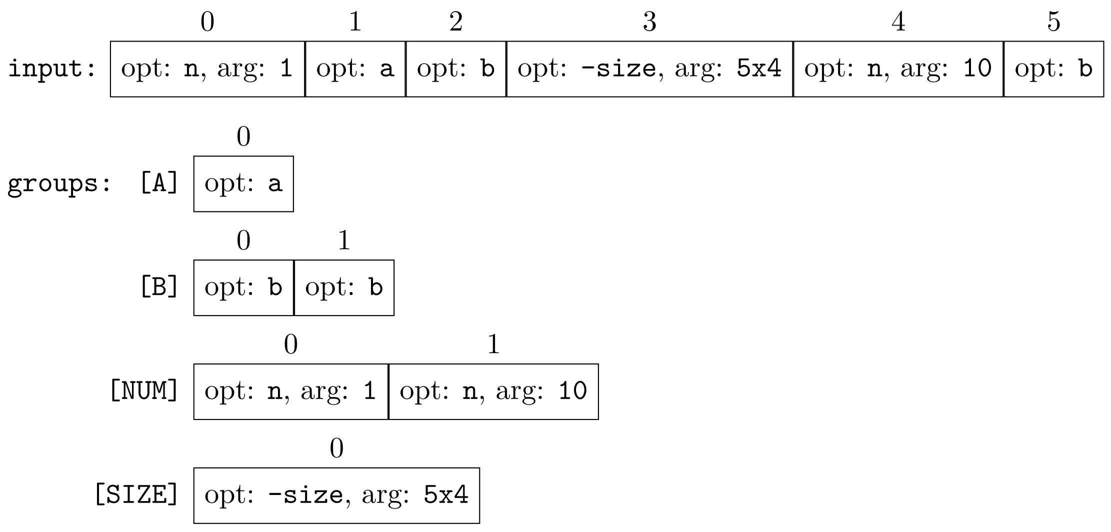


Figure 4.3: Parsed option structures for example command line argument: `./utility -n 1 -ab -size=5x4 -n 10 -b`.
The `input` object is just a vector and `groups` is a map of vectors with an enum key.

- Evaluate an `--enable-foo` / `--disable-foo` pair where the last one used wins:

```
if (!options[Opt::FOO].empty()) {
    if(options[Opt::FOO].back().prop() == Props::ENABLE)
        ...
    else // disable
        ...
}
```

- Cumulative option (`-v` verbose, `-vv` more verbose, `-vvv` even more verbose):

```
int verbosity = options[Opt::VERBOSE].size();
```

- Iterate over all `--file=<fname>` arguments:

```
for (auto& opt : options[Opt::FILE]) {
    std::string& fname = opt.arg;
    ...
}
```

- If required, the `input` storage can be used to process some or all arguments in the order they were given on command line:

```
for (auto& opt : options.input) {
    if (opt.spec() == Opt::NUM) {
        int n = Parse::Int(opt.arg);
        ...
    }
    else if (opt.spec() == Opt::FILE) {
        std::string& fname = opt.arg;
        ...
    }
    ...
}
```

All of these features make the option parser easy to use and save utility developers from a lot of work. Yet it is still powerful to cope with complex arguments and custom data types, while checking arguments and accepting option files without requiring utility code.

5 Implementation of Dictionary-Learning Algorithm SPSTFM

The “SParse-representation-based SpatioTemporal reflectance Fusion Model” (SPSTFM) algorithm is mainly based on learning an overcomplete dictionary pair and using sparse coding. The next sections explain the method in detail using a slightly different nomenclature as [HS12]. Also there are a few optional additions to the method. Since the focus is on the method itself, a simplified image structure compared to Table 2.2b is used without loss of generality, which contains fixed dates as shown in Table 5.1.

5.1 Sampling patches

SPSTFM works with patches. A patch is a small square region of the image I. Neighbouring patches can overlap. For processing a patch is not used as a small square image, but instead in a lexicographically stacked vector form, which is called *signal*. The process of sampling and stacking is depicted in Figure 5.1. We refer to it as *patch signal* $p_i \in \mathbb{R}^n$ to make clear that this vector is representing a $\sqrt{n} \times \sqrt{n}$ patch. A set of patch signals can be used as columns of a matrix P. Such a matrix is called *patch matrix*.

As can be seen in the image structure in Table 5.1, for date 1 and 3 there are two corresponding images: fine and coarse resolution images. Optimally these do only differ in resolution, but in practice they are maybe recorded at different times and with different sensors, which have even different bands and bit depth. So, the images themselves of the same date do not match exactly, but it is assumed that the image differences match better. Therefore only the difference image patch matrices are used and denoted by $P_{ji} := P_j - P_i$, where P_j and P_i would be the patch matrices corresponding to date j and i, respectively. Equivalently P_{ji} can be sampled from the difference image $I_{ji} := I_j - I_i$.

A patch matrix does not hold every patch signal from an image, rather a selection of N typical or important patch signals. The patch signals can be selected in different ways.

Table 5.1: Structure of images for SPSTFM, where for dates 1 and 3 image pairs are given and the fine resolution image for date 2 is to be predicted.

date / resolution	1	2	3
fine	$I_{\text{f},1}$	$I_{\text{f},2}$	$I_{\text{f},3}$
coarse	$I_{\text{c},1}$	$I_{\text{c},2}$	$I_{\text{c},3}$

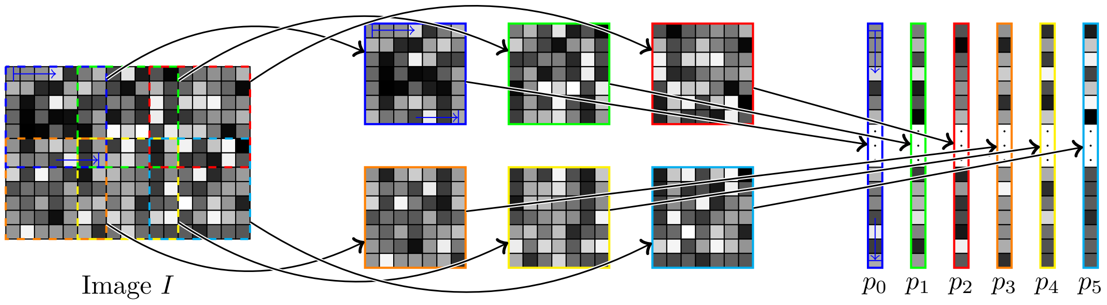


Figure 5.1: Sampling of a tiny image I. Here the image on the left has 2×3 patches. Each has a size of 7×7 pixels with 2 pixels overlap on each side, which are also the values used in [HS12]. These patches are considered one-by-one (middle) and lexicographically stacked (i. e. row-major) to form patch signals (right). The stacking order is indicated by some arrows for the first patch (blue).

The most simple one is to choose them randomly. A more promising one is to select the N patch signals with the most variance. This is the default option.

Now let $P_{\mathrm{f},31}$ and $P_{\mathrm{c},31}$ denote the $n \times N$ patch matrices for fine and coarse resolution, respectively. So $P_{\mathrm{f},31}$ contains the patches [HS12] suggests to subtract mean μ_c and divide by the variance σ_c of $P_{\mathrm{c},31}$ from both patch matrices for normalization:

$$\hat{P}_{\mathrm{c},31} := (P_{\mathrm{c},31} - \mathbf{1}_{n \times N}\, \mu_\mathrm{c})\, \frac{1}{\sigma_\mathrm{c}} \quad \text{and} \quad \hat{P}_{\mathrm{f},31} := (P_{\mathrm{f},31} - \mathbf{1}_{n \times N}\, \mu_\mathrm{c})\, \frac{1}{\sigma_\mathrm{c}}.$$

Then $\hat{P}_{\mathrm{c},31}$ and $\hat{P}_{\mathrm{f},31}$ would be the training data. For the sake of simplicity the training data is denoted by $P_{\mathrm{f},31}$ and $P_{\mathrm{c},31}$ in the following sections.

There are other options possible than using the mean and variance of the coarse resolution difference patch matrix. The general form is

$$\hat{P}_{\mathrm{c},31} := (P_{\mathrm{c},31} - \mathbf{1}_{n \times N}\, a_\mathrm{c})\, \frac{1}{b_\mathrm{c}} \quad \text{and} \quad \hat{P}_{\mathrm{f},31} := (P_{\mathrm{f},31} - \mathbf{1}_{n \times N}\, a_\mathrm{f})\, \frac{1}{b_\mathrm{f}}. \tag{5.1}$$

There is an option to choose a_c and a_f:

- coarse resolution mean for both (as above): $a_\mathrm{c} = a_\mathrm{f} = \mu_\mathrm{c}$
- fine resolution mean for both: $a_\mathrm{c} = a_\mathrm{f} = \mu_\mathrm{f}$
- separate resolution means: $a_\mathrm{c} = \mu_\mathrm{c}$, $a_\mathrm{f} = \mu_\mathrm{f}$
- no subtraction of any mean: $a_\mathrm{c} = a_\mathrm{f} = 0$

Another independent option is provided analogously to choose b_c and b_f:

- coarse resolution factor for both: $b_\mathrm{c} = b_\mathrm{f} = s_\mathrm{c}$
- fine resolution factor for both: $b_\mathrm{c} = b_\mathrm{f} = s_\mathrm{f}$

- separate resolution factors: $b_\mathrm{c} = s_\mathrm{c}$, $b_\mathrm{f} = s_\mathrm{f}$
- no division of any factor: $b_\mathrm{c} = b_\mathrm{f} = 1$

and s stands either for the variance or for the standard deviation (selectable with an option). Usually the standard deviation is used for normalization. In contrast to [HS12] this implementation uses the difference image instead of the difference patch matrix for computation of mean, variance and standard deviation. This should give more accurate results, since it contains more data.

The normalization operations have to be done for the coarse resolution patches at reconstruction again and the inverse operations for the fine resolution patches. This is described briefly in Section 5.4.

Note that this is about the normalization of differences. These are invariant to a mean shift in the image, consider $(I_j + \mu) - (I_i + \mu) = I_j - I_i$. The mean used in (5.1) is the mean of the difference. Hence, by default, no subtraction of any mean is performed. Dividing both difference patch matrix by the same factor should not have an effect. Using separate factors could be valuable for images that have different deviations for fine and coarse resolutions. This is evaluated in Section 5.6.2.

5.2 Dictionaries and Sparse Coding

A dictionary $D \in \mathbb{R}^{n \times m}$, $m < N$ is a matrix, which columns are modified patch signals. The patch signals in a dictionary are called atoms. These can be used as linear combination to represent an arbitrary patch signal p from the image: $p = D\,\lambda$.

For SPSTFM overcomplete dictionaries are used, i. e. $m > n$. This allows to have a sparse representation $\lambda \in \mathbb{R}^m$, such that only a few atoms are used to represent p.

However, this requires a sparse coding algorithm to find λ. The approach from [NW+07] that is used here is just briefly summarized. Actually the problem can be stated as

$$\lambda = \arg\min_{\lambda'} \|\lambda'\|_0 \quad \mathrm{s.\,t}\ p = D\,\lambda',$$

where $\|\cdot\|_0$ denotes the number of nonzero elements. To allow for a small error this can be reformulated in a relaxed form:

$$\lambda = \arg\min_{\lambda'} \|\lambda'\|_0 \quad \mathrm{s.\,t}\ \|p - D\,\lambda'\|_2^2 < \varepsilon,$$

where ε is a tolerance. Now, the term $\|\lambda'\|_0$ makes it a hard problem, but it can be approximated under certain conditions [RBE10], by an L^1-norm. There are different algorithms that are based on this formulation. [HS12] suggests to use a *Gradient Projection for Sparse Representation* (GPSR) algorithm [NW+07]. There, the problem is reformulated to an unconstrained optimization problem:

$$\min_{\lambda} \frac{1}{2}\|p - D\,\lambda\|_2^2 + \tau\|\lambda\|_1, \tag{5.2}$$

where τ balances the sparsity of the solution. A larger τ means a sparser solution at the cost of accuracy. [NW+07] suggests to choose $\tau = 0.1\|D^\top p\|_\infty$. Equation (5.2) is equivalent to a standard quadratic problem

$$\min_\lambda F(\lambda), \quad \text{where } F(\lambda) := \frac{1}{2}\lambda^\top D^\top D\,\lambda + (\tau \mathbf{1}_{1\times m} - p^\top D)\,\lambda.$$

This can also be reformulated to become a bound-constrained quadratic program (BCQP), see [NW+07]. The GPSR algorithm is based on this formulation. In this work the GPSR-BB variant with monotonic behaviour is chosen. The main loop of the GPSR-BB algorithm stops when relative change in the objective function is small enough, i. e.

$$\frac{F(\lambda_k) - F(\lambda_{k-1})}{F(\lambda_{k-1})} < \epsilon, \tag{5.3}$$

where $\epsilon = 10^{-6}$ is chosen as default option and $5 < k < 5000$ is the iteration count. In the reference implementation of GPSR mentioned in [NW+07] this is stop criterion 1. It is also worth to mention that using a smaller ϵ results in more iterations and this gives usually a sparser representation. After that λ debiased ($\|\lambda\|_1$ introduces a bias), but only if the number of non-zero elements is not more than the dimension. Debiasing is an iterative optimization with the Conjugate Gradient (CG) method that starts at λ_k and optimizes for (5.2) with $\tau = 0$. However, the CG method is modified such that it does not add any non-zero elements to λ, so the sparsity is not changed. For debiasing in this work at least one iteration is done, but a very loose debiasing tolerance of 10^{-1} or 10^{-2} is used as default option. To sum it up, with this algorithm a good sparse representation λ respective to an appropriate dictionary D can be found for any patch signal p.

5.3 Dictionary-Pair Training

Since in the image structure in Table 5.1 there are fine and coarse resolutions, also two corresponding dictionaries D_f and D_c are required. These are initialized with the first m columns of the training data $P_{\mathrm{f},31}$ and $P_{\mathrm{c},31}$, respectively. Then, optionally, each patch, independent of resolution, can be normalized to the Euclidean length of 1, i. e. $\|p_{\mathrm{f},i}\|_2 = \|p_{\mathrm{c},i}\|_2 = 1\ \forall i$, where $p_{\mathrm{f},i}$ and $p_{\mathrm{c},i}$ are the i-th coarse and fine patch signal, respectively. However, this does not allow for different scales of coarse and fine resolution atoms. Alternatively, all patches can be scaled with the same factor. This includes the special case of the factor being 1 in which case the patch signal is taken from the difference image unchanged. Another way of normalization is that all patch signal pairs can be scaled with a different factor, such that one of the two patches is normalized and the other divided by the same factor, i. e. $s := \max(\|p_{\mathrm{f},i}\|, \|p_{\mathrm{c},i}\|)$ and then $p_{\mathrm{f},i} \leftarrow p_{\mathrm{f},i}\,\frac{1}{s}$ and $p_{\mathrm{c},i} \leftarrow p_{\mathrm{c},i}\,\frac{1}{s}$. This results in a norm less or equal to 1. The two latter methods are rather similar since they do not change the ratio between the norms of fine and coarse resolution patches in each pair, but only between different pairs.

