Structureless global bundle adjustment in MicMac
Les grandes lignes

MicMac Team
corresp: ewelina.rupnik@ign.fr

April 28, 2019

Contents

1 Relative orientations NO_AllOri2Im

The code launches TestLib NO_AllOri2Im (cf. Alg. 1) for a pattern of images. Inside the program, it iterates over all images and launches TestLib NO_Ori2Im for all connected images (cf. Alg. 2 and Fig. 1). The main method that calculates the relative orientation between an image pair is detailed in Alg. 4.

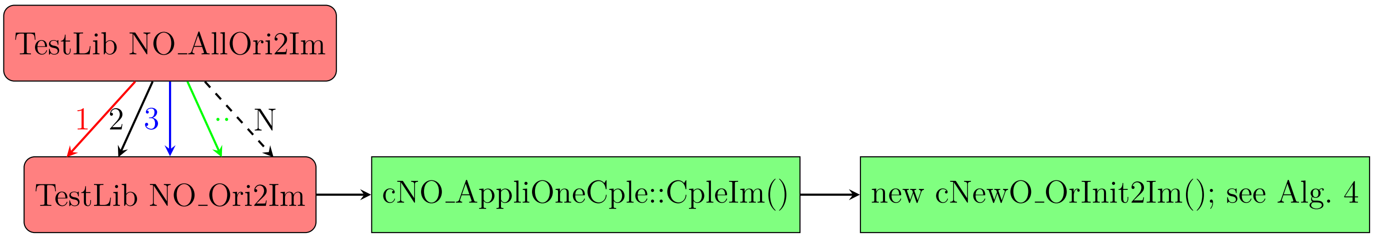

Figure 1: The relative orientation workflow. 1–N signify nodes (i.e. images) in the connectivity graph. Computation is done in parallel.

1.1 Main classes

- cNO_AppliOneCple – manager, prepares data for cNewO_OrInit2Im, contains the image pair and their names, multiple tie-pts structure, a directory/file manager ...
- cNewO_OrInit2Im – class managing the calculation of a relative orientation
- cXml_O2IComputed – saves the relative orientation result

1.2 Algorithms

Algorithm 1 TestLib NO_AllOri2Im def in TestAllNewOriImage_main

$aVIm$ – vector of images in pattern
$aHom$ – homologous points

▷ Launch rel ori caculation for each image pair

for $aK = 0; aK < aVIm \rightarrow size(); aK++$ **do**
  aVH – vector of overlapping images (with common tie-pts)
  for $aKH = 0$; $aKH < aVH \rightarrow size(); aKH++$ **do**
    aIm1 = aVIm[aK]
    aIm2 = aVH[aKH]
    TestLib NO_Ori2Im aIm1 aIm2
  end for
end for

Algorithm 2 TestLib NO_Ori2Im def in TestNewOriImage_main

$BenchNewFoncRot()$

▷ Preparation of data, i.e. create multiple tie-pts structures, read orientations if InOri

$cNO\_AppliOneCple\ anAppli(argc, argv)$

▷ Calculate relative orientation

cNewO_OrInit2Im * aCple = anAppli.CpleIm();

▷ Save result to xml

$cXml\_Ori2Im\&aXml = aCple- > XmlRes()$

$MakeFileXML(aXml, anAppli.NameXmlOri2Im(true))$

$MakeFileXML(aXml, anAppli.NameXmlOri2Im(false))$;

Algorithm 3 anAppli.CpleIm() def in NewOri/cNewO_CpleIm.cpp

▷ Initialise an object of class $cNewO\_OrInit2Im$

return new $cNewO_OrInit2Im(mGenOri, mQuick, mIm1, mIm2, \&mMergeStr, mTestSol, mRotInOr$

Algorithm 4 cNewO_OrInit2Im Initial orientation of an image pair

Creates *PackReduit* which is a subset of 500 tie-pts chosen heuristically
Save photogrammetric points for the image pair, i.e. HomFloatSym.dat
Initialise the cXml_Ori2Im for the image pair

if *GpsIsInit* **then**
  Do something
end if

Calculate a homography between 2d points with $cElHomographie :: RobustInit$
Save results (including residuals, overlap hom, 2d ellipses) to *cXml_O2IComputed* class, object *aXCmp*

▷ Test if the scene is not locally planar
ElRotation3D * aRP = TestOriPlanePatch()
if aRP **then**
  AmelioreSolLinear(*aRP,)
end if
▷ Test the essential matrix with minimum pts and RANSAC (8pts algo)
ElRotation3D aMRR = TestcRanscMinimMatEss()
AmelioreSolLinear(aMRR,);

▷ Test the essential matrix with more pts and RANSAC (8pts algo)
ElRotation3D aRR = RansacMatriceEssentielle()

▷ Test the essential matric - classical test, L1+L2, no RANSAC
ElRotation3D aR = (aL2 ? mPackPStd.MepRelPhysStd(1.0,true) : mPackStdRed.MepRelPhysStd(1.0,false)) ;

▷ Test global homography, robust estimation
cResMepRelCoplan aRMC = ElPackHomologue::MepRelCoplan(....
...
Save the new homography in aXCmp.HomWithR()
▷ Test pure rotation of the camera
cResMepCoc aRCoc= MEPCoCentrik(...
...
Save the rotation in aXCmp.RPure().Ori()

▷ Adjust the solution in a few iterations (until now direct linear solution)
ElRotation3D aSol = aBundle→OneIterEq(...

