MinWarping is a holistic method for local visual homing. The term holistic embraces methods where images are compared at the level of pixels, in contrast to methods which extract and compare local features. Given two panoramic images captured at two different positions in the plane, MinWarping produces two estimates: the angle of a home vector pointing from one location to the other, and a compass angle expressing the azimuthal rotation between the two images. This document describes a fast implementation of MinWarping in C++ based on a template library for SIMD vector processing (the latter is documented separately). It also provides details on new extensions of MinWarping.


This document relates to release CODE12 of WarpingSIMDStandAlone.¹

¹Available from www.ti.uni-bielefeld.de/html/people/moeller/tsimd_warplingsimd.html
Contents

1. Introduction 1

2. MinWarping Algorithm 1

3. Software Structure 4
   3.1. C++ SIMD Vector Template Library: SIMDVec ................................. 4
   3.2. Image Classes and Functions: SIMDImage, SIMDStack ..................... 4

4. Data Classes 6
   4.1. Warping Parameter: WarpingParameter ........................................... 6
   4.2. MinWarping Template: MinWarpingTemplate .................................... 6
   4.3. Scale-Plane Stack: WarpingSPS ...................................................... 7
       4.3.1. Organization of the Scale-Plane Stack ................................... 7
       4.3.2. Complementing the Scale-Plane Stack: complement() ................... 8
       4.3.3. Inversion of the Scale-Plane Stack: invertFrom() .................... 9
       4.3.4. Computation of Compass Estimate: compassEstimate() ........... 12
   4.4. Match Array: MinWarpingMatch .................................................. 12
       4.4.1. Inversion of Match Array: invertFrom() .................................. 12
       4.4.2. Inversion of a Solution: invertSolution() ............................. 14
       4.4.3. Averaging Match Arrays: averageOf() ................................... 14
       4.4.4. Finding the Best Match: bestMatchFull() .................................. 14
   4.5. Search Range: MinWarpingSearchRange ....................................... 15

5. Algorithm Classes 15
   5.1. Scale-Plane-Stack Computation: WarpingSPSComputation ..................... 15
       5.1.1. Version 1 .................................................................................. 21
       5.1.2. Version 2 .................................................................................. 22
   5.2. Search: MinWarpingFull, MinWarpingPartial .................................. 23
       5.2.1. Full Search: minWarping.full() ............................................. 23
       5.2.2. Partial Search: minWarping.partial() ...................................... 24

6. Warping Compound: WarpingCompound 25
   6.1. Scale-Plane-Stack Computation: computeSPS() .................................. 25
   6.2. Full Search: minWarping.full() .................................................... 26
   6.3. Partial Search: minWarping.partial() ........................................... 26
   6.4. Compass Search: minWarping.compassAcceleration() ....................... 27
   6.5. Double Search in Full, Partial, and Compass Search ....................... 27
   6.6. Finding the Best Match: minWarping.bestMatchFull() .................... 30
   6.7. Fine Search: minWarping.fine() .................................................. 30
   6.8. Time Measurements ........................................................................ 31

7. Test Programs 32
   7.1. Small Test Program ......................................................................... 32
   7.2. Large Test Program ........................................................................ 32

A. Floor and Modulo Rules 35

B. Derivation of Equation (8) 37

C. Angular Relationships in Test Program 38

References 40
1. Introduction

MinWarping is an insect-inspired method for local visual homing in the plane. In local visual homing, a target location is characterized by a panoramic snapshot image (“snapshot” in the following). The current panoramic image (“current view”) is interrelated to the snapshot to estimate two movement parameters: the angle of the “home vector” pointing from the current to the snapshot location, and the azimuthal rotation between the two images called “compass angle”. Following the repeatedly updated home vector will lead an agent to the target location. MinWarping is a “holistic” method where images are compared at the level of pixels, in contrast to methods based on the extraction and comparison of local features.

A detailed description of MinWarping is given by Möller et al. (2010), based on earlier methods developed by Möller (2009) and Franz et al. (1998). MinWarping is an algorithmic method, but a biologically plausible version of it has been developed (Möller, 2012). Different distance measures for MinWarping have been studied by Möller et al. (2014) with respect to illumination tolerance. MinWarping has been successfully applied for the visual navigation of cleaning robots (Gerstmayr-Hillen et al., 2013; Möller et al., 2013). Note that this document describes an implementation and therefore only provides a compact description of the method itself; only new extensions (double search) are described in detail. The reader is kindly referred to the papers cited above.

MinWarping is currently subject to the following constraints: First, it is only applicable to movements in the plane as only the two above-mentioned movement parameters are estimated (this constraint may be lifted in the future). Second, MinWarping assumes that all pixel visible in an image column have the same ground distance from the vantage point. Despite this “equal-distance assumption”, MinWarping produces home vectors of good precision in many indoor environments.

2. MinWarping Algorithm

MinWarping is based on a systematic search over the two movement parameters: the movement direction angle $\alpha$ and the compass angle $\psi$; see figure 1. The movement parameters $\alpha$ and $\psi$ describe the hypothetical movement tested by MinWarping from snapshot location $S$ to current location $C$ (with different azimuthal orientation in each location). If a landmark (in MinWarping an image column) is visible under an angle $x$ from the movement direction in the snapshot image, MinWarping searches for the same landmark (under different vertical magnifications, see below) in the current image at all possible angles $x + y$ (always only on the same side of the movement vector). The match quality for the best-matching landmark in the current view is summed over all angles $x$ in the snapshot. The parameter combination $(\alpha, \psi)$ with the minimal sum is the solution. The home-vector angle $\beta$ (from $C$ to $S$) is then determined from $\beta = \alpha - \psi + \pi$. Note that
neither the distances to the landmark $r$ and $r'$ nor the length of the movement $d$ are known.

MinWarping determines the match qualities for each parameter combination $(\alpha, \psi)$ from the following equation (with C-style array notation, parameter indices using letter $i$, pixel indices using letter $j$):

$$d[i_\psi][i_\alpha] = \sum_{j_x=0}^{w-1} \left\{ \min_{j_y \in T_y(j_x)} (D[T_p(j_x, j_y) \circ j_\theta(j_x, j_y(i_\psi))]) \right\}$$

In this equation, $i_\alpha, i_\psi$ are parameter indices corresponding to evenly spaced values of $\alpha, \psi$. The number of steps in $i_\alpha, i_\psi$ are $n_\alpha, n_\psi$ (parameter nAlpha, nPsi). If $w$ (parameter width, member $w$) is the width of a panoramic image (azimuthal dimension), $\Delta_\alpha, \Delta_\psi$ are the pixel step widths, with $w = \Delta_\alpha n_\alpha = \Delta_\psi n_\psi$. MinWarping computes a match quality (distance) $d[i_\psi][i_\alpha]$ for each combination $i_\alpha, i_\psi$.

Each match quality $d[i_\psi][i_\alpha]$ is computed by a summation over all pixel indices $j_x$ (corresponding to discrete values of the angle $x$). For each $j_x$, a minimum is determined over all possible values $j_y$ (corresponding to discrete values of the angle $y$) where the landmark seen at $x$ in the snapshot can appear in the current view. Which values of $j_y$ are consulted is encoded in the template, here expressed as set $T_y(j_x)$.

The minimum search consults a 3-dimensional array of column distances $D$, the so-called “scale-plane stack”. The scale-plane stack is formed by scale planes of the size $w \times w$. 

Figure 1: Parametrized movement and landmark search in the ground plane underlying MinWarping. S: snapshot location, C: current location, thick arrows: orientation, L: possible landmark locations.
which hold column distances between all column pairs of two panoramic images. An input image is either the preprocessed snapshot or current view or versions thereof magnified in vertical direction (around the horizon) according to a scale factor. The template is consulted to select a scale plane from the indices \( j_x, j_y \), here expressed as function \( T_p(j_x, j_y) \).

The valid entries (according to \( T_y(j_x) \)) of the template function \( T_p(j_x, j_y) \) are computed as follows: For each angle pair \( x, y \), the law of sines applied to the triangle SCL leads to a scale factor \( \sigma \)

\[
\sigma := \frac{r'}{r} = \frac{\sin x}{\sin(x + y)}
\]

which determines the distance ratio \( r'/r \). The distance ratio is in turn related to pixel elevations (with respect to the horizon), denoted by \( \gamma \) the in snapshot and \( \gamma' \) in the current view:

\[
\sigma := \frac{r'}{r} = \frac{\tan \gamma}{\tan \gamma'}
\]

For given angles \( x, y \), we can therefore determine how either the preprocessed current view or the snapshot have to be magnified in vertical direction around the horizon such that the column distance can be computed over the entire image height for pixel pairs with the same vertical index. For \( \sigma < 1 \), the snapshot is vertically magnified around the horizon (as \( r' < r \), the column appears larger in the current view) while the current view remains unchanged; for \( \sigma > 1 \), the current view is magnified (as \( r' > r \), the column appears larger in the snapshot) while the snapshot remains unchanged. The scale-plane stack is computed from a discrete set of scale factors. Scale factors computed from equation (2) in the template are mapped to the nearest scale factor from the discrete set; this mapping is stored in the template as \( T_p(j_x, j_y) \).

The discrete scale factors are chosen such that the roles of snapshot and current view can be exchanged in the so-called “double search” mode. Therefore, if a scale factor \( \sigma \) is contained, also the scale factor \( 1/\sigma \) is contained in the set. The scale factor 1 is always contained (and only once). The ratio of neighboring scale factors is constant. The discrete set of scale factors is computed from the number of scale planes \( n\text{ScalePlanes} \), the maximal scale factor \( \text{maxScaleFactor} \), and a threshold \( \text{maxThreshold} \) for scale factors which excludes all entries with scale factors above that threshold from the template.

Each scale plane stores column distances between two images. The horizontal coordinate

---

\(^2\)The present implementation expects panoramic input images with linear relationship between horizontal pixel coordinate and azimuth angle and between vertical pixel coordinate and elevation angle. The vertical resolution is variable (parameter \text{verticalResolution}; see section 5.1).
$j_\theta$ in each scale plane (second index of $D$) corresponds to the azimuth angle $\Theta = x + \alpha$ in the snapshot. Instead of using $j_\theta'$ (corresponding to the azimuth angle $\Theta' = \alpha - \psi + x + y$ in the current view), the vertical coordinate in each scale plane (third index of $D$) is $j_\delta$ corresponding to the angle $\delta = \Theta' - \Theta = y - \psi$. This storage format has two advantages: The visual compass can be computed faster by a summation over rows of the minimum of scale planes, and the inversion of the scale-plane stack for “double search” is accomplished by a fast data exchange between rows instead of a costly transposition. The second index $j_\delta$ of $D$ is computed from $j_\delta = (j_y - j_\psi)$ mod $w$, the third index $j_\Theta$ from $j_\Theta = (j_x + j_\alpha)$ mod $w$, with $j_\alpha(i_\alpha) = i_\alpha \Delta_\alpha$ and $j_\psi(i_\psi) = i_\psi \Delta_\psi$.

