Friedrich-Alexander-Universität Erlangen-Nürnberg

**Lehrstuhl für Multimediakommunikation und Signalverarbeitung**

Prof. Dr.-Ing. André Kaup

Forschungspraktikum

# Umsetzung der Intra-Prädiktion für Fisheye-Sequenzen in C++

von Markus Tauber

Juni 2017

Betreuer: Andrea Eichenseer

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

in diesem Bericht wird das Potential und die Grenzen von Intraprädiktionsverfahren für fisheye (Fischaugen) Sequenzen untersucht. Die Forschungsgrundlage bildet dabei der HEVC (High Efficiency Video Coding).
Fischaugenkameras erlauben die Aufnahme eines größeren Blickfeldes und sind daher in der Überwachungstechnik (weniger Kameras für mehr Überblick) sowie bei Fahrassistenzsystemen (frühere Erkennung von Objekten) in der Automobilindustrie sehr beliebt. Eine effiziente Komprimierung solcher Videosequenzen ist mit gängigen Kompressionsverfahren wie im HEVC nur begrenzt möglich, da diese für natürliche, unverzerrte Videosequenzen optimiert sind. [EBSK15] Ziel dieser Arbeit war es, ausgehend von einer bereits fertig vorhanden und optimierten Variante der Intraprädiktion in Matlab eine äquivalente Umsetzung in C/C++ zu implementieren, welche kompatibel zur HEVC Referenzsoftware ist. Zum Testen wurde dabei eine Referenzdatenbank mit synthetischen sowie realen Fischaugenbildern verwendet. [EK16] In den folgenden Kapiteln wird zunächst ein Überblick über die verschiedenen Ansätze zur Umsetzung gegeben und anschließend detailliert auf die Schwierigkeiten bei der Konvertierung von Matlab zu C++ eingegangen. Am Ende wird dann noch der derzeitige Fortschritt des Projekts beschrieben.

# Kapitel 2

# Realisierung einer effizienten Fischaugen-Intraprädiktion

In diesem Kapitel geht es um die konkrete Umsetzung einer effizienten Methode für die Intraprädiktion von Fischaugensequenzen. Dabei werden verschiedene Ansätze gezeigt und deren Umsetzbarkeit bewertet. Abschließend wird noch ein kurzes Fazit über den letztlich gewählten Weg gegeben. Grundlage für alle Versuche bildet eine Matlab Implementierung, bei der durch die Einführung spezieller Modi für die fisheye-prediction ein geringerer Prädiktionsfehler und damit theoretisch auch eine geringere Datenrate möglich ist. Theoretisch deshalb, weil ein geringerer Prädiktionsfehler für ein Bild nicht zwangsläufig auch zu einer geringeren Datenrate im HEVC führt. Durch die Kodierung der zusätzlichen Modi entsteht ein konstanter Overhead, welcher unter Umständen die Vorteile bei der Prädiktion überwiegt. Aus diesem Grund ist es auch erforderlich, dass der in Matlab erweiterte Algorithmus direkt in den HEVC implementiert wird, damit man den Vorteil gegenüber den aktuellen Prädiktionsmechanismen überprüfen kann. Der HEVC berechnet die Intraprädiktion für jedes Bild jeweils blockweise. Der Block kann eine Größe von 4x4 bis zu 64x64 Pixel annehmen. In der Matlab Implementierung werden diese Blockgrößen ebenfalls unterstützt.

## 2.1 Ubersicht über verwendete Software und Tools

Zunächst wird in diesem Abschnitt eine Übersicht über die verwendete Software und andere Tools gegeben. Für die Ausführung von Matlab Skripten wurde Matlab in der Version 2016b verwendet. [mat17] Bei der Umsetzung von Matlab zu C++ wurde OpenCV 3.2.0 verwendet. [ocv17] Die Fischaugenprädiktion in Matlab basiert auf der HEVC Referenzsoftware in der Version 16.12. [HM17] Um OpenCV auf dem Rechner zu installieren habe ich ein Installationsskript benutzt, welches im Anhang eingesehen werden kann (Listing A.1). Dabei ist anzumerken, dass die beiden Archive, welche dabei letztlich im Verlauf der Installation angelegt und entpackt werden in ein Standard und Contrib Archiv aufgeteilt sind (Zeile 4 und 5, Listing A.1). Das Contrib Archiv umfasst dabei Bibliotheken, welche nicht ohne weiteres benutzt werden können, da damit bestimmte Patente verletzt werden. Bei kommerziellem Einsatz müsste man hier entsprechend Acht geben.

## 2.2 Direkte Implementierung im HEVC

Die erste und naheliegende Idee war natürlich direkt im HEVC die neuen Modi auf geeignete Weise einzubauen. Dazu muss erstmal, nach Vertrautmachen mit der Referenzsoftware, die richtige Stelle im Source Code ausfindig gemacht werden. Die zu editierende Datei ist die `TComPrediction.cpp` und darin die Funktionen `predIntraAng` beziehungsweise `xPredIntraAng`.

Dieser Ansatz war nicht zielführend, da die Komplexität des HEVCs schnelle Änderungen in kurzer Zeit nicht zulässt und ein tieferes Verständnis des Codecs erfordert, was im Rahmen des Praktikums nicht möglich war. Auch eine angepasste Version des HEVCs, bei der die Ausführung von OpenCV Befehlen möglich ist wurde nicht verwendet. Die Idee war den jeweils aktuell zu prädizierenden Block auf dem man sich im Bild befindet erstmal in eine OpenCV Matrix zu kopieren, darauf dann die modifizierte Intraprädiktion zu berechnen und anschließend diese Matrix wieder in eine HEVC Matrix

umzuwandeln. Da ein Block samt seiner zugehörigen Referenzpixel nicht einfach direkt linear in Arrays abgelegt ist, sondern die entsprechenden Speicherbereiche durch Pointer adressiert werden, welche nicht immer auf den Anfang eines Blockes zeigen, lässt sich dies nicht ohne weiteres umsetzen.
Letztlich wurde dieser Ansatz auch deswegen nicht verwendet, weil man die Korrektheit der neuen Modi nur schwer überprüfen und debuggen kann.

## 2.3 Umsetzung in OpenCV

Das bereits im letzten Abschnitt erwähnte freie Programmbibliothek OpenCV stellt Algorithmen zur Bildverarbeitung zur Verfügung. Für die Umsetzung von Matlab zu C++ Code ist es erforderlich, dass man Schritt für Schritt oder Zeile für Zeile überprüfen kann, dass Variablen in Matlab und C++ identisch sind. Dazu wurden die entsprechenden Funktionen sowohl in Matlab als auch in C++ mit denselben Werten aufgerufen. Der Übersichtlichkeit halber haben alle Variablen und Funktionen in C++ die gleichen Namen und es wurde auch darauf geachtet, dass das Layout möglichst identisch sind. Zunächst wurde mit zufälligen Werten und Matrizen getestet. Um die Ausführung an bestimmten stellen in Matlab anzuhalten konnte man ein `pause;` einfügen in C++ ein `return;` oder `sleep(x);` wobei x für die Zeit in Sekunden, die der Prozess anhalten soll steht. Um Variablen auszugeben konnte man in C++ std::cout oder std::cerr verwenden, letzteres hat den Vorteil, dass dadurch Ausgaben im Matlab-Terminal rot eingefärbt werden, wodurch sich die Ausgaben von beiden Implementierungen leicht unterscheiden lassen. In Matlab können Variablen oder Operationen ausgegeben werden indem man entweder das Semikolon am Ende weglässt oder mit `disp(variable)` oder auch mit `fprintf()`. Ein Vergleich beider print-debugging Ansätze ist in den Code-Listings Listing 2.1 und Listing 2.2 ersichtlich. Gleichzeitig kann man sehen wie ein und dieselbe Matrix `predictedBlock` mit gleichen Dimensionen in Matlab und OpenCV deklariert werden.

**Listing 2.1: Print-Debugging in Matlab**

```
predictedBlock = zeros(blocksize,blocksize)
disp(predictedBlock);
fprintf("%f", predictedBlock);
```

**Listing 2.2: print-Debugging in C++ mit OpenCV**

```
using namespace std;
cv::Mat predictedBlock = cv::Mat::zeros(blocksize, blocksize,
    CV_64FC1);
cerr << predictedBlock << endl;
```

Die Unterschiede beim Deklarieren von verschiedenen Matrizen und deren Zugriff sind in den Listings Listing 2.3 und Listing 2.4 zu sehen.

