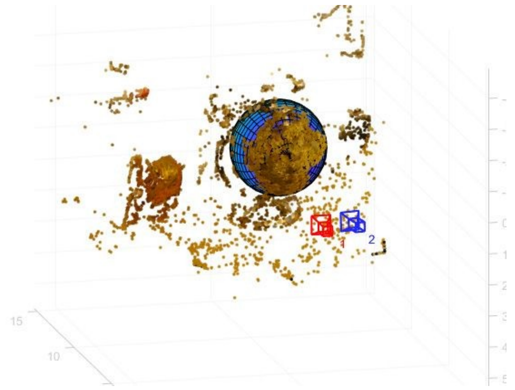
ORB-SLAM

(Simultaneous Localization And Mapping)



Oriented FAST and rotated BRIEF (ORB)

ORB （Oriented FAST and Rotated BRIEF）是 Oriented FAST + Rotated BRIEF 的缩写，是一种快速鲁棒的局部特征检测器，最早由 Ethan Rublee 等人在 2011 年提出，可以将其用于对象识别或 3D 重建等计算机视觉任务。它基于 FAST 关键点检测器和可视描述符 Brief 的修改版本，比 SIFT 快两个数量级。 ORB 是目前最快速稳定的特征点检测和提取算法，许多图像拼接和目标追踪技术利用 ORB 特征进行实现。

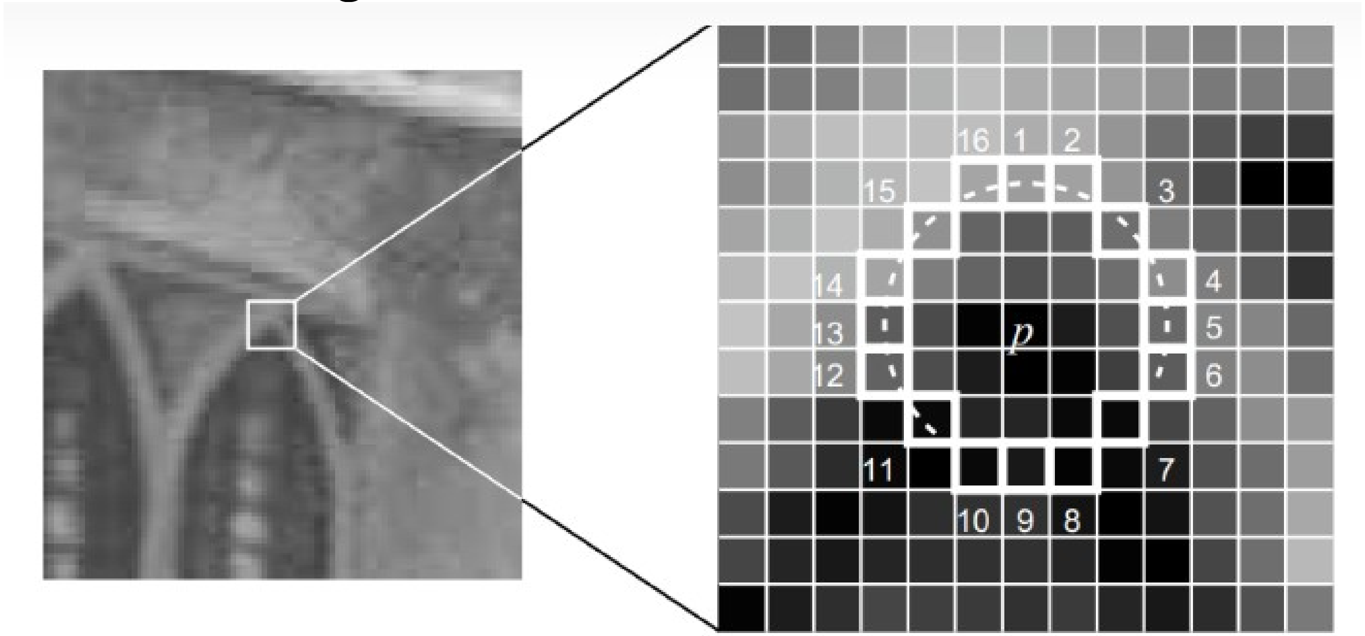
ORB 的关键点是在 FAST （Features from Accelerated Segments Test）关键点基础上进行了改进，主要是增加了特征点的主方向，称之为 Oriented FAST。描述子是在 BRIEF （Binary Robust Independent Elementary Features）描述子基础上加入了上述方向信息，称之为 Rotated BRIEF。

ORB = Oriented FAST （特征点） + Rotated BRIEF （特征描述）

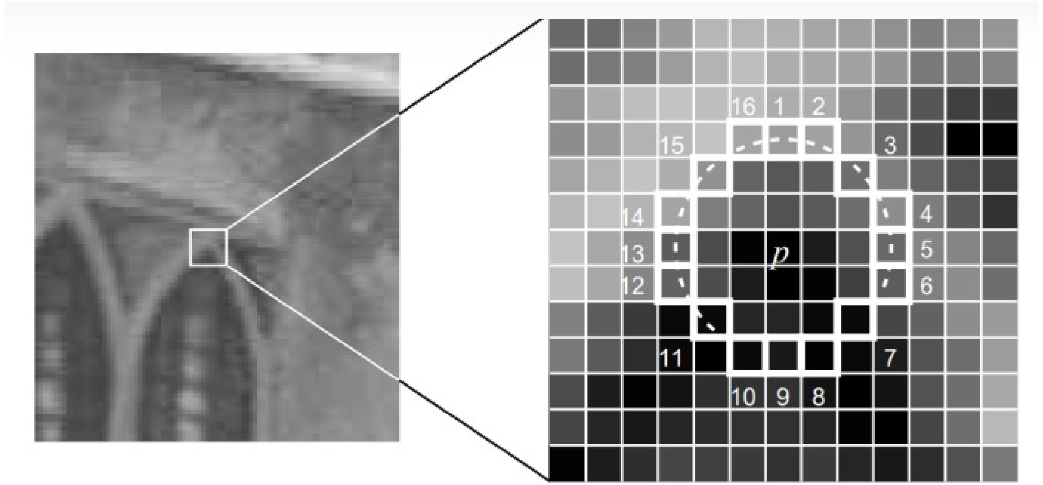
FAST

FAST：若某像素与其周围领域内足够多的像素点相差较大，则该像素可能是特征点。

Step1: 确定候选角点（Segment Test）



FAST



1. 选择某个像素 p， 其像素值为 I_p 。以 p 为圆心，半径为3， 确立一个圆，圆上有16个像素，分别为 $p_1, p_2, \ldots, p_{16}$
2. 确定一个阈值： t (比如 Ip 的 20%)。
3. 让圆上的像素的像素值分别与 p 的像素值做差，如果存在连续n个点满足 $I_x - Ip > t$ 或 $I_x - Ip < -t$ （其中 I_x 代表此圆上16个像素中的一个点），那么就把该点作为一个候选点。根据经验，一般令n=12(n 通常取 12，即为 FAST-12。其它常用的 N 取值为 9 和 11， 他们分别被称为 FAST-9，FAST-11).

FAST

Step2: 非极大值抑制

经过 Step 1 的海选后，还是会有很多个特征点。很可能大部分检测出来的点彼此之间相邻，要去除一部分这样的点，可以采用非最大值抑制的算法：

- 假设 P， Q 两个点相邻，分别计算两个点与其周围的 16 个像素点之间的差分和为 V。
- 去除 V 值较小的点，即把非最大的角点抑制掉。

Oriented FAST

尺度不变性：

1. 对图像做不同尺度的高斯模糊
2. 对图像做降采样 (隔点采样)
3. 对每层金字塔做 FAST 特征点检测
4. n 幅不同比例的图像提取特征点总和作为这幅图像的 oFAST 特征点。

Oriented FAST

旋转不变性：

1、在一个小的图像块 B 中，定义图像块的矩。

$$m_{pq} = \sum_{x,y} x^p y^q I(x, y)$$

2、通过矩可以找到图像块的质心

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}\right)$$

3、连接图像块的几何中心 O 与质心 C，得到一个方向向量 $\overrightarrow{OC}$ ，这就是特征点的方向

$$\theta = \operatorname{atan} 2\left(m_{01}, m_{10}\right)$$

BRIEF

RIEF 是 2010 年的一篇名为《 BRIEF:Binary Robust Independent Elementary Features 》的文章中提出， BRIEF 是对已检测到的特征点进行描述，它是一种二进制编码的描述子，摈弃了利用区域灰度直方图描述特征点的传统方法，采用二级制、位异或运算，大大的加快了特征描述符建立的速度，同时也极大的降低了特征匹配的时间，是一种非常快速，很有潜力的算法。

BRIEF

1、为减少噪声干扰，先对图像进行高斯滤波（方差为2，高斯窗口为9x9）

2、以特征点为中心，取SxS的邻域窗口。在窗口内随机选取一对（两个）点，比较二者像素的大小，进行如下二进制赋值。

$$\tau(\mathbf{p};\mathbf{x},\mathbf{y}) := \begin{cases} 1 & : \mathbf{p}(\mathbf{x}) < \mathbf{p}(\mathbf{y}) \\ 0 & : \mathbf{p}(\mathbf{x}) \geq \mathbf{p}(\mathbf{y}) \end{cases}$$

其中，p(x)，p(y)分别是随机点x=(u1,v1),y=(u2,v2)的像素值。

3、在窗口中随机选取N对随机点，重复步骤2的二进制赋值，形成一个二进制编码，这个编码就是对特征点的描述，即特征描述子。（一般N=256)

这个特征可以由n位二进制测试向量表示，**BRIEF描述子**：

$$f_n(\mathbf{p}) := \sum_{1 \leq i \leq n} 2^{i-1} \tau(\mathbf{p};\mathbf{x}_i,\mathbf{y}_i)$$

Rotated BRIEF

ORB算法采用关键点的主方向来旋转BEIEF描述子。

1、对于任意特征点，在31x31邻域内位置为 (x_i, y_i) 的n对点集，可以用2 x n的矩阵来表示：

$$S = \begin{pmatrix} x_1, \cdots, x_n \\ y_1, \cdots, y_n \end{pmatrix}$$

2、利用FAST求出的特征点的主方向 θ 和对应的旋转矩阵 R_θ ，算出旋转的 S_θ 来代表 S ：

$$\theta = \operatorname{atan} 2\,(m_{01}, m_{10})$$

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$S_\theta = R_\theta S$$

3、计算旋转描述子（**steered BRIEF**）：

$$g_n(p, \theta) := f_n(p) |\, (x_i, y_i) \in S_\theta$$

Oriented FAST and rotated BRIEF (ORB) 总结

- FAST 是用来寻找特征点的。 ORB 在 FAST 基础上通过金字塔、质心标定解决了尺度不变和旋转不变。即 oFAST 。
- BRIEF 是用来构造描述子的。 ORB 在 BRIEF 基础上通过引入 oFAST 的旋转角度和机器学习解决了旋转特性和特征点难以区分的问题。即 rBRIEF.