### 3. Software Structure

#### 3.1. C++ SIMD Vector Template Library: SIMDVec

The implementation of MinWarping is at the lowest level based on the “T-SIMD” C++ vector template library (Möller, 2016) which provides template class wrappers for built-in vector data types

```cpp
template <typename T, int SIMD_WIDTH> class SIMDVec;
```

and template function wrappers for vector intrinsics. Template parameters are the element type of the vectors ($T$) and the number of bytes in the vector data type (SIMD_WIDTH). Vector data types in turn are mapped by the compiler to vector registers, and vector intrinsics to vector instructions. Vector registers hold multiple elements of the same type, and vector instructions typically apply the same operation in parallel to all elements of its vector operands (“SIMD”: single instruction, multiple data). T-SIMD offers a simple way to change data types and vector extensions for entire portions of the code with minimal effort, and frees the programmer from many details of vector programming. MinWarping is very well suited for a SIMD vector implementation since it is a holistic method directly operating on arrays (images, scale-plane stack, match array) where the same operations are applied to groups of pixels. Note that the MinWarping implementation assumes that all arrays (scale-plane stack, match array) have a size which is a multiple of the SIMD vectors used in the processing (to avoid special treatment of the elements at the end of each row).

#### 3.2. Image Classes and Functions: SIMDImage, SIMDStack

At the second-lowest level, MinWarping uses a simple image processing library. It defines a template class SIMDImage and a template class SIMDStack which (conceptually) holds a stack of images with identical type and size:
template <typename T, int SIMD_WIDTH, int SIMD_ALIGN> class SIMDImage;
template <typename T, int SIMD_WIDTH, int SIMD_ALIGN> class SIMDStack;

Both template classes are derived from SharedSIMDPtr, a template class implementing a smart shared pointer to memory areas with a desired alignment (SIMD_ALIGN) and with the guarantee that the length of the area is a multiple of SIMD_WIDTH (to avoid truncation in SIMD processing):

template <typename T, int SIMD_WIDTH, int SIMD_ALIGN>
class SharedSIMDPtr;

Assignment, copy construction, and calls to view() generate a view of the SharedSIMDPtr argument instead of copying data; note that if null pointers (obtained from the default constructor) are processed in this way, the data is not shared if memory is assigned later (e.g. by calling resize()).

Both SIMDImage and SIMDStack have only few methods for construction, destruction, assignment, resizing, copying from / to external memory locations, filling, loading, saving, and row / image access, respectively (many of those inherited from SharedSIMDPtr). In both SIMDImage and SIMDStack, the data is allocated in a single block (via SharedSIMDPtr) for efficient SIMD access. SIMDStack data can be split into separate images by generating views using the methods of SharedSIMDPtr. Note that a SIMDImage which provides a view of a SIMDStack doesn’t hold data itself and therefore cannot be resized: A call to SIMDImage::resize() throws an exception if the new size doesn’t coincide with the current one.

The template parameter SIMD_ALIGN specifies the alignment of the data blocks in bytes. The data block always starts on an address that fulfills this alignment, i.e. it starts at an address divisible by SIMD_ALIGN. The row length of SIMDImage has to be chosen such that also each row starts on an aligned address. This allows aligned access by load and store functions operating on SIMDVector which is faster in some vector extensions and on some CPU architectures. This constraint restricts the width of images to a multiple of the number of elements of a SIMDVector with the same element type and SIMD_WIDTH. Therefore, all elements of a row can be processed in parallel by vector instructions; without this restriction, some elements at the end of a row would have to be processed in a special way which would be less efficient.

Image processing functions are defined as non-member template functions operating on SIMDImage and SIMDStack arguments. All functions test whether input and output images overlap — some functions reject overlapping images completely, other functions only allow full overlap but no partial overlap (these are functions directly mapping input to output pixels). The following functions are used in the first phase of MinWarping (computation of the scale-plane stack): verticalEdgeFilter(), magnifyAndScale(),
4. Data Classes

The classes and template classes described in this section primarily store data and provide only few public processing methods, typically for data re-organization. Since data classes and algorithm classes (section 5) are closely interrelated in the computations, data classes make many members available in the public section. Even if these are non-const, they are only supposed to be read, not written. The alternative (friend methods or access functions) would have been cumbersome.

4.1. Warping Parameter: WarpingParameter

The template class

```cpp
template <typename SPSType, int SIMD_WIDTH> class WarpingParameter;
```

holds information required in several parts of the code: Basic parameters such as image width \( w \), search steps \( n\alpha, n\Psi \) etc.; look-up tables for conversion from parameter indices to pixel indices (e.g. \( j\alpha\text{Vec} \)) and angles (e.g. \( \alpha\text{Vec} \)); and look-up tables for shuffling and modulo operations (e.g. \( \text{shuffleModulo} \)). The template argument \( \text{SPSType} \) is the type of the scale-plane stack entries (required for the computation of some parameters and look-up tables).

4.2. MinWarping Template: MinWarpingTemplate

The class \( \text{MinWarpingTemplate} \) stores the MinWarping template (formed by the set \( T_y \) and the function \( T_p \)), the vector of thresholds, and some parameters related to the template (minimal / maximal \( \rho \): \( \rho_{\text{min}}, \rho_{\text{max}} \)). The template itself is implemented as a contiguous array of structs containing arrays with information for each column \( (j_x) \) of the template. Each column array has the same length as elements in the set \( T_p(j_x) \). Each entry of a column array contains two items: The value for \( j_y \) (in the code inconsistently

\[ \text{This is an experimental feature: For each angle } x \text{ and } y, \text{ the ratio } \rho = \frac{d}{r} = \sin y / \sin(x + y) \text{ can be computed (which is non-negative for all valid entries of the template). Using the two thresholds, the template can be restricted to a range of } \rho \text{ values. Use } \rho_{\text{min}} = 0 \text{ and a large number e.g. } \rho_{\text{max}} = 100 \text{ to deactivate this feature.} \]
called \( i_y \), and the scale plane \( T_p(j_x, j_y) \) that has to be consulted for the pair \((j_x, j_y)\), encoded as offset to the start of the scale-plane stack. The values \( T_p(j_x, j_y) \) are computed from equation (2) and mapped to the nearest scale factor of the set used in the computation of the different scale planes by consulting the threshold vector.

### 4.3. Scale-Plane Stack: WarpingSPS

The template class

```
template <typename SPSType, int SIMD_WIDTH, int SIMD_ALIGN>
class WarpingSPS;
```

contains a scale-plane stack (in the form of a SIMDStack member). The type of its elements is given by the template parameter SPSType. The constructor

```
WarpingSPS(const WarpingParameter<SPSType,SIMD_WIDTH> &param,
           int numScaleFactors);
```

receives a reference to WarpingParameter which is stored internally as reference\(^4\) and the number of scale factors (i.e. the number of scale planes).

### 4.3.1. Organization of the Scale-Plane Stack

In the description of the MinWarping algorithm (section 2) it was assumed that each scale plane has the horizontal index \( j_\Theta \) and the vertical index \( j_\delta \). In the implementation, a “shuffled” version of this format is used. This is due to the way the search phase is parallelized using vector instructions: Multiple entries of the match array \( d \) for the same value of \( i_\Theta \) but neighboring values of \( i_\alpha \) are computed in parallel. Each scale plane is therefore shuffled within each row such that the pixel indices \( j_\alpha(i_\alpha) = i_\alpha \Delta_\alpha \) processed in parallel are contiguous in the shuffled row. The shuffled scale-plane stack \( D' \) is related to the unshuffled version \( D \) by

\[
D'[p][j_\delta][j'_\Theta] = D[p][j_\delta][j_\Theta]
\]

where the plane index \( p \) runs through all scale planes.

To describe the shuffling \((j_\Theta \rightarrow j'_\Theta)\)^\(5\), we use the abbreviations \( j := j_\Theta = (j_x + j_\alpha) \mod w \) (unshuffled coordinate), \( j' := j'_\Theta \) (shuffled coordinate), \( \Delta := \Delta_\alpha \), and \( n := n_\alpha \) (with

\(^4\)Note that this reference member makes the template class WarpingSPS non-copyable.

\(^5\)Please note the difference between \( j_\Theta \), which denotes the index corresponding to the current view azimuth angle \( \Theta' \), and \( j'_\Theta \), which denotes the shuffled index corresponding to \( j_\Theta \) (the azimuth index in the snapshot).
The shuffle mapping is defined as
\[
j' = \text{shuffle}(j) = (j \mod \Delta) b + \left\lfloor \frac{j}{\Delta} \right\rfloor \quad (5)\]
with \( b = n + s \), where \( s \) is the number of scale-plane elements in a SIMD vector. Shuffling rearranges a scale plane into \( \Delta \) blocks of block size \( b \) (see figure 2). The additional \( s \) columns at the end of each block are required for wrap-around operations on the horizontally cyclic scale plane. They are filled with the content of the first \( s \) columns of the block:
\[
D'[p][j][k \cdot b + n + e] := D'[p][j][k \cdot b + e] \quad \forall k \in [0, \Delta), \forall e \in [0, s) \quad (6)
\]
where \( k \) is the index of the block, and \( e \) is the index within the additional columns (corresponding to the element index in a SIMD vector).

The inverse mapping (unshuffle) is given by
\[
j = \text{unshuffle}(j') = \left( (j' \mod b) \Delta + \left\lfloor \frac{j'}{b} \right\rfloor \right) \mod w \quad (7)
\]
where the last modulo operation (by image width \( w \)) is required to transform indices in the \( \Delta \) gaps of width \( s \) in the shuffled scale-plane stack to valid indices in the unshuffled scale-plane stack.

Figure 2 visualizes the different formats of the scale-plane stack.

For the inversion of the scale-plane stack (section 4.3.3) and for the second way to compute the scale-plane stack (section 5.1.2), the following unshuffle-shuffle operation with an offset \( d \) is required:
\[
\text{shuffle } ([\text{unshuffle}(j') + d] \mod w) = ([i_{\text{blk}} + d] \mod \Delta) b + \left( i_{\text{off}} + \left\lfloor \frac{i_{\text{blk}} + d}{\Delta} \right\rfloor \right) \mod n \quad (8)
\]
where
\[
i_{\text{blk}} = \left\lfloor \frac{j'}{b} \right\rfloor \in [0, \Delta), \quad i_{\text{off}} = j' \mod b \in [0, b) \quad (9)
\]
are the index of the block in the scale-plane stack and the offset within the block, respectively. A proof for equation (8) is given in appendix B based on some modulo and floor rules listed in appendix A.

4.3.2. Complementing the Scale-Plane Stack: \texttt{complement()}  

Each of the \( \Delta \) blocks in the shuffled scale-plane stack contains \( s \) columns at the end which coincide with the first \( s \) columns of the block; see equation (6). The method \texttt{complement()} copies these columns from the start to the end of each block in all scale planes. It is called whenever a scale-plane stack is computed or inverted.
4.3.3. Inversion of the Scale-Plane Stack: invertFrom()

In the new “double search” version described in section 6.5, a second run of MinWarping is performed where the roles of snapshot and current view are exchanged. However, it is not necessary to compute the scale-plane stack a second time. Instead, the scale-plane stack from the first run can be inverted. This inversion has to aspects: First, the order of the scale planes is inverted, and second, each scale plane has to be rearranged.

Figure 3 visualizes that inversion of the scale-plane stack affects the order of the scale planes. Figure 4 is used to derive an equation for the rearrangement of each scale plane in the inversion. An entry in the original scale-plane stack has the coordinates \((\Theta_{ss}, \delta_{cv})\). This entry is transferred to \((\Theta_{cv}, \delta_{ss})\) in the inverted scale-plane stack. The relation between these variables can be derived from the landmark shifts \(\delta_{cv} = \Theta_{cv} - \Theta_{ss}\) and \(\delta_{ss} = \Theta_{ss} - \Theta_{cv}\), which leads to \(\delta_{ss} = -\delta_{cv}\) and \(\Theta_{ss} = \Theta_{cv} - \delta_{cv}\).