**Listing 2.3: Matrixdeklaration und Zugfriff in Matlab**

```
Matrix_1D = [1 2 3 4 5 6 7 8 9];
Matrix_2D = [1 2 3; 4 5 6; 7 8 9];
// returns first element
Matrix_1D(1);
// returns 8, third row and second column
Matrix_2D(3,2);
// returns the last element of the matrix
Matrix_2D(end);
```

Listing 2.4: Matrixdeklaration und Zugfriff mit OpenCV

```
1 cv::Mat Matrix_1D = (cv::Mat_<double>(1,9) << 1, 2, 3, 4 ,5 ,6 ,7
      ,8 ,9);
2 cv::Mat Matrix_2D = (cv::Mat_<double>(3,3) << 1, 2, 3, 4 ,5 ,6 ,7
      ,8 ,9);
3 // returns first element
4 Matrix_1D.at<double>(0,0);
5 // returns 8, which is located in the third row of column two
6 Matrix_2D.at<double>(2,1);
7 // returns the last element of the matrixs
8 Matrix_2D.at<double>(Matrix2D.rows-1,Matrix2D.cols-1);
```

Einer der Hauptunterschiede ist, dass Matlab beim Matrixindex bei 1 anfängt zu zählen und OpenCV bei 0. Das führt dazu, dass man bei jedem Index aus Matlab bei OpenCV eine 1 abziehen muss, um auf den richtigen Wert zuzugreifen.
Der allgemeine Konstruktor einer OpenCV Matrix sieht folgendermaßen aus `cv::Mat A(rows, cols, DATATYPE)`.
Dabei steht rows für die Anzahl und Zeilen und cols für die Anzahl an Spalten in der Matrix. Der Datentyp ist hier immer `CV_64FC1`, womit Bildpunkte als Gleitkommazahlen mit doppelter Präzision bei einem einzigen Kanal (monochromatisch) gespeichert werden. Es werden deswegen ausschließlich Doublewerte verwendet, da dieser Datentyp auch in Matlab standardmäßig genutzt wird, wenn man neue Variablen anlegt.

## 2.4 Matlab Interfaces

In diesem Abschnitt wird die Realisierung von Matlab Inferfaces erklärt. Solche Interfaces sind nötig, damit man aus einer Matlabumgebung direkt auf C/C++ Funktionen

zugreifen kann. Das Interface muss dafür Aufrufparameter an die entsprechende C/C++ Funktion weitergeben und in die entsprechenden C/C++ Datentypen umwandeln. Die Rückgabewerte der C/C++ Funktion müssen dann wiederum in Matlabstrukturen umgewandelt werden. Damit man das richtig versteht, schaut man sich dies am besten anhand eines Beispiels in Listing 2.5 an.

**Listing 2.5: Matrixdeklaration und Zugfriff mit OpenCV**

```
1 #include <mex.h>
2 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray
      *prhs[]) {
3   // input variables
4   // image
5   mxArray *image = mxDuplicateArray(prhs[0]);
6   double *imagePtr = mxGetPr(image);
7   const mwSize *image_dims = mxGetDimensions(image);
8   cv::Mat imageMat(image_dims[0], image_dims[1], CV_64FC1, imagePtr
     );
9   imageMat = imageMat.clone();
10  cv::transpose(imageMat, imageMat);
11  // blocksize
12  mxArray *blocksize = mxDuplicateArray(prhs[1]);
13  double *blocksizePtr = mxGetPr(blocksize);
14  int blocksizeInt = (int) blocksizePtr[0];
15  // showTest
16  mxArray *showTest = mxDuplicateArray(prhs[6]);
```

```
17  double *showTestPtr = mxGetPr(showTest);
18  bool showTestBool = showTestPtr[0] != 0 ? true : false;
19  // run c-function with
20  cv::Mat predictedImageFisheyeMat = cv::Mat::zeros(image_dims[0],
     image_dims[1], CV_64FC1);
21  int intraModeInt = runPred(imageMat, blocksize, showTest,
     predictedImageFisheyeMat);
22  // return variables
23  // intraMode
24  mxArray *intraMode = plhs[1] = mxCreateDoubleMatrix(1, 1, mxREAL)
     ;
25  double *intraModePtr = mxGetPr(intraMode);
26  intraModePtr[0] = (double) intraModeInt;
27  // predictedImageFisheye
28  mxArray *predictedImageFisheye = plhs[1] = mxCreateDoubleMatrix(
     HEIGHT, WIDTH, mxREAL);
29  double *predictedImageFisheyePtr = mxGetPr(predictedImageFisheye)
     ;
30  for (int y = 0; y < HEIGHT; y++) {
31    for (int x = 0; x < WIDTH; x++) {
32      predictedImageFisheyePtr[y*WIDTH+x] = (double)
     predictedImageFisheyeMat.at<double>(y,x);
33    }
34  }
```

```
35 }
```

In diesem Interface werden drei Eingabevariablen und eine Ausgabevariable behandelt. Bei der ersten Eingabevariable handelt es sich um das zu verarbeitende Eingabebild als Matrix gespeichert. Es muss zunächst immer eine `mxArray` Variable erstellt und anschließend ein Doublepointer auf diese Variable mit `mxGetPr(mxArray *)` angelegt werden (siehe Zeile 5 und 6). Im Falle einer `cv::Mat` Variable kann dieser Pointer dann direkt als Parameter dem Konstruktor übergeben werden, man muss dann aber auch noch die `clone()` Funktion aufrufen, damit die Werte auch wirklich kopiert werden (siehe Zeile 7 und 8). Die in Zeile 9 durchgeführte Operation ist notwendig, dass die Matrixinhalte an der richtigen Stelle befinden. Grund dafür ist, dass die Matlabmatrix Spalte für Spalte im Speicher liegt und OpenCV diese wiederum Zeile für Zeile abspeichert. Durch `cv::transpose(cv::Mat in, cv::Mat dst)` wird die gesamte Matrix transponiert, so dass die Matlab und OpenCV Matrix wieder identisch sind. Natürlich müssen OpenCV Matrizen dann ebenfalls in Matlab wieder transponiert werden. Also in diesem Fall predictedImageFisheyeMat. Alle Aufrufvariablen müssen zunächst als Doublepointer referenziert werden und anschließend dann in die entsprechenden C++ Primitiven gecastet werden. Beispiel für einen Integerwert findet man in den Zeilen 12 bis 14 und in den Zeilen 16 bis 18 ein Beispiel für einen Boolean. Der boolsche Wert wird dadurch definiert, dass alles ungleich einer 0 als `true` interpretiert wird.
Insgesamt war es nötig zwei Matlabinterfaces zu implementieren. Für die Matlab Funktion `ae_intraPredictionFisheye_v1(...)` in der `ae_intraPredictionFisheye_v1.m` ist das Interface in `mex_ae_intraPredictionFisheye_v1.cpp` implementiert worden. Die Funktion `ae_predictBlockFisheye_v6(...)` aus der `ae_predictBlockFisheye_v6.m` ist im Interface `mex_ae_predictBlockFisheye_v6.cpp` implementiert.
Zum Kompilieren solcher Interfaces braucht man einen speziellen angepassten Compiler von Matlab. Im einfachsten Fall lassen sich solche Interfaces für Matlab folgendermaßen kompilieren:

`mex source1.cpp source2.cpp ... sourceXY.cpp` Heraus kommt dann ein File das mit `mexa.64` endet (`source1.mexa64`). Diese mexa64 files müssen sich im working directory von Matlab befinden, damit man die C++ Funktion anschließend direkt innerhalb eines Matlabskripts aufrufen kann. Da ebenfalls OpenCV verwendet wurde, mussten die entsprechenden Bibliotheken dazugelinkt werden, so dass der vollständige Aufruf zum Kompilieren eines Interfaces folgendermaßen aussieht:

```
mex mex_ae_predictBlockFisheye_v6.cpp ae_predictBlockFisheye_v6.cpp
helper.cpp -I/fisheye_prediciton/lib/opencv3/include
-L/fisheye_prediciton/lib/opencv3/lib
-lopencv_core -lopencv_highgui -lopencv_imgproc
```

# Kapitel 3

# Probleme bei der Portierung von Matlab zu OpenCV

Es gibt diverse einfache aber auch komplizierter Probleme, die beim Portieren von Matlab zu OpenCV auftreten und auf die nun in den folgenden Abschnitten ausführlich eingegangen wird. Grundsätzliches Problem bei der Portierung war, dass Matlab eine wesentlich umfangreichere Funktionsbibliothek hat und man fehlende Funktionen in OpenCV manuell nachbauen muss. Außerdem lässt die Matlabsyntax zuweilen eine wesentlich kompaktere Schreibweise zu, so dass die C++ Implementierung in der Regel um einige Zeilen umfangreicher ausfällt. Besonders auffällig ist das bei der Umsetzung des Meshgrids gewesen, mit dem die verzerrten Koordinaten erzeugt werden. Siehe dazu vor allem Abschnitt 3.3.

## 3.1 OpenCV find and Matlab find function

Die erste Funktion welche sich in Matlab und OpenCV stark unterscheiden ist die `find()` Funktion. Da die Unterschiede in beiden Fällen zu gravierend waren wurde eine eigene `find()` Funktion in OpenCV implementiert.