▷ Triangulate tie-pts, calculate 3D ellipses, and save
... aXCmp.Elips() = anElips3D

Keep track of computational time with *cXml_O2ITiming* & *aTiming*

Algorithm 5 cElRotation3D * TestOriPlanePatch def in *photogramphgr_mep_patch_plane.cpp*

to do

Algorithm 6 cElRotation3D * TestcRanscMinimMatEss def in *photogramphgr_mep_patch_plane.cpp*

to do

Algorithm 7 cElRotation3D * RansacMatriceEssentielle def in *photogramphgr_mep_patch_plane.cpp*

to do

Algorithm 8 cElRotation3D * MEPCoCentrik def in *photogramphgr_mep_patch_plane.cpp*

to do

2 Generate point hom per triplet NO_AllImTriplet

The goal of this program is to gather information about tie-points of multiplicity 3, i.e. tie-points with track length 3 and thus visible in all images of a triplet. The program *TestLib NO_AllImTriplet* launches the $CPP\_GenAllImP3()$ function the defined in *cNewO_PointsTriples.cpp*. Inside the $CPP\_GenAllImP3()$, for each image within the called pattern the following programme is executed in parallel *TestLib NO_OneImTriplet*.

Then, the *TestLib NO_OneImTriplet*, calls the $CPP\_GenOneImP3()$ function (see Alg. 9) defined in *cNewO_PointsTriples.cpp*. Inside this function, it creates an object of class *cAppli_GenPTripleOneImage*, and launches this class' member function – $GenerateTriplets()$ (cf. Fig. 10).

2.1 Main classes

- *cAppli_GenPTripleOneImage*

2.2 Algorithms

Algorithm 9 *CPP_GenOneImP3* def in *cNewO_PointsTriples.cpp*

▷ Create object of class *cAppli_GenPTripleOneImage* (in constructor initialise *mVCams* which contains all images connected to the current image; the first element of *mVCam* is the current/master image

cAppli_GenPTripleOneImage anAppli(argc,argv);

▷ Launch triplet point generation, see Alg. 10

anAppli.GenerateTriplets();

Algorithm 10 $cAppli_GenPTripleOneImage :: GenerateTriplets()$ def in *cNewO_PointsTriples.cpp*

$mVCams$ – vector of all images

▷ For every two images connected to the current image (i.e. $mVCams[0]$), calculate the triplet points

for $aK = 0; aK < mVCams \to size(); aK++$ **do**
  for $aK = 0; aK < mVCams \to size(); aK++$ **do**
    GenerateTriplet(aKC1,aKC2);
  end for
end for

Algorithm 11 $cAppli_GenPTripleOneImage :: GenerateTriplet(in, int)$ def in *cNewO_PointsTriples.cpp*

▷ Load all tie-pts for an image pair
▷ Collect tie-pts with multiplicity 3
▷ Save the triplet points to a file

$mNM- > WriteTriplet(aName3, aVP1Exp, aVP2Exp, aVP3Exp, aVNb);$

3 Generate triplets NO_GenTripl

The goal here is to build a set of optimal triplet pairs from the existing relative orientations. As is illustrated schematically in Fig. 2 the TestLib NO_GenTripl launches GenTriplet_main (see Alg. 12) for a pattern of images. Then, for each edge of the epipolar graph, the Alg. 15 is called.