视觉 SLAM 示例

- Monocular Visual Simultaneous Localization and Mapping

Monocular Visual Simultaneous Localization and Mapping

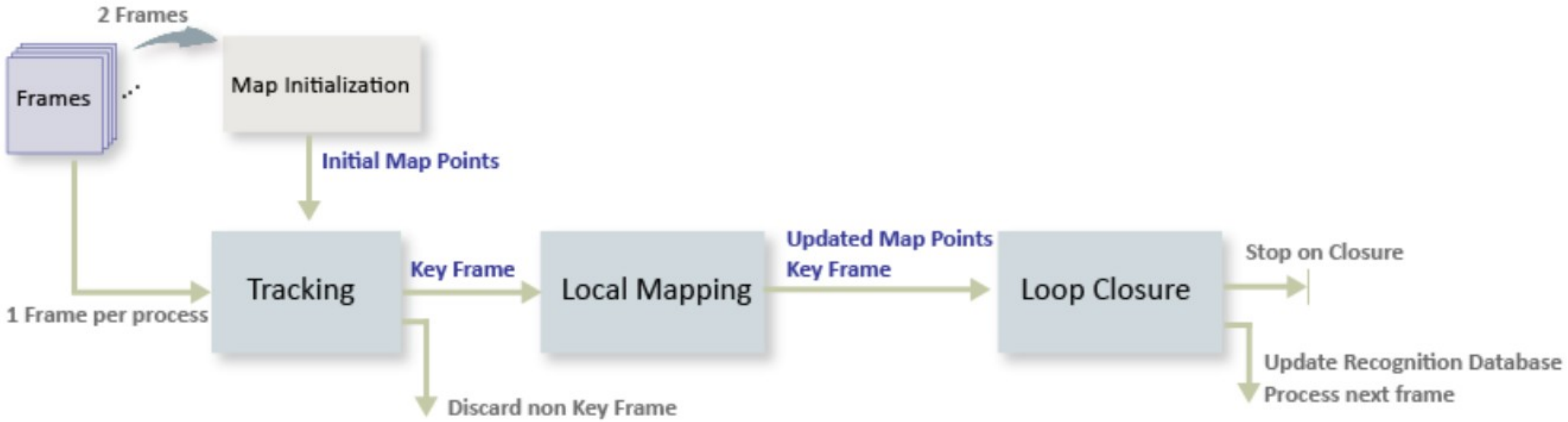
Visual simultaneous localization and mapping (vSLAM), refers to the process of calculating the position and orientation of a camera with respect to its surroundings, while simultaneously mapping the environment. The process uses only visual inputs from the camera. Applications for vSLAM include augmented reality, robotics, and autonomous driving.
This example shows how to process image data from a monocular camera to build a map of an indoor environment and estimate the trajectory of the camera. The example uses ORB-SLAM, which is a feature-based vSLAM algorithm.

Glossary

The following terms are frequently used in this example:

- **Key Frames**: A subset of video frames that contain cues for localization and tracking. Two consecutive key frames usually involve sufficient visual change.
- **Map Points**: A list of 3-D points that represent the map of the environment reconstructed from the key frames.
- **Covisibility Graph**: A graph consisting of key frame as nodes. Two key frames are connected by an edge if they share common map points. The weight of an edge is the number of shared map points.
- **Essential Graph**: A subgraph of covisibility graph containing only edges with high weight, i.e. more shared map points.
- **Place Recognition Database**: A database used to recognize whether a place has been visited in the past. The database stores the visual word-to-image mapping based on the input bag of features. It is used to search for an image that is visually similar to a query image.

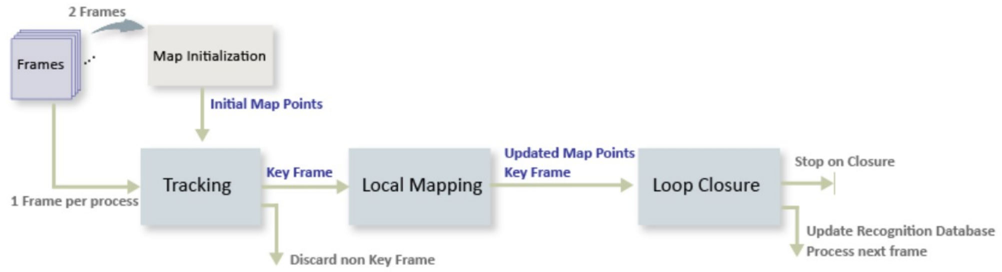
Overview of ORB-SLAM



Overview of ORB-SLAM

The ORB-SLAM pipeline includes:

- **Map Initialization**: ORB-SLAM starts by initializing the map of 3-D points from two video frames. The 3-D points and relative camera pose are computed using triangulation based on 2-D ORB feature correspondences.
- **Tracking**: Once a map is initialized, for each new frame, the camera pose is estimated by matching features in the current frame to features in the last key frame. The estimated camera pose is refined by tracking the local map.
- **Local Mapping**: The current frame is used to create new 3-D map points if it is identified as a key frame. At this stage, bundle adjustment is used to minimize reprojection errors by adjusting the camera pose and 3-D points.
- **Loop Closure**: Loops are detected for each key frame by comparing it against all previous key frames using the bag-of-features approach. Once a loop closure is detected, the pose graph is optimized to refine the camera poses of all the key frames.



Download and Explore the Input Image Sequence

The data used in this example are from the TUM RGB-D benchmark.

```
baseDownloadURL =
'https://vision.in.tum.de/rgbd/dataset/freiburg3/rgbd_dataset_freiburg3_long_office_hous
ehold.tgz';
dataFolder      = fullfile(tempdir, 'tum_rgbd_dataset', filesep);
options         = weboptions('Timeout', Inf);
tgzFileName     = [dataFolder, 'fr3_office.tgz'];
folderExists    = exist(dataFolder, 'dir');

Create a folder in a temporary directory to save the downloaded file
if ~folderExists
    mkdir(dataFolder);
    disp('Downloading fr3_office.tgz (1.38 GB). This download can take a few minutes.')
    websave(tgzFileName, baseDownloadURL, options);

    Extract contents of the downloaded file
    disp('Extracting fr3_office.tgz (1.38 GB) ...')
    untar(tgzFileName, dataFolder);
```

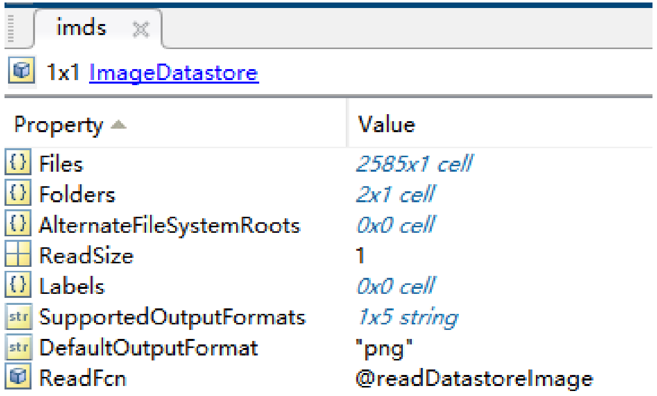
Download and Explore the Input Image Sequence

Create an imageDatastore object to inspect the RGB images.

```
imageFolder   =
["./",'rgbd_dataset_freiburg3_long_office_household/rgbd_dataset_freiburg3_long_office_household/
rgb/'];
imds          = imageDatastore(imageFolder);
```

Inspect the first image

```
currFrameIdx = 1;
currI = readimage(imds, currFrameIdx);
himage = imshow(currI);
```



imds

1x1 ImageDatastore

Property	Value
Files	2585x1 cell
Folders	2x1 cell
AlternateFileSystemRoots	0x0 cell
ReadSize	1
Labels	0x0 cell
SupportedOutputFormats	1x5 string
DefaultOutputFormat	"png"
ReadFcn	@readDatastoreImage

Map Initialization

The ORB-SLAM pipeline starts by initializing the map that holds 3-D world points. This step is crucial and has a significant impact on the accuracy of final SLAM result. Initial ORB feature point correspondences are found using matchFeatures between a pair of images. After the correspondences are found, two geometric transformation models are used to establish map initialization:

- Homography: If the scene is planar, a homography projective transformation is a better choice to describe feature point correspondences.
- Fundamental Matrix: If the scene is non-planar, a fundamental matrix must be used instead.

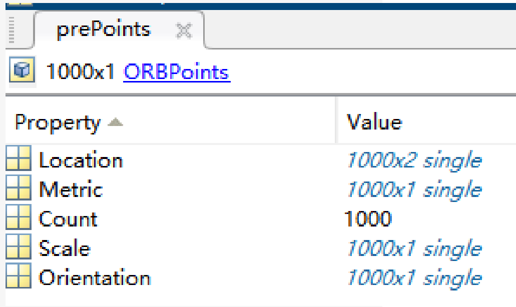
Map Initialization

The homography and the fundamental matrix can be computed using estimateGeometricTransform2D and estimateFundamentalMatrix, respectively. The model that results in a smaller reprojection error is selected to estimate the relative rotation and translation between the two frames using relativeCameraPose. Since the RGB images are taken by a monocular camera which does not provide the depth information, the relative translation can only be recovered up to a specific scale factor.

Given the relative camera pose and the matched feature points in the two images, the 3-D locations of the matched points are determined using triangulate function. A triangulated map point is valid when it is located in the front of both cameras, when its reprojection error is low, and when the parallax of the two views of the point is sufficiently large.