Listing 3.1: Beispiel find in Matlab

```
1 lessThan = find(rot_y_row_undist <= pixel_coord);
2 greaterThan = find(rot_y_row_undist >= pixel_coord);
3 lessThan = lessThan(1);
4 greaterThan = greaterThan(end);
```

Listing 3.2: Beispiel find in OpenCV

```
1 double lessThan[2];
2 double greaterThan[2];
3 ...
4 find_min_or_max(rot_y_row_undist, rot_y_row_undist.rows,
      rot_y_row_undist.cols, true, pixel_coord, lessThan, true);
5 find_min_or_max(rot_y_row_undist, rot_y_row_undist.rows,
      rot_y_row_undist.cols, false, pixel_coord, greaterThan, false);
6 ...
7 void find_min_or_max(cv::Mat &m, int rows, int cols, bool min,
      double val, double* result, bool first) {
8   result[0] = -1;
9   result[1] = -1;
10  if (min) {
11    for (int i = 0; i < cols; i++) {
12      for (int j = 0; j < rows; j++) {
13        if (m.at<double>(j,i) <= val) {
```

```
14            result[0] = j;
15            result[1] = i;
16            if (first)
17              return;
18          }
19        }
20      }
21    }
22    if (!min) {
23      for (int i = 0; i < cols; i++) {
24        for (int j = 0; j < rows; j++) {
25          if (m.at<double>(j,i) >= val) {
26            result[0] = j;
27            result[1] = i;
28            if (first)
29              return;
30          }
31        }
32      }
33    }
34 }
```

Was einem bei den beiden Listings Listing 3.1 und Listing 3.2 sofort auffällt ist, dass die OpenCV Version um einiges umfangreicher ist, das liegt daran, dass die `find()` Funktion manuell nachgebaut wurde. Die Matlab `find()` Funktion gibt ohne weitere

Aufrufparameter einen Vektor zurück, in welchem die Positionen (Indizes) der Elemente der übergebenen Matrix, welche ungleich null sind, gespeichert sind. Allerdings kann man diesen Selektionsprozess auch manipulieren indem man zum Beispiel `>= XY` oder `<= XY` hinter die Matrix schreibt, dann wird nur noch nach Elementen in der Matrix gesucht, welche der `find()` Funktion übergebenen "Maske" entsprechen (Zeile 1 und 2, Listing 3.1). Da jedoch in den Zeilen 3 und 4 lediglich ein einziges Element aus diesem Vektor extrahiert wird, wurde eine genau für diesen Fall angepasste `find()` Funktion in OpenCV implementiert. Die Funktion gibt nur den ersten oder letzten Index zurück und kann nur nach Minimas oder Maximas suchen. Beliebige Suchmasken werden nicht unterstützt, sondern müssten ebenfalls nachgebaut werden. Es mag durchaus möglich sein, dass auch die OpenCV eigene `find()` Funktion mit Suchmasken aufgerufen werden kann, im Rahmen des Praktikums wurde diese Lösung allerdings nicht gefunden. Damit keine Verwechslung auftritt heißt die selbstgebaute `find()` Funktion hier auch `find_min_or_max(...)` (Zeile 7, Listing 3.2). Als Aufrufparameter hat man die zu durchsuchende Matrix `m` (als Referenz), die Anzahl der Zeilen als `rows`, die Spalten als `cols`. Der Boolean `min` entscheidet, ob nach dem Mininum (`min=true`) oder Maximum (`min=false`) gesucht werden soll. Der Double `val` ist der mit den Matrix Elementen zu vergleichende Wert, der Doublepointer `result` zeigt auf ein Double-Array in dem das Ergebnis (der Index) gespeichert werden. Der Boolean `first` entscheidet ob man den Index des absoluten (`first=false`) oder des ersten (`first=true`) Minimums/Maximums sucht. In der Funktion selbst wird zunächst der Wert der boolschen Variable `min` überprüft (Zeile 10, Listing 3.2) und anschließend die übergebene Matrix `m` Element für Element iteriert, was durch die zwei For-Schleifen, deren Abbruchbedingung durch `rows` und `cols` eingestellt wird, realisiert wird ()Zeilen 11 und 12 Listing 3.2). In jeder Iteration wird das derzeitige Element mit dem übergebendem `val` verglichen, ist der Wert kleiner oder gleich (`min=true`), wird der Spalten und Zeilenindex in den Zeilen 14 und 15 (??) ins `result` Array geschrieben. Sofern die boolsche Variable `first` nicht `true` ist, wird bis zum Ende der Matrix weiter iteriert (Zeile 16, 17 in Listing 3.2). Die komplette Logik ist für den alternativen Fall (`min=false`) noch einmal in den Zeilen

22 bis 33 (Listing 3.2) implementiert.

## 3.2 Modulo in C/C++ und Matlab)

In diesem Abschnitt möchte ich einen kleinen Einschub über die Modulofunktion in C++ und Matlab machen, denn leider sind beide unterschiedlich definiert.
Der Standardmodulooperator (%) ist in C/C++ für negative Werte nicht definiert, weswegen man eine kleine Hilfsfunktion schreiben muss. Diese kann in Listing 3.3 eingesehen werden.

**Listing 3.3: Modulo für negative Werte in C/C++**

```
1 int ifract_tmp =  mod(((x+1) * intraPredAngleTemp), 32);
2 ...
3 int mod(int a, int b) {
4   return (a % b + b) % b;
5 }
```

In Zeile 1 von Listing 3.3 sieht man Aufruf der `mod` Funktion in der Datei `ae_predictBlockFisheye_v6.cpp`. Die Variable `intrPredAngleTemp` kann einen negativen Wert haben. Der Funktion müssen zwei Integerwerte (`a` und `b`) übergeben werden. Die Funktion stellt sicher, dass auch bei Modulo mit negativen Werten die richtigen Ergebnisse herauskommen. Zum Beispiel: `mod(-23,-9) => -5`. Wenn man den Modulooperator häufig verwenden muss, empfiehlt es sich immer diese Hilfsfunktion zu definieren.

## 3.3 Meshgrid

In diesem Abschnitt geht es um die Unterschiede bei der Implementierung des Meshghrids. Das Meshgrid ist in der Matlab Datei `ae_intraPredictionFisheye_v1.m`

implementiert und man findet dafür einen einzigen Befehl in Matlab:

```
[x, y] = meshgrid(1:width+2*blocksize,1:height+2*blocksize);
```

Leider besitzt OpenCV keine eigene Meshgrid Funktion, weswegen diese wiederum manuell nachgebaut werden musste. Die Implementierung dieser Funktion sieht man in Listing 3.4.

**Listing 3.4: Meshgrind Funktion in OpenCV**

```
1 // cartesian coordinates x and y
2 cv::Mat x;
3 cv::Mat y; //= cv::Mat::zeros(height, width, CV_64FC1);
4 meshgrid2D(cv::Range(1, width+2*blocksize+1), cv::Range(1, height
      +2*blocksize+1), x, y);
5 ...
6 void meshgrid(const cv::Mat &xgv, const cv::Mat &ygv, cv::Mat &X,
      cv::Mat &Y){
7 cv::repeat(xgv.reshape(1,1), ygv.total(), 1, X);
8 cv::repeat(ygv.reshape(1,1).t(), 1, xgv.total(), Y);
9 }
10
11 void meshgrid2D(const cv::Range &xgv, const cv::Range &ygv, cv::Mat
       &X, cv::Mat &Y) {
12 std::vector<double> t_x, t_y;
13 for (int i = xgv.start; i <= xgv.end; i++) t_x.push_back((double)i)
      ;
14 for (int i = ygv.start; i <= ygv.end; i++) t_y.push_back((double)i)
```

```
    ;
15 meshgrid(cv::Mat(t_x), cv::Mat(t_y), X, Y);
16 }
```

Für die Umsetzung waren gleich zwei Hilfsfunktionen nötig. Aus der Main heraus ruft man die Funktion `meshgrid2D(...)` auf (hier wurde lediglich eine Funktion für 2D grids gebaut, Matlab erlaubt auch 3D grids), die wiederum die `meshgrid(...)` Hilfsfunktion aufruft (Zeile 4 und 15, Listing 3.4). Auch hier werden `cv::Mat` Matrizen immer als Referenz übergeben, damit nicht unnötige Latenz durch Kopiervorgänge entsteht.

## 3.4 Pol2Cart und Cart2Pol

In diesem Abschnitt möchte ich auf zwei weitere Funktionen eingehen, die es so in OpenCV nicht gibt und die man selbst implementieren muss.
Dabei handelt es sich um die Umwandlung von kartesischen zu polaren Koordinaten und umgekehrt. `Cart2pol` wandelt kartesische in polare Koordinaten um, die Implementierung für OpenCV ist in Listing 3.5 zu sehen.

**Listing 3.5: Cart2pol Funktion in OpenCV**

```
1 void cart2pol(cv::Mat &xc, cv::Mat &yc, cv::Mat &phi, cv::Mat &r) {
2   // phi
3   for (int i = 0; i < phi.cols; i++) {
4     for (int j = 0; j < phi.rows; j++) {
5       phi.at<double>(j,i) = atan2(yc.at<double>(j,i), xc.at<double
    >(j,i));
6     }
```

```
7   }
8   // r
9   for (int i = 0; i < r.cols; i++) {
10    for (int j = 0; j < r.rows; j++) {
11      r.at<double>(j,i) = hypot(xc.at<double>(j,i), yc.at<double>(j
      ,i));
12    }
13  }
14 }
```

Bei der Implementierung wurde der vorliegender Source Code von Matlab konvertiert. Dazu schaut man sich am besten die Doku zur selben Funktion im Anhang an Listing A.2.
Die andere Funktion wandelt polare wieder in kartesische Koordinaten zurück und heißt pol2cart und ist in Listing 3.6 einzusehen.