Now the dictionaries have to be trained. Although the dictionaries might fit to the first training patch signals optimally (depending on normalization), since they were initialized with them, the goal is rather to have a dictionary-pair that is optimized for the whole training data:

$$\{D_\mathrm{f}^*, D_\mathrm{c}^*, \Lambda^*\} = \underset{D_\mathrm{f}, D_\mathrm{c}, \Lambda}{\arg\min} \left\{ \|P_{\mathrm{f},31} - D_\mathrm{f}\,\Lambda\|_F^2 + \|P_{\mathrm{c},31} - D_\mathrm{c}\,\Lambda\|_F^2 + \tau\,\|\Lambda\|_1 \right\} \tag{5.4}$$

Hereby, the columns λ_i of Λ are sparse representation coefficients. In [HS12] they are determined by both resolutions. Such a representation λ_i can be found by using a stacked patch signal $[p_{\mathrm{f},i}; p_{\mathrm{c},i}]$ and a stacked dictionary $[D_\mathrm{f}; D_\mathrm{c}]$ in the GPSR algorithm to find the best common representation for

$$\begin{pmatrix} p_{\mathrm{f},i} \\ p_{\mathrm{c},i} \end{pmatrix} \approx \begin{pmatrix} D_\mathrm{f} \\ D_\mathrm{c} \end{pmatrix} \lambda_i.$$

In addition, alternative optimization objectives for λ_i are provided in this implementation:

- Using the coarse resolution only: $p_{\mathrm{c},i} \approx D_\mathrm{c}\,\lambda_i$
- Using the fine resolution only: $p_{\mathrm{f},i} \approx D_\mathrm{f}\,\lambda_i$
- Averaging the coefficients of both resolutions: $p_{\mathrm{f},i} \approx D_\mathrm{f}\,\lambda_{\mathrm{f},i}$, $p_{\mathrm{c},i} \approx D_\mathrm{c}\,\lambda_{\mathrm{c},i}$ and $\lambda_i := \frac{1}{2}(\lambda_{\mathrm{f},i} + \lambda_{\mathrm{c},i})$.

All of these options basically optimize λ_i for (5.2), but with different P and D. Doing this for every i yields Λ. Note, there is only one Λ instead of one per resolution. The algorithm is not stated here. The reader is referred to [NW$^+$07] and the reference implementation mentioned there with an URL. Finding Λ by using the GPSR algorithm in this way for each column is the first step of the iterative training process to solve (5.4).

The second step is to update the dictionary columns in a pairwise manner. For that K-SVD [AEB06] is adapted. Let $P \in \mathbb{R}^{n \times N}$ be the training data, $D \in \mathbb{R}^{n \times m}$ the dictionary and $\Lambda \in \mathbb{R}^{m \times N}$ the sparse representation coefficients. Then, when updating the k-th atom of D, the nonzero entries in the k-th row of Λ are considered. These belong to the representations that use the k-th atom. Let ω_k be the set of their indices, i. e. $\omega_k := \{i \mid \Lambda_{k,i} \neq 0, 1 \leq i \leq N\}$. Now, let P_{ω_k} be the reduced training data, which includes only the columns of P with the indices in ω_k, analogously Λ_{ω_k} the reduced coefficients and I_k the $m \times m$ identity matrix, but with a zero in the k-th diagonal entry. Then the error matrix

$$E_k = P_{\omega_k} - D\,I_k\,\Lambda_{\omega_k}$$

represents the remaining error without the k-th atom, since I_k sets the coefficients for the k-th atom to zero. Now one applies a Singular Value Decomposition (SVD)

$$E_k = U\,\Delta\,V^\top,$$

which gives orthogonal U and V and diagonal Δ, to receive the best rank 1 approximation. This is $E_k \approx u_1\,\Delta_{11}\,v_1^\top$, where u_1 is the first column of U, Δ_{11} is the first (and thus largest) singular value and v_1 is the first column of V. To apply this approximation, the k-th column of D will be set to u_1 and the k-th row of Λ_{ω_k} (i. e. the nonzero entries in the k-th row of

Algorithm 1 Atom update (concatenated version)

```
Input concatenated difference patch matrix $P_{31}$ as $P$, concatenated dictionary $D$, sparse coefficients $\Lambda$, index $k$
Output updated concatenated dictionary column $d_k$, updated coefficient row $\lambda_k$
  function K-SVD($P$, $D$, $\Lambda$, $k$)
      $\lambda_k \leftarrow$ row $k$ of $\Lambda$
      $\omega_k \leftarrow$ indices of nonzero entries in $\lambda_k$
      $P_{\omega_k} \leftarrow$ columns of $P$ with indices $\omega_k$
      $\Lambda_{\omega_k} \leftarrow$ columns of $\Lambda$ with indices $\omega_k$
      $I_k \leftarrow m \times m$ identity matrix, but with $k$-th diagonal entry 0
      $E_k \leftarrow P_{\omega_k} - D\, I_k\, \Lambda_{\omega_k}$
      $[U, \Delta, V^\top] \leftarrow SVD(E_k)$
      if option is scale coefficients or option is scale dictionary normal then
          return $d_k \leftarrow u_1$ and $\lambda_k(\omega_k) \leftarrow v_1\, \Delta_{1,1}$     ▷ $\lambda_k(\omega_k)$ selects the elements at $\omega_k$
      else if option is scale dictionary direct then
          return $d_k \leftarrow u_1\, \Delta_{1,1}$ and $\lambda_k(\omega_k) \leftarrow v_1$
      end if
  end function
```

Λ) will be set to v_1 multiplied by Δ_{11}. Alternatively the k-th dictionary column can receive the multiplication with Δ_{11} instead of the coefficients. An option is provided for that.

Now this column update can be done in several ways. The easiest is to use *concatenated* training data and dictionaries, i. e.

$$P := \begin{pmatrix} P_{\mathrm{f},31} \\ P_{\mathrm{c},31} \end{pmatrix} \quad \text{and} \quad D := \begin{pmatrix} D_{\mathrm{f}} \\ D_{\mathrm{c}} \end{pmatrix}$$

This is one option and shown in Algorithm 1. For the difference between *scale dictionary normal* and *scale dictionary direct*, see Algorithm 2 and its description, where all options yield a different output. However, [HS12] mentions that this *Concatenated* option is not appropriate for typical satellite images, which can have large differences in “amplitude and variance”.

The *concatenated* option handles the dictionaries as one unit. The other options do separate SVDs for fine and coarse resolutions. Since the SVD is not unique it must be taken care of the signs of the updated column and row. Neglecting that could result in an inversion of atoms, which would break the mapping between them for any following updates. Also, the update of the coefficients cannot be done twice. It is very important that each column update in both dictionaries is done with the same coefficients. Otherwise they would loose their implicit mapping over time. This would also happen if the coefficients were not updated, which is available as an option (disable online learning), but not further mentioned. Nevertheless, there are now two options. Either the coefficients of a single resolution can be used for the coefficient update or the average of both.

Let us now discuss these steps from the beginning in more detail. First denote the error matrices for both resolution by

$$E_{\mathrm{f},k} = P_{\mathrm{f},\omega_k} - D_\mathrm{f}\, I_k\, \Lambda_{\omega_k} \quad \text{and} \quad E_{\mathrm{c},k} = P_{\mathrm{c},\omega_k} - D_\mathrm{c}\, I_k\, \Lambda_{\omega_k},$$

where P_{f,ω_k} is the reduced matrix of P_f and P_{c,ω_k} is the reduced matrix of P_c. Then let

$$E_{\mathrm{f},k} = U_\mathrm{f}\, \Delta_\mathrm{f}\, V_\mathrm{f}^\top \quad \text{and} \quad E_{\mathrm{c},k} = U_\mathrm{c}\, \Delta_\mathrm{c}\, V_\mathrm{c}^\top.$$

Now set $v := v_{\mathrm{f},1} \cdot v_{\mathrm{c},1}$, where $v_{\mathrm{f},1}, v_{\mathrm{c},1}$ are the respective first columns. This scalar product of the normalized new coefficients is utilized to recognize a different sign between the first columns of the two SVDs. v should be optimally (if $E_{\mathrm{f},k} = E_{\mathrm{c},k}$) 1 or -1 and in case of a -1, the sign will be changed. Due to the different resolutions and other reasons the errors will be different and thus the sign is changed if $v < 0$, i. e. multiplying by $\operatorname{sign}(v)$ (assuming $v \neq 0$ in which case the sign would not be changed). So, to update the dictionary atom the k-th column of D_c is set to $u_{\mathrm{c},1}$ and the k-th column of D_f to $\operatorname{sign}(v)\, u_{\mathrm{f},1}$, where $u_{\mathrm{c},1}$ and $u_{\mathrm{f},1}$ are the respective first column of U_c and U_f. For the update of the coefficients one of the following options can be used:

- The average of both new coefficients is used as update coefficient. Then the k-th row of Λ_{ω_k} will be set to $\dfrac{v_{\mathrm{c},1}\, \Delta_{\mathrm{c},11} + \operatorname{sign}(v)\, v_{\mathrm{f},1}\, \Delta_{\mathrm{f},11}}{2}$.
- Or the coefficients of the coarse resolution only is used. Then the k-th row of Λ_{ω_k} will be set to $v_{\mathrm{c},1}\, \Delta_{\mathrm{c},11}$.
- Or, finally, the coefficients of the fine resolution is used. Then the k-th row of Λ_{ω_k} will be set to $\operatorname{sign}(v)\, v_{\mathrm{f},1}\, \Delta_{\mathrm{f},11}$.

Again, as an alternative, the singular values can go into the dictionary instead into the coefficients. This is again provided as an option. However, in this split-up K-SVD variant this makes a real difference apart from just scaling. Using $u_{\mathrm{c},1}\, \Delta_{\mathrm{c},11}$ for the k-th column of D_c and $\operatorname{sign}(v)\, u_{\mathrm{f},1}\, \Delta_{\mathrm{f},11}$ for the k-th column of D_f enables different scales for the different resolutions. Since there is only one coefficient vector, which both resolutions share in the dictionaries, the coefficients cannot be used to match different scales of the resolutions. Scaling the atoms to the appropriate singular value could be valuable for images with a very different deviation of values of fine and coarse resolution. The coefficients can then have the norm 1. For their update all is done in the same way, except they are not multiplied with the singular values anymore. This is referred to as *scale dictionary direct*. As a variant the atoms can be divided by the larger singular value and the coefficients multiplied with it. So with $s := \max(\Delta_{\mathrm{c},11}, \Delta_{\mathrm{f},11})$ the k-th column of D_c receives $u_{\mathrm{c},1}\, \Delta_{\mathrm{c},11}\, \frac{1}{s}$ and the k-th column of D_f receives $\operatorname{sign}(v)\, u_{\mathrm{f},1}\, \Delta_{\mathrm{f},11}\, \frac{1}{s}$, while the singular values in the coefficient update are replaced by s. The effects of these options are shown in Section 5.6.2. This modified K-SVD algorithm with all the described options is stated in Algorithm 2

In [HS12] it seems like for finding the coefficients Λ the concatenated matrices are used and for the column updates of D_f and D_c the averaging procedure, but this is not clearly described. Although the options for coefficients and column updates seem to be independent it does not make sense to mix them arbitrarily: Concatenated and averaged can be mixed, but aside from that the same option should be used for coefficients and column

Algorithm 2 Atom update (separate resolutions)

```
Input concatenated difference patch matrix $P_{31}$ as $P$, concatenated dictionary $D$, sparse
  coefficients $\Lambda$, index $k$
Output updated concatenated dictionary column $d_k$, updated coefficient row $\lambda_k$
  function K-SVD($P$, $D$, $\Lambda$, $k$)
      $\lambda_k \leftarrow$ row $k$ of $\Lambda$
      $\omega_k \leftarrow$ indices of nonzero entries in $\lambda_k$
      $P_{\omega_k} \leftarrow$ columns of $P$ with indices $\omega_k$
      $\Lambda_{\omega_k} \leftarrow$ columns of $\Lambda$ with indices $\omega_k$
      $I_k \leftarrow m \times m$ identity matrix, but with $k$-th diagonal entry 0
      $E_k \leftarrow P_{\omega_k} - D\, I_k\, \Lambda_{\omega_k}$
      $\begin{pmatrix} E_{\mathrm{f},k} \\ E_{\mathrm{c},k} \end{pmatrix} = E_k$          ▷ Split up into fine and coarse resolutions
      $[U_\mathrm{f}, \Delta_\mathrm{f}, V_\mathrm{f}^\top] \leftarrow SVD(E_{\mathrm{f},k})$
      $[U_\mathrm{c}, \Delta_\mathrm{c}, V_\mathrm{c}^\top] \leftarrow SVD(E_{\mathrm{c},k})$
      $v \leftarrow v_{\mathrm{f},1} \cdot v_{\mathrm{c},1}$          ▷ $v_{\mathrm{f},1}$ and $v_{\mathrm{c},1}$ are the first columns of $V_\mathrm{f}$ and $V_\mathrm{c}$, respectively
      if option is *scale coefficients* then
          $d_k \leftarrow \begin{pmatrix} \mathrm{sign}(v)\, u_{\mathrm{f},1} \\ u_{\mathrm{c},1} \end{pmatrix}$
      else if option is *scale dictionary normal* then
          $d_k \leftarrow \begin{pmatrix} \mathrm{sign}(v)\, u_{\mathrm{f},1}\, \Delta_{\mathrm{f},11} \\ u_{\mathrm{c},1}\, \Delta_{\mathrm{c},11} \end{pmatrix} \dfrac{1}{\max(\Delta_{\mathrm{f},11}, \Delta_{\mathrm{c},11})}$
          $\Delta_{\mathrm{f},11}, \Delta_{\mathrm{c},11} \leftarrow \max(\Delta_{\mathrm{f},11}, \Delta_{\mathrm{c},11})$
      else if option is *scale dictionary direct* then
          $d_k \leftarrow \begin{pmatrix} \mathrm{sign}(v)\, u_{\mathrm{f},1}\, \Delta_{\mathrm{f},11} \\ u_{\mathrm{c},1}\, \Delta_{\mathrm{c},11} \end{pmatrix}$
          $\Delta_{\mathrm{f},11}, \Delta_{\mathrm{c},11} \leftarrow 1$
      end if
      if option is *average* then          ▷ $\lambda_k(\omega_k)$ selects the elements at indices $\omega_k$
          $\lambda_k(\omega_k) \leftarrow \frac{1}{2}(\mathrm{sign}(v)\, v_{\mathrm{f},1}\, \Delta_{\mathrm{f},11} + v_{\mathrm{c},1}\, \Delta_{\mathrm{c},11})$
      else if option is *fine* then
          $\lambda_k(\omega_k) \leftarrow \mathrm{sign}(v)\, v_{\mathrm{f},1}\, \Delta_{\mathrm{f},11}$
      else if option is *coarse* then
          $\lambda_k(\omega_k) \leftarrow v_{\mathrm{c},1}\, \Delta_{\mathrm{c},11}$
      end if
      return $d_k$ and $\lambda_k$
  end function
```

Algorithm 3 Dictionary Training

```
Input concatenated difference patch matrix $P_{31}$ as $P$
Output concatenated trained dictionary $D$
  function TRAIN($P$)
      initialize $D$ by first columns of $P$ and normalize
      repeat
          for all patch signals $p_i$ from $P$ do                    ▷ Find sparse coefficients $\Lambda$
              $\lambda_i = \text{GPSR}(p_i, D)$ and put $\lambda_i$ in $i$-th column of $\Lambda$
          end for
          for all atoms $d_k$ in $D$ and corresponding coefficient row $\lambda_k$ do ▷ Update $D$ and $\Lambda$
              $[d_k, \lambda_k] = \text{K-SVD}(P, D, \Lambda, k)$
          end for
      until stopping condition satisfied                              ▷ see Section 5.5
      return $D$
  end function
```

update. As will be seen in the next section, using the coarse resolution seems to be especially sound, since then the dictionaries are optimized for their actual use case, since in reconstruction only coarse patches are available. The influence of the different options is tested in Section 5.6.