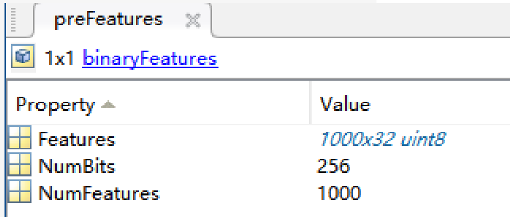
Map Initialization

```
% Set random seed for reproducibility
rng(0);

% Create a cameraIntrinsics object to store the camera intrinsic parameters.
% The intrinsics for the dataset can be found at the following page:
% https://vision.in.tum.de/data/datasets/rgbd-dataset/file_formats
% Note that the images in the dataset are already undistorted, hence there
% is no need to specify the distortion coefficients.
focalLength    = [535.4, 539.2];    % in units of pixels
principalPoint = [320.1, 247.6];    % in units of pixels
imageSize      = size(currI,[1 2]);  % in units of pixels
intrinsics     = cameraIntrinsics(focalLength, principalPoint, imageSize);

% Detect and extract ORB features
scaleFactor = 1.2;
numLevels   = 8;
numPoints   = 1000;
[preFeatures, prePoints] = helperDetectAndExtractFeatures(currI, scaleFactor, numLevels, numPoints);

currFrameIdx = currFrameIdx + 1;
firstI       = currI; % Preserve the first frame

isMapInitialized  = false;
```


prePoints

1000x1 ORBPoints

Property	Value
Location	*1000x2 single*
Metric	*1000x1 single*
Count	1000
Scale	*1000x1 single*
Orientation	*1000x1 single*

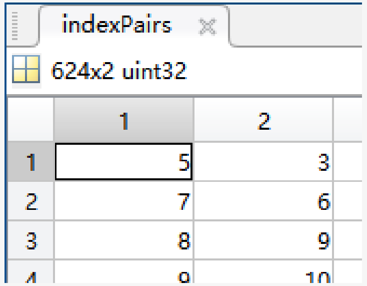

preFeatures

1x1 binaryFeatures

Property	Value
Features	*1000x32 uint8*
NumBits	256
NumFeatures	1000

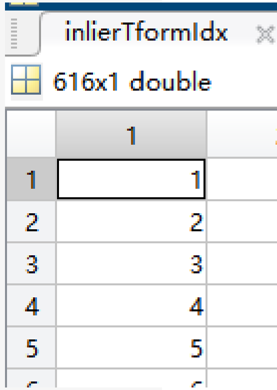
```matlab
function [features, validPoints] = helperDetectAndExtractFeatures(Irgb, ...
    scaleFactor, numLevels, numPoints, varargin)

% In this example, the images are already undistorted. In a general
% workflow, uncomment the following code to undistort the images.
%
% if nargin > 4
%     intrinsics = varargin{1};
% end
% Irgb  = undistortImage(Irgb, intrinsics);

% Detect ORB features
Igray  = im2gray(Irgb);

points = detectORBFeatures(Igray, 'ScaleFactor', scaleFactor, 'NumLevels', numLevels);

% Select a subset of features, uniformly distributed throughout the image
points = selectUniform(points, numPoints, size(Igray, 1:2));

% Extract features
[features, validPoints] = extractFeatures(Igray, points);
end
```

```
% Map initialization loop
while ~isMapInitialized && currFrameIdx < numel(imds.Files)
    currI = readimage(imds, currFrameIdx);
    [currFeatures, currPoints] = helperDetectAndExtractFeatures(currI, scaleFactor, numLevels, numPoints);
    currFrameIdx = currFrameIdx + 1;

    % Find putative feature matches
    indexPairs = matchFeatures(preFeatures, currFeatures, 'Unique', true, 'MaxRatio', 0.9, 'MatchThreshold', 40);
    preMatchedPoints  = prePoints(indexPairs(:,1),:);
    currMatchedPoints = currPoints(indexPairs(:,2),:);

    minMatches = 100;    % If not enough matches are found, check the next frame
    if size(indexPairs, 1) < minMatches
        continue
    end

    preMatchedPoints  = prePoints(indexPairs(:,1),:);
    currMatchedPoints = currPoints(indexPairs(:,2),:);
    % Compute homography and evaluate reconstruction
    [tformH, scoreH, inliersIdxH] = helperComputeHomography(preMatchedPoints, currMatchedPoints);
    % Compute fundamental matrix and evaluate reconstruction
    [tformF, scoreF, inliersIdxF] = helperComputeFundamentalMatrix(preMatchedPoints, currMatchedPoints);

    % Select the model based on a heuristic
    ratio = scoreH/(scoreH + scoreF);    ratioThreshold = 0.45;
    if ratio > ratioThreshold
        inlierTformIdx = inliersIdxH;    tform          = tformH;
    else
        inlierTformIdx = inliersIdxF;    tform          = tformF;
    end
```


indexPairs

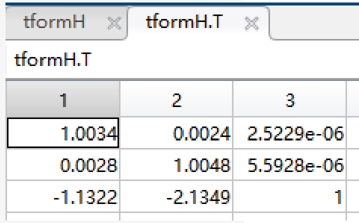
624x2 uint32

	1	2
1	5	3
2	7	6
3	8	9
4	9	10


inlierTformIdx

616x1 double

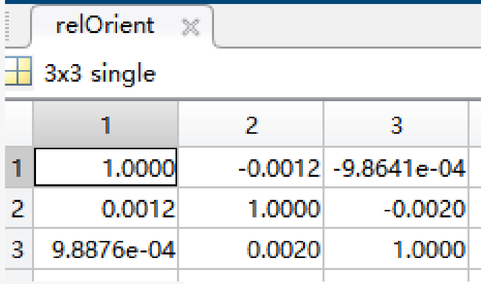
	1
1	1
2	2
3	3
4	4
5	5


tformH | tformH.T

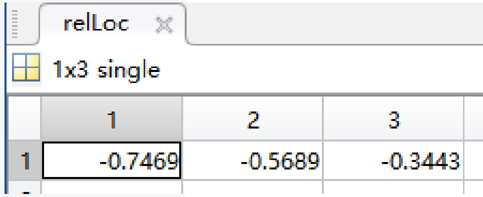
tformH.T

1	2	3
1.0034	0.0024	2.5229e-06
0.0028	1.0048	5.5928e-06
-1.1322	-2.1349	1

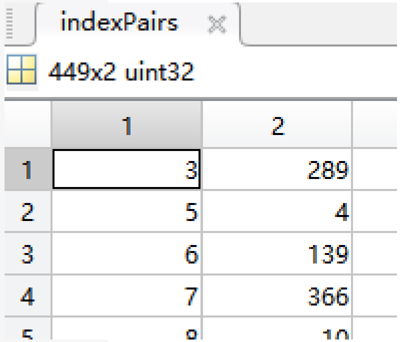
```matlab
    % Computes the camera location up to scale. Use half of the
    % points to reduce computation
    inlierPrePoints  = preMatchedPoints(inlierTformIdx);
    inlierCurrPoints = currMatchedPoints(inlierTformIdx);
    [relOrient, relLoc, validFraction] = relativeCameraPose(tform, intrinsics, ...
        inlierPrePoints(1:2:end), inlierCurrPoints(1:2:end));

    % If not enough inliers are found, move to the next frame
    if validFraction < 0.9 || numel(size(relOrient))==3
        continue
    end

    % Triangulate two views to obtain 3-D map points
    relPose = rigid3d(relOrient, relLoc);
    minParallax = 1; % In degrees
    [isValid, xyzWorldPoints, inlierTriangulationIdx] = helperTriangulateTwoFrames(...
        rigid3d, relPose, inlierPrePoints, inlierCurrPoints, intrinsics, minParallax);

    if ~isValid
        continue
    end

    % Get the original index of features in the two key frames
    indexPairs = indexPairs(inlierTformIdx(inlierTriangulationIdx),:);
    isMapInitialized = true;
    disp(['Map initialized with frame 1 and frame ', num2str(currFrameIdx-1)])
end % End of map initialization loop
```


relOrient
3x3 single

	1	2	3
1	1.0000	-0.0012	-9.8641e-04
2	0.0012	1.0000	-0.0020
3	9.8876e-04	0.0020	1.0000


relLoc
1x3 single

	1	2	3
1	-0.7469	-0.5689	-0.3443


indexPairs
449x2 uint32

	1	2
1	3	289
2	5	4
3	6	139
4	7	366

```
Map initialized with frame 1 and frame 26
```