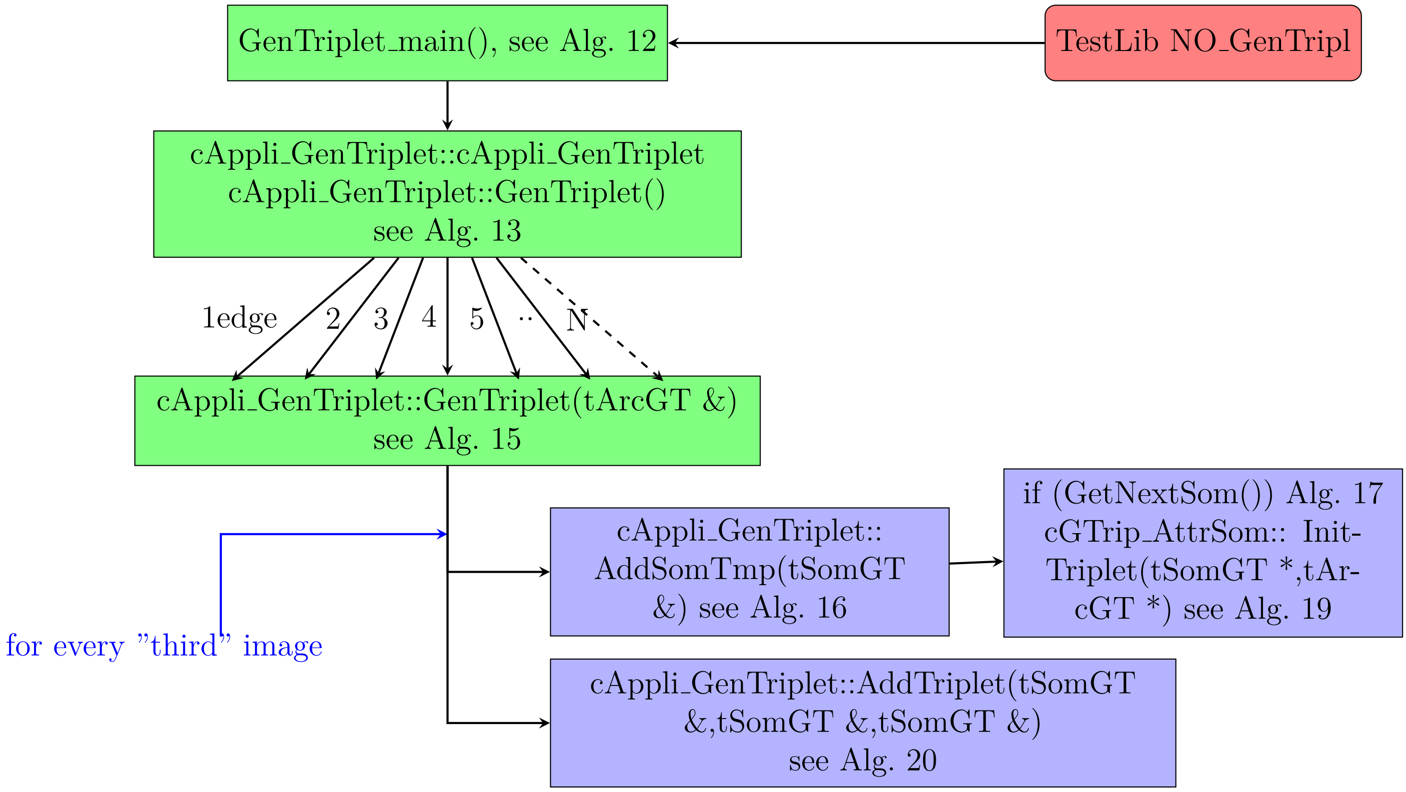

Figure 2: The triplet generation workflow. The enumerated arrows launch the GenTriplet method for each edge (i.e. a pair of nodes and the edge connecting them) of the epipolar graph. The methods in purple are launched for each "third" image attached to each edge of the graph.

3.1 Main classes, some typedefs and variables

- *cAppli_GenTriplet*
- *cGTrip_AttrSom* – class corresponding to a triplet node (its orientation parameters etc)
- *cGTrip_AttrASym* – ?
- *cGTrip_AttrArc* – class containing relative rotation corresponding to an edge
- *cResTriplet* – class containing *cXml_Ori3ImInit* (xml save)

– *cTripletInt* – typedef to $cTplTriplet < int >$ containing ids of images within a triplet

– *tSomGT* – a node of the graph

– *tArcGT* – an edge of the graph

– *tGrGT* – a graph

* *mHautBase* – height, used e.g. in the B to H ratio calcul

* $mMapTriplets$ – contains all triplets (for each triplet it stores the identifiers of nodes defining the node and the resulting orientations)

* $mTopoTriplets$ – contains a list of all accepted triplets (xml of type $Xml_TopoTriplet$)

* $mTriOfCple$ – stores edges (i.e. 2 images) and a list of "third" images for each edge

* $mVSomVois$ – a vector of nodes that are neighbouring with an edge in consideration (i.e., they're outside that edge)

* $mVSomEnCourse$ – vector of "third" images that are currently connected to an edge

* $mVSomSelected$ – vector containing the selected triplets

3.2 Algorithms

$$|C_2 + \lambda \cdot C_3 \quad U_3 \quad U_1| = 0 \tag{1}$$

$$\lambda = -\frac{|C_2 \quad U_3 \quad U_1|}{|C_3 \quad U_3 \quad U_1|} \tag{2}$$

Figure 3: Computation of orientations of a triplet consistent with relative orientations. See also Alg. 19

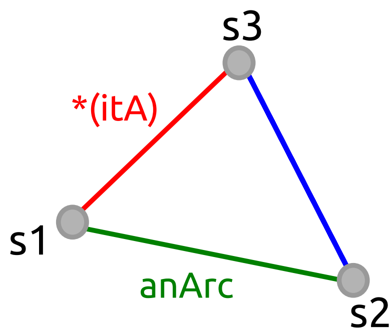

Figure 4: A triplet defined by 3 nodes and three edges, illustration of the Alg. 15. The green edge is the edge entering the $GenTriplet(tArcGT\&anArc)$ method. If s3 exists, there is an edge between s1 and s3. If $(mGrT.edge\_s1s2(anArc.s2(), aS3))$, the blue edge connecting s2 and s3 exists, and therefore a complete triplet is defined.

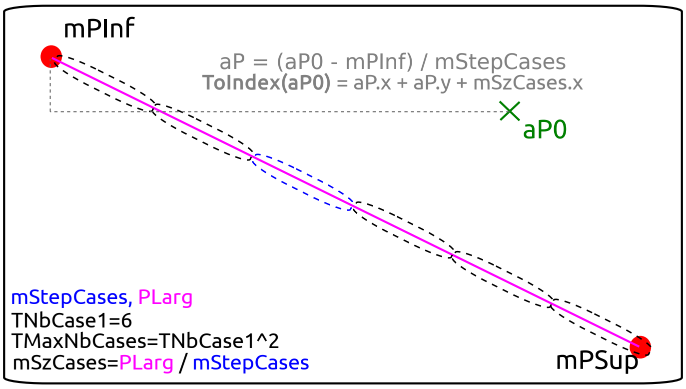

Figure 5: Illustration of the tie-pts indexing method implemented in $ToIndex(const Pt2df \& aP0)$.