Also the normalization of the initial dictionary depends on the singular value scaling option for the dictionary update. Using the singular values to scale the coefficients induces that all atoms have the same length. To be consistent, the dictionary should also be initialized in this way and this is done by normalization of every coarse and fine resolution patch signal independently. On the other hand, when using the fine and coarse resolution singular values to scale the fine and coarse resolution atoms separately, they have different length with the ratio of the singular values. Therefore the dictionary should be initialized with natural ratios between fine and coarse resolution patches. This is done by either using the singular values directly to scale the atoms, in which case the initial dictionary should not be normalized at all (factor 1), or by additionally dividing the atoms and multiplying the coefficients by the larger singular value, in which case the initial dictionary should be normed in a pairwise manner. Then the initial dictionary is consistent with the trained one, which makes sense.

To optimize (5.4) these two steps – finding sparse coefficients and updating the columns in the dictionaries – are repeated until the dictionary is trained. The condition, when to stop is discussed in Section 5.5, since for one option reconstruction is used, which is discussed in the next section. The training is summarized in Algorithm 3. There only the concatenated patch matrix and dictionary are used. However, from these the fine or coarse resolution parts can be extracted easily, which is not explicitly shown for GPSR. However, the algorithm does not change; the desired resolution parts have just to be used in calling arguments (and for the average option both resolutions separately and the results averaged). For K-SVD there are two versions, Algorithm 1 (concatenated option) and 2 (other options), from which the correct one should be selected. In Algorithm 2 the remaining options are stated explicitly.

5.4 Reconstruction

In the reconstruction stage the trained dictionary-pair is used to predict a fine resolution image from a coarse resolution image, compare date 2 in Table 5.1. However, the dictionaries are trained for difference image patches. Hence, the coarse resolution image of date 2 is not used directly, but as part of a difference image patch matrix. There are two choices: $\tilde{P}_{\mathrm{c},21}$ or $\tilde{P}_{\mathrm{c},23}$, which are patch matrices that include every patch from the difference image $I_{\mathrm{c},21}$ and $I_{\mathrm{c},21}$, respectively. Since for the test data or patch matrices $P_{\mathrm{c},21}$ and $P_{\mathrm{c},23}$ a normalization (5.1) has been done, this has also to be done for the patches in $\tilde{P}_{\mathrm{c},21}$ and $\tilde{P}_{\mathrm{c},23}$. The normalization must be done with the same values as done for the test data.

Before elaborating the best choice for $\tilde{P}_{\mathrm{c},21}$ and $\tilde{P}_{\mathrm{c},23}$, for the sake of simplicity, the general process is discussed. Let $\tilde{P}_{\mathrm{c}}$ be a coarse resolution difference image patch matrix. The goal is to predict the unknown fine resolution image I_{f}. This is done patch-by-patch. Let p_{c} be a coarse resolution patch from $\tilde{P}_{\mathrm{c}}$ and p_{f} the corresponding (unknown) fine resolution patch. Then the GPSR algorithm with D_{c} and p_{c} yields a sparse representation λ, which corresponds to the problem

$$\lambda = \min_{\lambda^*} \frac{1}{2} \| p_{\mathrm{c}} - D_{\mathrm{c}}\, \lambda^* \|_2^2 + \tau \| \lambda^* \|_1. \tag{5.5}$$

This sparse representation is now used with the fine resolution dictionary to predict p_{f}:

$$p_{\mathrm{f}} = D_{\mathrm{f}}\, \lambda \tag{5.6}$$

Inversion of the normalization can be done like shown in the following.

$$\tilde{p}_{\mathrm{f}} = p_{\mathrm{f}}\, b_{\mathrm{f}} + a_{\mathrm{f}}$$

This procedure can be done for every patch in $\tilde{P}_{\mathrm{c}}$ and give a corresponding fine resolution patch. However, these do not match exactly in the overlapping regions. This will be handled later on after the different resolutions have been combined.

SPSTFM claims [HS12] to be able to cope well with structural changes, like large build-up areas, even when using dates with long gaps in between. Their paper uses an example with two years between the images. This ability to handle structural changes is largely based upon weighting both resolution's patches $\tilde{P}_{\mathrm{c},21}$ and $\tilde{P}_{\mathrm{f},23}$ patch-by-patch according to the structural change in that patch. Let $p_{\mathrm{f},21}$ be a patch reconstructed using the corresponding patch $p_{\mathrm{c},21}$ with (5.5) and (5.6) and $p_{\mathrm{f},23}$ reconstructed from $p_{\mathrm{c},23}$. Then they are combined in the following way:

$$p_{\mathrm{f},2} = w_1\,(p_{\mathrm{f},1} + p_{\mathrm{f},21}) + w_3\,(p_{\mathrm{f},3} + p_{\mathrm{f},23}), \tag{5.7}$$

where $p_{\mathrm{f},1}$ and $p_{\mathrm{f},3}$ are the corresponding patches from the images $I_{\mathrm{f},1}$ and $I_{\mathrm{f},3}$. The weights w_1 and w_3 are determined with help of an improved build-up index [HSXZ10]. The continuous build-up index is as follows:

$$BU_c = \frac{SWIR - NIR}{SWIR + NIR} - \frac{NIR - Red}{NIR + Red}, \tag{5.8}$$

where *SWIR* is a shortwave infrared band, *NIR* near infrared and *Red* just the visible red band. In this work that value will not be calibrated with sample images (can be done optional), but instead only checked whether it is greater zero, in which case it is considered as build-up pixel and otherwise not. Let the number of build-up pixels that changed from $p_{\mathrm{c},1}$ to $p_{\mathrm{c},2}$ divided by the number of pixels n be v_1 and from $p_{\mathrm{c},2}$ to $p_{\mathrm{c},3}$ be v_3 (average number of build-up pixel changes per pixel). Then the weights are calculated as

$$w_i = \begin{cases} \frac{1}{2} & \text{if } v_i = v_j = 0 \\ 1, & \text{if } v_j - v_i > \delta \vee (v_i = 0 \wedge v_j \neq 0) \\ 0, & \text{if } v_i - v_j > \delta \vee (v_j = 0 \wedge v_i \neq 0) \\ \dfrac{v_1\, v_3}{v_i\, v_1 + v_i\, v_3}, & \text{otherwise} \end{cases} \qquad \text{for } i = 1, j = 3 \text{ or } i = 3, j = 1. \tag{5.9}$$

Hereby the weighting difference limit δ limits the difference of the weights and switches completely to one patch, when δ is exceeded. In [HS12] and in the implementation by default $\delta = 0.2$.

As an alternative option, and for cases where the required bands are not available, e. g. single-channel images, a simplified weighting based on the normed average absolute difference per patch is developed. There $v_i = \frac{\sum |p_{\mathrm{c},2i}|}{n\,M}$ is directly used, where M the maximum value in $\tilde{P}_{\mathrm{c},21}$ and $\tilde{P}_{\mathrm{c},23}$. The calculation of w_i stays the same.

Now, since every patch for the image $I_{\mathrm{f},2}$ is available, these are averaged in the overlapping regions to build that image. This is basically all for the reconstruction and summarized in Algorithm 4.

5.5 Stopping condition

Now, to the question about when the training should stop. This implementation provides several options for that. Firstly, it can be selected, which value should be considered. [HS12] suggests to use the objective function $\|P - D\,\Lambda\|_F^2 + \tau\|\Lambda\|_1$ with concatenated training data and dictionary. However, τ can be different for every column and thus the τ in the shown objective function could be interpreted as the largest τ. Another interpretation is also possible, where each column λ_i of Λ is multiplied with the corresponding τ_i and norm by the number of elements, so formally

$$E = (\|P - D\,\Lambda\|_F^2 + \max(\tau_i)\|\Lambda\|_1) \cdot \frac{1}{N\,n} \tag{5.10}$$

or

$$E = (\|P - D\,\Lambda\|_F^2 + \|\Lambda\ \mathrm{diag}((\tau_i))\|_1) \cdot \frac{1}{N\,n}. \tag{5.11}$$

P and D can be the concatenated versions, coarse or fine. A forth option for averaging coarse and fine is also available to be consistent with similar options for the dictionary training.

A third, completely different measure is the reconstruction error for a test set. For that we use K randomly selected patches from $\tilde{P}_{\mathrm{f},31}$ and $\tilde{P}_{\mathrm{c},31}$ as test data Q_{f} and Q_{c},

Algorithm 4 Reconstruction

```
Input  trained dictionaries $D_\mathrm{f}$, $D_\mathrm{c}$, images $I_{\mathrm{f},1}$, $I_{\mathrm{f},3}$, $I_{\mathrm{c},1}$, $I_{\mathrm{c},2}$, $I_{\mathrm{c},3}$
Output  predicted image $I_{\mathrm{f},2}$
  function Reconstruct($D_\mathrm{f}$, $D_\mathrm{c}$, $I_{\mathrm{f},1}$, $I_{\mathrm{f},3}$, $I_{\mathrm{c},1}$, $I_{\mathrm{c},2}$, $I_{\mathrm{c},3}$)
      $\tilde{P}_{\mathrm{f},21} \leftarrow$ Predict($I_{\mathrm{c},2} - I_{\mathrm{c},1}, D_\mathrm{f}, D_\mathrm{c}$)
      $\tilde{P}_{\mathrm{f},23} \leftarrow$ Predict($I_{\mathrm{c},2} - I_{\mathrm{c},3}, D_\mathrm{f}, D_\mathrm{c}$)
      $\tilde{P}_{\mathrm{f},1} \leftarrow$ convert $I_{\mathrm{f},1}$ to $n \times K$ patch matrix
      $\tilde{P}_{\mathrm{f},3} \leftarrow$ convert $I_{\mathrm{f},3}$ to $n \times K$ patch matrix
      for $i \leftarrow 1, \ldots, K$ do
          $w_1, w_3 \leftarrow$ weights for patch $i$                              ▷ see Section 5.4
          $p_{\mathrm{f},2} \leftarrow w_1\,(p_{\mathrm{f},1} + p_{\mathrm{f},21}) + w_3\,(p_{\mathrm{f},3} + p_{\mathrm{f},23})$
      end for
      $\tilde{P}_{\mathrm{f},2} \leftarrow$ made with column vectors
      return $I_{\mathrm{f},2} \leftarrow$ use patches from $\tilde{P}_{\mathrm{f},2}$, average overlapping areas $p_{\mathrm{f},2}$
  end function

Input  coarse resolution difference image $I_{\mathrm{c},2X}$, trained dictionaries $D_\mathrm{f}$, $D_\mathrm{c}$
Output  predicted difference patch matrix $\tilde{P}_{\mathrm{f},2X}$
  function Predict($I_{\mathrm{c},2X}$, $D_\mathrm{f}$, $D_\mathrm{c}$)
      $\tilde{P}_{\mathrm{c},2X} \leftarrow$ convert $I_{\mathrm{c},2X}$ to $n \times K$ patch matrix    ▷ $K$ is the total number of patches
      for $i \leftarrow 1, \ldots, K$ do
          $\lambda_i = \mathrm{GPSR}(p_{\mathrm{c},i}, D_\mathrm{c})$                    ▷ $p_{\mathrm{c},i}$ is column $i$ of $\tilde{P}_{\mathrm{c},2X}$
          $p_{\mathrm{f},i} \leftarrow D_\mathrm{f}\,\lambda_i$ and put $p_{\mathrm{f},i}$ in the $i$-th column of $\tilde{P}_{\mathrm{f},2X}$
      end for
      return $\tilde{P}_{\mathrm{f},2X} \leftarrow$ made with column vectors $p_{\mathrm{f},i}$
  end function
```

respectively, which are patch matrices. Then in each iteration a reconstruction can be done for each column with (5.5) and (5.6). Let the reconstructed patches be $\hat{Q}_\mathrm{f}$. Then the error is computed as

$$E = \|Q_\mathrm{f} - \hat{Q}_\mathrm{f}\|_1 \cdot \frac{1}{K\,n}. \tag{5.12}$$

Now there is a value that can be used as stopping criterion. [HS12] sets the acceptable "error of the objective function between two consecutive iterations" to $\varepsilon = 0.3$.

- This could be the absolute difference of $|E^{j-1} - E^j| < \varepsilon$.
- Alternatively the signed difference $E^{j-1} - E^j < \varepsilon$ can be checked,
- relative variants $\dfrac{|E^{j-1} - E^j|}{|E^{j-1}|} < \varepsilon$ or
- $\dfrac{E^{j-1} - E^j}{E^{j-1}} < \varepsilon$ or also
- the value itself $E^j < \varepsilon$.

The default stopping condition uses (5.11) and stops at $E^{j-1} - E^j < 10^{-10}$. The signed difference has the property that the training stops when the objective function value becomes worse, because then $E^{j-1} - E^j < 0 \Leftrightarrow E^j > E^{j-1}$. But it also stops when the objective function value improves less than 10^{-10}, which can be considered as converged.

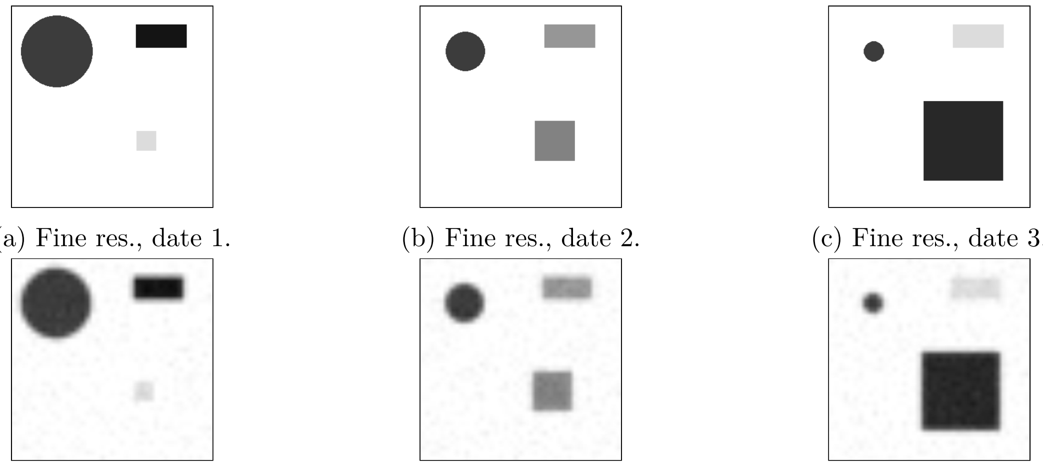
Apart from the above described stopping conditions, it is also possible to set a minimum and a maximum number of iterations. When setting both to the same value, the number of iterations is fixed.