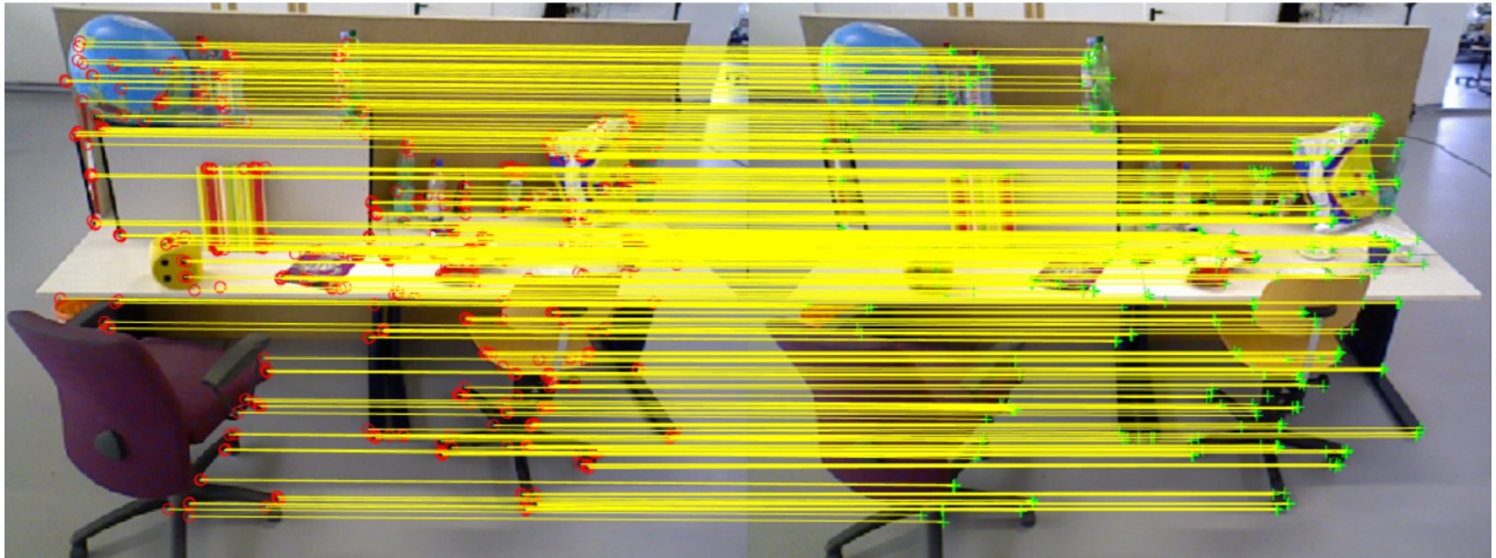
```
function [H, score, inliersIndex] = helperComputeHomography(matchedPoints1, matchedPoints2)

[H, inliersLogicalIndex] = estimateGeometricTransform2D( ...
    matchedPoints1, matchedPoints2, 'projective', ...
    'MaxNumTrials', 1e3, 'MaxDistance', 4, 'Confidence', 90);

inlierPoints1 = matchedPoints1(inliersLogicalIndex);
inlierPoints2 = matchedPoints2(inliersLogicalIndex);

inliersIndex  = find(inliersLogicalIndex);

locations1 = inlierPoints1.Location;
locations2 = inlierPoints2.Location;
xy1In2     = transformPointsForward(H, locations1);
xy2In1     = transformPointsInverse(H, locations2);
error1in2  = sum((locations2 - xy1In2).^2, 2);
error2in1  = sum((locations1 - xy2In1).^2, 2);

outlierThreshold = 6;

score = sum(max(outlierThreshold-error1in2, 0)) + ...
    sum(max(outlierThreshold-error2in1, 0));
end
```

```matlab
function [F, score, inliersIndex] = helperComputeFundamentalMatrix(matchedPoints1, matchedPoints2)

[F, inliersLogicalIndex]   = estimateFundamentalMatrix( ...
    matchedPoints1, matchedPoints2, 'Method','RANSAC',...
    'NumTrials', 1e3, 'DistanceThreshold', 0.01);

inlierPoints1 = matchedPoints1(inliersLogicalIndex);
inlierPoints2 = matchedPoints2(inliersLogicalIndex);

inliersIndex  = find(inliersLogicalIndex);

locations1    = inlierPoints1.Location;
locations2    = inlierPoints2.Location;

% Distance from points to epipolar line
lineIn1   = epipolarLine(F', locations2);
error2in1 = (sum([locations1, ones(size(locations1, 1),1)].* lineIn1, 2)).^2 ...
    ./ sum(lineIn1(:,1:2).^2, 2);
lineIn2   = epipolarLine(F, locations1);
error1in2 = (sum([locations2, ones(size(locations2, 1),1)].* lineIn2, 2)).^2 ...
    ./ sum(lineIn2(:,1:2).^2, 2);

outlierThreshold = 4;

score = sum(max(outlierThreshold-error1in2, 0)) + ...
    sum(max(outlierThreshold-error2in1, 0));

end
```

```
function [isValid, xyzPoints, inlierIdx] = helperTriangulateTwoFrames(...
    pose1, pose2, matchedPoints1, matchedPoints2, intrinsics, minParallax)

[R1, t1]   = cameraPoseToExtrinsics(pose1.Rotation, pose1.Translation);
camMatrix1 = cameraMatrix(intrinsics, R1, t1);

[R2, t2]   = cameraPoseToExtrinsics(pose2.Rotation, pose2.Translation);
camMatrix2 = cameraMatrix(intrinsics, R2, t2);

[xyzPoints, reprojectionErrors, isInFront] = triangulate(matchedPoints1, ...
    matchedPoints2, camMatrix1, camMatrix2);

% Filter points by view direction and reprojection error
minReprojError = 1;
inlierIdx  = isInFront & reprojectionErrors < minReprojError;
xyzPoints  = xyzPoints(inlierIdx ,:);

% A good two-view with significant parallax
ray1       = xyzPoints - pose1.Translation;
ray2       = xyzPoints - pose2.Translation;
cosAngle   = sum(ray1 .* ray2, 2) ./ (vecnorm(ray1, 2, 2) .* vecnorm(ray2, 2, 2));

% Check parallax
isValid = all(cosAngle < cosd(minParallax) & cosAngle>0);
end
```

Map Initialization

```
if isMapInitialized
   close(himage.Parent.Parent); Close the previous figure
   Show matched features
   hfeature = showMatchedFeatures(firstI, currI, prePoints(indexPairs(:,1)), ...
      currPoints(indexPairs(:, 2)), 'Montage');
else
   error('Unable to initialize map.')
end
```

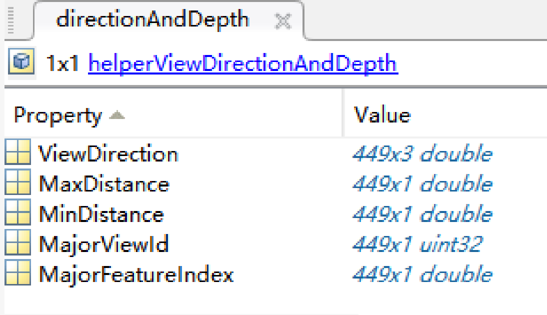


Store Initial Key Frames and Map Points

After the map is initialized using two frames, you can use imageviewset, worldpointset and helperViewDirectionAndDepth to store the two key frames and the corresponding map points:

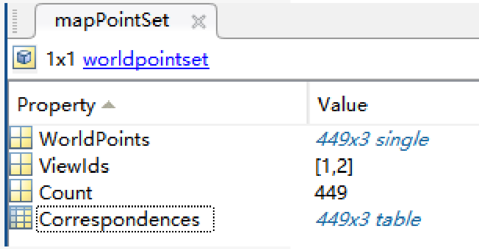
- imageviewset stores the key frames and their attributes, such as ORB descriptors, feature points and camera poses, and connections between the key frames, such as feature points matches and relative camera poses. It also builds and updates a pose graph. The absolute camera poses and relative camera poses of odometry edges are stored as rigid3d objects. The relative camera poses of loop-closure edges are stored as affine3d objects.
- worldpointset stores 3-D positions of the map points and the 3-D into 2-D projection correspondences: which map points are observed in a key frame and which key frames observe a map point.
- helperViewDirectionAndDepth stores other attributes of map points, such as the mean view direction, the representative ORB descriptors, and the range of distance at which the map point can be observed.

```matlab
% Create an empty imageviewset object to store key frames
vSetKeyFrames = imageviewset;
% Create an empty worldpointset object to store 3-D map points
mapPointSet   = worldpointset;
% Create a helperViewDirectionAndDepth object to store view direction and depth
directionAndDepth = helperViewDirectionAndDepth(size(xyzWorldPoints, 1));

% Add the first key frame. Place the camera associated with the first
% key frame at the origin, oriented along the Z-axis
preViewId     = 1;
vSetKeyFrames = addView(vSetKeyFrames, preViewId, rigid3d, 'Points', prePoints, 'Features', preFeatures.Features);

% Add the second key frame
currViewId    = 2;
vSetKeyFrames = addView(vSetKeyFrames, currViewId, relPose, 'Points', currPoints, 'Features', currFeatures.Features);

% Add connection between the first and the second key frame
vSetKeyFrames = addConnection(vSetKeyFrames, preViewId, currViewId, relPose, 'Matches', indexPairs);

% Add 3-D map points
[mapPointSet, newPointIdx] = addWorldPoints(mapPointSet, xyzWorldPoints);

% Add observations of the map points
preLocations  = prePoints.Location;
currLocations = currPoints.Location;
preScales     = prePoints.Scale;
currScales    = currPoints.Scale;

% Add image points corresponding to the map points in the first key frame
mapPointSet   = addCorrespondences(mapPointSet, preViewId, newPointIdx, indexPairs(:,1));

% Add image points corresponding to the map points in the second key frame
mapPointSet   = addCorrespondences(mapPointSet, currViewId, newPointIdx, indexPairs(:,2));
```


directionAndDepth

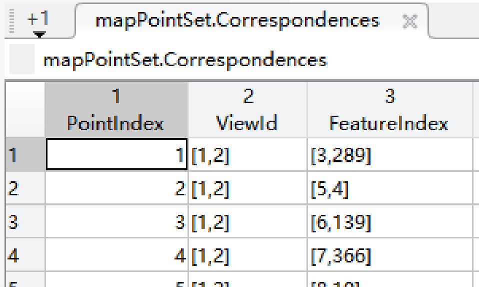
1x1 helperViewDirectionAndDepth

Property	Value
ViewDirection	449x3 double
MaxDistance	449x1 double
MinDistance	449x1 double
MajorViewId	449x1 uint32
MajorFeatureIndex	449x1 double


mapPointSet

1x1 worldpointset

Property	Value
WorldPoints	449x3 single
ViewIds	[1,2]
Count	449
Correspondences	449x3 table


mapPointSet.Correspondences

mapPointSet.Correspondences

	1 PointIndex	2 ViewId	3 FeatureIndex
1	1	[1,2]	[3,289]
2	2	[1,2]	[5,4]
3	3	[1,2]	[6,139]
4	4	[1,2]	[7,366]
5	5	[1,2]	[illegible]