From \(\delta_{ss} = -\delta_{cv}\) we conclude that data are transported from row \(j_{\delta}\) in the original scale plane to row \((w - j_{\delta}) \mod w\) in the rearranged scale plane (where \(w\) was added since we only work with non-negative indices because the C modulo operator does not fulfill equation (25) for negative arguments). Rearrangement within a row is described by \(\Theta_{ss} = \)
Figure 3: Inversion of the scale-plane stack inverts the order of the scale planes. Left: pairing of input images in the first run (snapshot SS against current view CV), right: second run (CV against SS). On the left side, the inversion-symmetric scale-factor set is visualized (with \( \sigma \) increasing in the direction of the arrow). Vertical magnification is indicated by arrows, with the number of arrows corresponding to the strength of the magnification. The crossing lines in the center depict the inverted order of scale planes.

Figure 4: Inversion of the scale-plane stack requires a rearrangement of each scale plane. Here the top SS-CV and bottom CV-SS pairing of figure 3 is used as an example.

\[ \Theta_{cv} - \delta_{cv} \] (which computes the source location from the destination location), but it has to be taken into account that the scale plane is stored in shuffled order. If \( j'_{src} \) is the horizontal index corresponding to \( \Theta_{ss} \) in the original shuffled scale plane, and \( j'_{dst} \) the horizontal index corresponding to \( \Theta_{cv} \) in the inverted shuffled scale plane, we can transform \( \Theta_{ss} = \Theta_{cv} - \delta_{cv} \) into

\[ j'_{src} = \text{shuffle} \left( \left[ \text{unshuffle} \left( j'_{dst} \right) + w - j_{\delta} \right] \mod w \right). \]  

(10)

This equation corresponds to equation (8), with \( d = w - j_{\delta} \).

For efficiency reasons, we want to transfer an entire SIMD vector of \( s \) elements from \( j'_{src} \ldots j'_{src} + s - 1 \) to \( j'_{dst} \ldots j'_{dst} + s - 1 \). The index \( j'_{dst} \) runs in steps of \( s \) through each block, the index \( j'_{src} \) is computed from equation (10), and \( s \) elements are transferred in one vector load-store operation. For that we have to guarantee that \( s \) contiguous destination indices correspond to \( s \) contiguous source indices. If the indices \( j'_{dst} \) (\( j' \) in equation (8))
of all \( s \) elements lie in the same block of width \( b \), the variable \( i_{blk} \) in equation (9) is constant and therefore the first term of equation (8) is constant as well. This means that all source indices lie in the same block (but not necessarily in the same block as the destination indices). Also the indices \( i_{off} \) in equation (9) are contiguous in this case. The second expression (floor operation) in the second term in equation (8) is constant. Without the modulo operation \((\text{mod}\ n)\), the second term would produce contiguous indices for contiguous \( i_{off} \), but the modulo operation causes the source indices to wrap around to the start of the block after the end of the first \( n \) elements of the block. So, while \( j'_{src} \) always lies outside the gap, other indices in the range \( j'_{src} \ldots j'_{src} + s - 1 \) may extend into the gap. However, since the first elements \( 0 \ldots s - 1 \) of the block are copied to the gap elements \( n \ldots n + s - 1 \) (equation (6)), data can be copied from contiguous indices even in the case of wrap-around.

The implementation of the inversion of the scale-plane stack is shown below (note that pixel indices are inconsistently called \( i* \) in the code):

```c
void
invertFrom(const WarpingSPS<SPSType,SIMD_WIDTH,SIMD_ALIGN> &spsFrom)
{
    ...
    int S = stack.numPlanes;
    SIMDVec<SPSType,SIMD_WIDTH> d;
    for (int s = 0; s < S; s++) {
        SPSType *planeTo = stack[s].data;
        SPSType *planeFrom = spsFrom.stack[(S-1)-s].data;
        for (int iDeltaCV = 0; iDeltaCV < w; iDeltaCV++) {
            SPSType *planeToRow = planeTo + modulo2wpw[-iDeltaCV];
            SPSType *planeFromRow = planeFrom + modulo2wpw[iDeltaCV];
            for (int blkStart = 0; blkStart < wSPS; blkStart += blkSize) {
                for (int iDstS = blkStart; iDstS < blkStart + nAlpha;
                     iDstS += simd_sps_elems) {
                    int iSrcS = shuffleModuloPW[unshuffle[iDstS] - iDeltaCV];
                    d = loadu<SIMD_WIDTH>(planeFromRow + iSrcS);
                    store(planeToRow + iDstS, d);
                }
            }
        }
    }
    complement();
}
```

Note that the additional \( s \) columns in each block are complemented after the inversion.

At least for full search (section 5.2.1), the time required for the inversion of the scale-plane stack is negligible compared to the time required for the search.
4.3.4. Computation of Compass Estimate: \texttt{compassEstimate()}

The function

\begin{verbatim}
template<typename CompassType>
void
compassEstimate
(SIMDImage<CompassType,SIMD_WIDTH,SIMD_ALIGN>& compass) const;
\end{verbatim}

computes a compass estimate for $n_\psi$ values as described by Möller et al. (2010), which can either be used directly to compute an estimate of $\psi$ without the more costly search phase (see table 4 in Möller et al., 2014), or is used in the search with compass acceleration (section 6.4). The function first computes the minimum through all scale planes and then adds all entries along the horizontal dimension of the minimum scale plane. If the compass estimate is used directly it has to be taken into account that the index of the resulting compass image corresponds to $-\psi$.

4.4. Match Array: \texttt{MinWarpingMatch}

The template class

\begin{verbatim}
template<typename MatchType, int SIMD_WIDTH, int SIMD_ALIGN>
class MinWarpingMatch;
\end{verbatim}

contains the match qualities $d[i_\psi][i_\alpha]$ computed in equation (as a \texttt{SIMDImage} of type \texttt{MatchType} with $i_\alpha$ varying fastest). After a partial search (see section 5.2.2), part of the match array will contain invalid entries. The value used to mark invalid entries is passed to the constructor.

4.4.1. Inversion of Match Array: \texttt{invertFrom()}

Double search (section 6.5) first produces a match array from the processing of snapshot vs. current view, then inverts the scale-plane stack (see section 4.3.3), and uses the inverted scale-plane stack to produce a match array for current view vs. snapshot. Both match arrays are averaged to produce a joint match array. However, the indices of the second match array (corresponding to angles $\alpha_2, \psi_2$) differ from those of the first match array (corresponding to angles $\alpha_1, \psi_1$). Therefore the match array of the second search has to be rearranged (inverted) before averaging.
Figure 5: Relation between angles $\alpha_1, \psi_1$ from the first search and angles $\alpha_2, \psi_2$ from the second search in the double search mechanism.

Figure 5 explains how the angles are interrelated:

\[
\begin{align*}
\psi_2 & = 2\pi - \psi_1 \\
\alpha_2 & = \pi + \alpha_1 - \psi_1
\end{align*}
\]  

(11) \hspace{1cm} (12)

After conversion to indices using $\psi_i = 2\pi i_\psi / n_\psi$ and $\alpha_i = 2\pi i_\alpha / n_\alpha$ with $i = 1, 2$ we obtain:

\[
\begin{align*}
i_\psi_2 & = (n_\psi - i_\psi_1) \mod n_\psi \\
i_\alpha_2 & = \left( n_\alpha + \frac{n_\alpha}{2} + i_\alpha_1 - \frac{n_\alpha i_\psi_1}{n_\psi} \right) \mod n_\alpha.
\end{align*}
\]

(13) \hspace{1cm} (14)

Note that $n_\psi$ and $n_\alpha$ were added to avoid negative arguments to the modulo operator. The third term in equation (14) is a bit problematic if $n_\alpha \neq n_\psi$: The current implementation uses an integer division, but rounding may be more appropriate.

The function `invertFrom()` is implemented as follows (note that the code uses indices 0, 1 instead of 1, 2 as in the derivation above):

```c
1 void invertPsiRangeFrom(const MinWarpingMatch &otherMatch,
2 const int iPsi1Start, const int iPsi1End)
3 {
4    ...
5    int nAlpha2 = nAlpha / 2;
6    for (int iPsi0 = iPsi0Start; iPsi0 <= iPsi0End; iPsi0++) {
7        int iPsi1 = (nPsi - iPsi0) % nPsi;
8        int iPsi1_nAlpha = iPsi1 * nAlpha;
```
4.4.2. Inversion of a Solution: invertSolution()

Equations (13, 14) are also used in the method

```c
void invertSolution(int iAlpha0, int iPsi0, int &iAlpha1, int &iPsi1);
```

This method is used for double fine search (see section 6.7).

4.4.3. Averaging Match Arrays: averageOf()

The method

```c
void averageOf(const MinWarpingMatch &match1, const MinWarpingMatch &match2);
```

computes the average of two match arrays, typically of an original one from the first run and of one inverted (section 4.4.1) from the match array resulting from the second run of double search (section 6.5). If at least one of the match arrays contains an invalid entry, the resulting entry is invalid as well.

4.4.4. Finding the Best Match: bestMatchFull()

The method

```c
void bestMatchFull(int &iAlphaMin, int &iPsiMin, MatchType &dMin);
```
searches through the match array $d[i_\psi][i_\alpha]$ for the minimal entry and returns the corresponding value and the arguments $i_\alpha, i_\psi$.

The function

```cpp
void extremaExceptInvalid(MatchType &min, MatchType &max);
```

returns the minimal and maximal entry of the match array but ignores the invalid values. It is typically used for visualization only.

### 4.5. Search Range: MinWarpingSearchRange

The template class

```cpp
template <typename SPSType, int SIMD_WIDTH, int SIMD_ALIGN>
class MinWarpingSearchRange;
```

is used to compute and hold index ranges for $i_\alpha$ and $i_\psi$ for partial and compass searches. For $i_\alpha$, indices consider the vector-wise processing in this dimension. Member variables `alphaIndices` and `psiIndices` store vectors of indices. The methods `setPartialRange()` and `setFullCompassRange()` are used by `WarpingCompound` (section 6) to update these vectors.

### 5. Algorithm Classes

The classes and template classes described in this section implement algorithms. They typically hold no data but operate on the data classes described in section 4. Algorithms are mostly implemented as virtual functions (defining an abstract interface). The Min-Warping algorithm is split into two phases: the computation of the scale-plane stack (section 5.1) and the search (section 5.2).

#### 5.1. Scale-Plane-Stack Computation: WarpingSPSComputation

The template class

\footnote{The term “range” is somewhat misleading since actually arbitrary sets of indices can be stored.}
template<typename ImgType,
    typename ProcType, typename MeasType, typename SPSType,
    int SIMD_WIDTH, int SIMD_ALIGN>
class WarpingSPSComputation;

is an abstract interface for the computation of the scale-plane stack.\textsuperscript{7} The abstract interface is designed for the computation of normalized measures (such as measures based on correlation). \textit{ImgType} is the type of the input images, \textit{ProcType} is the type used for image processing, \textit{MeasType} is the type used to compute and store components of distance measures, and \textit{SPSType} is the storage type for the scale-plane stack.