Stopping the training does not necessarily mean that the dictionary from the last iteration is used. There is an option, called *Best Shot Dictionary*, that uses the dictionary from the iteration where the test set error was the lowest.

5.6 Experiments

This section describes results from different experiments with three different sets of images. First the test image sets are introduced. Then a very extensive test section follows, which compares different configuration options to see, what influence they have and also to optimize the default options. Finally, a comparison is drawn between fused products from SPSTFM and from STARFM / ESTARFM.

The artificial image set is similar to the one described in [HS12]. So the image background is white, a circular structure changes its size, a non-square rectangular structure changes its value and a squared structure changes its size and value. The clear, homogeneous structures make it easy to spot artefacts and the behaviour on size or value changing objects. The images are shown in Figure 5.2 in original size and lossless, so they can be extracted from the digital version. However, they are also available together with the code as test images. To compare it with real dimensions, a fine resolution of 30 m per pixel and a coarse resolution of 250 m per pixel is assumed. The stated pixel dimensions refer to the fine resolution.


(a) Fine res., date 1. (b) Fine res., date 2. (c) Fine res., date 3.

(d) Coarse res., date 1. (e) Coarse res., date 2. (f) Coarse res., date 3.

Figure 5.2: Artificial images. Coarse resolution images use bilinear interpolation and have an signal-to-noise-ratio of 35 dB.

The artificial images have a width and height of 335 pixels and use 8-bit unsigned type (0 represents black, 255 represents white). The coarse resolution image is made from the fine resolution image by downscaling by $\frac{25}{3}$, then adding noise with a signal to noise ratio of 35 dB, and finally scaling back. Scaling is done with with bilinear interpolation (and anti-aliasing for the downscaling). The circle has its centre located at pixel coordinates $(75.5, 75.5)$ and a fixed value of 60. Its radius changes from 60 pixels (1800 m) to 33 pixels (990 m) to 17 pixels (510 m). The non-square rectangle has its centre located at pixel position $(250, 50)$ and a fixed size of 85×39. Its values change from 20 to 150 to 220. The square has its centre located at $(225, 225)$. Its size changes from 33 pixels (990 m) to 67 pixels (2010 m) to 133 pixels (3990 m). Its value changes from 220 to 130 to 40.

The single-channel images are real near-infrared (NIR) 16-bit unsigned integer images taken from MODIS and LANDSAT satellites. The dates are the days 158, 238 and 254 in 2016. Their centre coordinates are 12° 28' 6.35" E, 54° 16' 7.96" N and they have a size of 300×300. These are shown (as small size previews) in Figure 5.3. Note the rather larger time gap between the first two dates (80 days) and the small gap between the second two dates (16 days). One can see a huge difference in brightness, which is shown in the histogram in Figure 5.4. There one can notice that the brightness is not only shifted, but also the range of the values (contrast) is different. These are problems that arise in real images due to different recording properties across different satellites. These images are not always used in full size, but cropped to 708×708 with a 4 pixel border on each side for the prediction area (see Section 4.1.3). This results in a 700×700 image. This is located at $(700, 700)$. The cropped image covers then a bit water in its top left corner, but mostly land.

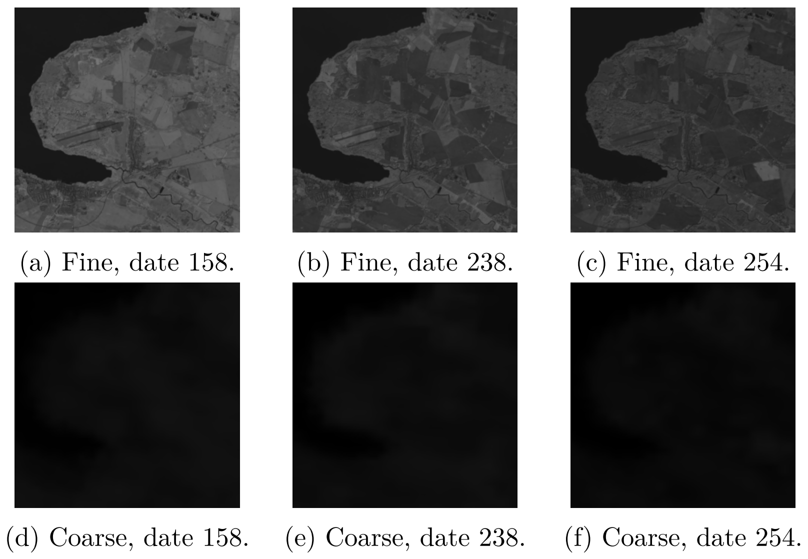

(a) Fine, date 158. (b) Fine, date 238. (c) Fine, date 254. (d) Coarse, date 158. (e) Coarse, date 238. (f) Coarse, date 254.

Figure 5.3: Real single-channel images. Coarse resolution images use bilinear interpolation. There is a huge difference in brightness between fine and coarse resolution images.

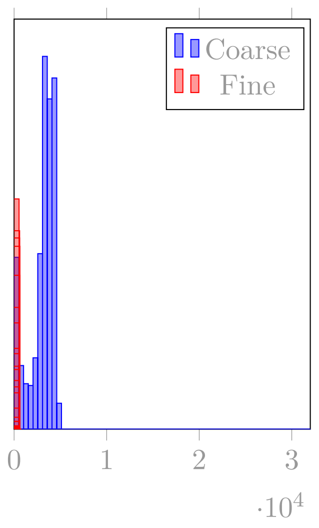


Fig. 5.4: Histogram of fine (red) and coarse (blue) images of date 158.

The multi-channel images are real 16-bit unsigned integer images taken from MODIS and LANDSAT satellites. They have four channels: Red, Shortwave-Infrared (SWIR), Near-Infrared (NIR) and Green. This should allow to use the improved build-up index, see (5.8) for weighting. Their centre coordinates are 6° 57' 37.08" E, 51° 19' 31.38" N and they have a size of 444 × 444. The dates are the days 220, 236 and 243 in 2016. There is no preview for the images.

The image error is measured as the mean absolute difference (MAD) between the result image $\tilde{I}_{f,2}$ (also called fused product) and the reference image $I_{f,2}$. This is very similar to (5.12), but using the images instead of the patch matrices:

$$\mathrm{MD} := \frac{1}{w\,h} \|\tilde{I}_{f,2} - I_{f,2}\|_1, \tag{5.13}$$

where w and h are width and height of the images, respectively. A lower error means higher quality. However, for low depth images (8-bit) the MAD is lower than for the same images represented in a higher depth.

5.6.1 Configuration tests

From first experiences with SPSTFM a default configuration is chosen. This is not optimal yet, but used for the configuration tests. To make it complete here the configuration test default options are listed:

- dictionary initialization: normalize each patch in each resolution separately

- dictionary size: 256
- number training samples: 2000
- patch size: 7
- patch overlap: 2
- minimum number of training iterations: 0 (however, one iteration is always done to check convergence, except the maximum number of training iterations is also 0)
- maximum number of training iterations: 20 (arbitrarily chosen)
- subtract mean value: no
- divide by: coarse resolution image variance
- sampling strategy: most variance
- weight difference limit $\delta = 0.2$
- use build-up index for weights: no (since the images must have the required bands and this might not be the default case)
- band indices of red, NIR, SWIR for build-up index: 0, 1, 2 (only used if build-up index is used)
- threshold of build-up index: 0 (only used if build-up index is used)
- all resolution options in training (GPSR, K-SVD, objective function): coarse
- K-SVD singular values weight: coefficients
- stopping condition function: objective (5.11) (using coarse resolution)
- stopping condition: signed difference, with value 10^{-10}, i. e. $E^{j-1} - E^{j} < 10^{-10}$
- test set size: 4000 (only used if test set error is used as stopping condition function)
- GPSR options:
  - main algorithm tolerance: 10^{-7} (for training and reconstruction)
  - debiasing tolerance: 10^{-1} for training and 10^{-2} for reconstruction
  - number of iterations k for the main algorithm: $5 \leq k \leq 5000$
  - number of iterations k for the debiasing: $1 \leq k \leq 200$
  - sparsity balancing factor τ: automatic ($\tau = 0.1 \, \|D^{\top} p\|_{\infty}$)
  - use continuation: yes (speeds up GPSR algorithm)

Table 5.2: Average time for training and reconstruction. Average taken over 75 iterations.

	Concatenated	Coarse	Averaged	Fine	Concat. / Avg.
Computation time	152 s	428 s	622 s	728 s	506 s

In the following tests we deviate with some options from the above defaults to see their influence and try to improve the default options. The changed options are stated there.

The first test tries to answer the question from Section 5.3 which resolution should be selected to find the sparse representation coefficients. These are then used to initialize the K-SVD algorithm, which updates the dictionary and again the coefficients. The following five combinations are tested here:

- *Concatenated* – use the concatenated dictionary and patch matrix to find the coefficients and also update the dictionary in a concatenated fashion directly. [HS12] states that [YWHM10] uses this, but for satellite images it were not appropriate. This approach can be found in Algorithm 1.
- *Coarse* – use the coarse resolution only to find the sparse representation coefficients and in K-SVD only the updated coefficients stem from the coarse resolution SVD. This and the remaining options below use Algorithm 2.
- *Fine* – the same as *Coarse*, but with the fine resolution.
- *Averaged* – both resolutions are used and then the average of the coefficients is used.
- *Concatenated / Averaged* – for the GPSR algorithm, which finds the sparse representation coefficients, the concatenated dictionaries and patch matrices are used. For the coefficient update in K-SVD the new coefficients of both resolutions are averaged.

These combinations are used for fusing the shapes image set with 75 iterations. After each training iteration a reconstruction is done to measure the error. Also the test set error is recorded to see whether real error and test set error correlate. The plot with the errors is shown in Figure 5.5. One can recognize a good match of the error shapes, especially for *Concatenated* and *Averaged* options. The errors of *Averaged* and *Fine* are very noisy. The highest average error is the one of the *Concatenated* option, which never goes below the untrained dictionary error. The lowest average and absolute error is the one with the *Coarse* option, *Concatenated / Averaged* is the second best option. One can also recognize that the interesting behaviour happens in about the first 15 iterations.

However, not only the errors depend on the resolution, but also the computation time. The computation times for training and reconstruction is averaged over the 75 iteration for each option. The result is shown in Table 5.2. The times for the resolutions are all different. The *Coarse* option is the second shortest. The shortest is *Concatenated*, but this had the largest error. The other options required more computation time and have also a larger error than *Coarse*. So it is reasonable to keep *Coarse* as default option.

Let us consider the computation time in more detail and split it up into separate tasks. So, generally, SPSTFM can be separated into training phase and reconstruction phase. The

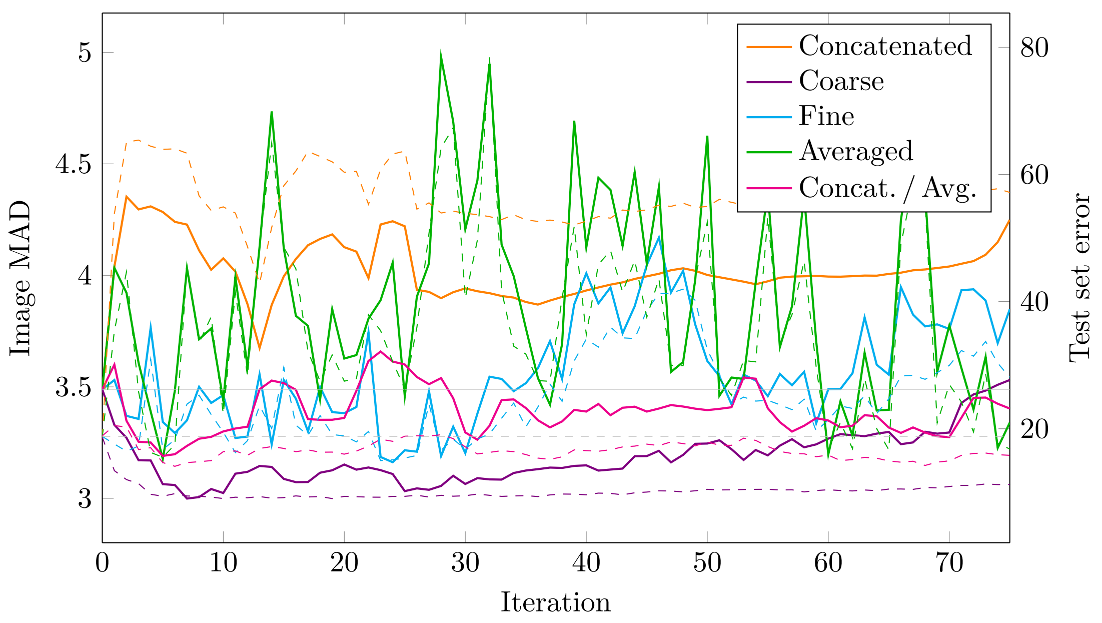
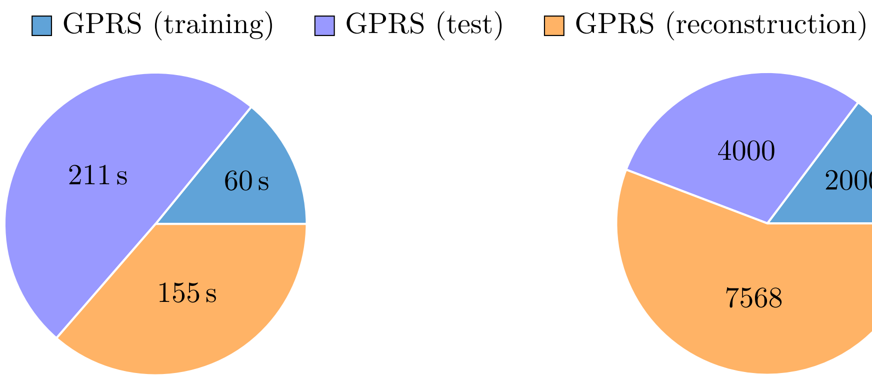


Figure 5.5: Errors of different training resolutions. Straight lines show the real error (MAD), dashed lines the test set error. The grey lines show the error values of the untrained dictionary.

first step in the training is to find the sparse representation coefficients using the GPSR algorithm. These are then used in the K-SVD algorithm that updates the dictionaries and coefficients. Optionally, there can be a test set to estimate the error. For that the GPSR algorithm is used as well. After the training phase, the reconstruction is done. For that the GPSR algorithm is used again. Since for reconstruction many patches have non-zero weights, even two GPSR evaluations have two be done for these patches. Figure 5.6a shows the ratio of times and Figure 5.6b shows the number of GPSR evaluations for the different tasks. The options include the default 2000 training samples and use 4000 test samples. The image consists of 4356 patches and in total there are 7568 non-zero weights. The time for the K-SVD algorithm is negligible (0.67 s) and not shown. The computation time of the sparse representation coefficients for training fits approximately to the ratio of total evaluations. The computation time for the test set coefficients is longer than expected and the time for the reconstructions is shorter than expected. For both tasks the same main GPSR tolerance of 10^{-7} has been used. The GPSR algorithm requires with increasing training iterations less time to find the coefficients, although it is always initialized with zero-coefficients. Finding the reason for these behaviours is left for further work. Nevertheless, the GPSR algorithm determines the runtime. So it would be worth to look for improvements to that algorithm in general. However, there are also some options involved that determine the behaviour of GPSR.