**Listing 3.6: Pol2cart Funktion in OpenCV**

```
1 void pol2cart(cv::Mat &phi, cv::Mat &r, cv::Mat &xc, cv::Mat &yc) {
2   assert(xc.cols == yc.cols && xc.rows == yc.rows);
3   for (int i = 0; i < xc.cols; i++) {
4     for (int j = 0; j < xc.rows; j++) {
5       xc.at<double>(j,i) = r.at<double>(j,i)*cos(phi.at<double>(j,i
      ));
6       yc.at<double>(j,i) = r.at<double>(j,i)*sin(phi.at<double>(j,i
      ));
7     }
```

```
8   }
9 }
```

Was sofort auffällt ist, dass sich bei der Rückwandlung in kartesische Koordinaten eine Schleife einsparen lässt, weil durch die `assert` Anweisung in Zeile zwei (Listing 3.6) sichergestellt ist, dass jeweils über alle Elemente der Matrix iteriert wird. Wahrscheinlich wäre diese Optimierung auch in der `cart2pol(...)` Funktion (Listing 3.5) möglich, müsste jedoch getestet werden. Momentan wird diese mit zwei Schleifen realisiert. Natürlich müssen auch hier, im Gegensatz zu Matlab, die eigentlichen Rückgabewerte als `cv::Mat` Referenzen übergeben werden, da es in C/C++ nur möglich ist jeweils einen Wert pro Funktion zurückzugeben.

## 3.5 Komplexwertige Matrizen

Komplexwertige Matrizen zu berechnen stellt in Matlab überhaupt kein Problem dar. Ganz anders sieht es in OpenCV und C/C++ allgemein aus. Sofern ein Berechnung zu einem komplexwertigen Ergebnis führt, kann diese Berechnung nicht ohne weiteres in C/C++ ausgeführt werden. Es braucht hierfür einen eigenen Datentyp, doch dieser lässt sich nicht ohne weiteres in eine OpenCV Matrix einkopieren. Außerdem fehlen dann doch einige typische trigonometrische Funktionen, oder sind erst in neuere C++ Standards verfügbar, so dass man keine andere Wahl hat als selber für die Aufteilung von Ergebnissen in Imaginär- und Realteil zu sorgen. Im Klartext bedeutet dies, dass man zwei `cv::Mats` braucht, eine speichert den Realteil und die andere den Imaginärteil. Sobald man diese Aufteilung hat muss man allgemein ganz anders rechnen und muss auch manche trigonometrische Funktionen selber nachbauen. Im Folgenden wird nun auf die speziellen Herausforderungen im Umgang mit komplexwertigen Matrizen aus der Sicht von OpenCV eingegangen.

Es mussten genau genommen zwei trigonometrische Funktionen umgesetzt werden. Die erste war der komplexe Arkussinus. Die `asin()` Funktion ist nur für einen bestimm-

ten Definitionsbereich reelwertig und ansonsten komplexwertig. In C++ ist die `asin()` Funktion für komplexe Zahlen erst ab C++11 verfügbar. Auf Wikipedia ([asi17]) findet man zwar eine mathematische Formel für den komplexen Arkussinus, aber diese funktioniert erst, wenn man bereits eine Aufsplittung in Real- und Imaginärteil hat. Daher musste auf die C (nicht C++) Version des komplexen Arkussinus zurückgegriffen werden. Um das im Detail zu erörtern wird das Listing ?? genauer betrachtet.

**Listing 3.7: Asin mit komplexen Zahlen**

```
1 #include <complex.h>
2 #include <complex>
3 void complex_asin(complex<double> &value, double val) {
4 double _Complex arg = val;
5 double _Complex asin_temp = casin(arg);
6 value.real(creal(asin_temp));
7 value.imag(cimag(asin_temp));
8 }
```

Wie in den ersten beiden Zeilen (Listing 3.7) zu sehen ist, muss sowohl die `complex.h` (C) als auch die `complex` (C++) eingebunden werden. Die `double _Complex` Struktur stammt dann aus der `complex.h` und der Datentyp `complex<double>` aus der `complex`. Die C Struktur wird nur für den Aufruf der `casin(arg)` in Zeile fünf (Listing 3.7) benötigt. Anschließend kann man auf `value.real` und `value.imag` zugreifen, wobei man zum Extrahieren des Real- und Imaginärteils wiederum eine C typische Funktion verwenden muss (`creal()` unf `cimag()`). In den Zeilen 6 und 7 (Listing 3.7) wird quasi C und C++ gleichzeitig verwendet.
Anschließend kann man mit C++ aber auch auf komplexen Zahlen ohne großen Aufwand arithmetische Operationen durchführen. Der komplexe Tangens muss aber wieder implementiert werden.

**Listing 3.8: Komplexe Tangens Funktion in C++**

```
1 void complex_tan(complex<double> &value) {
2 double x = value.real();
3 double y = value.imag();
4 value.real(sin(2*x)/(cos(2*x)+cosh(2*y)));
5 value.imag(sinh(2*y)/(cos(2*x)+cosh(2*y)));
6 }
```

In Listing 3.8 sieht man eine komplexe Tangensfunktion. Zunächst werden die Real- und Imaginärteile in zwei Doubles zwischengespeichert (Zeile 2,3 Listing 3.8) und anschließend beide Komponenten seperat in Form von `sin/sinh` und `cos/cosh)` Funktionen ausgerechnet.

Der ganze Ablauf ist in Listing 3.9 zu sehen. Der Source Code befindet sich in der `ae_intraPredictionFisheye_v1.cpp` und wird zur Berechnung von `r_col`, `r_row` und `r_blk` gebraucht. Außerdem wird dort auch die Konvertierung zurück in eine reelwertige Matrix gezeigt.

**Listing 3.9: Komplexwertige Matrizen**

```
1 // r_row_undist
2 for (int i = 0; i <  r_row_undist_real.cols; i++) {
3   for (int j = 0; j <  r_row_undist_real.rows; j++) {
4     complex<double> temp(0,0);
5     complex_asin(temp, r_row.at<double>(j,i) / 2 / flp);
6     temp *= 2;
7     complex_tan(temp);
8     temp *= flp;
```

```
 9      r_row_undist_real.at<double>(j,i) = temp.real();
10      r_row_undist_imag.at<double>(j,i) = temp.imag();
11    }
12 }
13 // conversion back to real values single cv::Mat
14 cv::Mat r_row_undist = r_row_undist_real.clone();
15 for (int i = 0; i < r_row_undist_real.cols; i++) {
16   for (int j = 0; j < r_row_undist_real.rows; j++) {
17     if (r_row_undist_real.at<double>(j,i) < 0 || r_row_undist_imag.
       at<double>(j,i) < 0) {
18       r_row_undist.at<double>(j,i) = (r_180_undist + (r_180_undist-
       sqrt(r_row_undist_real.at<double>(j,i)*r_row_undist_real.at<
       double>(j,i)+r_row_undist_imag.at<double>(j,i)*r_row_undist_imag
       .at<double>(j,i))));
19     }
20   }
21 }
```

# Kapitel 4

# Ausblick

Nachdem nun ausreichend ausführlich auf die Unterschiede und die damit verbundenen Schwierigkeiten bei der Konvertierung von Matlab zu C++ eingegangen wurde, soll zuletzt noch ein Ausblick gegeben werden, wie von dem derzeitigen Stand aus weitergemacht werden könnte. In der aktuellen Version funktioniert die Prädiktion komplett identisch zur Matlabimplementierung. Das heißt für beliebige Eingabebilder wird immer derselbe PSNR erreicht. Die C++ Implementierung hat dabei ein wesentlich geringere Latenz, was auch zu erwarten war, dennoch benötigt die Prädiktion eines einzelnen Bildes circa 30 Sekunden. In Matlab vergehen dabei aber bis zu 90 Sekunden. Sicherlich kann man die OpenCV Implementierung noch an der ein oder anderen Stelle, womöglich auch gravierend, optimieren. Absolute Priorität hat zunächst die Implementierung in den HEVC, um herauszufinden, ob das ganze wirklich einen Kompressionsvorteil ergibt. Dazu müsste man mindestens zwei Dateien im HEVC editieren, wenn nicht sogar noch mehr. Eine genaue Inventur der zu ändernden Abschnitte ist während des Praktikums nicht erfolgt und ist auch nicht trivial.
Wenn man nach der Implementierung herausfindet, dass sich die Fischaugen spezifischen Anpassungen durchaus lohnen, würde als nächstes die Optimierung der aktuellen OpenCV Implementierung im Fokus stehen. Dies ist insofern wichtig, da die Performance und damit auch die Umsetzbarkeit vor allem für den Decoder von Bedeutung sind. Wenn der Decoder nicht mehr in Echtzeit dekodieren kann funktionieren interaktive

Anwendungen oder Liveübertragungen nicht mehr.