\texttt{WarpingSPSComputation} provides the following pure virtual or empty functions (typically used by \texttt{computeSPS()}) which need to be defined or redefined in derived classes:

\begin{verbatim}
virtual void convertImage(const SIMDImage<ImgType,SIMD_WIDTH,SIMD_ALIGN> &in,
    SIMDImage<ProcType,SIMD_WIDTH,SIMD_ALIGN> &out) = 0;
\end{verbatim}

converts an image from \textit{ImgType} to \textit{ProcType}, the latter being used for the preprocessing. In the simplest case, this function just converts the image pixel by pixel, but for images with invalid pixels (e.g. resulting from robot structures visible in the image or undefined regions from image tilt) it has to be guaranteed that invalid pixels in the image \texttt{in} are also marked as invalid in \texttt{out}, where the values used to mark invalid pixels can differ between the two pixel types.

\begin{verbatim}
virtual void preprocessing(const SIMDImage<ProcType,SIMD_WIDTH,SIMD_ALIGN> &input,
    SIMDImage<ProcType,SIMD_WIDTH,SIMD_ALIGN> &output,
    const WarpingParameter<SPSType,SIMD_WIDTH> &par) = 0;
\end{verbatim}

performs preprocessing from image \textit{input} to image \textit{output}, e.g. edge-filtering.

\begin{verbatim}
virtual double correctHorizon(double horizon) = 0;
\end{verbatim}

is used to correct the horizon of the image if it is affected by the preprocessing (e.g. if edge filtering removes one row).

\begin{verbatim}
virtual void copyImage(const SIMDImage<ProcType,SIMD_WIDTH,SIMD_ALIGN> &img,
    double multiplyScale,
    SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &outImg) = 0;
\end{verbatim}

\textsuperscript{7}Note that the name refers to Warping, not MinWarping, since the scale-plane stack would be computed in the same way also for 2D-Warping.
converts an image from ProcType to MeasType by scaling each pixel by multiplyScale (the type MeasType being used to compute the components of distance measures). It is typically used together with magnifyImage() to generate the image pairs for the computation of each scale plane.

virtual void
magnifyImage(const SIMDImage<ProcType,SIMD_WIDTH,SIMD_ALIGN> &img,
    double verticalResolution, double horiz,
    int interpolation,
    double magnifyScale,
    double multiplyScale,
    SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &outImg) = 0;

is used together with copyImage() to generate image pairs for the scale-plane stack. It receives a parameter verticalResolution (radians per pixel), the location of the horizon, an integer specifying interpolation (e.g. 0 for nearest-neighbor and 1 for linear interpolation), a magnification factor magnifyScale (corresponding to \( \sigma \)), and a factor multiplyScale (with the same meaning as in copyImage()).

virtual void
reArrange(SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &img,
    const WarpingParameter<SPSType,SIMD_WIDTH> &par);

is used to re-arrange the input image img. It is used in the second version of the scale-plane-stack computation (section 5.1.2). It is an empty function by default.

virtual void
columnMeasure(const SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &input,
    SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &output,
    const WarpingParameter<SPSType,SIMD_WIDTH> &par) = 0;

is used to compute some measure over the columns of the input image and stores the result in the single-row output image. Note that this computation works on the type MeasType.

virtual void
jointMeasure
    (const SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &input1,
    const SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &input2,
    const SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &colMeasRaw1,
    const SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &colMeasRaw2,
    SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &output,
    SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &colMeas1,
    SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &colMeas2,
    SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &colMeas12,
    const WarpingParameter<SPSType,SIMD_WIDTH> &par) = 0;
is used to compute a joint measure over the two images \texttt{input1} and \texttt{input2}, e.g. the sum of absolute differences over column pairs. Also this function operates on images with element type \texttt{MeasType}. For images with invalid pixels it is necessary to also pass the two column measures through this function, here from the inputs \texttt{colMeasRaw1} and \texttt{colMeasRaw2} to the outputs \texttt{colMeas1}, \texttt{colMeas2}, and \texttt{colMeas12} (a combination of both).\footnote{Thanks to Annika Hoffmann for pointing this out.} If no invalid pixels can be present in the images, the column-measure output can just be a view or a copy of the column measure input and the combined column-measure output is not required. However, if invalid pixels are present, the column measure of one image is influenced by the invalid pixels in the other image. If invalid pixels are only located as contiguous regions at the bottom of image columns, the column-measure input could contain pre-computed accumulated values of the column measure, and the function would compute \texttt{colMeasRaw12}. If invalid pixels can occur anywhere, the \texttt{colMeasRaw1} and \texttt{colMeasRaw2} images could be empty (and the corresponding output images unused) since the computation of the \texttt{colMeas12} has to be done completely inside \texttt{jointMeasure()}.\footnote{Note that this mechanism may not be general enough for arbitrary normalized distance measures: \texttt{idealPixelScale()} (see below) only considers overflow in the denominator measure, but not overflow in sub-measures computed on the two input images on which the denominator measure depends. Summation in the denominator is used in all currently implemented scale-plane measures, and avoiding overflow in the sum also avoids overflow in the summands.}

virtual void

normalization

{\begin{verbatim}
    (const SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &jointMeasure,
    const SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &ssMeasure,
    const SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &cvMeasure,
    const SIMDImage<MeasType,SIMD_WIDTH,SIMD_ALIGN> &combMeasure,
    double postScale,
    SIMDImage<SPSType,SIMD_WIDTH,SIMD_ALIGN> &scalePlane,
    const WarpingParameter<SPSType,SIMD_WIDTH> &par) = 0;
\end{verbatim}}

takes measures (type \texttt{MeasType}) computed from \texttt{columnMeasure()} or \texttt{jointMeasure()} (parameters \texttt{ssMeasure} and \texttt{cvMeasure}) and from \texttt{jointMeasure()} (parameters \texttt{combMeasure} and \texttt{jointMeasure}) and computes a normalized measure that is stored in the scale plane \texttt{scalePlane} (type \texttt{SPSType}). The factor \texttt{postScale} is applied to the normalized measure before it is stored in the scale plane.

virtual double

maxDenom

{\begin{verbatim}
    (const SIMDImage<ImgType,SIMD_WIDTH,SIMD_ALIGN> &img) const = 0;
\end{verbatim}}

computes the maximum denominator of the normalized measure for an example image \texttt{img}; a maximum over multiple images should be computed and passed to \texttt{idealPixelScale()}.\footnote{Note that this mechanism may not be general enough for arbitrary normalized distance measures: \texttt{idealPixelScale()} (see below) only considers overflow in the denominator measure, but not overflow in sub-measures computed on the two input images on which the denominator measure depends. Summation in the denominator is used in all currently implemented scale-plane measures, and avoiding overflow in the sum also avoids overflow in the summands.}
virtual double
measureMax () const = 0;

has to return the maximal value of the normalized measure (without post-normalization, i.e. such as it is defined mathematically; see e.g. the range specifications in equations (15) and (16)). The minimal normalized measure is always assumed to be zero.

virtual bool
validForWidth(int width) const;

returns true iff input images of a given width can be processed. This function has a default implementation.

The following methods are currently non-virtual:

double
idealPixelScale(double maxDenom,
    double measTypeMax = double(SIMDTypeInfo<MeasType>::max())) const;

computes the ideal pixel scale (for scaling of pixels in transformations to MeasType) from the maximum denominator computed for a set of images (using the method maxDenom()). The parameter measTypeMax is the maximal value MeasType can hold; as a default, the maximum of the data type is used, but for SIMDFloat a different value could be used. The value return from pixelScale() is chosen such that the maximum denominator of a normalized measure multiplied by this value does not exceed the range of MeasType.

double
idealPostScale(double maxElemSearchPhase,
    double spsTypeMax = double(SIMDTypeInfo<SPSType>::max())) const;

computes the ideal post scale, the factor used to transform a normalized measure into an element of the scale-plane stack. It is computed from maxElemSearchPhase — the maximal value an element of the scale-plane stack should not exceed in the search phase to avoid overflow there —, and either the maximum of the type SPSType (default) or some other value (e.g. used if SPSType is SIMDFloat). Two conditions are met: The maximal value of the normalized measure (from measureMax(), see above) multiplied by the post scale should neither exceed spsTypeMax nor maxElemSearchPhase.

virtual void
computeSPS
    (const SIMDImage<ImgType,SIMD_WIDTH,SIMD_ALIGN> &ss,
const SIMDImage<ImgType,SIMD_WIDTH,SIMD_ALIGN> &cv,
double verticalResolution,
double horizon,
const std::vector<double> &scaleFactors,
int interpolation,
double pixelScale,
double postScale,
WarpingSPS<SPSType,SIMD_WIDTH,SIMD_ALIGN> &sps) = 0;

computes the entire scale-plane stack (type SPSType) from snapshot (ss) and current view (cv) of type ImgType. It calls the virtual functions explained above. The parameters have the following meaning:

- `verticalResolution` (radians per pixel) is the vertical resolution of the input images; if it is less or equal than zero, an approximative version of equation (3) is used ($r'/r = \gamma/\gamma'$) and the parameter value is ignored, otherwise magnification uses equation (3).
- `horizon` is the position of the horizon (relates to the row index but can be non-integer),
- `scaleFactors` is an array of scale factors $\sigma$ used in the computation of the different scale planes,
- `interpolation` relates to the image magnification and is 0 for nearest-neighbor interpolation or 1 for linear interpolation,
- `pixelScale` is a pixel factor which is used in the transformation from ImgType to MeasType, and
- `postScale` is factor which is used in the transformation from normalized measures to SPSType.

Currently, the following template classes are derived from the template class WarpingSPSComputation:

- A template class WarpingSPSComputation1 is derived from WarpingSPSComputation where the method computeSPS() is defined as it is the same for all derived classes.
- A template class WarpingSPSComputation2 is derived from WarpingSPSComputation1; it defines the methods convertImage(), magnifyImage() and copyImage().
- A template class WarpingSPSComputationEdgeAbs derived from WarpingSPSComputation2 defines the methods preprocessing() (edge-filtering), correctHorizon() (subtract 1), columnMeasure() (sum of absolute values), and maxDenom() (for edge-filtering and sum of absolute
values).

- Two intermediate template classes `WarpingSPSComputationEdgeAbs[12]` define `normalization()` (1, 2) and `reArrange()` (2 only) for the version with element-wise rewrite to the SPS (1) and with image re-arrangement and block-wise rewrite to the SPS (2).

- From the template classes `WarpingSPSComputationEdgeAbs[12]`, the template classes `WarpingSPSComputationEdgeNSAD[12]` and `WarpingSPSComputationEdgeASC[12]` are derived which implement the methods `jointMeasure()` and `measureMax()`.

NSAD refers to the distance measure “normalized sum of absolute differences”

\[
J_{NSAD}(a', b') = \frac{\sum |a'_i - b'_i|}{\sum |a'_i| + \sum |b'_i|} \in [0, 1]
\]

(15)

and ASC+ to a version of “approximated sequential correlation” (Möller et al., 2014):

\[
J_{ASC+}(a', b') = 1 - \frac{\sum_i (|a'_i + b'_i| - |a'_i - b'_i|)}{\sum_i |a'_i| + \sum_i |b'_i|} \in [0, 2].
\]

(16)

where the computation of the numerator is accelerated by using

\[
|a'_i + b'_i| - |a'_i - b'_i| = 2 \max\{\min(a'_i, b'_i), -\max(a'_i, b'_i)\}.
\]

(17)

The vectors \(a'\) and \(b'\) hold a pair of edge-filtered image columns, and \(i\) is a row index. NSAD and ASC+ perform about equally well without strong illumination changes, but NSAD performs noticeably better under strong illumination changes (cross-database experiment with snapshots from database “day” and current views from database “night”), see table 1.