Hence, regarding the GPSR algorithm several combinations of different tolerance options are tested. There is the main algorithm tolerance (5.3) and the debiasing tolerance. All 9 combinations of the following training tolerances values are tested:



(a) Average computation times.

(b) Number GPSR evaluations.

Figure 5.6: Computation times and number of evaluations for one training iteration and reconstruction. The times are averaged over 75 iterations.

- main algorithm tolerance in the dictionary training: 10^{-5}, 10^{-6}, 10^{-7}
- debiasing tolerance in the dictionary training: 10^{-1}, 10^{-2}, 10^{-5}

and then the 9 combinations of these reconstruction tolerances:

- main algorithm tolerance in the reconstruction stage: 10^{-5}, 10^{-6}, 10^{-7}
- debiasing tolerance in the reconstruction stage: 10^{-1}, 10^{-2}, 10^{-5}

The fusions are done with 15 iterations with the artificial image set Figure 5.2. The test set has not been used here. Figure 5.7 shows the averaged training times against the image error (MAD) for different main algorithm tolerances in the training phase. Varying this, directly influences how precise – and thus also how long – the GPSR algorithm optimizes the representation coefficients. One recognizes a strong influence on the runtime, but only a rather slight influence on the average error. To make it more clear, 5 training iterations with main algorithm tolerance 10^{-5} cost approximately as much as one iteration with main algorithm tolerance 10^{-7}. Still, training only with a tolerance of 10^{-5} will not approach the error that a tolerance of 10^{-7} reaches.

However, there is also the debiasing tolerance in the training. It did not have a strong effect on the computation time. Let us have a look on the image error. The image error against the iteration for different debiasing tolerances is shown in Figure 5.8 for different main algorithm tolerances. One can recognize that a lower debiasing tolerance is not always beneficial. Only for a low main tolerance a low debiasing tolerance improves the accuracy further in the later iterations. This behaviour might give rise to an adaptive method of varying the GPSR tolerances in the training phase. So first a few iterations with a high tolerance for main algorithm and debiasing could be done. Then a few iterations with a medium tolerances and finally a few or one iteration with low tolerance could be done. This might give a low error with less computation time required compared to directly using low tolerances. This improvement is left for further work though.

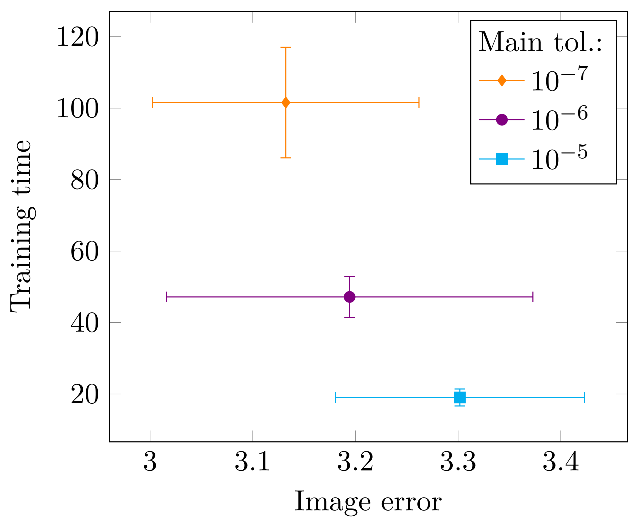
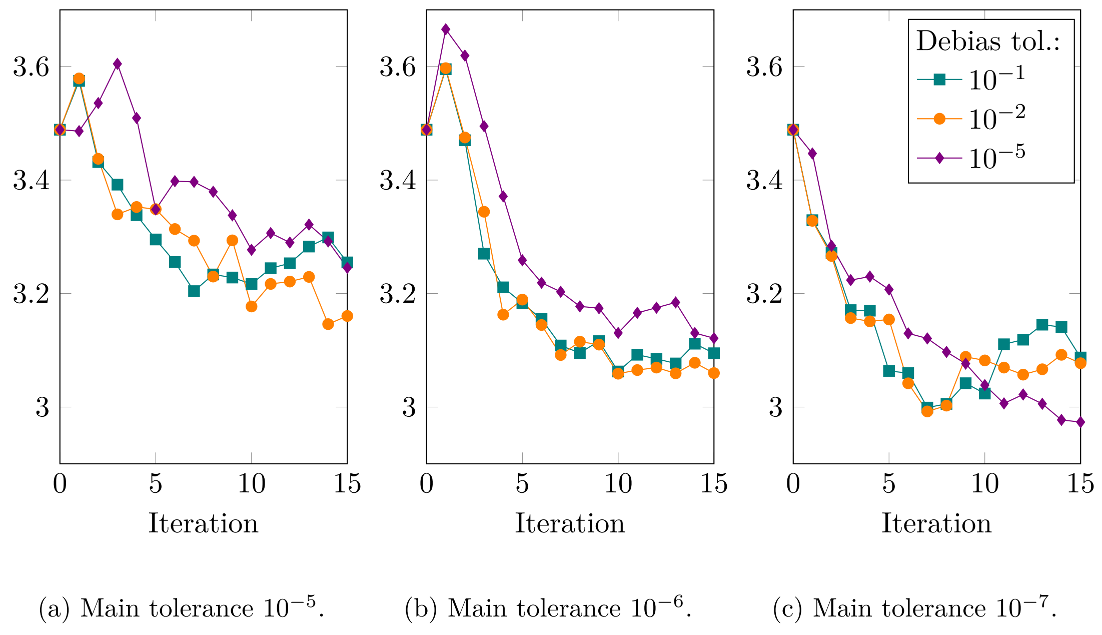


Figure 5.7: Average computation times (training only) and image errors (with standard deviations as error bars) for different main loop tolerances of the GPSR algorithm. The debiasing tolerance is fixed to 10^{-2} since its influence on the computation time was negligible. Averaging is done over 15 iterations. The main loop tolerance in the reconstruction stage is 10^{-7} and the debias tolerance 10^{-2}. The artificial image set has been used.



(a) Main tolerance 10^{-5}. (b) Main tolerance 10^{-6}. (c) Main tolerance 10^{-7}.

Figure 5.8: Average image errors across iterations with different debias tolerances of the GPSR algorithm in the training process. The main tolerance in the reconstruction stage is 10^{-7} and the debias tolerance 10^{-2}. The artificial image set has been used.

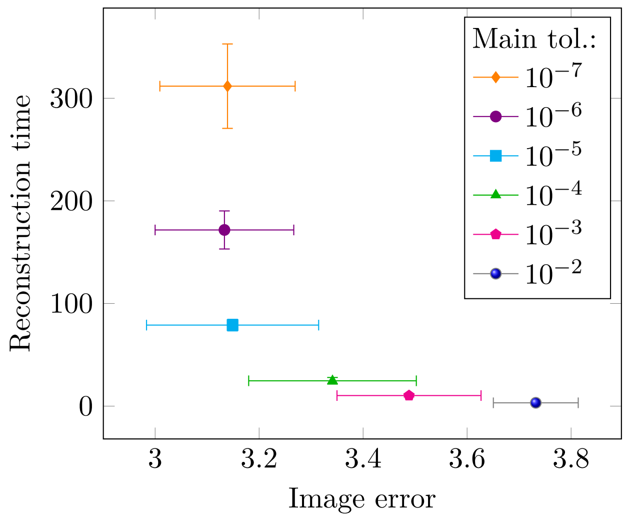
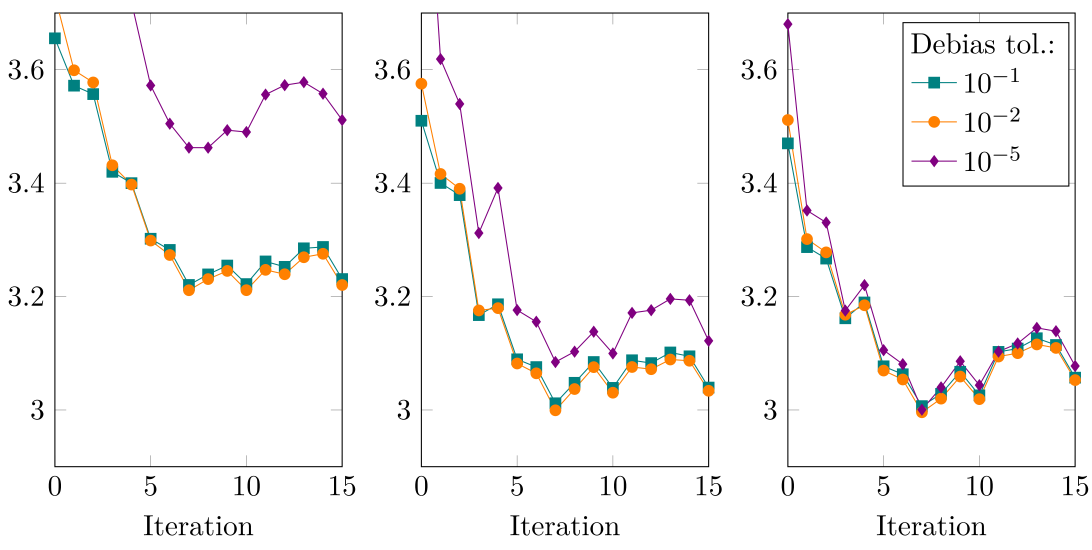


Figure 5.9: Average computation times (reconstruction only) and image errors (with standard deviations as error bars) for different main loop tolerances of the GPSR algorithm. The debiasing tolerance is fixed to 10^{-2} since its influence on the computation time was negligible. Averaging is done over 15 iterations. The main loop tolerance in the training stage is 10^{-7} and the debias tolerance 10^{-1}. The artificial image set has been used.

The training is a large portion in Figure 5.6a, but note, that the training phase is only required once per time series. In a typical time series, like shown in Table 2.1, date 0 and 16 are used for training. These resolution pair dates are the same for the fusion of every date in between. Therefore, once the dictionary is trained, it will be used to reconstruct the images of dates 1 – 15. So this means the computation time for reconstruction has also to be considered for different GPSR tolerances in the reconstruction phase. Figure 5.9 shows the averaged reconstruction time against the image error (MAD) for different main algorithm tolerances in the reconstruction phase. Note, the tolerances in the training are fixed to default, so these configurations use all the same dictionary. It is interesting that the reconstruction time changes very much (more than $\times 3$) from main tolerance 10^{-5} to 10^{-7}, while the error is nearly the same. The error even increases slightly from main tolerances 10^{-6} to 10^{-7}. So it does not seem to be worth to use a low main tolerance in reconstruction. However, using a main tolerance less than 10^{-5} increases the error. So a value near 10^{-5} appears to be the best choice here.

Now consider the debiasing tolerances in the reconstruction phase. Figure 5.10 shows the errors against the iteration for different debiasing tolerances and for different main tolerances. Here, a similar behaviour can be seen. A low debias tolerance is not beneficial at all. In the case of a high main tolerance of 10^{-5} or 10^{-4} it even increases the error considerably. Next, a deeper look into the reasons for this behaviour is taken.

To make the effect of a low debiasing tolerance in the reconstruction better visible, a bad trained dictionary is used. Therefore the *Concatenated* option for dictionary and patch matrix in the training are used and only one iteration is done. Then the effect is visible

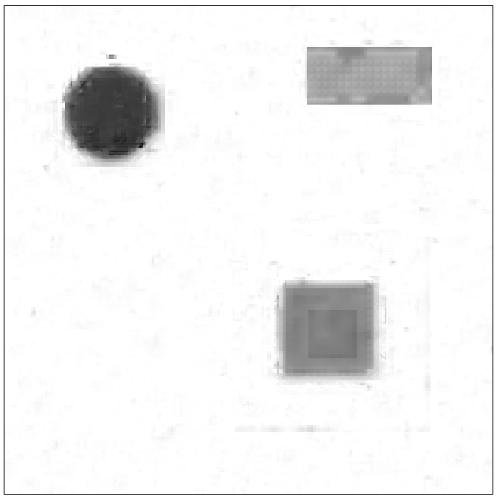


(a) Main tolerance 10^{-4}. (b) Main tolerance 10^{-5}. (c) Main tolerance 10^{-6}.

Figure 5.10: Average image errors across iterations with different debias tolerances of the GPSR algorithm in the reconstruction process. The main tolerance in the training stage is 10^{-7} and the debias tolerance 10^{-1}. Therefore all dictionaries at the same iterations are equal. The artificial image set has been used.

for the very noisy artificial image set (Figure 5.2). Figure 5.11 compares two images with rather opposite configurations. Figure 5.11a uses a low tolerance (10^{-5}) for debiasing in reconstruction stage, while Figure 5.11b uses a high tolerance (10^{-1}). One can see, that the low tolerance gives a lot of artefacts in this situation. In combination with a high tolerance (10^{-5}) for the main algorithm in reconstruction stage this effect is even stronger. This failure of handling noise is kind of expected and also mentioned in [NW^{+}07] for strong debiasing. The tolerances in the training do not influence this effect in a visible way.

When looking at Figure 5.11a one might notice the artefacts in the non-square rectangle. These do not stem from the noise, but rather from a bad trained dictionary in general. However, the question arises why there are so large differences in the quality across this area. This is shown larger in Figure 5.12 with an even more extreme example. There only the upper right non-square rectangle is shown. It should be homogeneous, compare to the reference image in Figure 5.2b. In the bad result some patches are too bright and too structured. And some patches seem to be predicted quite well although they are neighbours. Also near the border many patches look quite well. Let us now consider the two neighboured patches in the red marked area; one is bad, one is ok. However, if we analyze why this result for these both patches is so different, it turns out that both are actually badly reconstructed. The main difference is that the weights for the left patch exceeds slightly the weighting difference limit δ, see (5.9) and the black area in the top right of Figure 5.13. The left patch is therefore made by only one predicted patch. The right patch is slightly below the weighting difference limit and in this case the weighted


(a) Main alg. tol.: 10^{-5}, debiasing tol.: 10^{-5}.