# Anhang A

# Anhang Kapitel

## A.1 Anhang Abschnitt

Listing A.1: OpenCV Installationsskript

```
1 #!/bin/bash
2
3 # constants
4 OPENCV_ARCHIVE="required_dependencies/opencv-3.2.0.zip"
5 OPENCV_CONTRIB_ARCHIVE="required_dependencies/opencv_contrib-3.2.0.
      zip"
6 INSTALL_PATH="opencv3"
7 TMP_PATH="build"
8
9 # functions
10 function error_exit
11 {
```

```
12      echo "${1:-"Unknown Error"}" 1>&2
13      exit 1
14 }
15
16
17
18 ##### MAIN #####
19
20 # are we in right path? => .
21 scriptName=${0##*/}
22 if [ ! -f "./$scriptName" ] ; then
23   error_exit "$LINENO: '$scriptName' is called from wrong path.
       Change in console to cwd '.'!"
24 fi
25
26 # do required files exist?
27 if [ ! -f $OPENCV_ARCHIVE ] ; then
28   error_exit "$LINENO: '$OPENCV_ARCHIVE' does not exist."
29 fi
30 if [ ! -f $OPENCV_CONTRIB_ARCHIVE ] ; then
31   error_exit "$LINENO: '$OPENCV_CONTRIB_ARCHIVE' does not exist."
32 fi
33
34 # create required folders
```

```
35 echo "create required folders"
36 rm -rf $INSTALL_PATH
37 mkdir $INSTALL_PATH
38 if [ $? != 0 ] ; then
39   error_exit "$LINENO: 'mkdir $INSTALL_PATH' failed"
40 fi
41 rm -rf $TMP_PATH
42 mkdir $TMP_PATH
43 if [ $? != 0 ] ; then
44   error_exit "$LINENO: 'mkdir $TMP_PATH' failed"
45 fi
46
47 # extract archives
48 echo "extract archives folders"
49 unzip $OPENCV_ARCHIVE -d $TMP_PATH 1>/dev/null
50 if [ $? != 0 ] ; then
51   error_exit "$LINENO: extracting '$OPENCV_ARCHIVE' failed"
52 fi
53 unzip $OPENCV_CONTRIB_ARCHIVE -d $TMP_PATH 1>/dev/null
54 if [ $? != 0 ] ; then
55   error_exit "$LINENO: extracting '$OPENCV_CONTRIB_ARCHIVE' failed"
56 fi
57
58 # compile OpenCV
```

```
59 echo "compile OpenCV"
60 release_dir=$TMP_PATH"/opencv-3.2.0/release"
61 mkdir $release_dir
62 if [ $? != 0 ] ; then
63   error_exit "$LINENO: 'mkdir release_dir' failed"
64 fi
65 cd $release_dir
66 cmake_args="-D PYTHON_INCLUDE_DIR=$(python -c "from distutils.
      sysconfig import get_python_inc; print(get_python_inc())") -D
      PYTHON_LIBRARY=$(python -c "import distutils.sysconfig as
      sysconfig; print(sysconfig.get_config_var('LIBDIR'))") -D
      BUILD_NEW_PYTHON_SUPPORT=ON -D BUILD_PYTHON_SUPPORT=ON -D
      WITH_OPENMP=ON -D CMAKE_BUILD_TYPE=RELEASE -D
      CMAKE_INSTALL_PREFIX=../../../$INSTALL_PATH -D
      OPENCV_EXTRA_MODULES_PATH=../../opencv_contrib-3.2.0/modules .."
67 #~ cmake_args="-D CMAKE_BUILD_TYPE=RELEASE -DBUILD_SHARED_LIBS=OFF
      -D CMAKE_INSTALL_PREFIX=../../../$INSTALL_PATH -D
      OPENCV_EXTRA_MODULES_PATH=../../opencv_contrib-master/modules ..
      " # static libraries
68 cmake $cmake_args
69 if [ $? != 0 ] ; then
70   error_exit "$LINENO: 'cmake' failed"
71 fi
72 make -j6
```

```
73 if [ $? != 0 ] ; then
74   error_exit "$LINENO: 'make' failed"
75 fi
76 make install
77 if [ $? != 0 ] ; then
78   error_exit "$LINENO: 'make install' failed"
79 fi
80 cd ../../../
81
82 # remove build path
83 rm -rf $TMP_PATH
```

Listing A.2: Cart2pol Funktion in Matlab

```
1 function [th,r,z] = cart2pol(x,y,z)
2 %CART2POL Transform Cartesian to polar coordinates.
3 %   [TH,R] = CART2POL(X,Y) transforms corresponding elements of
    data
4 %   stored in Cartesian coordinates X,Y to polar coordinates (angle
     TH
5 %   and radius R).  The arrays X and Y must be the same size (or
6 %   either can be scalar). TH is returned in radians.
7 %
8 %   [TH,R,Z] = CART2POL(X,Y,Z) transforms corresponding elements of
9 %   data stored in Cartesian coordinates X,Y,Z to cylindrical
```

```
10 %   coordinates (angle TH, radius R, and height Z).  The arrays X,Y
       ,
11 %   and Z must be the same size (or any of them can be scalar).  TH
       is
12 %   returned in radians.
13 %
14 %   Class support for inputs X,Y,Z:
15 %      float: double, single
16 %
17 %   See also CART2SPH, SPH2CART, POL2CART.
18
19 %   Copyright 1984-2005 The MathWorks, Inc.
20
21 th = atan2(y,x);
22 r = hypot(x,y);
```

# Literaturverzeichnis

[asi17] Wikipedia: *Wikipedia Komplexwertige Argumente für Arkussinus.* https://de.wikipedia.org/wiki/Arkussinus_und_Arkuskosinus#Komplexe_Argumente, 2017. – (Besucht am: 11.07.2017)

[EBSK15] Eichenseer, A. ; Bätz, M. ; Seller, J. ; Kaup, A.: A hybrid motion estimation technique for fisheye video sequences based on equisolid reprojection. In: *2015 IEEE International Conference on Image Processing (ICIP)*, 2015, S. 3565–3569

[EK16] Eichenseer, A. ; Kaup, A.: A data set providing synthetic and real-world fisheye video sequences. In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2016, S. 1541–1545

[HM17] Fraunhofer HHI: *HEVC 16.12.* https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/tags/HM-16.2/, 2017. – (Besucht am: 11.07.2017)

[mat17] Matlab: *Matlab 2016b.* https://www.mathworks.com/products/new_products/release2016b.html, 2017. – (Besucht am: 11.07.2017)

[ocv17] OpenCV: *Open Computer Vision.* http://opencv.org/opencv-3-2.html, 2017. – (Besucht am: 11.07.2017)

# Fisheye Prediciton with OpenCV Support

Generated by Doxygen 1.8.5

Wed May 24 2017 13:24:19

# Contents

# Chapter 1

# File Index

## 1.1 File List

Here is a list of all documented files with brief descriptions:

# Chapter 2

# File Documentation

## 2.1 ae_intraPredictionFisheye_v1.cpp File Reference

```
#include <unistd.h>
#include <math.h>
#include <assert.h>
#include <limits.h>
#include <complex.h>
#include <complex>
#include <iostream>
#include "opencv2/opencv.hpp"
#include "ae_predictBlockFisheye_v6.hpp"
#include "ae_intraPredictionFisheye_v1.hpp"
#include "helper.hpp"
```

### Functions

- void meshgrid (const cv::Mat &xgv, const cv::Mat &ygv, cv::Mat &X, cv::Mat &Y)
- void meshgrid2D (const cv::Range &xgv, const cv::Range &ygv, cv::Mat &X, cv::Mat &Y)
- void complex_asin (complex< double > &value, double val)
- void complex_tan (complex< double > &value)
- double sgn (double d)
- void pol2cart (cv::Mat &phi, cv::Mat &r, cv::Mat &xc, cv::Mat &yc)
- void cart2pol (cv::Mat &xc, cv::Mat &yc, cv::Mat &phi, cv::Mat &r)
- void ae_intraPredictionFisheye_v1 (cv::Mat &image, int blocksize, int borderFOV, int modesToTestF[], int modes_size, int modesToTestC[], int modesC_size, bool doScale, bool showTest, bool replaceRef, bool variableModes, cv::Mat &predictedImageConventional, cv::Mat &predictedImageFisheye, cv::Mat &intraModesFisheye, cv::Mat &intraModesConventional)

### 2.1.1 Detailed Description

Source File for fisheye intra prediction for the complete image. Contains all the function definitions for the intra prediction of fisheye images, especially the construction of a meshgrid.

### 2.1.2 Function Documentation

#### 2.1.2.1 void ae_intraPredictionFisheye_v1 ( cv::Mat & *image,* int *blocksize,* int *borderFOV,* int *modesToTestF[],* int *modes_size,* int *modesToTestC[],* int *modesC_size,* bool *doScale,* bool *showTest,* bool *replaceRef,* bool *variableModes,* cv::Mat & *predictedImageConventional,* cv::Mat & *predictedImageFisheye,* cv::Mat & *intraModesFisheye,* cv::Mat & *intraModesConventional* )

Performs the intra prediciton on one image. This functions performs the normal and the fisheye prediction on one image, so that the results can later be compared. The image is iterated in blocks of the size blocksizexblocksize. It also prepares the meshgrid at the start of each image, which is necessary for the enhanced intra prediction for fisheye sequences.

**Parameters**

| | |
|---|---|
| *image* | is a OpenCV Mat in which the image is saved. |
| *blocksize* | is an integer, which sets the blocksize for each block iterating the image. It can have the values 4, 8, 16, 32, 64. Every other values are not permitted. |
| *borderFOV* | is an integer, gives inforMation about the border Field of View. |
| *modesToTestF* | is an integer array, which saves the modes for the enhanced fisheye intra prediction. The allowed value per element is between 0 and 67. Every other value is ignored. |
| *modes_size* | is an integher, indicating the length of the modesToTestF array. |
| *modesToTestC* | is an integer array, which saves the modes for the conventional intra prediction. The allowed values per element is between 0 and 67. But if you want to compare conventional with fisheye prediction only the values 0 to 35 should be used, because otherwise the results should be the same. |
| *modesC_size* | is an integer, indicating the length of the modesToTestC array. |
| *doScale* | is a boolean, indicating if the scaling is active or not. True means scaling is on, false scaling is off. |
| *showTest* | is a boolean, indicating if additional debbuging tests should be performed and showed. If True test are shown. |
| *replaceRef* | is a boolean, indicating if the reference pixel row and column should be replaced or not by static values. If true the reference Pixels are replaced. |
| *variableModes* | is boolean, indicating if variableModes are allowed or not. If true variable modes are allowed. |
| *predictedImage-Conventional* | is a OpenCV Mat reference, in which the conventionally predicted image is saved to. |
| *predictedImage-Fisheye* | is a OpenCV Mat reference, in whick the fisheye predicted image is saved to. |
| *intraModes-Fisheye* | is a OpenCV Mat reference, in which the used fisheye prediciton mode for each block is saved to. This is needed for evaluation purposes. |
| *intraModes-Conventional* | is a OpenCV Mat reference, in which the used fisheye prediciton mode for each block is saved to. This is needed for evaluation purposes. |

#### 2.1.2.2 void cart2pol ( cv::Mat & *xc,* cv::Mat & *yc,* cv::Mat & *phi,* cv::Mat & *r* )

Converts cartesian coordinates to polar coordinates.