Algorithm 12 *GenTriplet_main* def in cNewO_OldGenTriplets.cpp

---

▷ Initialise the object class

cAppli_GenTriplet anAppli(argc,argv);

▷ Call the GenTriplet()

anAppli.GenTriplet();

Algorithm 13 *cAppli_GenTriplet :: cAppli_GenTriplet* constructor def in cNewO_OldGenTriplets.cpp

---

▷ Some member variables

mVecAllSom - a vector of all nodes
mMapS - a map of graph nodes and the respective image names
mGrT - the epipolar graph
tSubGrGT SubAll allows to iterate over a subgraph of edges attached to a node

▷ Initialise the nodes of the graph

mVecAllSom, mMapS

▷ Initialise the edges of the graph

for Each stereo pair **do**
  Get respective nodes from the map of nodes (aS1,aS2)
  Get the relative orientation from XML file
  Test the edge from aS1 to aS2 ?
  Add the edge to the graph mGrT
end for

Algorithm 14 *cAppli_GenTriplet :: GenTriplet()* def in cNewO_OldGenTriplets.cpp

---

▷ Iterate over edges

for All edges between nodes in mVecAllSom **do**
  Call GenTriplet(*itA);
end for

Algorithm 15 *cAppli_GenTriplet* :: *GenTriplet(tArcGT&anArc)* def in cNewO_OldGenTriplets.cpp

▷ Only symmetric edges are dealt with

if (!anArc.attr().IsDirASym()) return;

▷ Load tie-pts for the stereo pair

▷ Get the distance between the two most distant points (mPInf,mPSup) in the pair model; return if it is too large

▷ Some variables

mCurPMed – median point calculated on all tie-pts of a pair
mCurS1 = & (anArc.s1()); – the aS1 in Fig. 4
mCurS2 = & (anArc.s2()); – the aS2 in Fig. 4

▷ Initialise the triplets. See Fig. 4 for illustration

for all edges attached to mCurS1/aS1 **do**
  Get the second node $(*itA).s2()$ for an edge in consideration and call it aS3 of the triplet
  if there is an edge between aS3 and aS2 of the anArc **then**
    call $AddSomTmp(aS3)$, see Alg. 16
  end if
end for

▷ Iterate over all connected images that are found in mVSomEnCourse and select the "best" ones. The number of selected triplets is defined by $TQuickNbMaxTriplet$ and $TStdNbMaxTriplet$. See Alg. 17

while aSom = GetNextSom() **do**
  $AddTriplet(*aSom, mCurArc->s1(), mCurArc->s2());$ see Alg. 20
end while

Algorithm 16 *cAppli_GenTriplet* :: *AddSomTmp(tSomGT&aS)* def in cNewO_OldGenTriplets.cpp

$mVSomVois.push\_back(\&aS);$

▷ If an approx orientation of triplet is possible, add the "third" image to the current nodes

if **then** $InitTriplet$ (see Alg. 19)
  $mVSomEnCourse.push\_back(\&aS);$
end if

Algorithm 17 $cAppli\_GenTriplet :: GetNextSom()$, update the mVSomSelected; def in cNewO_OldGenTriplets.cpp

▷ Leave if your objective is reached, i.e. there are no "third" images in the current nodes or if the number of selected "third" images is already bigger than the $aNbMaxTriplet$
$if(mVSomEnCourse.empty())return0;$
$if(int(mVSomSelected.size()) > aNbMaxTriplet)return0;$

▷ Get the best GainGlob for all available "third" images and store it in: aGainMax, aRes, aIndexRes
▷ Remove the best image from mVSomEnCourse
$mVSomEnCourse.erase(mVSomEnCourse.begin() + aIndexRes);$
▷ And save it in the selected images
$mVSomSelected.push_back(aRes);$
▷ Update the cost/gain of the remaining images giventhe best result in $aRes$
for all images in mVSomEnCourse **do**
    $tSomGT * aSom = mVSomEnCourse[aK];$
$aSom- > attr().UpdateCost(aSom, aRes);$ see Alg. 18
end for

Algorithm 18 $cGTrip_AttrSom :: UpdateCost(tSomGT * aSomThis, tSomGT * aSomSel)$, def in cNewO_OldGenTriplets.cpp

▷ Calculates the new gain as a function of b sur h between aSom and the aSomSel being the best node
$aNewGain = mAppli- > GainBSurH(aSomThis, aSomSel);$

▷ Get Pds and Dens that will help to define the new global gain/cost
$int * aPdsGlob = mAppli- > Pds();$
$int * aDens2 = aSomSel- > attr().mDens;$

▷ Re-define the gain accordingly:
for All "cases" (here 6*6) **do**
    int aD2 = aDens2[aK];
    ElSetMin(mGain[aK], mDens[aK] * (aNewGain * aD2 + TQuant*(TQuant-aD2)));
    mGainGlob += mGain[aK] * aPdsGlob[aK];
end for

Algorithm 19 *cGTrip_AttrSom :: InitTriplet(tSomGT * aSom, tArcGT * anA12)* def in cNewO_OldGenTriplets.cpp

▷ Some variables

$anA13$ – edge from s1 to s3 (see Fig. 4)
$anA23$ – edge from s2 to s3 (see Fig. 4)
$aR21$ – affine rotation (Rot et tr) corresponding to edge between s1 and s2
$aR31$ – affine rotation (Rot et tr) corresponding to edge between s1 and s3
$aR31Bis$ – affine rotation (Rot et tr) corresponding to edge between s1 and s3 calculated from aR21 and aR32
$aR32$ – affine rotation (Rot et tr) corresponding to edge between s2 and s3
$aVP1$,$aVP2$,$aVP3$ – vectors containing tie-pts visible in the triplet
mNb – vector that contains information about tie-pts in an indexed form