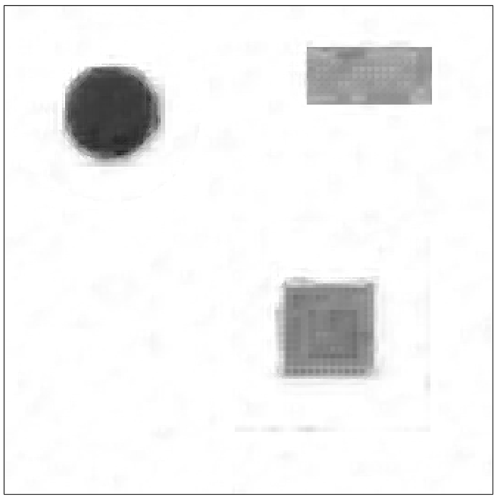

(b) Main alg. tol.: 10^{-7}, debiasing tol.: 10^{-1}.

Figure 5.11: Artefacts for different GPSR tolerances at reconstruction stage. Main tolerance in the training is 10^{-7} and debiasing tolerance in the training is 10^{-1}. The dictionary is trained with one iteration.

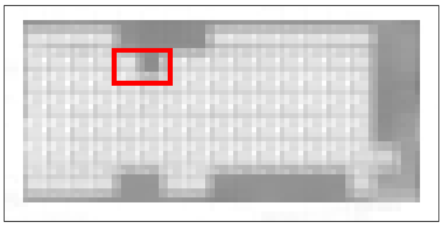


Figure 5.12: Artefacts in the area of the rectangle. Concatenated samples and dictionary are used in the training. Only one iteration is done.

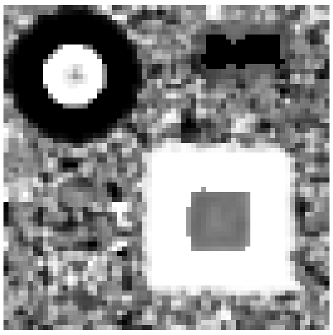


Figure 5.13: Weights $w1$ for the artificial image set, see also Figure 5.2.

average of both predicted patches gives a rather good result although each of them for itself is bad. When looking at the representation coefficients for these patches, the GPSR algorithm finds a very low number of atoms (four in this example) to represent the low resolution patches quite accurately, but the corresponding high resolution patches do not fit at all. The test set error value (5.12) reflects this mismatch quite well in this case, as is rises from untrained state to trained state by a factor of 2.8. This can also be seen in Figure 5.5, where the *Concatenated* image error curve and its test set set error rises steeply in the first 2 iterations. That the averaging lowers the error for appropriate weights also explains why the test set error can be much higher than the actual image (note the two different scales in Figure 5.5). So this analysis supports the reliability that the test set error is a good indicator for the quality of the dictionary.

Considering the weights in Figure 5.13 they appear to have more variance in comparison to [HS12]. In the background area, where nothing but noise changes, the weights seem to be too noisy. Especially in the non-square rectangular region the weights should not hit the weight difference limit δ. One could increase δ or choose a different weighting method. [HS12] does not state how the weights in their artificial image set were calculated, but it seems the method is different than the alternative option, described in Section 5.4. Increasing δ would not help against the noisy behaviour, so generally using a different weighting method could provide a better solution. Also, a low δ might be important in cases where strong changes appear (structures, clouds, etc.) and these patches should be not used at all.

Let us discuss the image error estimation further. So there is a rather reliable method, which is very costly to evaluate – the test set error. The costly part is finding the sparse representation coefficients for them. However, the objective functions has not yet been considered. These are extremely cheap to evaluate. Furthermore there is one alternative, which has not been mentioned yet. Using the training set error yields a similar method to the test set error, but for free. When using the default resolution option *Coarse*, the sparse representation coefficients for low resolution samples from the training set are required for training anyway. However, these can be utilized to predict the fine resolution patches and calculate the MAD analogue to (5.12) to estimate the dictionary quality. The performance of this and the other options can be compared with Figure 5.14. Both objective functions show very similar behaviour, but it is different to the actual error. Note, that in the first iteration the objective functions values increase significantly, while the actual error decreases. The test set error comes close to the shape of the actual error. Nevertheless, the behaviour of test set error and actual error are expected to be different, since the actual error is much lower because of the weighting, which often smoothes errors out, as described above. Also some spikes may appear from single patches that are used often and that have a large error, because the image has much repetitions. The training set error is very similar to the test set error, but seems to be a bit smoother. However, it could replace the test set error, which takes a large amount of time in the training, see Figure 5.6a.

There are still options, which have not been discussed yet, like whether or not to subtract the mean value. The sampling strategy *Random* has also not been considered so far. So Figure 5.15 shows the influence of these options and their test set error. It can be seen, that subtracting the mean value increases the error for the artificial image set permanently. It does not approach the level of the default configuration. Surprisingly the random sampled

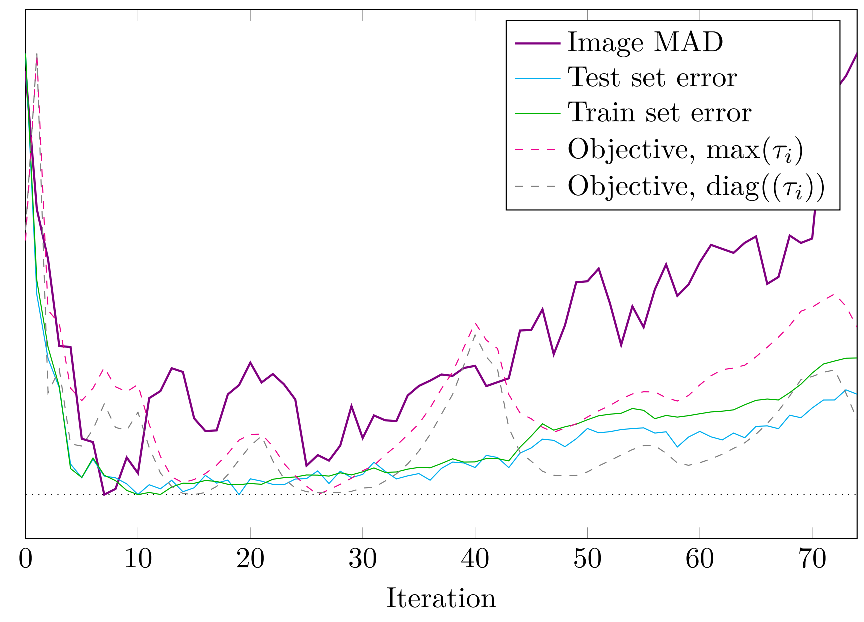


Figure 5.14: Stopping criterion functions behaviour compared with the actual image error. The functions are scaled to fit in one plot; only the shape should be shown here.

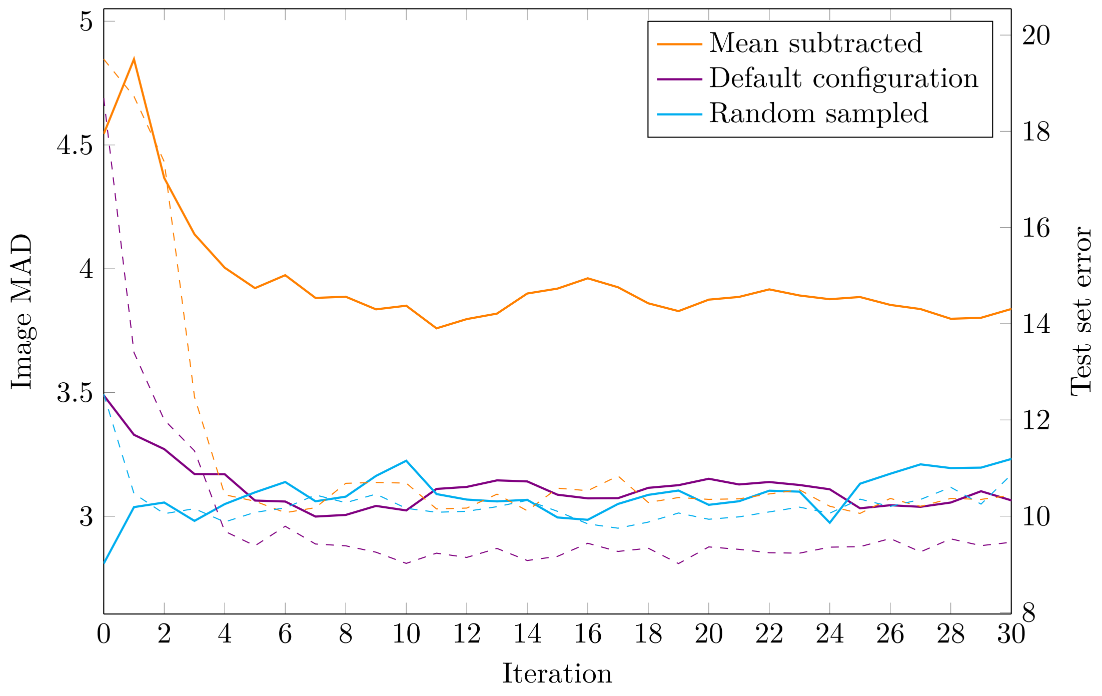


Figure 5.15: Errors for different options. Image MAD are plotted as straight lines and test set MAD as dashed lines.

initialized dictionary gives the best result overall. The corresponding test set error is initially also the lowest, but strangely decreases in the first iterations, while the actual error with the random sampled training data increases. Since the actual image MAD is not available in practice, this could not have detected. Also, after a few iterations the MAD is comparable to the one of the default configuration. It might well be that the low initial error in the random sampled configuration is a coincidence due to the white background. All values above 255 are limited to 255 and thus have no error at all. The test would be more meaningful with a light grey background, but the white background is used to be comparable with [HS12].

Finally, before the comparison with STARFM and ESTARFM is made, some preparing tests are performed. In Figure 5.4 can be seen one effect that appear in real images: different data ranges across fine and coarse resolution. The mean value is not a problem since only differences are used and therefore the different means cancels out (see Section 5.1). However, the difference in standard deviations is still present in the difference image. There are options to handle this and the next test compares these options.

- The *default* configuration has the same options as described in the beginning of this section. Summarizing the relevant options, this configuration divides all difference samples by the variance of the coarse resolution difference images, as suggested by [HS12], but does not subtract any mean. The dictionary is normalized at initialization in the way that every fine and every coarse patch has Euclidean length 1. For the dictionary update with K-SVD in the training the singular values are used to scale the coefficients. The following configurations list only the differences to the *default* configuration.
- The *div. sep. var.* configuration divides the fine and coarse difference samples by the variance of fine and coarse resolution difference images, respectively.
- The *div. sep. std. dev.* configuration divides the fine and coarse difference samples by the standard deviation of fine and coarse resolution difference images, respectively.
- The *scale dict. direct* configuration uses for the initial dictionary the option to scale all patches by the same factor and for the dictionary update via K-SVD the *scale dictionary direct* option, which scales the dictionary with the singular values and in addition divides by a fixed factor. In both cases the factor is chosen such that the norm of the first fine atom is 1. These options fit together as explained in Section 5.3.
- The *scale dict. normal* configuration uses for the initial dictionary the option to scale all patch pairs such that one of them is normalized and the other has a smaller norm. For the dictionary update via K-SVD the *scale dictionary normal* option is used, which scales the dictionary with the singular values, but divides each atom by the larger singular value. See Section 5.3 for details and why these options fit together as well.
- The *div. / scale dict. direct* configuration combines the *scale dict. direct* options with the *div. sep. std. dev.* options.
- The *div. / scale dict. normal* configuration combines the *scale dict. normal* options with the *div. sep. std. dev.* options.

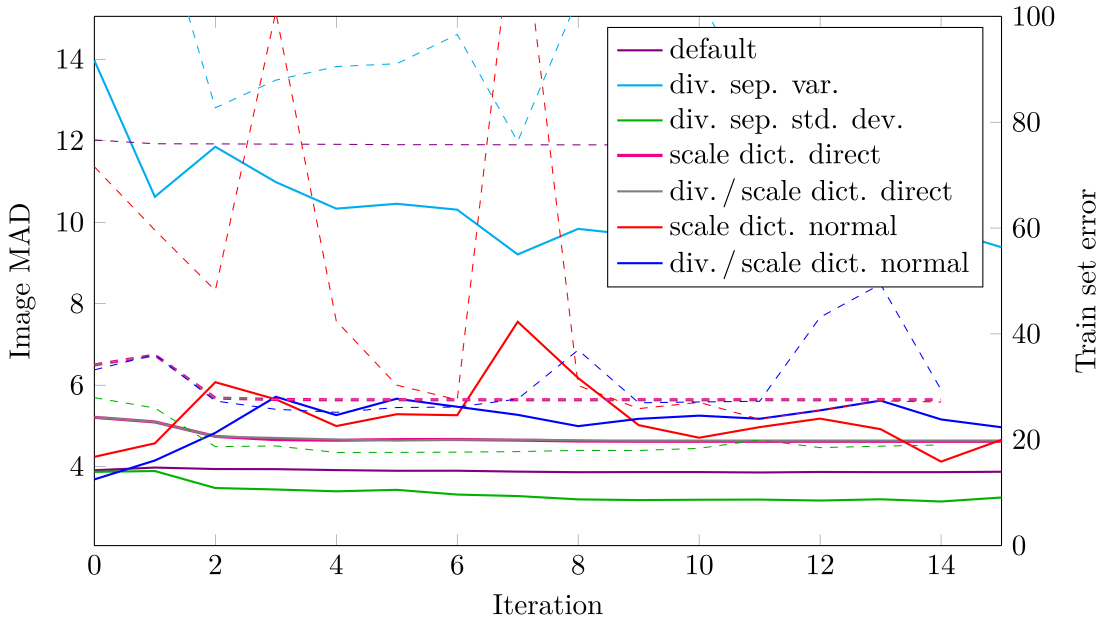


Figure 5.16: This plot shows the behaviour of different options to handle images that have different standard deviation in fine and coarse resolution images. The solid lines show the actual Image MAD and the dashed lines the training set error.

The test performs 15 iterations and shows also the training set error to estimate the dictionary quality. Figure 5.16 shows all curves in one plot. The *default* configuration has got a rather low image MAD, but this only due to the weighting. The dictionary has a bad quality, which can be seen from the large test set error of around 75. The resulting image shows strong artefacts, like Figure 5.12, which indicates a bad dictionary quality. The *div. sep. var.* configuration shows the largest image and test set error. This means, dividing by variance is not the right way to normalize an image. On the contrary *div. sep. std. dev.* configuration shows the lowest image and test set error. Dividing by the separate standard deviations is an elegant way of handling differences in the data ranges of fine and coarse resolutions. It makes the data ranges of the training data equal. The dictionary is then trained for this data. At reconstruction stage the coarse patches are normalized to this very data range, coefficients found, the fine resolution patch predicted, which then is denormalized to its natural data range. Thus the dictionary does not need to be able to match different data ranges for fine and coarse resolution patches. The *(div. /) scale dict. direct* configurations perform equally, but slightly worth than the *div. sep. std. dev.* configuration. The *scale dict. normal* configuration has large deficits in the train set error, which makes it unreliable. The *div. / scale dict. normal* configuration improves the behaviour but does not outperform the other good configurations. So for real image sets like Figure 5.3 the *div. sep. std. dev.* configuration seems to be the best one. A test with this configuration on the original artificial image set gives a minimum image MAD of 3.1 and a test set error of 17.4, while the default configuration gives an image MAD of 3.1 and a test set error of 18.5. Hence, it is reasonable to use the option to divide by separate standard deviation by

Table 5.3: Errors for different methods with the artificial image set from Figure 5.2.