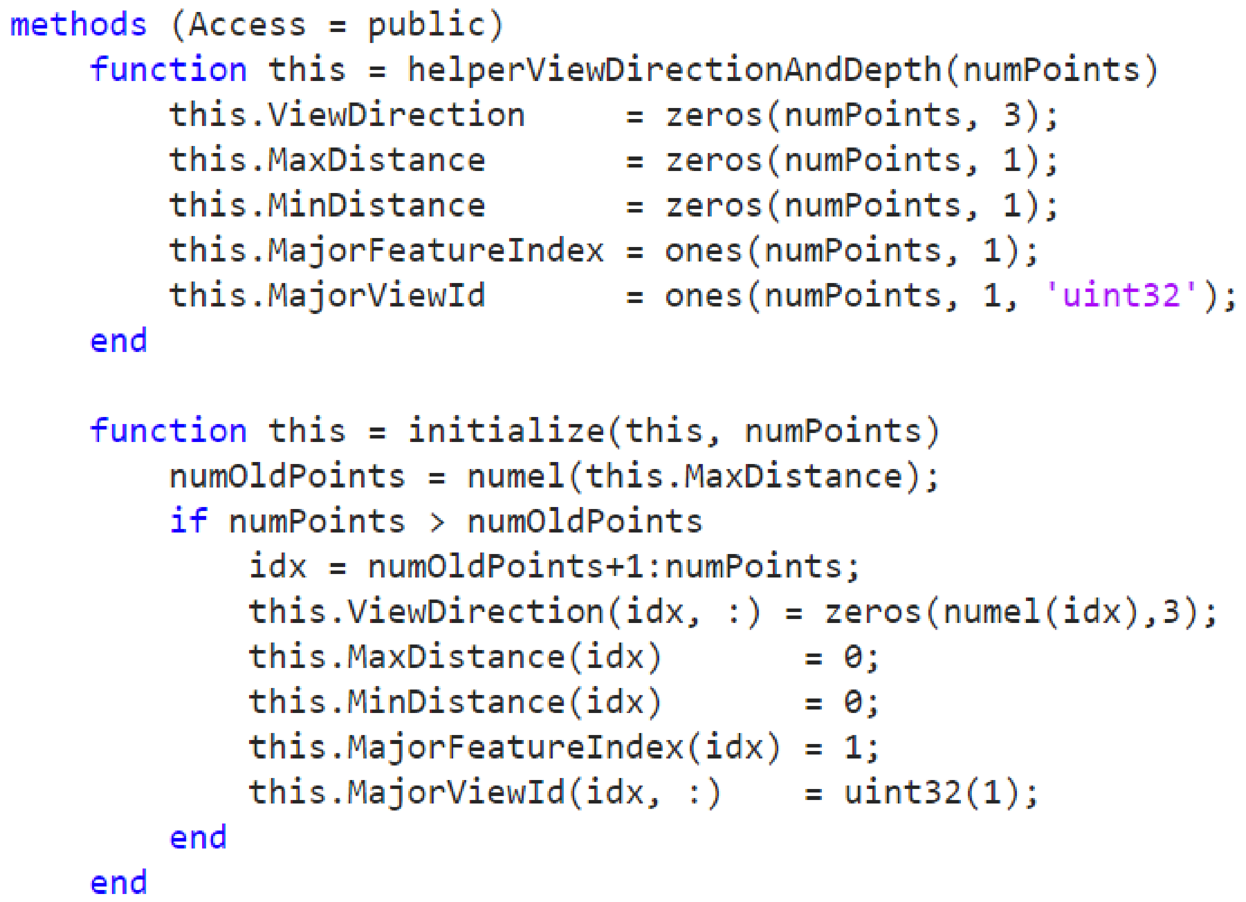
helperViewDirectionAndDepth

```
helperViewDirectionAndDepth Object for storing view direction and depth
Use this object to store map points attributes, such as view direction, predicted depth range,
and the ID of the major view that contains the representative feature descriptor.

viewAndDepth = helperViewDirectionAndDepth(numPoints) returns a
helperViewDirectionAndDepth object.

  helperViewDirectionAndDepth properties:
  ViewDirection        - An M-by-3 matrix representing the view direction
  MaxDistance          - An M-by-1 vector representing the maximum scale invariant distance
  MinDistance          - An M-by-1 vector representing the minimum scale invariant distance
  MajorViewId          - An M-by-1 vector containing the Id of the major view that contains the
                              representative feature descriptor
  MajorFeatureIndex       - An M-by-1 vector containing the index of the representative feature in
the
                              major view
```

helperViewDirectionAndDepth

```
classdef helperViewDirectionAndDepth
    properties
        %ViewDirection  An M-by-3 matrix representing the view direction of
        %   each map point
        ViewDirection

        %MaxDistance An M-by-1 vector representing the maximum scale
        %   invariant distance for each map point
        MaxDistance

        %MinDistance An M-by-1 vector representing the minimum scale
        %   invariant distance for each map point
        MinDistance

        %MajorViewId An M-by-1 vector containing the Id of the major view
        %   that contains the representative feature descriptor for each
        %   map point
        MajorViewId

        %MajorFeatureIndex An M-by-1 vector containing the index of the
        %   representative feature in the major view for each map point
        MajorFeatureIndex
    end
```

helperViewDirectionAndDepth



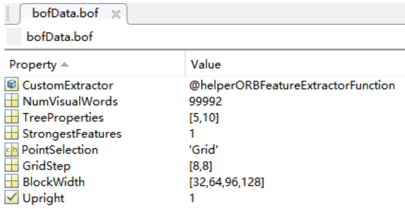
```
methods (Access = public)
    function this = helperViewDirectionAndDepth(numPoints)
        this.ViewDirection     = zeros(numPoints, 3);
        this.MaxDistance       = zeros(numPoints, 1);
        this.MinDistance       = zeros(numPoints, 1);
        this.MajorFeatureIndex = ones(numPoints, 1);
        this.MajorViewId       = ones(numPoints, 1, 'uint32');
    end

    function this = initialize(this, numPoints)
        numOldPoints = numel(this.MaxDistance);
        if numPoints > numOldPoints
            idx = numOldPoints+1:numPoints;
            this.ViewDirection(idx, :) = zeros(numel(idx),3);
            this.MaxDistance(idx)       = 0;
            this.MinDistance(idx)       = 0;
            this.MajorFeatureIndex(idx) = 1;
            this.MajorViewId(idx, :)    = uint32(1);
        end
    end
```

Initialize Place Recognition Database

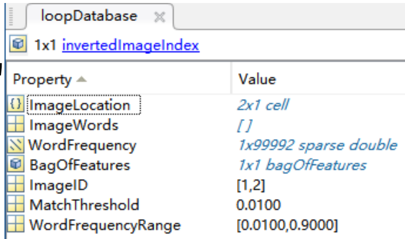
Loop detection is performed using the bags-of-words approach. A visual vocabulary represented as a bagOfFeatures object is created offline with the ORB descriptors extracted from a large set of images in the dataset by calling:
bag = bagOfFeatures(imds,'CustomExtractor', @helperORBFeatureExtractorFunction, 'TreeProperties', [5, 10], 'StrongestFeatures', 1);
where imds is an imageDatastore object storing the training images and helperORBFeatureExtractorFunction is the ORB feature extractor function. See Image Retrieval with Bag of Visual Words for more information.
The loop closure process incrementally builds a database, represented as an invertedImageIndex object, that stores the visual word-to-image mapping bag of ORB features.
Load the bag of features data created offline
bofData   = load('bagOfFeaturesDataSLAM.mat');

Initialize the place recognition database
loopDatabase   = invertedImageIndex(bofData.bof,"SaveFeatureLocations'

Add features of the first two key frames to the database
addImageFeatures(loopDatabase, preFeatures, preViewId);


bofData.bof
bofData.bof

Property	Value
CustomExtractor	@helperORBFeatureExtractorFunction
NumVisualWords	99992
TreeProperties	[5,10]
StrongestFeatures	1
PointSelection	'Grid'
GridStep	[8,8]
BlockWidth	[32,64,96,128]
Upright	1


loopDatabase
1x1 invertedImageIndex

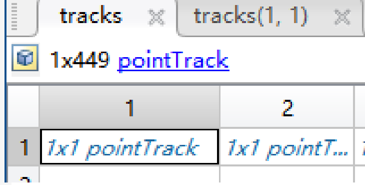
Property	Value
ImageLocation	2x1 cell
ImageWords	[]
WordFrequency	1x99992 sparse double
BagOfFeatures	1x1 bagOfFeatures
ImageID	[1,2]
MatchThreshold	0.0100
WordFrequencyRange	[0.0100,0.9000]

Refine and Visualize the Initial Reconstruction

Refine the initial reconstruction using bundleAdjustment, that optimizes both camera poses and world points to minimize the overall reprojection errors. After the refinement, the attributes of the map points including 3-D locations, view direction, and depth range are updated. You can use helperVisualizeMotionAndStructure to visualize the map points and the camera locations.

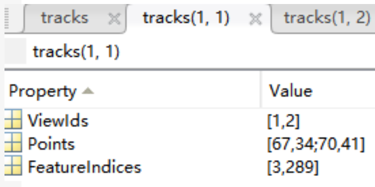
```
% Run full bundle adjustment on the first two key frames
tracks       = findTracks(vSetKeyFrames);
cameraPoses  = poses(vSetKeyFrames);

[refinedPoints, refinedAbsPoses] = bundleAdjustment(xyzWorldPoints, tracks, ...
    cameraPoses, intrinsics, 'FixedViewIDs', 1, ...
    'PointsUndistorted', true, 'AbsoluteTolerance', 1e-7,...
    'RelativeTolerance', 1e-15, 'MaxIteration', 20, ...
    'Solver', 'preconditioned-conjugate-gradient');

% Scale the map and the camera pose using the median depth of map points
medianDepth   = median(vecnorm(refinedPoints.'));
refinedPoints = refinedPoints / medianDepth;

refinedAbsPoses.AbsolutePose(currViewId).Translation = ...
    refinedAbsPoses.AbsolutePose(currViewId).Translation / medianDepth;
relPose.Translation = relPose.Translation/medianDepth;

% Update key frames with the refined poses
vSetKeyFrames = updateView(vSetKeyFrames, refinedAbsPoses);
vSetKeyFrames = updateConnection(vSetKeyFrames, preViewId, currViewId, relPose);

% Update map points with the refined positions
mapPointSet   = updateWorldPoints(mapPointSet, newPointIdx, refinedPoints);

% Update view direction and depth
directionAndDepth = update(directionAndDepth, mapPointSet, vSetKeyFrames.Views, newPointIdx, true);

% Visualize matched features in the current frame
close(hfeature.Parent.Parent);
featurePlot   = helperVisualizeMatchedFeatures(currI, currPoints(indexPairs(:,2)));
```

```
medianDepth =
  single
   28.5561
```


tracks | tracks(1, 1)

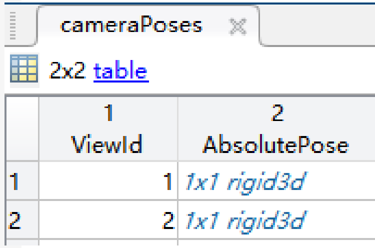
1x449 pointTrack

	1	2
1	1x1 pointTrack	1x1 pointT...


tracks | tracks(1, 1) | tracks(1, 2)