NSAD and ASC+ are implemented in two versions which are described below.

5.1.1. Version 1

In the first version (template classes `WarpingSPSComputationEdgeNSAD1` and `WarpingSPSComputationEdgeASC1`), the two input images are not re-arranged (method `reArrange()` remains empty), thus both snapshot and current view are stored with horizontal indices corresponding to the azimuth angles \((\Theta, \Theta')\). The same horizontal indices are also used for the measures computed from `columnMeasure()`.

For the joint measure computed by `jointMeasure()`, both the horizontal and vertical index relate to the azimuth angle. However, scale planes are organized with the horizontal index corresponding to \(\Theta\) and the vertical index corresponding to \(\delta = \Theta' - \Theta\).
Table 1: Comparison of mean angular errors (mean AE, in radians) of home vector ($\beta$) and compass ($\psi$) for MinWarping with NSAD and ASC+ as distance measure. In all cases, NSAD improves the mean angular errors ($w = 288$, $n_\alpha = n_\psi = 96$, 9 scale planes with maximal scale factor 2.0 and maximal threshold 2.5, Butterworth cutoff 0.10, full search, double search, all snapshot/current view pairs in each second grid row and column).

<table>
<thead>
<tr>
<th>databases</th>
<th>NSAD mean AE $\beta$</th>
<th>NSAD mean AE $\psi$</th>
<th>ASC+ mean AE $\beta$</th>
<th>ASC+ mean AE $\psi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>living1 day-day</td>
<td>0.052</td>
<td>0.026</td>
<td>0.054</td>
<td>0.029</td>
</tr>
<tr>
<td>living2 day-day</td>
<td>0.058</td>
<td>0.030</td>
<td>0.062</td>
<td>0.031</td>
</tr>
<tr>
<td>living3 day-day</td>
<td>0.044</td>
<td>0.023</td>
<td>0.046</td>
<td>0.024</td>
</tr>
<tr>
<td>living4 day-day</td>
<td>0.035</td>
<td>0.029</td>
<td>0.039</td>
<td>0.030</td>
</tr>
<tr>
<td>living1 day-night</td>
<td>0.209</td>
<td>0.063</td>
<td>0.251</td>
<td>0.080</td>
</tr>
<tr>
<td>living2 day-night</td>
<td>0.236</td>
<td>0.147</td>
<td>0.259</td>
<td>0.202</td>
</tr>
<tr>
<td>living3 day-night</td>
<td>0.381</td>
<td>0.124</td>
<td>0.560</td>
<td>0.180</td>
</tr>
<tr>
<td>living4 day-night</td>
<td>0.116</td>
<td>0.038</td>
<td>0.120</td>
<td>0.040</td>
</tr>
</tbody>
</table>

and are shuffled (see section 4.3.1). Therefore, when a scale plane is computed using the virtual function `normalization()` (ending in the image processing function `computeScalePlaneNormalizedAdd1()`), the entries have to be inserted sequentially into the scale plane which leads to performance penalties due to inefficient cache use. The function `computeScalePlaneNormalizedAdd1()` takes a single element from the current-view measure, duplicates it into all elements of a vector, and interrelates this vector with a vector of elements from the snapshot measure and the joint measure. Only aligned load and store instructions are used.

### 5.1.2. Version 2

In the second version (template classes `WarpingSPSComputationEdgeNSAD2` and `WarpingSPSComputationEdgeASC2`), the two images are shuffled in `reArrange()` in the same way as in the computation of the scale-plane stack (see section 4.3.1). If $j_s$ is the index in the shuffled snapshot and $j_c$ is the index in the shuffled current view, we have $j_s = \text{shuffle}(j_\Theta)$, $j_\Theta = \text{unshuffle}(j_s)$, and $j_c = \text{shuffle}(j_\Theta')$ where $j_\Theta' = (j_\Theta + j_\delta) \mod w$. Index $j_s$ coincides with the horizontal index of the scale planes. We can derive the following relation for $j_c$:

$$j_c = \text{shuffle}(j_\Theta') = \text{shuffle}([j_\Theta + j_\delta] \mod w) = \text{shuffle}([\text{unshuffle}(j_s) + j_\delta] \mod w).$$

This is identical to equation (8) with $d = j_\delta$. Following the same argumentation as in section 4.3.3, contiguous indices $j_s$ in the shuffled snapshot lead to contiguous indices $j_c$. 

Table 2: Comparison of run-times (in ms) between version 1 and 2 of the scale-plane-stack computation for the NSAD measure \((w = 288, 9\) scale planes) for different Intel CPUs. The column “SIMD” shows which SIMD vector instruction set was used.

<table>
<thead>
<tr>
<th>CPU</th>
<th>clock</th>
<th>SIMD</th>
<th>NSAD1</th>
<th>NSAD2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom N2600</td>
<td>1.60GHz</td>
<td>SSSE3</td>
<td>29.7</td>
<td>37.1</td>
</tr>
<tr>
<td>i7-3610QM</td>
<td>2.30GHz</td>
<td>SSE4.2</td>
<td>3.4</td>
<td>3.3</td>
</tr>
<tr>
<td>i7-4790K</td>
<td>4.00GHz</td>
<td>SSE4.2</td>
<td>2.8</td>
<td>2.3</td>
</tr>
<tr>
<td>i7-4790K</td>
<td>4.00GHz</td>
<td>AVX2</td>
<td>2.1</td>
<td>1.4</td>
</tr>
</tbody>
</table>

(possibly extending into the \(\Delta\) gaps) in the current view. Thus, in row \(j_{s}\) of a scale plane, elements \(j_{s} \ldots j_{s} + 1\) can be computed by interrelating elements \(j_{s} \ldots j_{s} + s - 1\) in the snapshot (and in the column measure of the snapshot which has the same horizontal order) to elements \(j_{c} \ldots j_{c} + s - 1\) in the current view (and its column measure). Therefore the loop structure in `computeScalePlaneNormalizedAdd2()` differs from the one in `computeScalePlaneNormalizedAdd1()`: \(s\) contiguous elements of vectors from snapshot and current view are paired with each other (rather than duplicating elements in one vector). However, in this case one of the vector load instructions must be capable of unaligned access. Table 2 compares the time required for the computation of the scale-plane stack for different Intel CPUs using different SIMD vector instruction sets. A noticeable gain of NSAD2 over NSAD1 is achieved for the most modern CPU, in particular if used with its native AVX2 instruction set. However, for the Atom CPU with SSSE3, NSAD1 is faster. A possible explanation is a penalty for unaligned memory access in the older architectures.

5.2. Search: **MinWarpingFull, MinWarpingPartial**

In the search phase of MinWarping, a match array (template class `MinWarpingMatch`) is computed from a scale-plane stack (template class `WarpingSPS`) based on equation (1), using the MinWarping template (template class `MinWarpingTemplate`) in the process.

5.2.1. **Full Search: MinWarpingFull**

In a full search, the search extends over the entire range of values for \(i_{\alpha} = 0 \ldots n_{\alpha} - 1\) and \(i_{\psi} = 0 \ldots n_{\psi} - 1\). The template class

```cpp
template <typename SPSType, typename MatchType, int SIMD_WIDTH, int SIMD_ALIGN>
class MinWarpingFull
{}
```
is the abstract interface for full search. Currently, only a single template class `MinWarpingFull_XPAY` is derived from `MinWarpingFull` and defines `search()`. In the derived class, equation (1) is not implemented in the “natural” loop order indicated by the equation (“PAXY”: $\psi \to \alpha \to x \to y$), but in the order “XPAY” ($x \to \psi \to \alpha \to y$). Search in the order XPAY was the fastest or among the fastest for different architectures of a range of different loop orders. In contrast to PAXY where summation is done in a register, summation has to be done in memory (on the match array) for XPAY. It is not clear why this loop order is better than the natural one. One guess is that fixing the value of $x$ in the three inner loops causes the search to operate only on one block of the shuffled scale-plane stack at a time. An attempt to enforce block-wise processing even in the outer loop by varying $x$ in steps of $\Delta_x$ resulted in minor performance improvements and only with AVX2 (SIMD_WIDTH=32).

### 5.2.2. Partial Search: MinWarpingPartial

In a partial search, the range of indices $i_\alpha$ and $i_\psi$ is restricted.\(^ {10} \) The template class `MinWarpingSearchRange` (see section 4.5) is used to encode the search ranges for the two indices:

```cpp
template <typename SPSType, typename MatchType, int SIMD_WIDTH, int SIMD_ALIGN>
class MinWarpingPartial
{
public:
  virtual ~MinWarpingPartial() {}
  virtual void
  search(const MinWarpingTemplate &minTemplate,
          const WarpingSPS<SPSType,SIMD_WIDTH,SIMD_ALIGN> &sps,
          const MinWarpingSearchRange<SPSType,SIMD_WIDTH,SIMD_ALIGN> &range,
          MinWarpingMatch<MatchType,SIMD_WIDTH,SIMD_ALIGN> &match,
          int jAlphaOff = 0, int jPsiOff = 0) = 0;
};
```

The arguments `jAlphaOff` and `jPsiOff` are used for fine search (see section 6.7). Currently, only a single template class `MinWarpingPartial_XPAY` is derived from `MinWarpingPartial` and defines `search()`. As in full search, the loop order is “XPAY”. At the beginning of partial search, the match array is filled with invalid, and all entries that will be touched by the search are initialized with zero (since XPAY accumulates the match values).

---

\(^ {10} \)Please note that “partial search” is unfortunately used in two meanings in this document: In this paragraph it refers a general mechanism for searching through arbitrary parts of the parameter space. In section 6.3 it refers to a search through contiguous ranges in the two parameter spaces.
6. Warping Compound: WarpingCompound

The entire functionality of the data and algorithm classes described above is combined in the template class

```
template <typename SPSType, typename MatchType,
          typename CompassType,
          int SIMD_WIDTH, int SIMD_ALIGN>
class WarpingCompound;
```

where SPSType is the element type of the scale-plane stack, MatchType is the element type of the match array, and CompassType is the element type used for the compass estimate. WarpingCompound contains one instance of WarpingParameter (section 4.1), two instances of WarpingSPS (section 4.3), and 8 instances of MinWarpingMatch (section 4.4), the duplicated instances being used for double search (section 6.5). WarpingCompound currently contains two instantiations of the template class ComplexSearch, one for MinWarping (minWarping), one for 2D-Warping (warping) for which a test implementation is provided as well (not described here). These instantiations define the search functions described below. The constructor of WarpingCompound is defined as

```
WarpingCompound(int width, int nAlpha, int nPsi, int nScalePlanes,
                 double maxScaleFactor,
                 double maxThresholdMinWarping,
                 double rhoMinMinWarping, double rhoMaxMinWarping)
                 double rhoMinWarping, double rhoMaxWarping,
                 int nRhoWarping)
```

Its parameters are explained in section 4.1 and section 4.2. The suffixes MinWarping and Warping indicate that these parameters are only used for one of the warping methods.