▷ Calculate orientations consistent within a triplet (origin in s1) using the known relative orientations. We search for a scale factor λ for which the three directions stemming from three images (i.e. nodes) and corresponding to a tie-point will intersect in 3D. We use the coplanarity condition as the mathematical model to calculate it. Its determinant shall be null for the solution to exist. Because we will have as many λs as there is tie-points, the final result is the median of all results. Since the multiple points are not really considered in this approach, it will be ill-defined for three colinear perspective center.

for all almost tie-pts in $aVP1$ (aStep defines the subsample) **do**

▷ Get normalised vector directions

$aU31$ – direction corresponding to tie-pt in s3 calculated from R31
$aU2$ – direction corresponding to tie-pt in s2
$aU1$ – direction corresponding to tie-pt in s1
$aU31Bis$ – direction corresponding to tie-pt in s3 calculated from R31Bis

$aVL13$, $aVL23$ are vectors containing all the results calculated using two combinations

▷ Calculate intersection of the points using s1, s2

aPInt12 = InterSeg(aC1,aC1+aU1,aC2,aC2+aU2,OkI);

▷ Impose that s3 is found at the intersection of line s1 s3 and on the bundle stemming from aPInt12 and going towards U31.Save result to aVLByInt

CoordInterSeg(aC1,aC3,aPInt12,aPInt12+aU31,OkI,p,q);

▷ Calculate the perspective center of s3 from aVL13 and aVL23 and aVLByInt

Pt3dr aC31 = aC3 / MedianeSup(aVL13);
Pt3dr aC32 = aC2 + aV3Bis * MedianeSup(aVL23);
mC3 = (aC31 + aC32) / 2.0;
Pt3dr aC3I = aC3 * MedianeSup(aVLByInt);

▷ The final solution for the perspective center is the one from multiple points (i.e. from the inverse aVLByInt). As far as the rotation, the avarege result from R31 et R31Bis is taken (orthonormality is imposed with nearest rotation.)

mC3 = aC3I;
mM3 = NearestRotation((aR31.Mat() + aR31Bis.Mat())*0.5);

end for

▷ Calculate gain and density

$aGain1$, b sur h between s1 and s3
$aGain2$, b sur h between s2 and s3
$aGain = ElMin(aGain1, aGain2)$;

▷ Inside this method, the mNb vector is updated. The size of the vector is equal to $TMaxNbCases$ and it encodes indexed position in an image. To update the mNb, each tie-pt is first indexed with ToIndex() (see Fig. 3.2). The output of the indexing is an int from 0 to $TMaxNbCases$ and it indicates the position within the mNb vector that will be incremented.
$InitNb(aVP1)$;

▷ Using the mNb we now calculate gain scores for each *case element* of the indexed image. The gain depends on points' density in a given *case element*
for all element cases i.e. $mSzCases.x * mSzCases.y$, see Fig. 3.2 **do**
  $mDens[aK]$ – the density per case
  $mGain[aK]$ – gain per case
  $mGainGlob$ – global gain
end for

Algorithm 20 *cAppli_GenTriplet* :: *AddTriplet(tSomGT&, tSomGT&, tSomGT&)* def in cNewO_OldGenTriplets.cpp

▷ Get the corresponding nodes

aA1, aA2, aA3

▷ Normalise the orientations so that the first node has and identity rotation matrix and the bases between the first and the second node is equal to 1. To do that, multiply orientations of nodes aA2 and aA3 by the inverse of aA1; then, divide the orientations by the length of the base between aA1 and aA2.
aR1Inv = aA1.R3().inv();
aR1 = aR1Inv*aA1.R3();
aR2 = aR1Inv*aA2.R3();
aR3 = aR1Inv*aA3.R3();
double aD = euclid(aR2.tr());
aR2 = ElRotation3D(aR2.tr()/aD,aR2.Mat(),true);
aR3 = ElRotation3D(aR3.tr()/aD,aR3.Mat(),true);

▷ Iterate over all (triplet) tie-pts. For each tie-pt in each image get its direction with *AddSegOfRot*. The directions are stored in vectors aW. Intersect the directions in 3D with *InterSeg*. Get the residuals in images, store the mean value in aVRes vector.

▷ Check if this triplet already exists in the $mMapTriplets$. If it does and the median residual of the current triplet is larger than that of the triplet already stored in the map, it returns from the AddTriplet method. Otherwise it continues.

▷ Save. Update the $mMapTriplets$, the $mTopoTriplets$ and the $mTriOfCple$.

4 ***Optimisation I*** NO_AllImOptTrip

The goal of this program is to optimise the calculated triplet orientations by (i) filtering tie-pts, (ii) testing with $TestOPA()$ algorithm based on resection and $TestTomasiKanade()$ algorithm suited for long focal lengths. Finally, bundle adjustement $SolveBundle3Image$ is called for each triplet. Schematic workflow of the algorithms is presented in Fig. 6.