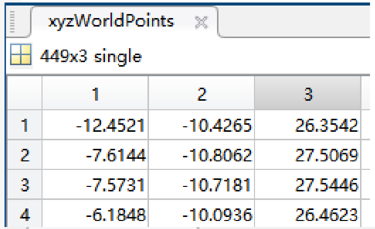
tracks(1, 1)

Property	Value
ViewIds	[1,2]
Points	[67,34;70,41]
FeatureIndices	[3,289]


cameraPoses

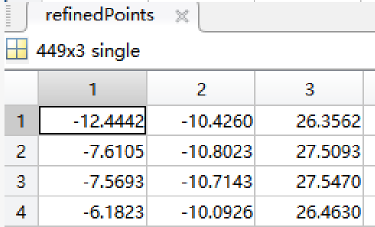
2x2 table

	1 ViewId	2 AbsolutePose
1	1	1x1 rigid3d
2	2	1x1 rigid3d


xyzWorldPoints

449x3 single

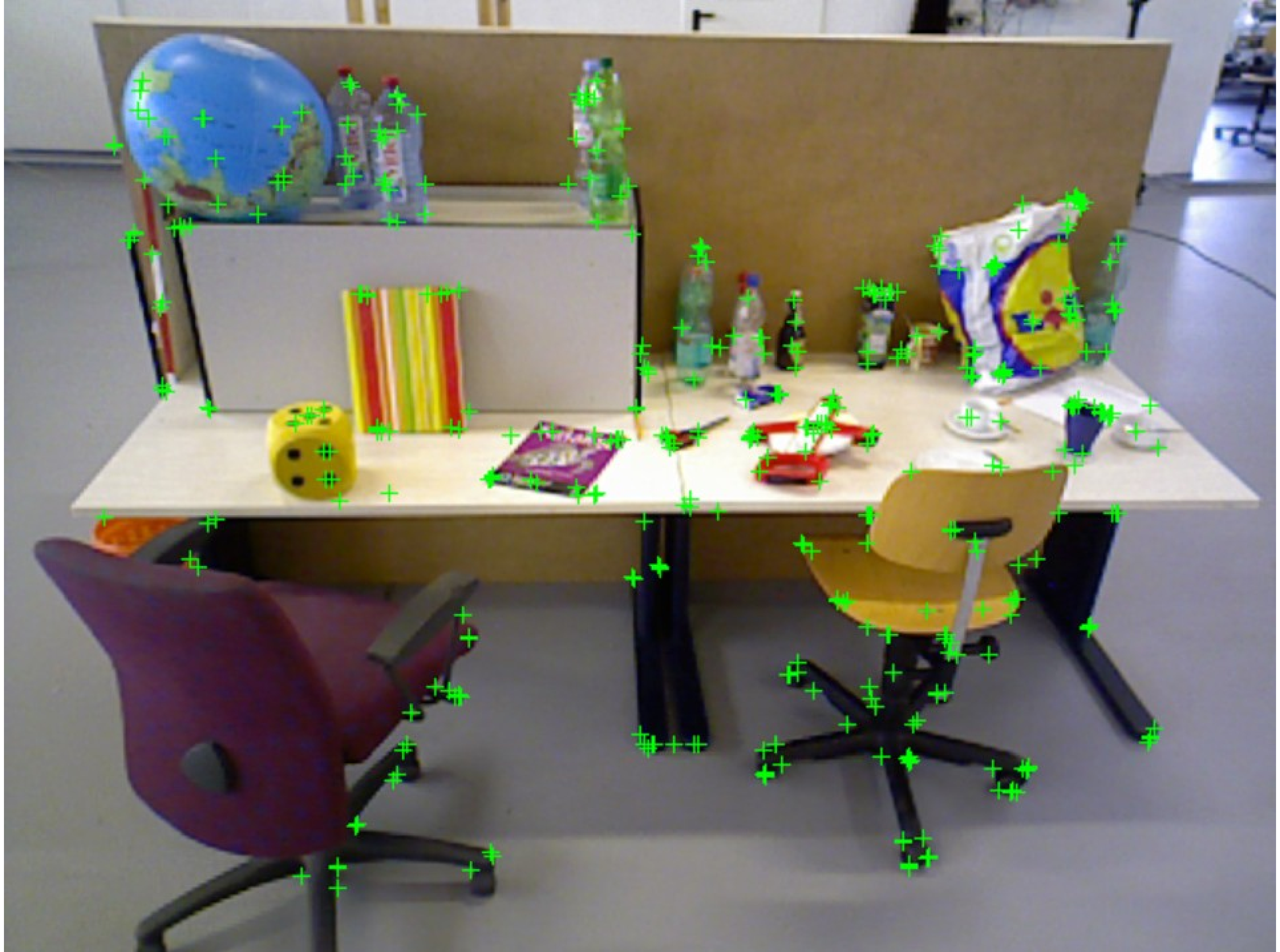
	1	2	3
1	-12.4521	-10.4265	26.3542
2	-7.6144	-10.8062	27.5069
3	-7.5731	-10.7181	27.5446
4	-6.1848	-10.0936	26.4623


refinedPoints

449x3 single

	1	2	3
1	-12.4442	-10.4260	26.3562
2	-7.6105	-10.8023	27.5093
3	-7.5693	-10.7143	27.5470
4	-6.1823	-10.0926	26.4630

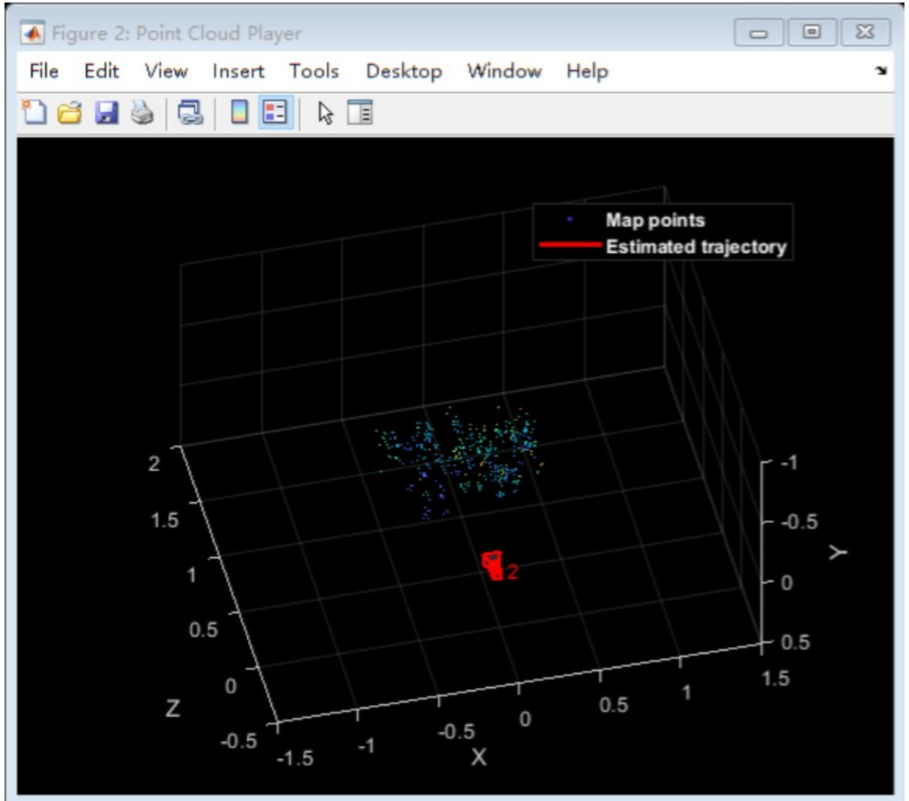
```
classdef helperVisualizeMatchedFeatures < handle
%helperVisualizeMatchedFeatures show the matched features in a frame
    properties (Access = private)
        Image
        Feature
    end

    methods (Access = public)
        function obj = helperVisualizeMatchedFeatures(I, featurePoints)
            locations= featurePoints.Location;

            % Plot image
            hFig  = figure;
            hAxes = newplot(hFig);
            % Set figure visibility and position
            hFig.Visible = 'on';
            movegui(hFig, [300 220]);

            % Show the image
            obj.Image = imshow(I, 'Parent', hAxes, 'Border', 'tight');
            title(hAxes, 'Matched Features in Current Frame');
            hold(hAxes, 'on');
            % Plot features
            plot(featurePoints, hAxes, 'ShowOrientation',false, ...
                'ShowScale',false);
            obj.Feature = findobj(hAxes.Parent,'Type','Line');
        end

        function updatePlot(obj, I, featurePoints)
            locations = featurePoints.Location;
            obj.Image.CData   = I;
            obj.Feature.XData = locations(:,1);
            obj.Feature.YData = locations(:,2);
            drawnow limitrate
        end
    end
end
```