Method	STARFM	ESTARFM	STARFM	SPSTFM	SPSTFM	SPSTFM
Pair date	1	both	3	1	both	3
Image MAD	14.3	17.1	9.8	6	3.2	7.6

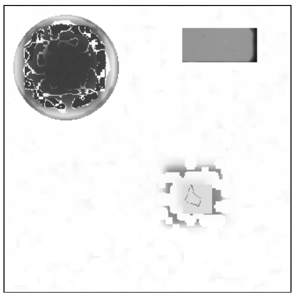
default.

5.6.2 Comparison with STARFM and ESTARFM

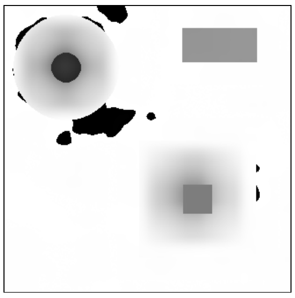
This section shows test results not only of SPSTFM products, but also of STARFM and ESTARFM. STARFM actually requires just three images as input, while the other models require five, see Tables 2.2a and 2.2b. To make the comparison between STARFM and SPSTFM fair, STARFM is used with both possibilities; using dates 1 and 2 (denoted by *from left*) and using dates 3 and 2 (denoted by *from right*) with the dates as given in Table 5.1. To make it even fairer, SPSTFM is also used without weighting in the reconstruction stage to get results *from left* and *from right* as well. This is equivalent to using $w_1 = 1$, $w_3 = 0$ and $w_1 = 0$, $w_3 = 1$, respectively. The other images are still required for training. Nevertheless, the STARFM settings are not optimized for this problem, because their optimization is out of scope of this work. So STARFM might not give the best results that it could give.

The configuration used in this section is slightly different to the one used in the previous section. Based on the configuration results of the GPSR tolerances, here a main algorithm tolerance of 10^{-6} is used for training and 10^{-5} for reconstruction. The debias tolerance is 10^{-1} for training and reconstruction. Also, the test set is not used, since the reconstruction is done for every iteration for testing purposes and – if needed – the training set error is available anyway. From the last test in the previous section, see Figure 5.16 and its context, it came out that in any case the fine and coarse training samples can be divided by the standard deviations of the fine and coarse difference images, respectively. So this option is used here as well. The prediction area is set to have a border of 5 pixels.

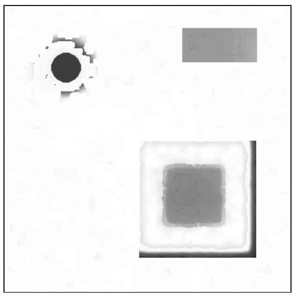
The first test uses again the artificial image set from Figure 5.2. The minimum errors of the different models and input dates are shown in Table 5.3. The (best) result images are shown in Figure 5.17. In the products *from right* there is a strong error in the border in the bottom and right of the square shape. This might stem from a slight shift of the coarse resolution image because of the reduced image size when downsampled. Apart from that some more interesting observations can be drawn. STARFM could not deal with the circle shape, which might be resolvable with different settings. In [HS12] the circle is much better predicted and just has a very blurred edge. Also the result of ESTARFM does look very strange. The black regions around the circle and the square look like an overflow. When replacing these areas by white, the error is 10.3. For this an implementation in IDL has been used, but the implementation in the image fusion framework yields similar results. Especially the large blurs around the circle and the square seem to be specific for ESTARFM. So ESTARFM as well as STARFM cannot handle structural changes well. Also worth to mention is that STARFM (from right) and ESTARFM matched the non-



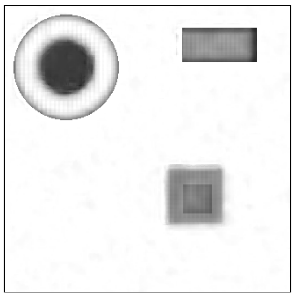
(a) STARFM from left.



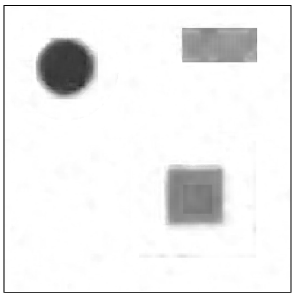
(b) ESTARFM.



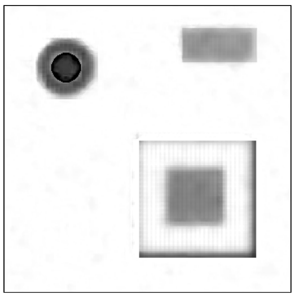
(c) STARFM from right.



(d) SPSTFM from left.



(e) SPSTFM with weights.



(f) SPSTFM from right.

Figure 5.17: Products for artificial image set with different fusion models, input images and settings. Compare with Figure 5.2b.

square rectangle area quite well. SPSTFM with weights gave the overall best results and reproduce the size changing shapes quite well, but the non-square rectangle not as good as ESTARFM. SPSTFM *from left* shows a border around the circle, which should not be there. Similarly SPSTFM *from right* has a border around the square. These stem from the difference image, which SPSTFM uses as actual inputs. There the borders are blurred because of the coarse resolution. The predicted fine resolution patches apparently follow this blurred behaviour and thus fails to add enough to reach white in these borders. However, by using the weights these effects almost vanish. So indeed the weights are a strong component that makes SPSTFM suitable for structural changes, but also without weights the results are not too bad.

Before using real images, which includes several additional effect, the artificial set (Figure 5.2) is used again in its modified variant. For that the pixel values of the coarse resolution images are divided by 5 to simulate a different standard deviation, similarly as in the single channel set (Figure 5.3). Table 5.4 shows the minimum errors of the different models and input dates. The (best) result images are shown in Figure 5.18. The first observation is that STARFM completely fails at matching the pixel values (colour) of both rectangular shapes. These heavily depend on the input image pair date. So, not even the fixed size object is reconstructed in a reasonable way. The size changing shapes have wrong sizes and are mostly solid filled. The error is huge in both STARFM products, as can be seen from Table 5.4. The ESTARFM (imagefusion implementation) product matches all values very accurately. However, at the non-square rectangle has got again a strange artefact, which might vanish with the right settings. Fixing this manually lets the error decrease to

Table 5.4: Errors for different methods with a modified artificial image set as described.

Method	STARFM	ESTARFM	STARFM	SPSTFM	SPSTFM	SPSTFM
Pair date	1	both	3	1	both	3
Image MAD	21.5	12.5	32.7	5.7	3.3	7.3

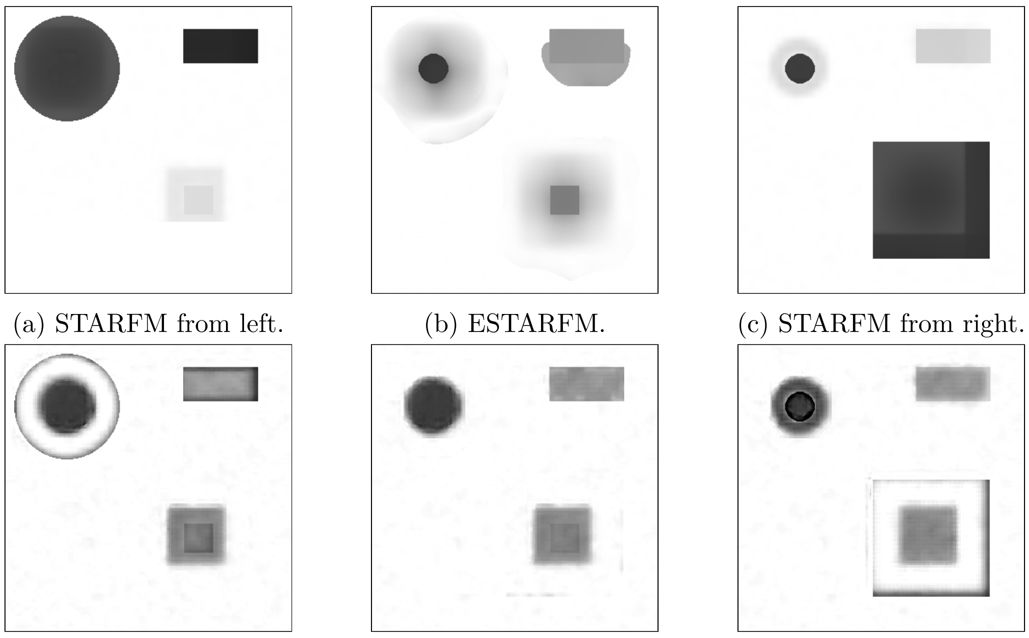

(a) STARFM from left. (b) ESTARFM. (c) STARFM from right.
(d) SPSTFM from left. (e) SPSTFM with weights. (f) SPSTFM from right.

Figure 5.18: Products for the modified artificial image set (dark coarse resolution images) with different fusion models, input images and settings. Compare with Figure 5.2b.

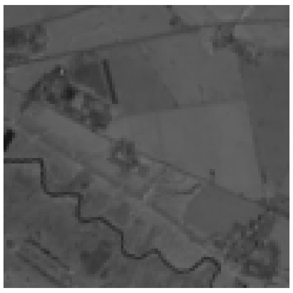
Table 5.5: Errors for different methods with the semi-real set from Figure 5.3.

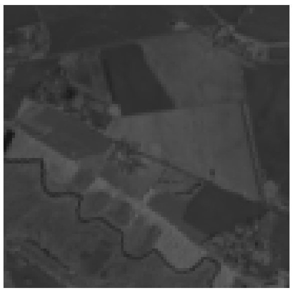
Method	STARFM	ESTARFM	STARFM	SPSTFM	SPSTFM	SPSTFM
Pair date	158	both	254	158	both	254
Image MAD	1446	1035	803	1257	637	629

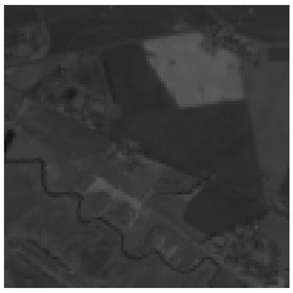
10.5. This error is still rather large due to the failure in matching the sizes of the circle and square. The SPSTFM product are very similar to the ones from Figure 5.17. The products *from left* and *from right* are indeed slightly better, as they do not show a patch pattern around the circle and the square. The errors in Table 5.4 confirm that. However, the error for the weighted product has slightly increased, but in an insignificant order (0.1). So SPSTFM can handle images with different standard deviations without preprocessing.

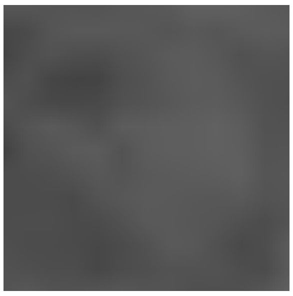
The next tests are performed with some of the real images from Figure 5.3. However, before using the complete set, the effects of the coarse resolution images should be ignored. They do not only differ in standard deviation (see Figure 5.4), but also suffer in quality. So first only the fine resolution images are used with artificial coarse resolution images made from the fine resolution images. The coarse resolution images have been downsampled (with anti-aliasing) and upsampled again to reduce the resolution. Therefore the histograms of the fine resolution images look the same as the ones of the corresponding coarse resolution images. These images do not hold structural changes, although the time dates are spread across 96 days. In practice structural changes are not very common. For easier comparison of the fused products with the reference image, the lower right 100×100 region (of the prediction area) is zoomed in Figure 5.19. The selected region shows different grey value areas across the three images. Some areas turn a lot darker from date 158 to date 238, which stay dark in date 254. Other areas turn dark from date 238 to date 254, while one area shows the opposite behaviour. So this 100×100 region is appropriate to take a close look how the algorithms behave. From the artificial coarse resolution images it is obvious how little one can recognize even from optimal images of that resolution. The real images have even more issues by not being very precise in some areas.

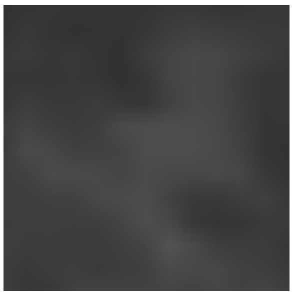
The image MADs for the set with the artificial coarse resolution images are shown in Table 5.5 and the lower right 100×100 region of the fused products is shown in Figure 5.20. ESTARFM (imagefusion implementation) performs worse than STARFM *from right.* The results of SPSTFM are better than the ones from STARFM and ESTARFM. For SPSTFM and STARFM the error *from right* is about the half as the one *from left.* When using weights for SPSTFM, the error even increases slightly. From looking at Figure 5.20f one can see that borders and small areas that change are not matched accurately. Basically it seems to increase and decrease the pixel values as the coarse resolution image suggests. However, reconstruction of details from the coarse resolution seems to be not realistic with real images. The training did not work well with these images as the lowest error *with weights* and *from right* occurred with the initial dictionary before training. So the whole training procedure has not been successful. Nevertheless, the errors are rather moderate. Note, that the images have a depth of 16-bit and the data range, as can be seen from Figure 5.4, is effectively 15-bit. So an error of 642.5 here corresponds to an error of $642.5 \cdot \frac{2^8-1}{2^{15}-1} = 5$ in an 8-bit

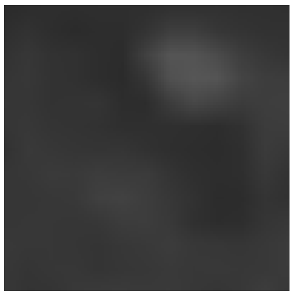

(a) Date 158, fine.

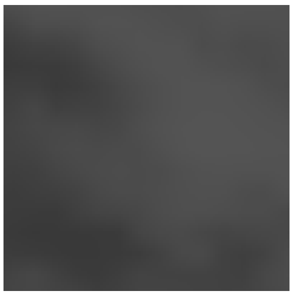

(b) Date 238, fine.

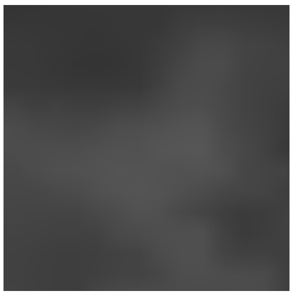

(c) Date 254, fine.


(d) Date 158, art. coarse.


(e) Date 238, art. coarse.


(f) Date 254, art. coarse.


(g) Date 158, coarse.


(h) Date 238, coarse.