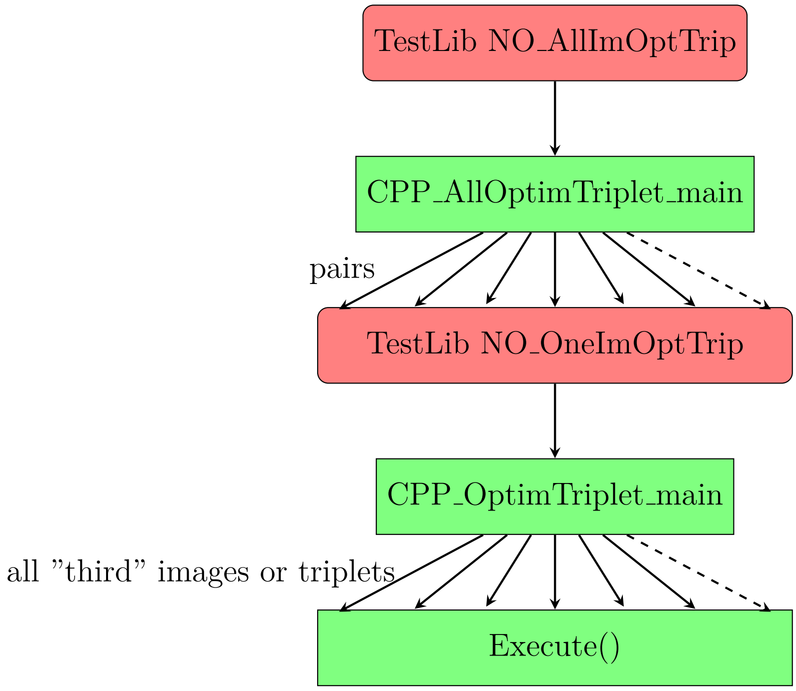

Figure 6: Triplet optimisation I workflow.

4.1 Main classes, some typedefs and variables

- *cAppliOptimTriplet* – manager

4.2 Algorithms

Algorithm 21 *CPP_AllOptimTriplet_main* constructor def in cNewO_OptimTriplet.cpp

▷ Iterate over all couples in $ListeCpleOfTriplets.xml$ (must be coherent with pattern too!) and launch the $TestLibNO\_OneImOptTrip$ programme for each.

Algorithm 22 *CPP_OptimTriplet_main* constructor def in cNewO_OptimTriplet.cpp

▷ Call the object of the "managing" class

cAppliOptimTriplet anAppli(argc,argv,true);

Algorithm 23 $cAppliOptimTriplet$ :: $cAppliOptimTriplet$ constructor def in cNewO_OptimTriplet.cpp

▷ Check whether testing with Tomas-Kanade algo shall be done (specific to long focal lengths)
$mModeTTK$, $mWithTTK$

▷ For a pair of images, get a list of its "third" images forming a triplet with that couple. Launch:
$Execute()$

Algorithm 24 *cAppliOptimTriplet :: Execute()* constructor def in cNewO_OptimTriplet.cpp

▷ Define the number of selected points. If Quick mode is on (true by default) 200 points are used, otherwise its 500 points.
QuickDefNbMaxSel=200, StdDefNbMaxSel=500;
mNbMaxSel = mQuick ? QuickDefNbMaxSel : StdDefNbMaxSel;

mNbMaxInit = mQuick ? QuickNbMaxInit : StdNbMaxInit ;

▷ Calculate a mean focal length $mFoc$

▷ Load images to the vector mIms. Create variables mIm1, mIm2, mIm3 for each image of the triplet.Load pairs of images to vector mPairs. Create variables mP12, mP13, mP23 for each couple of the triplet.

▷ Load tie-pts with the $VFullPtOf3()$ method.

▷ "Attenuate" the number of selections with $\frac{a \cdot b}{a+b}$ as shown below:
int aNbFull3 = $mIm2->VFullPtOf3().size()$;
int aNb3 = round_up(CoutAttenueTetaMax(aNbFull3,mNbMaxSel));
mNbMaxSel = aNb3;

▷ Filter tie-pts according to set parameters
$mSel3 = IndPackReduit(mIm2->VFullPtOf3(), mNbMaxInit, mNbMaxSel);$

▷ Filter tie-pts for Tomas-Kanade tests
$mTKNbMaxSel = ElMin(mTKNbMaxSel, aNbFull3);$
$mTKSel3 = IndPackReduit(mIm2->VFullPtOf3(), ElMin(aNbFull3, 5 * mTKNbMaxSel), ElMin(aNbFull3, mTKNbMaxSel));$

▷ Update some variables for all triplet images
$mIms[aK]->SetReduce(mSel3, mTKSel3);$
$mFullH123.push\_back(\&(mIms[aK]->VFullPtOf3()));$
$mRedH123.push\_back(\&(mIms[aK]->VRedPtOf3()));$
$mTK\_H123.push\_back(\&(mIms[aK]->VTK\_PtOf3()));$

▷ Calculate a global residual which is the mean residual of residuals calculated on pairs 12, 13 and 23
mBestResidu = ResiduGlob();

▷ TestOPA

for (int aKP=0 ; aKP¡int(mPairs.size()) ; aKP++) **do**
    TestOPA(*(mPairs[aKP])); see Alg. 25
end for

▷ TestTomasiKanade

TestTomasiKanade(); see Alg. 27

▷ Perform bundle adjustemnt on the result
$SolveBundle3Image(mFoc, aR21, //mIm2->Ori(), aR31, //mIm3->Ori(), aPMed, aBOnH, mRedH123, mP12->RedHoms(), mP13->RedHoms(), mP23->RedHoms(), mPds3, aParamSB3I);$

▷ Calculate and save the "ellipse"