**Parameters**

| | |
|---|---|
| *xc* | is a OpenCV Mat reference and holds the x cartesian coordinates. |
| *yc* | is a OpenCV Mat reference and holds the y cartesian coordinates. |
| *phi* | is a OpenCV Mat reference to which the phi values of the polar coordinates are saved to. |
| *r* | is a OpenCV Mat reference to which the radius values of the polar coordinates are saved to. |

#### 2.1.2.3 void complex_asin ( complex< double > & *value,* double *val* )

Calculates the inverse sinus of a complex number. This function calculates the inverse sinus of a complex number. Because there is no c++ (at least not before c++11) implementation to directly calculate asin on a complex value, this function is needed.

**Parameters**

| | |
|---|---|
| *value* | is a reference to a c++ complex double. The result of the calculation is saved in here. |
| *val* | is a double value, on which the calculation of the asin is performed. Because the result of asin is not always real valued, the corresponding real and imaginary parts of the result have to be assinged to c++ complex number. |

#### 2.1.2.4 void complex_tan ( complex< double > & *value* )

Calculates tangens of a complex number. This function calculates the tangens of a complex number. Because there is no c++ (at least not before c++11) implementation to directly calculate tan on a complex value, this function is needed. The function calculates each part of the complex number (real and imaginary) seperately.

**Parameters**

| | |
|---|---|
| *value,is* | a reference to a c++ complex double, on which the tangens is calculated. Each part (rean and imaginary) is calculated seperately. |

#### 2.1.2.5 void meshgrid ( const cv::Mat & *xgv,* const cv::Mat & *ygv,* cv::Mat & *X,* cv::Mat & *Y* )

Helper Function for a two dimensional meshgrid. This function is only called whithin the meshgrid2D function, it also has the same input variables. This function is not supposed to be called directly.

**Parameters**

| | |
|---|---|
| *xgv* | is a OpenCV Range indicating the start and end of the X meshgrid. |
| *ygv* | is a OpenCV Range indicating the start and end of the Y meshgrid. |
| *X* | is a OpenCV Mat reference, in which the rows meshgrid is saved to. |
| *Y* | is a OpenCV Mat reference, in which the columns meshgrid is saved to. |

#### 2.1.2.6 void meshgrid2D ( const cv::Range & *xgv,* const cv::Range & *ygv,* cv::Mat & *X,* cv::Mat & *Y* )

Calcualtes a meshgrid. This function calculates a two-dimensional meshgrid, similiar to Matlabs meshgrid function. It is necessary for the fisheye intra prediciton and is only dependent on the resolution of the input image.

**Parameters**

| | |
|---|---|
| *xgv* | is a OpenCV Range indicating the start and end of the X meshgrid. |
| *ygv* | is a OpenCV Range indicating the start and end of the Y meshgrid. |
| *X* | is a OpenCV Mat reference, in which the rows meshgrid is saved to. |
| *Y* | is a OpenCV Mat reference, in which the columns meshgrid is saved to. |

#### 2.1.2.7 void pol2cart ( cv::Mat & *phi,* cv::Mat & *r,* cv::Mat & *xc,* cv::Mat & *yc* )

Converts polar coordinates to cartesian coordinates.

**Parameters**

| | |
|---|---|
| *phi* | is a OpenCV Mat reference and holds the phi values of the polar coordinates. |
| *r* | is a OpenCV Mat reference and holds the radius values of the polar coordinates. |
| *xc* | is a OpenCV Mat reference to which the x cartesian coordinates are saved to. |
| *yc* | is a OpenCV Mat reference to which the y cartesian coordinates are saved to. |

#### 2.1.2.8 double sgn ( double *d* )

Returns the sign of a double number.

**Parameters**

| | |
|---|---|
| *d* | is the double on which the sign check is performed. |

**Returns**

returns the sign of the value of d. If d is less than zero -1 is returned, if it is zero then 0 is returned and if d is bigger than zero 1 is returned.

## 2.2 ae_intraPredictionFisheye_v1.hpp File Reference

```
#include <Math.h>
#include <complex.h>
#include <complex>
#include "opencv2/opencv.hpp"
```

### Functions

- void ae_intraPredictionFisheye_v1 (cv::Mat &image, int blocksize, int borderFOV, int modesToTestF[], int modes_size, int modesToTestC[], int modesC_size, bool doScale, bool showTest, bool replaceRef, bool variableModes, cv::Mat &predictedImageConventional, cv::Mat &predictedImageFisheye, cv::Mat &intraModesFisheye, cv::Mat &intraModesConventional)
- void meshgrid2D (const cv::Range &xgv, const cv::Range &ygv, cv::Mat &X, cv::Mat &Y)
- void meshgrid (const cv::Mat &xgv, const cv::Mat &ygv, cv::Mat &X, cv::Mat &Y)
- void complex_asin (complex< double > &value, double val)
- void complex_tan (complex< double > &value)
- double sgn (double d)
- void cart2pol (cv::Mat &xc, cv::Mat &yc, cv::Mat &phi, cv::Mat &r)
- void pol2cart (cv::Mat &phi, cv::Mat &r, cv::Mat &xc, cv::Mat &yc)

### 2.2.1 Detailed Description

Header File for fisheye intra prediction for the complete image. Contains all the function declarations for the intra prediction of fisheye images, especially the construction of a meshgrid.

### 2.2.2 Function Documentation

#### 2.2.2.1 void ae_intraPredictionFisheye_v1 ( cv::Mat & *image*, int *blocksize*, int *borderFOV*, int *modesToTestF[]*, int *modes_size*, int *modesToTestC[]*, int *modesC_size*, bool *doScale*, bool *showTest*, bool *replaceRef*, bool *variableModes*, cv::Mat & *predictedImageConventional*, cv::Mat & *predictedImageFisheye*, cv::Mat & *intraModesFisheye*, cv::Mat & *intraModesConventional* )

Performs the intra prediciton on one image. This functions performs the normal and the fisheye prediction on one image, so that the results can later be compared. The image is iterated in blocks of the size blocksizexblocksize. It also prepares the meshgrid at the start of each image, which is necessary for the enhanced intra prediction for fisheye sequences.

**Parameters**

| | |
|---|---|
| *image* | is a OpenCV Mat in which the image is saved. |
| *blocksize* | is an integer, which sets the blocksize for each block iterating the image. It can have the values 4, 8, 16, 32, 64. Every other values are not permitted. |
| *borderFOV* | is an integer, gives inforMation about the border Field of View. |
| *modesToTestF* | is an integer array, which saves the modes for the enhanced fisheye intra prediction. The allowed value per element is between 0 and 67. Every other value is ignored. |
| *modes_size* | is an integher, indicating the length of the modesToTestF array. |
| *modesToTestC* | is an integer array, which saves the modes for the conventional intra prediction. The allowed values per element is between 0 and 67. But if you want to compare conventional with fisheye prediction only the values 0 to 35 should be used, because otherwise the results should be the same. |
| *modesC_size* | is an integer, indicating the length of the modesToTestC array. |
| *doScale* | is a boolean, indicating if the scaling is active or not. True means scaling is on, false scaling is off. |
| *showTest* | is a boolean, indicating if additional debbuging tests should be performed and showed. If True test are shown. |
| *replaceRef* | is a boolean, indicating if the reference pixel row and column should be replaced or not by static values. If true the reference Pixels are replaced. |
| *variableModes* | is boolean, indicating if variableModes are allowed or not. If true variable modes are allowed. |
| *predictedImage-Conventional* | is a OpenCV Mat reference, in which the conventionally predicted image is saved to. |
| *predictedImage-Fisheye* | is a OpenCV Mat reference, in whick the fisheye predicted image is saved to. |
| *intraModes-Fisheye* | is a OpenCV Mat reference, in which the used fisheye prediciton mode for each block is saved to. This is needed for evaluation purposes. |
| *intraModes-Conventional* | is a OpenCV Mat reference, in which the used fisheye prediciton mode for each block is saved to. This is needed for evaluation purposes. |

#### 2.2.2.2 void cart2pol ( cv::Mat & *xc,* cv::Mat & *yc,* cv::Mat & *phi,* cv::Mat & *r* )

Converts cartesian coordinates to polar coordinates.

**Parameters**

| | |
|---|---|
| *xc* | is a OpenCV Mat reference and holds the x cartesian coordinates. |
| *yc* | is a OpenCV Mat reference and holds the y cartesian coordinates. |
| *phi* | is a OpenCV Mat reference to which the phi values of the polar coordinates are saved to. |
| *r* | is a OpenCV Mat reference to which the radius values of the polar coordinates are saved to. |

#### 2.2.2.3 void complex_asin ( complex< double > & *value,* double *val* )

Calculates the inverse sinus of a complex number. This function calculates the inverse sinus of a complex number. Because there is no c++ (at least not before c++11) implementation to directly calculate asin on a complex value, this function is needed.