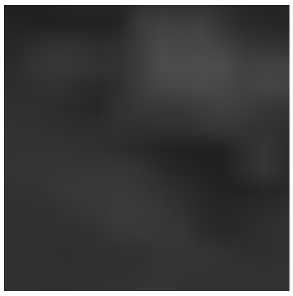
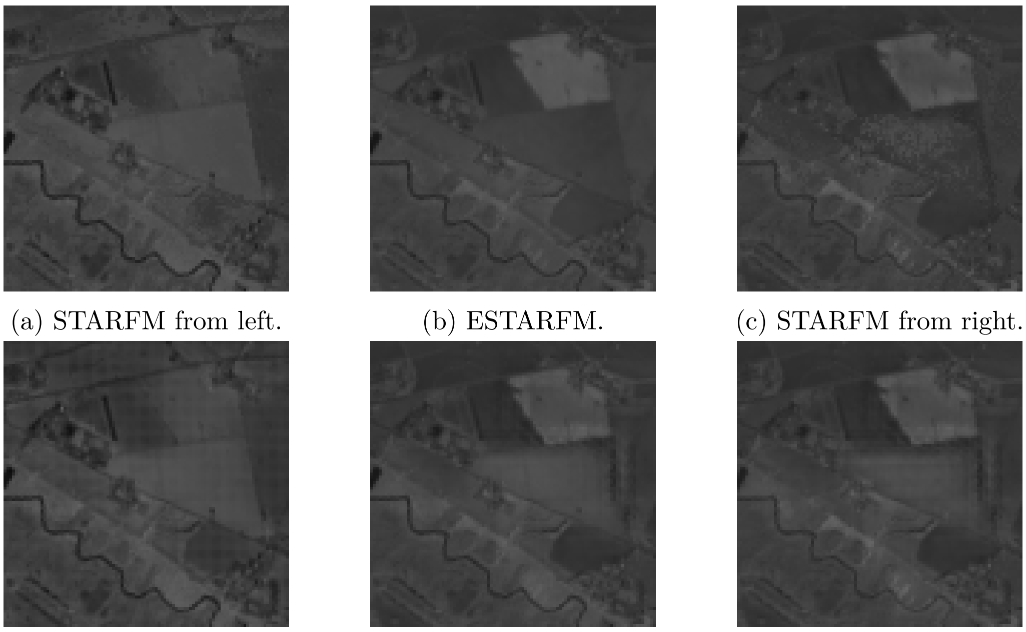

(i) Date 254, coarse.

Figure 5.19: Lower right 100×100 region of Figure 5.3, where some areas change their grey value (crop growth) differently. The fine resolution images are shown in the top, the artificial coarse resolution images in the middle and the real coarse resolution images in the bottom. The latter images are multiplied by the ratio of the standard deviations between fine difference image and coarse difference image, which is 4.7178, only to make the image content more visible. This multiplication is not used for fusion.


(a) STARFM from left. (b) ESTARFM. (c) STARFM from right.

(d) SPSTFM from left. (e) SPSTFM with weights. (f) SPSTFM from right.

Figure 5.20: Products for semi-real image set with different fusion models, input images and settings.

Table 5.6: Errors for different methods with the real single-channel set from Figure 5.3.

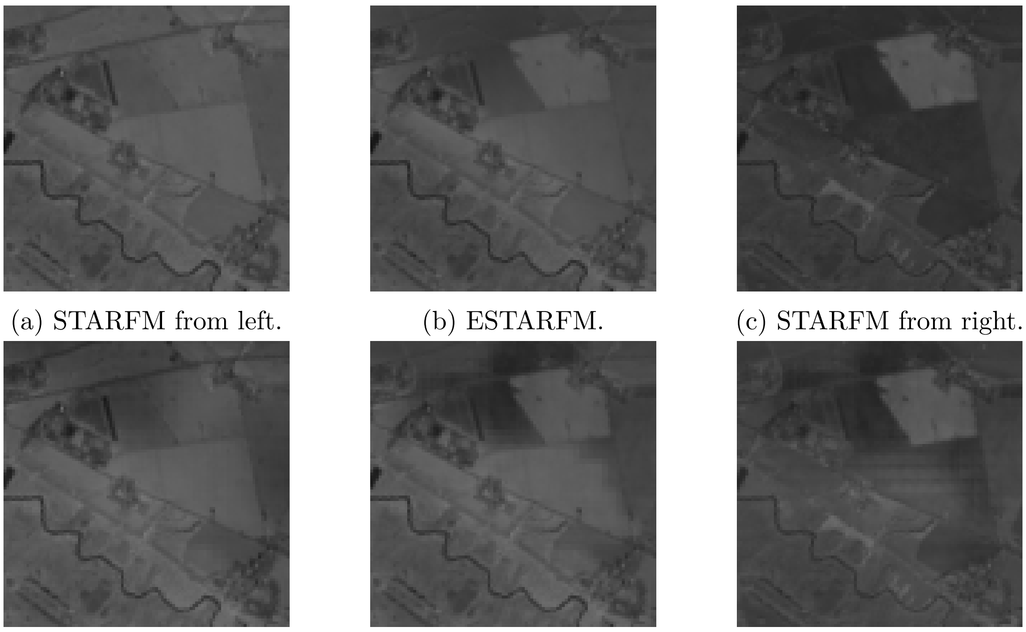
Method	STARFM	ESTARFM	STARFM	SPSTFM	SPSTFM	SPSTFM
Pair date	158	both	254	158	both	254
Image MAD	4355	2707	1447	4068	3014	1629

image.

Next the test with the real coarse resolution follows. The images MADs are shown in Table 5.6. The lower right 100 × 100 region of the corresponding fused products is shown in Figure 5.21. The results are much worse in comparison to the semi-real image set. Here, STARFM performs best, but with an error, which is as large as the maximum error with the artificial coarse resolution image, see Table 5.5. Indeed SPSTFM with the *div. sep. std. dev. / scale dict. normal* configuration from the end of the previous section performs a bit better than the selected option. It yields an error of 1444 *from right.* Generally none of the fused products seems to match the values on the changing areas. Maybe the quality of the coarse resolution image is to bad in this image set.

It is also worth to mention that the computational costs for a fusion with STARFM are very low. The fusion in this section could be finished in less than a second. ESTARFM takes considerably more time for a fusion, but still not as much as SPSTFM.

Finally, the multi-channel image set is used. However, the weighting with the build-up index does not work with the image as it is. This is because the bands red, SWIR and



(a) STARFM from left. (b) ESTARFM. (c) STARFM from right.

(d) SPSTFM from left. (e) SPSTFM with weights. (f) SPSTFM from right.

Figure 5.21: Products for the real single-channel image set with different fusion models, input images and settings.

NIR, that are involved in the build-up index (5.8), have different data ranges, which is not documented in the satellite image product specification. This can not easily be fixed by preprocessing (e. g. normalization), since this could also eliminate the desired properties. Actually the data could be correct and there might just be no structure at all in the dimension of the coarse resolution (250 m). The histograms of the important channels of the MODIS image with date 220 are shown in Figure 5.22. There one can easily see, that every NIR value is greater than any red value. Also the mean of SWIR is smaller than the mean of NIR. This results in $BU_c < 0$ for all pixels and all images. So there is no build-up change and thus all weights are 0.5. These weights are still used, but denoted by *equal weights* (eq. w.). The other weighting method chooses weights for each channel separately, just like for single-channel images (s.-c. w.). The image MADs for each channel are given in Table 5.7 For the errors with this test set holds that STARFM yields the worst results, followed by SPSTFM with single-channel weights. ESTARFM (imagefusion implementation) yields better results and SPSTFM with simple equal weights yields the best results. Also both SPSTFM methods gave their best result with an untrained dictionary, like with the single-channel real image set.

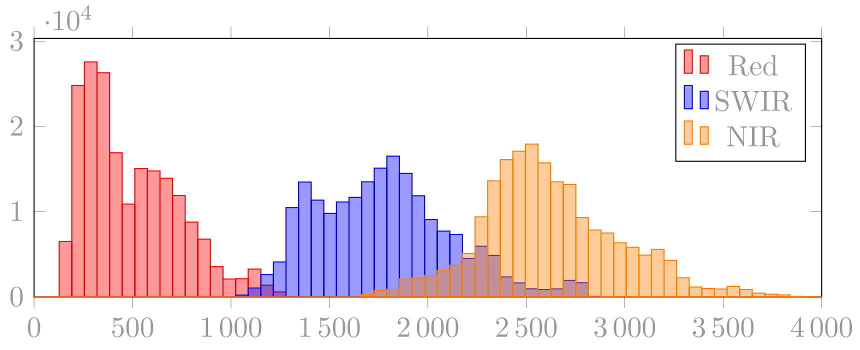


Figure 5.22: Histogram for red (red), SWIR (blue) and NIR (orange) channels of the multi-channel image set.

Table 5.7: Errors for different methods with the multi-channel real image set.

Method	STARFM	ESTARFM	STARFM	SPSTFM	SPSTFM
Date / Weighting	220	both	243	(s.-c. w.)	(eq. w.)
MAD channel 0	596	462	532	508	398
MAD channel 1	1019	907	1210	1024	744
MAD channel 2	1447	1215	1818	1406	955
MAD channel 3	509	374	555	426	277

6 Conclusions and further work

During this work a software framework for image fusion algorithms has been successfully developed. It consists of a library `libimagefusion`, currently three image fusion algorithms and utilities. The library is designed by a few easy-to-use core classes and interfaces that build the basis for image fusion algorithms. `libimagefusion` makes also use of very common third party libraries, like the multi-purpose *boost* library, the geo data library *GDAL* and the image processing library *OpenCV*. With their support the code base could be kept small but powerful. The core classes serve also as interface to these libraries and thus allowed to overcome some design flaws and unify the programming interface. This could help to make an `Image` object by far more const-correct than the underlying corresponding object of OpenCV. The same class makes use of GDAL for input and output and hides its complicated interface. The included algorithms – STARFM, ESTARFM and SPSTFM – use all the same interface due to object orientation, which improves consistency and allows to exchange them more easily. The framework also provides an option parser to support speed up the development of utilities. This has been utilized for some utilities and yields a powerful yet consistent command line interface. There are utilities to prepare images for imagefusion, to fuse images with the included algorithms and to compare a fused product with a reference image. The latter allows to measure the quality by several means, which can be helping to improve an algorithm. All the utilities have proven their usefulness in the development of SPSTFM.

SPSTFM is a rather complex image fusion algorithm, but could be successfully implemented. The detailed discussion about the mathematical background serves as documentation for the implementation. With lots of options that extend and improve the original algorithm the implementation goes beyond the sole realization of the algorithm as program. Several tests are invaluable to understand in which direction future improvement can go and what has been tried so far. These tests also showed what to choose as default options, gave justification for intuitively chosen options and also argued against to strict tolerances.

The test have been run mostly with the artificial image set from Figure 5.2. There SPSTFM behaved well and outperformed STARFM and ESTARFM clearly. However this image set mimics the one in [HS12] to get comparable results. But it seems this image set is prepared as show case for SPSTFM, which can handle it very well and STARFM and ESTARFM cannot handle the structural changes. For crop monitoring structural changes are less important. Often only the value of an area changes homogeneously without changing its size. For monitoring build-up areas, where structural changes are very common, image fusion is not really important. The changes happen so slowly that the frequency of fine resolution images should be enough for monitoring. In [HS12] one of the two real image examples uses images with a time gap of two years. The paper focuses on these changes and emphasizes this by choosing the build-up index for weighting. Certainly, this could be replaced by another index to calculate the weights. Maybe SPSTFM could be more valuable for applications where structural changes happen rapidly, i. e. inbetween 16 days. These kind of changes happen with catastrophes. So a water or burn index might be appropriate depending on the application. However, for catastrophes a real-time prediction might be valuable, too. With SPSTFM (and ESTARFM) image fusion is usually done for the past, since five images are required. Tests could show if the dictionary can be trained with past images and the weighting could also be used for extrapolation. Currently the sum of both weights may not exceed 1. This is left for further work. Nevertheless, the quality of the fused products of SPSTFM is not bad. For the multi-channel image set it performed even best, see Table 5.7. However, for the real images the training did not decrease the error significantly. This behaviour has to be researched to understand which properties of the images are responsible for that and if SPSTFM can be modified such that the dictionaries benefit from training for these kind of images.

Last but not least this is a software project and software can almost always be improved by resolving errors, inconsistencies or adding new features. This framework is no exception. There are many small points that would make the utilities or algorithms simply better. A useful example for development would be logging capabilities. The most images in image fusion are single-channel images or can be handled as such, but it would still be use full if the *image compare* utility could work with multi-channel images. Similarly, but with multiple images instead of channels: the *image geo crop* utility currently can only crop two images at once. When having more than two images two crop the utility must be called multiple times. It also cannot do a reprojection of geographical coordinate systems yet. Especially for MODIS and Landsat images this would be useful, since they use different coordinate systems. However, GDAL has a utility for that and the library supports this functionality. So it might be possible to use similar code as the GDAL utility for that. The GDAL utilities handle in general a lot of metadata, which would be very useful to support better by the imagefusion framework. Then the image fusion algorithm utilities could recognize more reliably no-data values and the valid data range, etc. Currently there is no utility for SPSTFM available, but this can quickly change. The most work would be to parse the options, which is already available. Nonetheless SPSTFM itself lacks a few important features. The most important are support for masks (in case of no data value) and padding. Mask support is in SPSTFM not as trivial as in STARFM or ESTARFM. If for example a patch is taken for training or reconstruction, which contains some pixels that do not represent data, these are often marked with data that is completely out of range (e. g. -28672). Using them could make the GPSR algorithm to find an actually bad optimum. So they have to be treated in some way. One way to deal with them could be to replace them by a mean value of the patch. But this is left for further work. Padding is the second important point, since currently it is not possible to fuse a whole image if the patches do not coincidentally fit exactly. Hence a prediction area with a small border is mandatory. This could be solved by padding when sampling out of the bounds of an image. So the image could be mirrored at its border. This would lift the restriction of the prediction area immediately. Furthermore small features like saving and loading dictionaries and weights could be added. There is still room for improvements in the computation time, by using an adaptive GPSR main loop tolerance, as discussed near Figure 5.7. What also could improve training time, would be to initialize the GPSR algorithm with the coefficients from the K-SVD algorithm. In the reconstruction stage for large images there occur often equal coarse resolution patches, which would actually have to be predicted only once, when they were cached. Then, quite far away when the framework is stable regarding its interface, algorithms and utilities, interfaces for other programming languages could be included. An interface to C would be good, since many libraries are written in C. Python is nowadays very popular for easy programming of computationally expensive operations. This can be done when a natively (C, C++) compiled library is accessible from Python. libimagefusion would be a good candidate for that. GDAL and OpenCV are also accessible from Python and C, so maybe it would be possible with a relative low amount of work. However, this is still complicated, since each language prefers its own style of interface and so these are not able to be translated one-to-one. The interfaces have to be designed in such a way that it feels natural to that language, which requires experience.

List of Figures