6.1. Scale-Plane-Stack Computation: computeSPS()

The computation of the scale-plane stack is accomplished in the method

```
template <typename ProcType, typename MeasType, typename ImgType>
void
```

11 Note that this reference member makes the template class WarpingCompound non-copyable.
12 A template class was used since the implementation of MinWarping and 2D-Warping is kept separate with different template classes for search templates and search algorithms.
which is just a wrapper that passes all arguments to the method `computeSPS()` of a template class derived from `WarpingSPSComputation` (see section 5.1). It updates the first copy of the scale-plane stack.

### 6.2. Full Search: `minWarping.full()`

Full search is implemented by a `ComplexSearch` wrapper method `full()` (shown with the template arguments inserted for `WarpingCompound::minWarping`) that calls the method `search()` of a template class derived from `MinWarpingFull` (see section 5.2.1):

```cpp
MinWarpingMatch<MatchType,SIMD_WIDTH,SIMD_ALIGN>*
full(MinWarpingFull<SPSType,MatchType,SIMD_WIDTH,SIMD_ALIGN>
    &searchAlgorithm, bool doubleSearch);
```

The parameter `doubleSearch` decides between single and double search (see section 6.5). The method returns a pointer to a match array containing the search result.

### 6.3. Partial Search: `minWarping.partial()`

Partial search in a range of the angles $\alpha$ and $\psi$ is implemented by the `ComplexSearch` wrapper method `partial()` (shown below with the template arguments inserted for `WarpingCompound::minWarping`). The method creates a `MinWarpingSearchRange` around estimates $\alpha_{\text{Est}}$ and $\psi_{\text{Est}}$ within a radius $\alpha_{\text{Radius}}$ and $\psi_{\text{Radius}}$ and passes this to the method `search()` of a template class derived from `MinWarpingPartial` (see section 5.2.2):

```cpp
MinWarpingMatch<MatchType,SIMD_WIDTH,SIMD_ALIGN>*
partial(MinWarpingPartial<SPSType,MatchType,SIMD_WIDTH,SIMD_ALIGN>
    &searchAlgorithm, bool doubleSearch,
    double alphaEst, double alphaRadius,
    double psiEst, double psiRadius);
```

The meaning of `doubleSearch` and the return value are the same as described above.
6.4. Compass Search: \texttt{minWarping.compassAcceleration()}

Compass search is a partial search (section 5.2.2) through the entire range of $i_{\alpha}$ but through selected values of $i_{\psi}$. The latter are computed from the compass estimate described in section 4.3.4. In the \texttt{ComplexSearch} wrapper method \texttt{compassAcceleration()} (shown below with the template arguments inserted for \texttt{WarpingCompound::minWarping}), a parameter \texttt{psiFraction} specifies which percentage of $i_{\psi}$ values with the best compass estimates is considered in the search:

\begin{verbatim}
MinWarpingMatch<MatchType,SIMD_WIDTH,SIMD_ALIGN>*
compassAcceleration
  (MinWarpingPartial<SPSType,MatchType,SIMD_WIDTH,SIMD_ALIGN>
   &searchAlgorithm,
   bool doubleSearch,
   double psiFraction);
\end{verbatim}

The meaning of \texttt{doubleSearch} and the return value are the same as described above.

6.5. Double Search in Full, Partial, and Compass Search

The search in equation (1) takes a column in the snapshot and searches for the best-matching column in the current view. We found that the homing performance of MinWarping can be improved by performing a second search where the role of snapshot and current view are exchanged (taking a current view column and searching for the best-matching column in the snapshot); this modification was suggested by Gedicke (2012). The match arrays resulting from the two search passes is averaged. Table 3 shows that double search typically improves the homing performance.\textsuperscript{13} This improvement can be explained as follows: MinWarping searches for \textit{each} column in the snapshot for the best-matching column in the current view, thus each snapshot column is part of a best-matching column pair, but not each current-view column; the second search produces column pairs where each current-view column but not each snapshot column is used.

For full, partial and compass search described above, versions with double search are executed if \texttt{doubleSearch = true}; they invert the scale-plane stack (section 4.3.3), perform a second search, invert the match array (section 4.4.1), and average the match arrays (section 4.4.3). The unresolved template code for full search is

\begin{verbatim}
template<
  template <typename,typename,int,int>
  class FullSearcherClass>
MinWarpingMatch<MatchType,SIMD_WIDTH,SIMD_ALIGN>*
\end{verbatim}

\textsuperscript{13}This result indicates that performance could possibly further be improved by computing the scale-plane stack for different representations of the input images (e.g. different low-pass filter strengths, edge filters with different order), perform a search for each scale-plane stack, and average the resulting match arrays.
Table 3: Comparison of mean angular errors (mean AE, in radians) of home vector (\(\beta\)) and compass (\(\psi\)) for MinWarping with single and double search. In all but one case (emphasized entries), double search improves the mean angular errors \((w = 288, n_q = n_\psi = 96, 9\) scale planes with maximal scale factor \(2.0\) and maximal threshold \(2.5\), full search, Butterworth cutoff \(0.10\), NSAD, all snapshot/current view pairs in each second grid row and column).

<table>
<thead>
<tr>
<th>databases</th>
<th>single search mean AE (\beta)</th>
<th>single search mean AE (\psi)</th>
<th>double search mean AE (\beta)</th>
<th>double search mean AE (\psi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>living1 day-day</td>
<td>0.072</td>
<td>0.035</td>
<td>0.052</td>
<td>0.026</td>
</tr>
<tr>
<td>living2 day-day</td>
<td>0.069</td>
<td>0.040</td>
<td>0.052</td>
<td>0.030</td>
</tr>
<tr>
<td>living3 day-day</td>
<td>0.059</td>
<td>0.031</td>
<td>0.044</td>
<td>0.023</td>
</tr>
<tr>
<td>living4 day-day</td>
<td>0.040</td>
<td>0.031</td>
<td>0.035</td>
<td>0.029</td>
</tr>
<tr>
<td>living1 day-night</td>
<td>0.371</td>
<td>0.177</td>
<td>0.209</td>
<td>0.063</td>
</tr>
<tr>
<td>living2 day-night</td>
<td>0.422</td>
<td>0.276</td>
<td>0.236</td>
<td>0.147</td>
</tr>
<tr>
<td>living3 day-night</td>
<td>0.370</td>
<td>0.155</td>
<td>0.381</td>
<td>0.124</td>
</tr>
<tr>
<td>living4 day-night</td>
<td>0.239</td>
<td>0.083</td>
<td>0.116</td>
<td>0.038</td>
</tr>
</tbody>
</table>

In partial double search in ranges of \(\alpha\) and \(\psi\), the search range centers have to be inverted as well (see section 4.4.1):

```cpp
3 template<template <typename,typename,int,int> class PartialSearcherClass>
4  MinWarpingMatch<MatchType,SIMD_WIDTH,SIMD_ALIGN> *
5      partial(PartialSearcherClass<SPSType,MatchType,SIMD_WIDTH,SIMD_ALIGN> &searchAlgorithm,
6      bool doubleSearch,  
7      double alphaEst, double alphaRadius,
8      double psiEst, double psiRadius)
9  {
```
In double search with compass acceleration, the compass estimate is computed again for the second search pass (it is probably possible to also just invert the compass estimate from the first pass):

```cpp
10  range(param.nAlpha, param.nPsi);
11  spsWasInverted = doubleSearch;
12  // first run
13  range.setPartialRange(alphaEst, alphaRadius, psiEst, psiRadius);
14  searchAlgorithm.search(*searchTemplate,
15    *(spsArray[SPS_ORIG]), range,
16    *(matchArray[MATCH_1]));
17  if (!doubleSearch)
18    return (matchResPtr = matchArray[MATCH_1]);
19  // second run
20  // we have to invert the search range centers as well for second
21  // computation
22  double psiEstInv = 2.0 * M_PI - psiEst;
23  double alphaEstInv = M_PI + alphaEst - psiEst;
24  range.setPartialRange(alphaEstInv, alphaRadius, psiEstInv, psiRadius);
25  searchAlgorithm.search(*searchTemplate,
26    *(spsArray[SPS_INV]), range,
27    *(matchArray[MATCH_2]));
28  // average
29  matchArray[MATCH_2_INV]->invertFrom(*(matchArray[MATCH_2]));
30  matchArray[MATCH_RES]->averageOf(*(matchArray[MATCH_1]),
31    *(matchArray[MATCH_2_INV]));
32  return (matchResPtr = matchArray[MATCH_RES]);
33 }
```

```
In double search with compass acceleration, the compass estimate is computed again for the second search pass (it is probably possible to also just invert the compass estimate from the first pass):