$cXml\_Elips3D\ anElips3D;$
$aXml.Elips() = anElips3D;$
$MakeFileXML(aXml, aNameSauveXml);$

Algorithm 25 $cAppliOptimTriplet :: TestOPA(cPairOfTriplet\&aPair)$ constructor def in cNewO_OptimTriplet.cpp

▷ Load filtered tie-pts mRedH123 into aVP and images to aI.

for All tie-pts in aVP **do**
Get direction of the point in aI1 and aI2 and store in aVA,aVB
▷ Intersect the two directions
$Pt3dranInter = InterSeg(aVA, aVB, OkI);$

▷ Store in aL32 the correspondence 3D - 2D for the "third" images aI3. Note that the 2D position is not a projection of the anInter.
$aL32.push\_back(Appar23(Pt2dr(aP3.x, aP3.y), anInter));$
end for
▷ Estimate orientation of aI3 from RANSAC-based resection
$ElRotation3DaR3 = aCSI.RansacOFPA(true, aNbRansac, aL32, \&anEcart);$

▷ Normalise the orientations so that the base between image1 and image2 is equal to 1 and the rotation matrix of image1 is identity (same as in Alg. 20)

▷ Test the solution by comparing residuals of the new solution with the former residuals.
TestSol(aVR); see Alg. 26

Algorithm 26 $cAppliOptimTriplet :: TestSol(conststd :: vector < ElRotation3D > \&aVR);$ def in cNewO_OptimTriplet.cpp

▷ Compute the residuals based on the aVR solution
$aResidu = ResiduGlob(aVR[0], aVR[1], aVR[2]);$

▷ If the current residual is smaller than the former best residual, update the orientations and the best residual
if then$aResidu < mBestResidu$
$mBestResidu = aResidu;$
for do$intaK = 0; aK < 3; aK + +$
$mIms[aK]- > SetOri(aVR[aK]);$
end for
end if

Algorithm 27 $cAppliOptimTriplet :: TestTomasiKanade();$ def in cNewO_OptimTriplet.cpp

▷ Orientate with Tomasi-Kanade algorithm
$std :: vector < ElRotation3D > aVR = OrientTomasiKanade$
$(aPrec, mTK\_H123, 3, 50, 1e - 5, (std :: vector < ElRotation3D > *)NULL);$

▷ Test the new solution and accept if smaller residual
TestSol(aVR);

5 *Optimisation II* NO_SolInit3

The goal of this program is ...

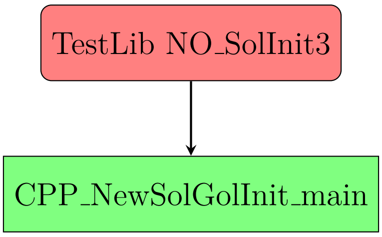

Figure 7: Triplet optimisation II workflow.

5.1 Main classes, some typedefs and variables

- $cAppli\_NewSolGolInit$ – the manager class
- $cRandNParmiQ$ – class for random selections
- $cNOSolIn\_AttrSom$ – class corresponding to a node, defined in SolInitNewOri.h
- $cNOSolIn\_AttrArc$ – class corresponding to an edge, defined in SolInitNewOri.h
- $cNOSolIn\_Triplet$ – triplet solution
- $cLinkTripl$ – it is an ordered $cNOSolIn\_Triplet$

– $typedef ElSomIterator < cNOSolIn\_AttrSom, cNOSolIn\_AttrArc > tItSNSI$ – iterator over nodes

– $typedef ElArcIterator < cNOSolIn\_AttrSom, cNOSolIn\_AttrArc > tItANSI$ – iterator over arcs

5.2 Algorithms

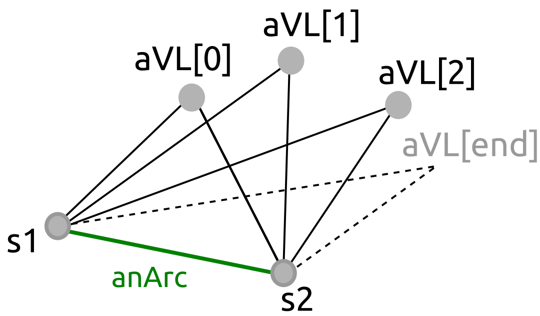

Figure 8: Illustration to Alg. 29. The coherence scores are calculated for all combinations of triplets, e.g. a couple of triplet $s1 - s2 - aVL[0]$ and $s1 - s2 - aVL[1]$.

Algorithm 28 $cAppli\_NewSolGolInit$ :: $cAppli_NewSolGolInit$; def in cNewO_SolGlobInit.cpp

▷ Load the graph, i.e. all the nodes/images and the edges : mMapS, mGr and mV3

▷ Calculate an average coherence score for each triplet.
$EstimCoherenceMed()$;

▷ Estimate robust rotations from available triplets
$EstimRotsArcsInit()$;

▷ dd
$EstimCoheTriplet()$;

▷ dd
$FilterTripletValide()$;

▷ dd
$mTestTrip->CalcCoherFromArcs(true)$;

▷ dd
$NumeroteCC()$;

▷ dd
$CalculOrient()$;

Algorithm 29 $EstimCoherenceMed()$; def in cNewO_SolGlobInit.cpp

▷ Calculate the number of triplet couples that share the same edge and store in: $aNbTT$