**Parameters**

| | |
|---|---|
| *value* | is a reference to a c++ complex double. The result of the calculation is saved in here. |
| *val* | is a double value, on which the calculation of the asin is performed. Because the result of asin is not always real valued, the corresponding real and imaginary parts of the result have to be assinged to c++ complex number. |

#### 2.2.2.4 void complex_tan ( complex< double > & *value* )

Calculates tangens of a complex number. This function calculates the tangens of a complex number. Because there is no c++ (at least not before c++11) implementation to directly calculate tan on a complex value, this function is needed. The function calculates each part of the complex number (real and imaginary) seperately.

**Parameters**

| | |
|---|---|
| *value,is* | a reference to a c++ complex double, on which the tangens is calculated. Each part (rean and imaginary) is calculated seperately. |

#### 2.2.2.5 void meshgrid ( const cv::Mat & *xgv,* const cv::Mat & *ygv,* cv::Mat & *X,* cv::Mat & *Y* )

Helper Function for a two dimensional meshgrid. This function is only called whithin the meshgrid2D function, it also has the same input variables. This function is not supposed to be called directly.

**Parameters**

| | |
|---|---|
| *xgv* | is a OpenCV Range indicating the start and end of the X meshgrid. |
| *ygv* | is a OpenCV Range indicating the start and end of the Y meshgrid. |
| *X* | is a OpenCV Mat reference, in which the rows meshgrid is saved to. |
| *Y* | is a OpenCV Mat reference, in which the columns meshgrid is saved to. |

#### 2.2.2.6 void meshgrid2D ( const cv::Range & *xgv,* const cv::Range & *ygv,* cv::Mat & *X,* cv::Mat & *Y* )

Calcualtes a meshgrid. This function calculates a two-dimensional meshgrid, similiar to Matlabs meshgrid function. It is necessary for the fisheye intra prediciton and is only dependent on the resolution of the input image.

**Parameters**

| | |
|---|---|
| *xgv* | is a OpenCV Range indicating the start and end of the X meshgrid. |
| *ygv* | is a OpenCV Range indicating the start and end of the Y meshgrid. |
| *X* | is a OpenCV Mat reference, in which the rows meshgrid is saved to. |
| *Y* | is a OpenCV Mat reference, in which the columns meshgrid is saved to. |

#### 2.2.2.7 void pol2cart ( cv::Mat & *phi,* cv::Mat & *r,* cv::Mat & *xc,* cv::Mat & *yc* )

Converts polar coordinates to cartesian coordinates.

**Parameters**

| | |
|---|---|
| *phi* | is a OpenCV Mat reference and holds the phi values of the polar coordinates. |
| *r* | is a OpenCV Mat reference and holds the radius values of the polar coordinates. |
| *xc* | is a OpenCV Mat reference to which the x cartesian coordinates are saved to. |
| *yc* | is a OpenCV Mat reference to which the y cartesian coordinates are saved to. |

#### 2.2.2.8 double sgn ( double *d* )

Returns the sign of a double number.

**Parameters**

| | |
|---|---|
| *d* | is the double on which the sign check is performed. |

**Returns**

returns the sign of the value of d. If d is less than zero -1 is returned, if it is zero then 0 is returned and if d is bigger than zero 1 is returned.

## 2.3 ae_predictBlockFisheye_v6.cpp File Reference

```
#include <unistd.h>
#include <math.h>
#include <assert.h>
#include <iostream>
#include "opencv2/opencv.hpp"
#include "ae_predictBlockFisheye_v6.hpp"
#include "helper.hpp"
```

### Macros

- #define **ANGLE_PARAMS** {32, 26, 21, 17, 13, 9, 5, 2, 0, -2, -5, -9, -13, -17, -21, -26, -32, -26, -21, -17, -13, -9, -5, -2, 0, 2, 5, 9, 13, 17, 21, 26, 32}
- #define **INVERSE_ANGLE_PARAMS** {NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, -4096, -1638, -910, -630, -482, -390, -315, -256, -315, -390, -482, -630, -910, -1638, -4096, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN, NAN}

### Functions

- void ae_rotateByDegrees (cv::Mat &x, cv::Mat &y, double angle, cv::Mat &u, cv::Mat &v, int cols, int rows)
- double find_minimum (cv::Mat &m, int rows, int cols)
- void find_min_or_max (cv::Mat &m, int rows, int cols, bool min, double val, double ∗result, bool first)
- int ae_predictBlockFisheye_v6 (cv::Mat &block, cv::Mat &refPxlRow, cv::Mat &refPxlCol, cv::Mat &x_row_undist, cv::Mat &y_row_undist, cv::Mat &x_col_undist, cv::Mat &y_col_undist, cv::Mat &x_blk_undist, cv::Mat &y_blk_undist, int modesToTest[], int modes_size, bool showTest, bool replaceRef, cv::Mat &predictedBlock)

### 2.3.1 Detailed Description

Source File for fisheye intra prediction on one block of the image. Contains all the function definitons for the intra prediction of fisheye image blocks.

### 2.3.2 Function Documentation

#### 2.3.2.1 int ae_predictBlockFisheye_v6 ( cv::Mat & *block,* cv::Mat & *refPxlRow,* cv::Mat & *refPxlCol,* cv::Mat & *x_row_undist,* cv::Mat & *y_row_undist,* cv::Mat & *x_col_undist,* cv::Mat & *y_col_undist,* cv::Mat & *x_blk_undist,* cv::Mat & *y_blk_undist,* int *modesToTest[],* int *modes_size,* bool *showTest,* bool *replaceRef,* cv::Mat & *predictedBlock* )

Does the fisheye intra prediction on the current block This function performs the optimized intra prediction for fisheye sequences.

**Parameters**

| | |
|---|---|
| *block* | is the current block of the picture of size blocksizexblocksize. |
| *refPxlRow* | is the reference pixel row above the picture. |
| *refPxlCol* | is the reference pixel column left of the picture. |
| *x_row_undist* | is the undistorted pixel row x-indices of the reference pixel row of the original image. |
| *y_row_undist* | is the undistorted pixel row y-indices of the reference pixel row of the original image. |
| *x_col_undist* | is the undistorted pixel colum x-indices of the reference pixel col of the original image. |
| *y_col_undist* | is the undistorted pixel colum y-indices of the reference pixel col of the original image. |
| *x_blk_undist* | is the undistorted pixel block x-indices of the original image. |
| *y_blk_undist* | is the undistorted pixel block y-indices of the original image. |
| *modesToTest* | is an integer array which has the value range of $<0..67>$. The conventional modes are from 0 to 34, the optimized modes are from 35 to 67. The order of the modes is not important any value outside the range is ignored. |
| *modes_size* | is an integer which saves the length of the modesToTest array. |
| *showTest* | is a boolean, indicating if additional debbuging tests should be performed and showed. If True test are shown. |
| *replaceRef* | is a boolean, indicating if the reference pixel row and column should be replaced or not by static values. If true the reference Pixels are replaced. |
| *predictedBlock* | is a reference to a OpenCV Mat with dimension blocksizexblocksize in whick the predicted Block of the image is stored. |

**Returns**

an integer indicating the used intra mode. Can be from 0 to 67. The mode returned also is the mode with the lowest SAD (orininal-predicted) from all tested modes in the modesToTest array.

#### 2.3.2.2 void ae_rotateByDegrees ( cv::Mat & *x,* cv::Mat & *y,* double *angle,* cv::Mat & *u,* cv::Mat & *v,* int *cols,* int *rows* )

Rotation Function for the intra prediction. This function is called in the ae_predictBlockFisheye_v6 function to init angular rotation for the modes 35 to 67. These modes are the optimized modes for intra prediciton.

**Parameters**

| | |
|---|---|
| *x* | ??? |
| *y* | ??? |
| *angle* | is a double with the angle value for rotation. |
| *u* | is a reference to the return value. |
| *v* | is a reference to the return value. |
| *cols* | is an integer, which says how many columns the input mats x and y have. The columns in the input mats have to be identical. |
| *rows* | is an integer, whick says how many rows the input mats x and y have. The rows in the input mats have to be identical. |

#### 2.3.2.3 void find_min_or_max ( cv::Mat & *m,* int *rows,* int *cols,* bool *min,* double *val,* double * *result,* bool *first* )

Find the indices of minimum or maximum in a OpenCV mat. This functions finds the minimum or maximum, indicated by boolean min, and saves the row and colum indices to the result pointer address.

**Parameters**

| | |
|---|---|
| *m* | is the mat in which the search is performed. |
| *rows* | is an integer, which says how many rows the input mat m has. |

| | |
|---|---|
| *cols* | is an integer, which says how many colums the input mat m has. |
| *min* | is a boolean, indicating if a minimum or maximum search is performed. If set to true minimum is searched, otherwise maximum is searched. |
| *val* | if val is a double and when set to a positive value, it searches for values bigger (max) or smaller (min) than this value. |
| *result* | is a pointer to a double array in which the indices of row and column are saved to. result[0] is the index of the row and result[1] is the index of the column. |
| *first* | is a boolean, indicating if the search should stop after finding the first value bigger or smaller than val or if it should continue to find the absolute maximum. |

#### 2.3.2.4 double find_minimum ( cv::Mat & *m,* int *rows,* int *cols* )

Finds the minimum in a OpenCV mat and returns it. This function searches for the minimum in the input mat m and returns it.