```cpp
1 template<template <typename,typename,int,int> class PartialSearcherClass>
2 MinWarpingMatch<MatchType,SIMD_WIDTH,SIMD_ALIGN>*
3 compassAcceleration
4 (PartialSearcherClass<SPSType,MatchType,SIMD_WIDTH,SIMD_ALIGN>
5   &searchAlgorithm,
6   bool doubleSearch,
7   double psiFraction)
8 {
9    SIMDImage<CompassType,SIMD_WIDTH,SIMD_ALIGN> compass;
10   MinWarpingSearchRange<SPSType,SIMD_WIDTH,SIMD_ALIGN>
11    range(param.nAlpha, param.nPsi);
12   spsWasInverted = doubleSearch;
13   // first run
14   spsArray[SPS_ORIG]->compassEstimate(compass);
15   range.setFullCompassRange(psiFraction, compass);
16   searchAlgorithm.search(*searchTemplate,
17     *(spsArray[SPS_ORIG]), range,
18     *(matchArray[MATCH_1]));
19   if (!doubleSearch)
20     return (matchResPtr = matchArray[MATCH_1]);
21   // second run
22   spsArray[SPS_INV]->invertFrom(*{spsArray[SPS_ORIG]});
23   spsArray[SPS_INV]->compassEstimate(compass);
24   range.setFullCompassRange(psiFraction, compass);
25   searchAlgorithm.search(*searchTemplate,
26     *(spsArray[SPS_INV]), range,
27     *(matchArray[MATCH_2]));
28   // average
29   matchArray[MATCH_2_INV]->invertFrom(*{matchArray[MATCH_2]});
30   matchArray[MATCH_RES]->averageOf(*{matchArray[MATCH_1]},
31     *(matchArray[MATCH_2_INV]));
32   return (matchResPtr = matchArray[MATCH_RES]);
33 }
```
6.6. Finding the Best Match: \texttt{minWarping.bestMatchFull()}

The method

\begin{verbatim}
void bestMatchFull(int &iAlphaMin, double &alphaMin,
int &iPsiMin, double &psiMin,
double &dMin);
\end{verbatim}

looks for the best match. It uses the method of the same name of \texttt{WarpingSPS} (section 4.4.4) and in addition provides the angular solution \((\alpha, \psi)\).

6.7. Fine Search: \texttt{minWarping.fine()}

The resolution of the two search parameters \(i_\alpha, i_\psi\) is typically worse than the pixel resolution: The search template is shifted over the scale-plane stack in steps of \(\Delta_\alpha, \Delta_\psi\). This limits the resolution of the result. To refine the solution, a fine search can be executed after one of the search operations described above. A fine search is a partial search between the neighbors of the best solution, but with pixel resolution. Here the parameters \(j_{\alpha, \psi}\) (see section 5.2.2) are varied systematically and the minimal match value at the coarse solution is determined. The offsets for the minimal match value are then used to compute corrected solution angles \(\alpha, \psi\).

The method \texttt{fine()} (template definition shown unresolved below) takes the \texttt{searchAlgorithm} (that should be the same as in the preceding search), the \texttt{doubleSearch} argument (typically also the same as before), and the best coarse solution \(i_{\alpha, \psi}\) (parameter indices). It computes the best pixel offsets \(j_{\alpha, \psi}\), the corrected best angular solution \((\alpha, \psi)\), and the refined match value \(\text{matchBest}\). It also updates an image \texttt{fineMatch} with all match values in pixel resolution. The optional parameter \texttt{surround} defines whether the search extends between the two neighboring solutions (false) or only covers the surrounding elements of the given solution (true).

\begin{verbatim}
template<template <typename,typename,int,int>
class PartialSearcherClass>
void fine
(PartialSearcherClass<SPSType,MatchType,SIMD_WIDTH,SIMD_ALIGN>
&searchAlgorithm,
bool doubleSearch, int iAlpha, int iPsi,
int &jAlphaOffBest, double &alphaBest,
int &jPsiOffBest, double &psiBest,
double &matchBest,
SIMDImage<MatchType,1,1> &fineMatch,
bool surround = false);
\end{verbatim}
Table 4: Mean angular errors (MAE, in radians) without fine search, with fine search and surround = false, with fine search and surround = true ($w = 288$, $n_x = n_y = 96$, 9 scale planes with maximal scale factor 2.0 and maximal threshold 2.5, partial search, search radii $\pi/4$, double search gridStep = 1, boxRad = 3; see section 7).

<table>
<thead>
<tr>
<th>databases</th>
<th>coarse MAE $\beta$</th>
<th>coarse MAE $\psi$</th>
<th>fine MAE $\beta$</th>
<th>fine MAE $\psi$</th>
<th>fine surround MAE $\beta$</th>
<th>fine surround MAE $\psi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>living1 day-day</td>
<td>0.067</td>
<td>0.021</td>
<td>0.060</td>
<td>0.013</td>
<td>0.063</td>
<td>0.013</td>
</tr>
<tr>
<td>living2 day-day</td>
<td>0.057</td>
<td>0.020</td>
<td>0.050</td>
<td>0.012</td>
<td>0.052</td>
<td>0.012</td>
</tr>
<tr>
<td>living3 day-day</td>
<td>0.068</td>
<td>0.020</td>
<td>0.061</td>
<td>0.013</td>
<td>0.063</td>
<td>0.013</td>
</tr>
<tr>
<td>living4 day-day</td>
<td>0.055</td>
<td>0.018</td>
<td>0.049</td>
<td>0.008</td>
<td>0.051</td>
<td>0.008</td>
</tr>
<tr>
<td>living1 day-night</td>
<td>0.266</td>
<td>0.031</td>
<td>0.265</td>
<td>0.028</td>
<td>0.265</td>
<td>0.028</td>
</tr>
<tr>
<td>living2 day-night</td>
<td>0.265</td>
<td>0.031</td>
<td>0.265</td>
<td>0.029</td>
<td>0.265</td>
<td>0.028</td>
</tr>
<tr>
<td>living3 day-night</td>
<td>0.474</td>
<td>0.076</td>
<td>0.476</td>
<td>0.074</td>
<td>0.475</td>
<td>0.074</td>
</tr>
<tr>
<td>living4 day-night</td>
<td>0.215</td>
<td>0.029</td>
<td>0.213</td>
<td>0.026</td>
<td>0.214</td>
<td>0.026</td>
</tr>
</tbody>
</table>

For double fine search, not only the best coarse solution has to be inverted according to equations (13, 14), but also the fine-resolution offsets have to be inverted. If we introduce angular offsets $\delta$ into equations (11, 12), we obtain

\[
\psi_2 + \delta_\psi = 2\pi - (\psi_1 + \delta_\psi_1) \tag{18}
\]
\[
\alpha_2 + \delta_\alpha = \pi + (\alpha_1 + \delta_\alpha_1) - (\psi_1 + \delta_\psi_1) \tag{19}
\]

By subtracting equations (11, 12), we obtain

\[
\delta_\psi_2 = -\delta_\psi_1 \tag{20}
\]
\[
\delta_\alpha_2 = \delta_\alpha_1 - \delta_\psi_1 \tag{21}
\]

which can easily be converted to offset indices with pixel resolution.

The mean angular errors in table 4 were obtained for a local grid search (gridStep = 1, boxRad = 3) with partial search and either no fine search, with fine search and surround = false, and with fine search and surround = true. The wider fine search produces the best results. Precision improvements are noticeable in tests with small illumination changes (day-day), particularly for the mean angular error of the compass ($\psi$).

6.8. Time Measurements

Table 5 lists computation times for the different search methods described above on different Intel CPUs with different SIMD vector instruction sets.
Table 5: Computation times (in ms) of the search phase for different Intel CPUs for full search, full search with fine search, partial search (search radii $\pi / 4$), partial search with fine search, search with compass acceleration (30%), search with compass acceleration with fine search ($w = 288$, $n_\alpha = n_\psi = 96$, 9 scale planes with maximal scale factor 2.0 and maximal threshold 2.5, double search, fine search with $\text{surround} = \text{false}$; gcc 4.6 on Atom, gcc 4.8 on other CPUs).

<table>
<thead>
<tr>
<th>CPU</th>
<th>clock [GHz]</th>
<th>SIMD</th>
<th>full</th>
<th>full +fine</th>
<th>part.</th>
<th>part. +fine</th>
<th>com.a.</th>
<th>com.a. +fine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom N2600</td>
<td>1.60</td>
<td>SSSE3</td>
<td>327.0</td>
<td>372.0</td>
<td>37.2</td>
<td>80.5</td>
<td>94.0</td>
<td>136.0</td>
</tr>
<tr>
<td>i7-3610QM</td>
<td>2.30</td>
<td>SSE4.2</td>
<td>22.2</td>
<td>23.9</td>
<td>2.7</td>
<td>4.4</td>
<td>6.9</td>
<td>8.5</td>
</tr>
<tr>
<td>i7-4790K</td>
<td>4.00</td>
<td>SSE4.2</td>
<td>16.0</td>
<td>17.1</td>
<td>1.9</td>
<td>3.1</td>
<td>4.9</td>
<td>6.0</td>
</tr>
<tr>
<td>i7-4790K</td>
<td>4.00</td>
<td>AVX2</td>
<td>9.5</td>
<td>10.8</td>
<td>1.6</td>
<td>2.9</td>
<td>3.0</td>
<td>4.2</td>
</tr>
</tbody>
</table>

7. Test Programs

7.1. Small Test Program

The small test program `warpingSIMDSmallTest.C` is helpful as an introduction to the use of the code. It provides a simple class `WarpingExample`. In the constructor, many parameters are set to default values. The method `run()` executes first and second phase of MinWarping and determines the best match. In the `main()` function, the function `WarpingExample::run()` is applied to fixed snapshot and current view from a fixed database and the results are printed to stdout.

7.2. Large Test Program

A comprehensive test program `warpingSIMDTest.C` is provided which demonstrates how to use the different components of this library.

The program has integer (INT) and double (DOUBLE) program parameters with default values which can be overwritten with environment parameters of the given name (e.g. in tcsh: `setenv firstPhase 2`). If the variable `printParameters` is set to `PRINT_OVERWRITTEN_PARAMETERS`, all parameters which deviate from their default values are printed to stderr. Call `warpingSIMDTest` without command line arguments to see all overwritten parameters.

The code creates an object `warping` of the class `WarpingBundle` (which is a wrapper of all components), influenced by the program parameters. The function `warping.run()` executes the two phases of MinWarping: scale-plane stack computation (section 5.1) and search (section 5.2), the latter including an optional fine search.
(section 6.7). It also searches for the best match. The computation times of the two phases are determined.

The main function offers different operations depending on the command line parameters. Results are printed to stdout (preceded by a string for a grep of the output), additional information is printed to stderr.

The first parameter `<action>` defines which action is executed.

For `<action> = x`, the program explores different values for image width ($w$) and step width ($\Delta_\alpha = \Delta_\psi$) and tries to create and run MinWarping. It prints all combinations to stdout for which this succeeded. This list can be consulted to select supported combinations of $w$ and $n_\alpha, n_\psi$ (depends on SIMD_WIDTH).

All other actions have at least the following 5 command-line arguments (in the given order):

- `<base>` is the base name of the image database, e.g. `living3`.
- `<ss>` is the suffix of the image database from which the snapshots are loaded, e.g. day or night.
- `<cv>` is the suffix of the image database from which the current views are loaded, e.g. day or night.
- `<suffix>` is the suffix characterizing the image database, e.g. Hh288sh (in this example meaning Hyperbolic mirror mapping, histogram equalization, width $w = 288$, all images with the same height).
- `<bw>` is a string characterizing the low-pass filter (Butterworth) cutoff parameter used in the preprocessing step when the databases were generated (using a different program not contained in the distribution), e.g. 0.10 (cutoff including 10% of the lowest frequencies).

When a database is loaded based on the specified parameters, the program randomly rotates images in azimuthal direction to make the navigation task more demanding. The image databases are described below.

For `<action> = p` (no further command-line arguments required), the program computes and prints the optimal `pixelScale` and `postScale`. These values can be used to set the program parameters with the same name. Grep stdout for `SCALING`.

For `<action> = s` the program computes the home vectors for a single snapshot (snapshot grid coordinates given in two additional command-line arguments) and all other current views, writes the home vectors into a file (into the directory `TESTPATH` specified in the program) which can be visualized by gnuplot. The program measures the average angular errors of home vector and compass and the average times required for the two phases of min-warping; grep stdout for `RESULTS`.

33