▷ Iterate over all nodes

for $tItSNSIanItS = mGr.begin(mSubAll); anItS.go_on(); anItS++$ **do**

$tSomNSI * aS1 = \&(*anItS);$

▷ Iterate over all edges attached to node aS1

for $tItANSIanItA = aS1->begin(mSubAll); anItA.go_on(); anItA++$ **do**

$tArcNSI\&anArc = (*anItA);$ current edge ▷ Get vector of "third" images (triplets) for that edge

$std :: vector < cLinkTripl > \&aVL = anArc.attr().ASym()->Lnk3();$

▷ Two iterations over all "thirds" to calculate some coherence scores between all available triplets for the anArc edge. See Fig. 8.

for $intaK1 = 0; aK1 < int(aVL.size()); aK1++$ **do**

for $intaK2 = aK1 + 1; aK2 < int(aVL.size()); aK2++$ **do**

if $aSel.GetNext()$ **then** i.e., only a number of calls are executed according to $aSel(ElMin(aNbTT, NbMaxATT), aNbTT)$. The $NbMaxATT = 100000$

$cNOSolIn\_Triplet * aTri2 = aVL[aK2].m3;$

▷ Checks coherence from the difference of coordinate systems in TriA and TriB

$aDCAB = DistCoherenceAtoB(\&anArc, aTri1, aTri2);$ see Alg. 30

$aNb = ElMin(aTri1->Nb3(), aTri2->Nb3());$

$aVPAB.push\_back(Pt2df(aDCAB, aNb));$

▷ Checks coherence from relative orientations

$aDC12 = DistCoherence1to2(\&anArc, aTri1, aTri2);$ see Alg. 31

$aVP12.push\_back(Pt2df(aDC12, aNb));$

end if

end for

end for

end for

end for

▷ Saves the median coherence for CohAB and Coh12

$mCoherMedAB = DefMedianPond(0, -1, aVPAB, 0);$

$mCoherMed12 = DefMedianPond(0, -1, aVP12, \&aKMed);$

Algorithm 30 $DistCoherenceAtoB(tArcNSI * anArc, cNOSolIn_Triplet * aTriA, cNOSolIn_Triplet * aTriB)$; def in cNewO_SolGlobInit.cpp

▷ Get orientations of the nodes s1 and s2 in the coordinate system of Triplet A and Triplet 2
aR1A, aR2A, aR2A, aR2B

▷ Calculate transformations from the c system of tripletA to tripletB. Calculate twice from s1 and s2, then take a mean
$aR1AtoB = aR1B * aR1A.inv()$
$aR2AtoB = aR2B * aR2A.inv()$;
$aMatA2B = NearestRotation((aR1AtoB.Mat() + aR2AtoB.Mat()) * 0.5)$;

▷ Calculate the deviation of rotations calculated from s1 and s2 wrt the mean rotation
$aD1 = (aMatA2B - aR1AtoB.Mat()).L2()$;
$aD2 = (aMatA2B - aR2AtoB.Mat()).L2()$;

▷ Calculate the base between s1 and s2 and transform from c sys of tripletA to tripletB (aVA12). Calculate the base in c sys of triplet B (aVB12)
$aVA12 = aMatA2B * (aR2A.tr() - aR1A.tr())$;
$aVB12 = aR2B.tr() - aR1B.tr()$;

▷ Transform the deviation of rotations to distance measure
$aDistRot = sqrt(aD1 + aD2) * (2.0/3)$;

▷ Calculate the B to H ratio as a harmonic mean (because of inverse of proportionality)
$aBOnH = MoyHarmonik(aTriA- > BOnH(), aTriB- > BOnH())$;
▷ Compute deviation on the s1-s2 base calculated from two different triplets. Multiply by BtoH
$aDistTr = DistBase(aVA12, aVB12) * aBOnH$;

▷ The output is the sum of deviation on rotation and the base.
return $aDistRot + aDistTr$;

Algorithm 31 $DistCoherence1to2(tArcNSI * anArc, cNOSolIn_Triplet * aTriA, cNOSolIn_Triplet * aTriB)$; def in cNewO_SolGlobInit.cpp

▷ The $RotationC2toC1$ return relative orientation of s2 wrt s1 (i.e. $R1^{-1} \cdot R2$). Here, it is calculated twice for two triplets and it is passed to $DistanceRot$ together with the B to H ratio harmonic mean.
return $DistanceRot(RotationC2toC1(anArc, aTriA), RotationC2toC1(anArc, aTriB), MoyHarmonik(aTriA- > BOnH(), aTriB- > BOnH()))$; see Alg. 32.

Algorithm 32 $DistanceRot(constElRotation3D\&aR1, constElRotation3D\&aR2, doubleaBSurH)$; def in cNewO_SolGlobInit.cpp

▷ DIfference of rotations

$aDif = aR1.Mat() - aR2.Mat();$

▷ Calculate L2 on the difference

$aDistRot = sqrt(aDif.L2());$

▷ Calculate the euclidean difference between two bases and multiply by the B to H ratio.

$aDistTr = DistBase(aR1.tr(), aR2.tr()) * aBSurH;$

▷ Return the sum of DistRot and DistTr

Algorithm 33 $EstimRotsArcsInit()$;; def in cNewO_SolGlobInit.cpp

Algorithm 34 $EstimCoheTriplet()$;; def in cNewO_SolGlobInit.cpp

Algorithm 35 $FilterTripletValide()$;; def in cNewO_SolGlobInit.cpp

6 Other

Thresholds are defined in *NewOri/NewOri.h*

Graphs are defined in *graphes/graphe.h*