**Parameters**

| | |
|---|---|
| *m* | is the input mat in which the search is performed. |
| *rows* | is an integer, which says how many rows the input mat m has. |
| *cols* | is an integer, which says how many columns the input mat m has. |

**Returns**

is a double value, which is the minimum in the input mat m.

## 2.4 ae_predictBlockFisheye_v6.hpp File Reference

```
#include "opencv2/opencv.hpp"
```

### Functions

- int ae_predictBlockFisheye_v6 (cv::Mat &block, cv::Mat &refPxlRow, cv::Mat &refPxlCol, cv::Mat &x_row_undist, cv::Mat &y_row_undist, cv::Mat &x_col_undist, cv::Mat &y_col_undist, cv::Mat &x_blk_undist, cv::Mat &y_blk_undist, int modesToTest[], int modes_size, bool showTest, bool replaceRef, cv::Mat &predictedBlock)
- void ae_rotateByDegrees (cv::Mat &x, cv::Mat &y, double angle, cv::Mat &u, cv::Mat &v, int cols, int rows)
- void find_min_or_max (cv::Mat &m, int rows, int cols, bool min, double val, double *result, bool first)
- double find_minimum (cv::Mat &m, int rows, int cols)

### 2.4.1 Detailed Description

Header File for fisheye intra prediction on one block of the image. Contains all the function declarations for the intra prediction of fisheye image blocks.

### 2.4.2 Function Documentation

#### 2.4.2.1 int ae_predictBlockFisheye_v6 ( cv::Mat & *block,* cv::Mat & *refPxlRow,* cv::Mat & *refPxlCol,* cv::Mat & *x_row_undist,* cv::Mat & *y_row_undist,* cv::Mat & *x_col_undist,* cv::Mat & *y_col_undist,* cv::Mat & *x_blk_undist,* cv::Mat & *y_blk_undist,* int *modesToTest[],* int *modes_size,* bool *showTest,* bool *replaceRef,* cv::Mat & *predictedBlock* )

Does the fisheye intra prediction on the current block This function performs the optimized intra prediction for fisheye sequences.

**Parameters**

| | |
|---|---|
| *block* | is the current block of the picture of size blocksizexblocksize. |
| *refPxlRow* | is the reference pixel row above the picture. |
| *refPxlCol* | is the reference pixel column left of the picture. |
| *x_row_undist* | is the undistorted pixel row x-indices of the reference pixel row of the original image. |
| *y_row_undist* | is the undistorted pixel row y-indices of the reference pixel row of the original image. |
| *x_col_undist* | is the undistorted pixel colum x-indices of the reference pixel col of the original image. |
| *y_col_undist* | is the undistorted pixel colum y-indices of the reference pixel col of the original image. |
| *x_blk_undist* | is the undistorted pixel block x-indices of the original image. |
| *y_blk_undist* | is the undistorted pixel block y-indices of the original image. |
| *modesToTest* | is an integer array which has the value range of $<$0..67$>$. The conventional modes are from 0 to 34, the optimized modes are from 35 to 67. The order of the modes is not important any value outside the range is ignored. |
| *modes_size* | is an integer which saves the length of the modesToTest array. |
| *showTest* | is a boolean, indicating if additional debbuging tests should be performed and showed. If True test are shown. |
| *replaceRef* | is a boolean, indicating if the reference pixel row and column should be replaced or not by static values. If true the reference Pixels are replaced. |
| *predictedBlock* | is a reference to a OpenCV Mat with dimension blocksizexblocksize in whick the predicted Block of the image is stored. |

**Returns**

an integer indicating the used intra mode. Can be from 0 to 67. The mode returned also is the mode with the lowest SAD (orininal-predicted) from all tested modes in the modesToTest array.

#### 2.4.2.2 void ae_rotateByDegrees ( cv::Mat & *x,* cv::Mat & *y,* double *angle,* cv::Mat & *u,* cv::Mat & *v,* int *cols,* int *rows* )

Rotation Function for the intra prediction. This function is called in the ae_predictBlockFisheye_v6 function to init angular rotation for the modes 35 to 67. These modes are the optimized modes for intra prediciton.

**Parameters**

| | |
|---|---|
| *x* | ??? |
| *y* | ??? |
| *angle* | is a double with the angle value for rotation. |
| *u* | is a reference to the return value. |
| *v* | is a reference to the return value. |
| *cols* | is an integer, which says how many columns the input mats x and y have. The columns in the input mats have to be identical. |
| *rows* | is an integer, whick says how many rows the input mats x and y have. The rows in the input mats have to be identical. |

#### 2.4.2.3 void find_min_or_max ( cv::Mat & *m,* int *rows,* int *cols,* bool *min,* double *val,* double * *result,* bool *first* )

Find the indices of minimum or maximum in a OpenCV mat. This functions finds the minimum or maximum, indicated by boolean min, and saves the row and colum indices to the result pointer address.

**Parameters**

| | |
|---|---|
| *m* | is the mat in which the search is performed. |
| *rows* | is an integer, which says how many rows the input mat m has. |

| | |
|---|---|
| *cols* | is an integer, which says how many colums the input mat m has. |
| *min* | is a boolean, indicating if a minimum or maximum search is performed. If set to true minimum is searched, otherwise maximum is searched. |
| *val* | if val is a double and when set to a positive value, it searches for values bigger (max) or smaller (min) than this value. |
| *result* | is a pointer to a double array in which the indices of row and column are saved to. result[0] is the index of the row and result[1] is the index of the column. |
| *first* | is a boolean, indicating if the search should stop after finding the first value bigger or smaller than val or if it should continue to find the absolute maximum. |

#### 2.4.2.4 double find_minimum ( cv::Mat & *m,* int *rows,* int *cols* )

Finds the minimum in a OpenCV mat and returns it. This function searches for the minimum in the input mat m and returns it.

**Parameters**

| | |
|---|---|
| *m* | is the input mat in which the search is performed. |
| *rows* | is an integer, which says how many rows the input mat m has. |
| *cols* | is an integer, which says how many columns the input mat m has. |

**Returns**

is a double value, which is the minimum in the input mat m.

## 2.5 helper.cpp File Reference

```
#include "helper.hpp"
```

### Functions

- int mod (int a, int b)

  *Mahtematical Modulo Function.*

### 2.5.1 Detailed Description

Contains helper functions. This File contains helper function definitions, which can be included by all other functions which may need them.

### 2.5.2 Function Documentation

#### 2.5.2.1 int mod ( int *a,* int *b* )

Mahtematical Modulo Function.

Calculates the result of a modulo with two integers. Instead of the standard modulo operation in c/c++ this function also is able to give the right results when the arguments are negative.

**Parameters**

| | |
|---|---|
| *a* | the first integer for the modulo operation. |
| *b* | the second integer for the modulo operation. |

**Returns**

the result of the operation a mod b.

## 2.6 helper.hpp File Reference

### Functions

- int mod (int a, int b)

  *Mahtematical Modulo Function.*

### 2.6.1 Detailed Description

Contains helper functions. This File contains helper function declarations, which can be included by all other functions which may need them.

### 2.6.2 Function Documentation

#### 2.6.2.1 int mod ( int *a*, int *b* )

Mahtematical Modulo Function.

Calculates the result of a modulo with two integers. Instead of the standard modulo operation in c/c++ this function also is able to give the right results when the arguments are negative.

**Parameters**

| | |
|---|---|
| *a* | the first integer for the modulo operation. |
| *b* | the second integer for the modulo operation. |

**Returns**

the result of the operation a mod b.

## 2.7 mex_ae_intraPredictionFisheye_v1.cpp File Reference

```
#include <math.h>
#include <matrix.h>
#include <mex.h>
#include <iostream>
#include "opencv2/opencv.hpp"
#include "ae_intraPredictionFisheye_v1.hpp"
```

### Macros

- #define **HEIGHT** image_dims[0]
- #define **WIDTH** image_dims[1]

### Functions

- void **mexFunction** (int nlhs, mxArray ∗plhs[], int nrhs, const mxArray ∗prhs[])

### 2.7.1 Detailed Description

Is a Mex interface for matlab. This interface can be called directly in matlab to run c/c++ code. This interface passes the arguments from matlab converts them into OpenCV and standard c/c++ data types and then passes these converted arguments to the c/c++ function. The return arguments are converted back to matlab variables. This interface is for the function ae_intraPredictionFisheye_v1.

## 2.8 mex_ae_predictBlockFisheye_v6.cpp File Reference

```
#include <math.h>
#include <matrix.h>
#include <mex.h>
#include <iostream>
#include "ae_predictBlockFisheye_v6.hpp"
#include "opencv2/opencv.hpp"
```

### Functions

- void **mexFunction** (int nlhs, mxArray ∗plhs[], int nrhs, const mxArray ∗prhs[])

### 2.8.1 Detailed Description

Is a Mex interface for matlab. This interface can be called directly in matlab to run c/c++ code. This interface passes the arguments from matlab converts them into OpenCV and standard c/c++ data types and then passes these converted arguments to the c/c++ function. The return arguments are converted back to matlab variables. This interface is for the function ae_predictBlockFisheye_v6.

# Index