For \texttt{\textless action\textgreater} = \texttt{v} the program performs a computation for a single snapshot (grid coordinates given in two additional command-line arguments) and a single current view (two more grid coordinates on the command line). Numerous images (e.g. scale-plane stack, match arrays) and other data (e.g. search template) produced by MinWarping are stored to the directory specified in \texttt{TESTPATH}.

For \texttt{\textless action\textgreater} = \texttt{a}, the program computes home vectors for multiple snapshots and multiple current views. Two additional command-line arguments specify the selection of the images: The argument \texttt{\textless gridStep\textgreater} defines a sub-grid of images to use, and the argument \texttt{\textless boxRad\textgreater} specifies the half-width (radius) of a grid box around the snapshot image from which current view images are selected. The program measures the average angular errors of home vector and compass and the average times required for the two phases of min-warping; grep stdout for \texttt{RESULTS}.

For \texttt{\textless action\textgreater} = \texttt{l} (with snapshot grid coordinates as additional command-line arguments), the program performs localization using the visual compass from the first phase of MinWarping. It writes a gnuplot file with the match values and prints the time measurement to stdout (code based on a contribution by Michael Horst).

Some scripts are provided which use \texttt{warpingSIMDTest} in the different \texttt{\textless action\textgreater} modes:

- \texttt{runWarpingSIMDTest} uses \texttt{\textless action\textgreater} = \texttt{s} and visualizes home-vector fields,
- \texttt{runWarpingSIMDTestVis} uses \texttt{\textless action\textgreater} = \texttt{v} and stores data for visualization,
- \texttt{runWarpingSIMDTestAll} uses \texttt{\textless action\textgreater} = \texttt{a} and just prints errors and time measurements, and
- \texttt{runWarpingSIMDTestPar} uses \texttt{\textless action\textgreater} = \texttt{p} and prints parameters,
- \texttt{runPlaceRecognitionTest} uses \texttt{\textless action\textgreater} = \texttt{x} and visualizes place recognition results.

In these scripts, the command-line arguments passed to \texttt{warpingSIMDTest} are defined as shell variables or are varied in loops.

Some details on the angular relationships in the test program are explained in appendix \textit{C}.

The living image databases (Kreft, 2007) used in the test program are located in the data directory (specified in \texttt{DATAPATH}). For each group (e.g. day and night), a description file (suffix .db) contains in a single line specifying image width, image grid borders xmin, xmax, ymin, ymax, vertical resolution (radians), and horizon. Note that the horizontal resolution has approximately (but not exactly) the same value as the vertical resolu-
tion (but only the latter is used in the code). The data directory contains sub-directories for the different image databases. Each image filename contains the grid coordinates and the cutoff frequency of Butterworth lowpass (3rd order) that was used to generate these images from the original camera images. The day/night databases with same base name were collected at the same positions under different conditions of illumination. Note that the images are not calibrated. They were collected using a camera equipped with a hyperbolic mirror. The parameters of this mirror were used to determine the angular coordinates of the pixels. The image center in the camera image and the horizon were determined by visual inspection. Also note that images have mathematically negative azimuth angles. In the full database tar ball, also the grid databases (collected by David Fleer and Michael Horst, see Fleer and Möller, 2017) are included.

Acknowledgements

The present SIMD version of MinWarping was developed based on a sequential integer-based implementation developed by Martin Krzykawski. Andreas Stöckel improved the storage of the MinWarping template. Michael Horst and David Fleer suggested code changes which improve the portability to other compilers. Localization functionality is based on code provided by Michael Horst. Wolfram Schenck tested the code for the Intel C Compiler (icc).

A. Floor and Modulo Rules

In the following, \( m, n, k, \) and \( l \) are always natural numbers. Division produces a real-valued result. Integer division produces integer results and is expressed as floor operation applied to the result of an integer division (a fraction).

For a floor operation on a sum of a real and an integer number we have\(^{14}\)

\[
[x + n] = [x] + n, \quad x \in \mathbb{R}; n \in \mathbb{N}.
\] (22)

For a nested floor operation on a fraction (i.e. nested integer division) we have

\[
\left\lfloor \frac{x}{mn} \right\rfloor = \left\lfloor \frac{x}{mn} \right\rfloor, \quad x \in \mathbb{R}; m, n \in \mathbb{N}.
\] (23)

For a floor operation on a fraction (i.e. integer division) of a sum we have

\[
\left\lfloor \frac{x + m}{n} \right\rfloor = \left\lfloor \frac{x}{n} \right\rfloor, \quad x \in \mathbb{R}; m, n \in \mathbb{N}.
\] (24)

\(^{14}\)See [https://en.wikipedia.org/wiki/Floor_and_ceiling_functions](https://en.wikipedia.org/wiki/Floor_and_ceiling_functions)
The modulo operator can be defined as:\(^{15}\)

\[
m \mod n = m - n \left\lfloor \frac{m}{n} \right\rfloor.
\] (25)

The modulo of a sum is

\[(m + k) \mod n = (m \mod n + k \mod n) \mod n.\] (26)

For the following special case, a modulo operation applied to a sum can be simplified:

\[(mn + k) \mod n = k \mod n.\] (27)

Proof of (27):

\[
(mn + k) \mod n, \quad \text{apply (25)}
\]

\[
= mn + k - n \left\lfloor \frac{mn + k}{n} \right\rfloor,
\]

\[
= mn + k - n \left\lfloor \frac{m + \frac{k}{n}}{n} \right\rfloor, \quad \text{apply (22)}
\]

\[
= mn + k - n \left(m + \left\lfloor \frac{k}{n} \right\rfloor\right),
\]

\[
= k - n \left\lfloor \frac{k}{n} \right\rfloor, \quad \text{apply (28)}
\]

\[
= k \mod n
\]

For subsequent modulo operations we have

\[
\{m \mod (kn)\} \mod n = m \mod n,
\] (28)

with the special case

\[(m \mod n) \mod n = m \mod n.\] (29)

Proof of (28):

\[
\{m \mod (kn)\} \mod n, \quad \text{apply (25)}
\]

\[
= \left(m - kn \left\lfloor \frac{m}{kn} \right\rfloor\right) \mod n, \quad \text{apply (25)}
\]

\[
= m - kn \left\lfloor \frac{m}{kn} \right\rfloor - n \left\lfloor \frac{m - kn \left\lfloor \frac{m}{kn} \right\rfloor}{n} \right\rfloor, \quad \text{apply (22)}
\]

\[
= m - kn \left\lfloor \frac{m}{kn} \right\rfloor - n \left(\left\lfloor \frac{m}{n} \right\rfloor - k \left\lfloor \frac{m}{kn} \right\rfloor\right),
\]

\[
= m - n \left\lfloor \frac{m}{n} \right\rfloor, \quad \text{apply (25)}
\]

\[
= m \mod n
\]

\(^{15}\)See https://en.wikipedia.org/wiki/Modulo_operation
An inner modulo in a sum can be omitted

\[(m \mod n + k) \mod n = (m + k) \mod n\]  \hspace{1cm} (30)

Proof of (30):

\[(m \mod n + k) \mod n, \hspace{0.5cm} \text{apply (26)}\]
\[= ([m \mod n] \mod n + k \mod n) \mod n, \hspace{0.5cm} \text{apply (29)}\]
\[= (m \mod n + k \mod n) \mod n, \hspace{0.5cm} \text{apply (26)}\]
\[= (m + k) \mod n\]

The following integer division rule is required in the derivation in appendix B:

\[\left\lfloor \frac{(m + kn) \mod (ln)}{n} \right\rfloor = \left\{ \frac{m}{n} \right\} + k \mod l\]  \hspace{1cm} (31)

Proof of (31):

\[\left\lfloor \frac{(m + kn) \mod (ln)}{n} \right\rfloor, \hspace{0.5cm} \text{apply (25)}\]
\[= \left\lfloor \frac{(m + kn) - (ln) \left\lfloor \frac{m + kn}{ln} \right\rfloor}{n} \right\rfloor,\]
\[= \left\lfloor \frac{m}{n} + k - l \left\lfloor \frac{m}{l} \right\rfloor \right\rfloor, \hspace{0.5cm} \text{apply (22) and (24)}\]
\[= \left( \left\lfloor \frac{m}{n} \right\rfloor + k \right) - l \left\lfloor \frac{m}{l} \right\rfloor, \hspace{0.5cm} \text{apply (25)}\]
\[= \left\{ \frac{m}{n} \right\} + k \mod l\]

\section*{B. Derivation of Equation (8)}

We first resolve the unshuffle function:

\[\text{shuffle } \left( \left[ \text{unshuffle} (j') + d \right] \mod w \right), \hspace{0.5cm} \text{apply (7)}\]
\[= \text{shuffle } \left\{ \left[ \left( [j' \mod b] \Delta + \left\lfloor \frac{j'}{b} \right\rfloor \right) \mod w + d \right] \mod w \}\]

The argument to the shuffle function can be simplified:

\[\left[ \left( [j' \mod b] \Delta + \left\lfloor \frac{j'}{b} \right\rfloor \right) \mod w + d \right] \mod w, \hspace{0.5cm} \text{apply (30)}\]
\[= \left( [j' \mod b] \Delta + \left\lfloor \frac{j'}{b} \right\rfloor + d \right) \mod w\]
We now insert this term into the shuffle function (5):

\[
\text{shuffle}\left\{\left(\left[j' \mod b\right] \Delta + \left\lfloor \frac{j'}{b} \right\rfloor + d\right) \mod w\right\}, \quad \text{apply (5)}
\]

\[
= \left(\left(\left[j' \mod b\right] \Delta + \left\lfloor \frac{j'}{b} \right\rfloor + d\right) \mod \Delta\right) b
\]

The first term of the sum can be simplified as follows:

\[
\left(\left(\left[j' \mod b\right] \Delta + \left\lfloor \frac{j'}{b} \right\rfloor + d\right) \mod \Delta\right) b, \quad \text{apply (28), } w = n\Delta
\]

\[
= \left(\left[j' \mod b\right] \Delta + \left\lfloor \frac{j'}{b} \right\rfloor + d\right) \mod \Delta, \quad \text{apply (27)},
\]

\[
= \left(\left\lfloor \frac{j'}{b} \right\rfloor + d\right) \mod \Delta, \quad \text{apply (9)},
\]

\[
= ([i_{\text{blk}} + d] \mod \Delta) b
\]

The second term of the sum can be simplified as follows:

\[
\left(\left[j' \mod b\right] \Delta + \left\lfloor \frac{j'}{b} \right\rfloor + d\right) \mod w\]

\[
= \left(j' \mod b + \left\lfloor \frac{j'}{b} \right\rfloor + d\right) \mod n, \quad \text{apply (9)},
\]

\[
= (i_{\text{off}} + \left\lfloor \frac{i_{\text{blk}} + d}{\Delta} \right\rfloor) \mod n
\]

By adding the simplified first and second term, we have proven equation (8):

\[
\text{shuffle}\left([\text{unshuffle}(j') + d] \mod w\right)
\]

\[
= ([i_{\text{blk}} + d] \mod \Delta) b + \left(i_{\text{off}} + \left\lfloor \frac{i_{\text{blk}} + d}{\Delta} \right\rfloor\right) \mod n
\]

C. Angular Relationships in Test Program

In the following, the relationships between angles as they appear in the test program warpingSIMDTest.C are explained. Figure 6 visualizes the relationships between the home vector angle $\beta^*$ which is expressed relative to the grid $x$ axis (code: true:
Figure 6: Angular relationships at the current view location $C$.

betaTrue, estimate: betaMin), the movement direction $\alpha$, the orientation change $\psi$, and the orientation of the current view in the world $\Theta$, all measured in mathematically positive direction. We obtain:

$$\beta^* - \pi = \Theta + \alpha - \psi. \quad (32)$$

In the test program, warping works on images with mathematically negative azimuth angles, thus we have the warping parameters $\tilde{\alpha} = -\alpha$ and $\tilde{\psi} = -\psi$. The array rotAngle in the code (symbol $\tilde{\Theta}$) contains rotation angles in mathematically negative direction as well, so we get:

$$\beta^* - \pi = -\tilde{\Theta} - \tilde{\alpha} + \tilde{\psi}. \quad (33)$$

This equation is used for the computation of the true alpha angle (in internal warping coordinates, mathematically negative):

```c
double alphaTrue = psiTrue - betaTrue + M_PI - rotAngle[x][y];
```

and for the computation of the estimated home vector angle in world coordinates:

```c
double betaMin = -alphaMin + psiMin + M_PI - rotAngle[x][y];
```

Note that psiMin (estimate) and psiTrue (true orientation change) are mathematically negative angles as well, with the former obtained from warping, the latter from

```c
double psiTrue = rotAngle[x][y] - rotAngle[xss][yss];
```

where the rotation angles are mathematically negative. The true home vector direction in world coordinates (relative to grid $x$ axis) is computed from

```c
double xTrueHome = xss - x, yTrueHome = yss - y;
double betaTrue = atan2(yTrueHome, xTrueHome);
```
References


Changes

January 8, 2018: Updated section 7.