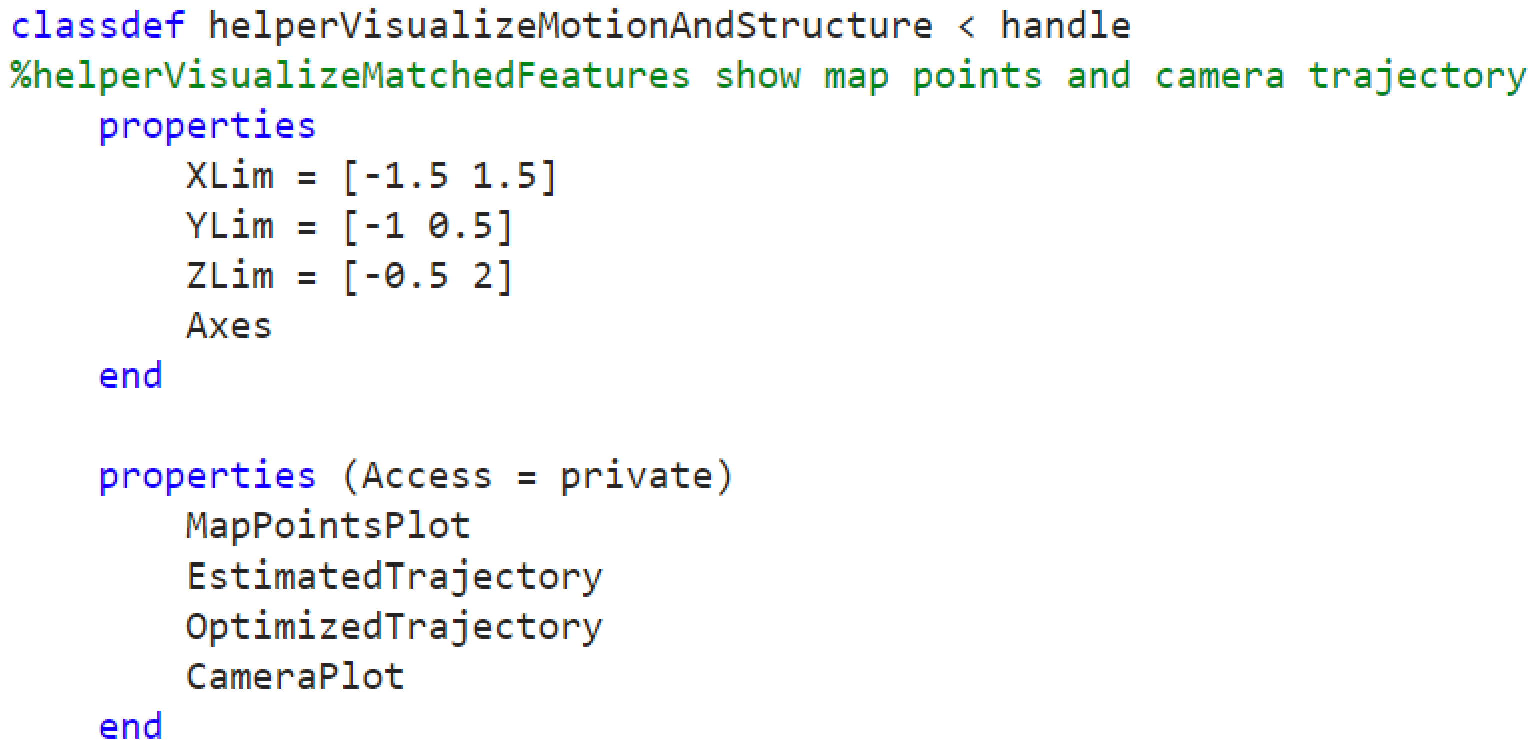
Refine and Visualize the Initial Reconstruction

Visualize initial map points and camera trajectory
mapPlot   = helperVisualizeMotionAndStructure(vSetKeyFrames, mapPointSet);

Show legend
showLegend(mapPlot);



helperVisualizeMotionAndStructure



```
classdef helperVisualizeMotionAndStructure < handle
%helperVisualizeMatchedFeatures show map points and camera trajectory
    properties
        XLim = [-1.5 1.5]
        YLim = [-1 0.5]
        ZLim = [-0.5 2]
        Axes
    end

    properties (Access = private)
        MapPointsPlot
        EstimatedTrajectory
        OptimizedTrajectory
        CameraPlot
    end
```

```matlab
methods (Access = public)
    function obj = helperVisualizeMotionAndStructure(vSetKeyFrames, mapPoints, varargin)

        if nargin > 2
            obj.XLim = varargin{1};
            obj.YLim = varargin{2};
            obj.ZLim = varargin{3};
        end

        [xyzPoints, currPose, trajectory]  = retrievePlottedData(obj, vSetKeyFrames, mapPoints);

        obj.MapPointsPlot = pcplayer(obj.XLim, obj.YLim, obj.ZLim, ...
            'VerticalAxis', 'y', 'VerticalAxisDir', 'down');

        obj.Axes  = obj.MapPointsPlot.Axes;
        obj.MapPointsPlot.view(xyzPoints);
        obj.Axes.Children.DisplayName = 'Map points';

        hold(obj.Axes, 'on');

        % Set figure position on the screen
        movegui(obj.Axes.Parent, [1000 200]);

        % Plot camera trajectory
        obj.EstimatedTrajectory = plot3(obj.Axes, trajectory(:,1), trajectory(:,2), ...
            trajectory(:,3), 'r', 'LineWidth', 2 , 'DisplayName', 'Estimated trajectory');

        % Plot the current cameras
        obj.CameraPlot = plotCamera(currPose, 'Parent', obj.Axes, 'Size', 0.05);
    end
```

```
methods (Access = private)
    function [xyzPoints, currPose, trajectory]  = retrievePlottedData(obj, vSetKeyFrames, mapPoints)
        camPoses    = poses(vSetKeyFrames);
        currPose    = camPoses(end,:); % Contains both ViewId and Pose

        % Ensure the rotation matrix is a rigid transformation
        R = double(currPose.AbsolutePose.Rotation);
        t = double(currPose.AbsolutePose.Translation);
        [U, ~, V] = svd(R);
        currPose.AbsolutePose.T = eye(4);
        currPose.AbsolutePose.T(4, 1:3) = t;
        currPose.AbsolutePose.T(1:3, 1:3) = U * V';

        trajectory  = vertcat(camPoses.AbsolutePose.Translation);
        xyzPoints   = mapPoints.WorldPoints;%(mapPoints.UserData.Validity,:);

        % Only plot the points within the limit
        inPlotRange = xyzPoints(:, 1) > obj.XLim(1) & ...
            xyzPoints(:, 1) < obj.XLim(2) & xyzPoints(:, 2) > obj.YLim(1) & ...
            xyzPoints(:, 2) < obj.YLim(2) & xyzPoints(:, 3) > obj.ZLim(1) & ...
            xyzPoints(:, 3) < obj.ZLim(2);
        xyzPoints   = xyzPoints(inPlotRange, :);
    end
```

Tracking

The tracking process is performed using every frame and determines when to insert a new key frame. To simplify this example, we will terminate the tracking process once a loop closure is found.

ViewId of the current key frame

```
currKeyFrameId   = currViewId;
```

ViewId of the last key frame

```
lastKeyFrameId   = currViewId;
```

Index of the last key frame in the input image sequence

```
lastKeyFrameIdx  = currFrameIdx - 1;
```

Indices of all the key frames in the input image sequence

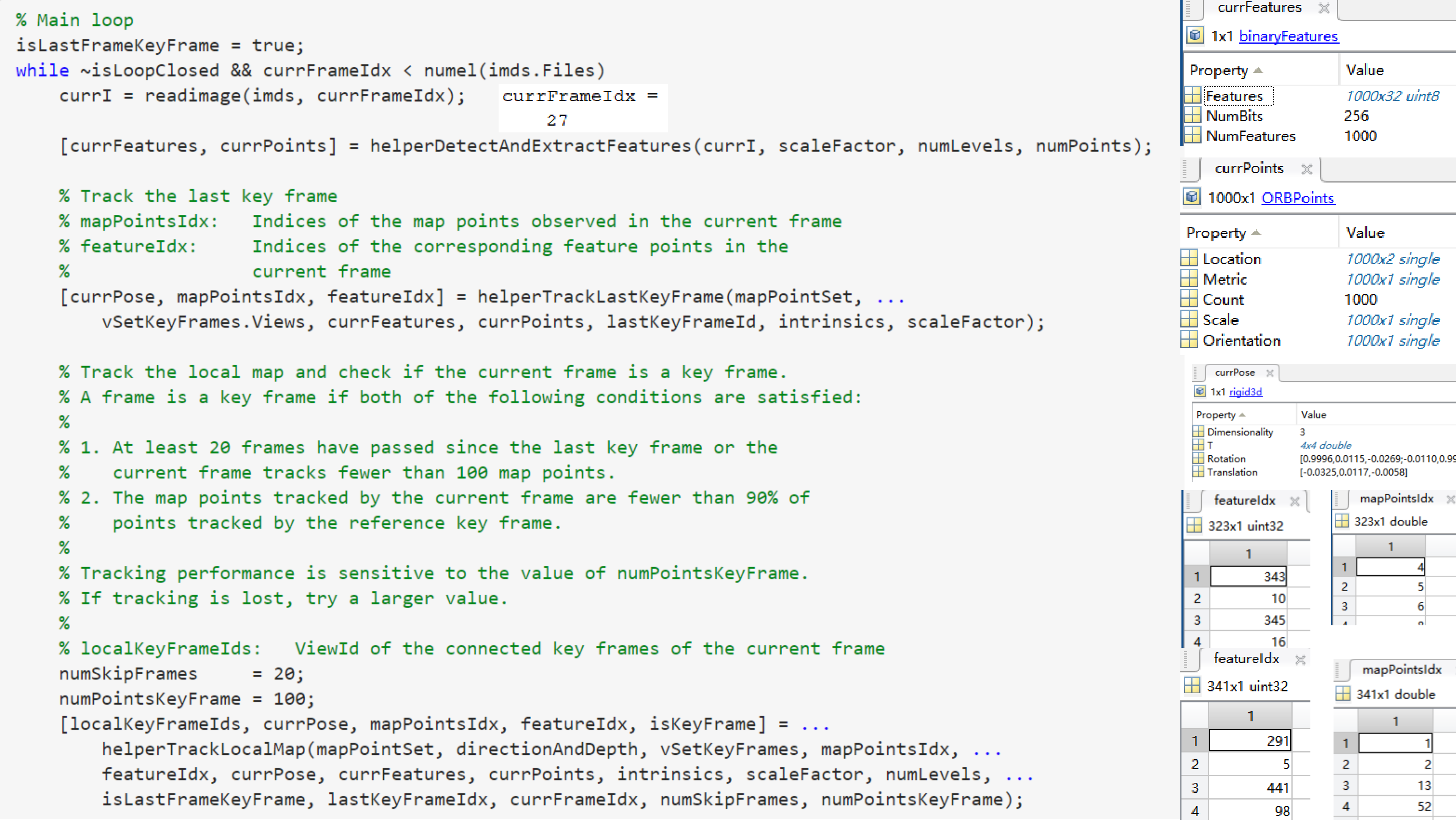
```
addedFramesIdx   = [1; lastKeyFrameIdx];
```

```
isLoopClosed     = false;
```

Tracking

Each frame is processed as follows:

1. ORB features are extracted for each new frame and then matched (using matchFeatures), with features in the last key frame that have known corresponding 3-D map points.
2. Estimate the camera pose with the Perspective-n-Point algorithm using estimateWorldCameraPose.
3. Given the camera pose, project the map points observed by the last key frame into the current frame and search for feature correspondences using matchFeaturesInRadius.
4. With 3-D to 2-D correspondence in the current frame, refine the camera pose by performing a motion-only bundle adjustment using bundleAdjustmentMotion.
5. Project the local map points into the current frame to search for more feature correspondences using matchFeaturesInRadius and refine the camera pose again using bundleAdjustmentMotion.
6. The last step of tracking is to decide if the current frame is a new key frame. If



```matlab
% Main loop
isLastFrameKeyFrame = true;
while ~isLoopClosed && currFrameIdx < numel(imds.Files)
    currI = readimage(imds, currFrameIdx);
    
    [currFeatures, currPoints] = helperDetectAndExtractFeatures(currI, scaleFactor, numLevels, numPoints);
    
    % Track the last key frame
    % mapPointsIdx:   Indices of the map points observed in the current frame
    % featureIdx:     Indices of the corresponding feature points in the
    %                 current frame
    [currPose, mapPointsIdx, featureIdx] = helperTrackLastKeyFrame(mapPointSet, ...
        vSetKeyFrames.Views, currFeatures, currPoints, lastKeyFrameId, intrinsics, scaleFactor);
    
    % Track the local map and check if the current frame is a key frame.
    % A frame is a key frame if both of the following conditions are satisfied:
    %
    % 1. At least 20 frames have passed since the last key frame or the
    %    current frame tracks fewer than 100 map points.
    % 2. The map points tracked by the current frame are fewer than 90% of
    %    points tracked by the reference key frame.
    %
    % Tracking performance is sensitive to the value of numPointsKeyFrame.
    % If tracking is lost, try a larger value.
    %
    % localKeyFrameIds:   ViewId of the connected key frames of the current frame
    numSkipFrames     = 20;
    numPointsKeyFrame = 100;
    [localKeyFrameIds, currPose, mapPointsIdx, featureIdx, isKeyFrame] = ...
        helperTrackLocalMap(mapPointSet, directionAndDepth, vSetKeyFrames, mapPointsIdx, ...
        featureIdx, currPose, currFeatures, currPoints, intrinsics, scaleFactor, numLevels, ...
        isLastFrameKeyFrame, lastKeyFrameIdx, currFrameIdx, numSkipFrames, numPointsKeyFrame);
```

```
currFrameIdx =
    27
```

currFeatures

1x1 binaryFeatures

Property	Value
Features	1000x32 uint8
NumBits	256
NumFeatures	1000

currPoints

1000x1 ORBPoints

Property	Value
Location	1000x2 single
Metric	1000x1 single
Count	1000
Scale	1000x1 single
Orientation	1000x1 single

currPose

1x1 rigid3d

Property	Value
Dimensionality	3
T	4x4 double
Rotation	[0.9996,0.0115,-0.0269;-0.0110,0.99
Translation	[-0.0325,0.0117,-0.0058]

featureIdx

323x1 uint32

	1
1	343
2	10
3	345
4	16

mapPointsIdx

323x1 double

	1
1	4
2	5
3	6

featureIdx

341x1 uint32

	1
1	291
2	5
3	441
4	98

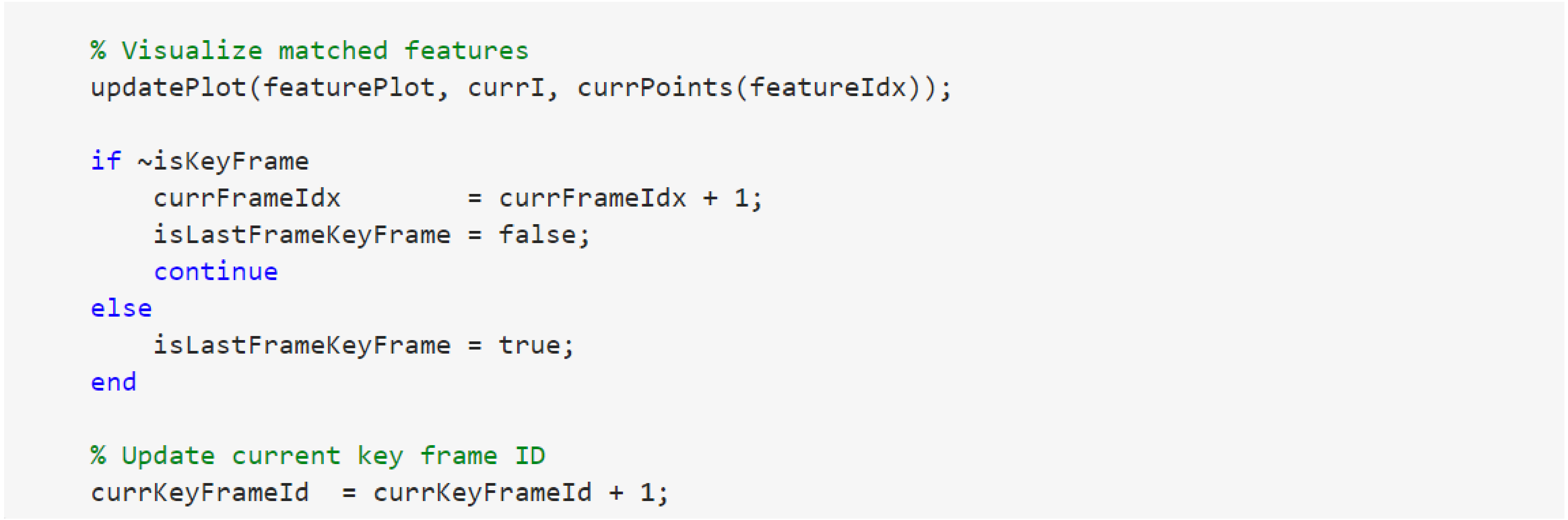
mapPointsIdx

341x1 double

	1
1	1
2	2
3	13
4	52

Tracking

If tracking is lost because not enough number of feature points could be matched, try inserting new key frames more frequently.



```
% Visualize matched features
updatePlot(featurePlot, currI, currPoints(featureIdx));

if ~isKeyFrame
    currFrameIdx        = currFrameIdx + 1;
    isLastFrameKeyFrame = false;
    continue
else
    isLastFrameKeyFrame = true;
end

% Update current key frame ID
currKeyFrameId  = currKeyFrameId + 1;
```

```
helperTrackLastKeyFrame Estimate the camera pose by tracking the last key frame
  [currPose, mapPointIdx, featureIdx] = helperTrackLastKeyFrameStereo(mapPoints,
  views, currFeatures, currPoints, lastKeyFrameId, intrinsics, scaleFactor) estimates
  the camera pose of the current frame by matching features with the
  previous key frame.

  This is an example helper function that is subject to change or removal
  in future releases.

  Inputs
  ------
  mapPoints         - A helperMapPoints objects storing map points
  views             - View attributes of key frames
  currFeatures      - Features in the current frame
  currPoints        - Feature points in the current frame
  lastKeyFrameId    - ViewId of the last key frame
  intrinsics        - Camera intrinsics
  scaleFactor       - scale factor of features

  Outputs
  -------
  currPose          - Estimated camera pose of the current frame
  mapPointIdx       - Indices of map points observed in the current frame
  featureIdx        - Indices of features corresponding to mapPointIdx
```

```matlab
function [currPose, mapPointIdx, featureIdx] = helperTrackLastKeyFrame(...
    mapPoints, views, currFeatures, currPoints, lastKeyFrameId, intrinsics, scaleFactor)

% Match features from the previous key frame with known world locations
[index3d, index2d]    = findWorldPointsInView(mapPoints, lastKeyFrameId);
lastKeyFrameFeatures  = views.Features{lastKeyFrameId}(index2d,:);
lastKeyFramePoints    = views.Points{lastKeyFrameId}(index2d);

indexPairs  = matchFeatures(currFeatures, binaryFeatures(lastKeyFrameFeatures),...
    'Unique', true, 'MaxRatio', 0.9, 'MatchThreshold', 40);

% Estimate the camera pose
matchedImagePoints = currPoints.Location(indexPairs(:,1),:);
matchedWorldPoints = mapPoints.WorldPoints(index3d(indexPairs(:,2)), :);

matchedImagePoints = cast(matchedImagePoints, 'like', matchedWorldPoints);
[worldOri, worldLoc, inlier, status] = estimateWorldCameraPose(...
    matchedImagePoints, matchedWorldPoints, intrinsics, ...
    'Confidence', 95, 'MaxReprojectionError', 3, 'MaxNumTrials', 1e4);

if status
    currPose=[];  mapPointIdx=[];  featureIdx=[];
    return
end

currPose = rigid3d(worldOri, worldLoc);

% Refine camera pose only
currPose = bundleAdjustmentMotion(matchedWorldPoints(inlier,:), ...
    matchedImagePoints(inlier,:), currPose, intrinsics, ...
    'PointsUndistorted', true, 'AbsoluteTolerance', 1e-7,...
    'RelativeTolerance', 1e-15, 'MaxIteration', 20);
    % Search for more matches with the map points in the previous key frame
    xyzPoints = mapPoints.WorldPoints(index3d,:);
    tform = cameraPoseToExtrinsics(currPose);
    [projectedPoints, isInImage] = worldToImage(intrinsics, tform, xyzPoints);
    projectedPoints = projectedPoints(isInImage, :);
    minScales    = max(1, lastKeyFramePoints.Scale(isInImage)/scaleFactor);
    maxScales    = lastKeyFramePoints.Scale(isInImage)*scaleFactor;
    r            = 4;
    searchRadius = r*lastKeyFramePoints.Scale(isInImage);
    indexPairs   = matchFeaturesInRadius(binaryFeatures(lastKeyFrameFeatures(isInImage,:)), ...
        binaryFeatures(currFeatures.Features), currPoints, projectedPoints, searchRadius, ...
        'MatchThreshold', 40, 'MaxRatio', 0.8, 'Unique', true);

    if size(indexPairs, 1) < 20
        currPose=[];  mapPointIdx=[];  featureIdx=[];
        return
    end
    % Filter by scales
    isGoodScale = currPoints.Scale(indexPairs(:, 2)) >= minScales(indexPairs(:, 1)) & ...
        currPoints.Scale(indexPairs(:, 2)) <= maxScales(indexPairs(:, 1));
    indexPairs  = indexPairs(isGoodScale, :);
    % Obtain the index of matched map points and features
    tempIdx            = find(isInImage); % Convert to linear index
    mapPointIdx        = index3d(tempIdx(indexPairs(:,1)));
    featureIdx         = indexPairs(:,2);
    % Refine the camera pose again
    matchedWorldPoints = mapPoints.WorldPoints(mapPointIdx, :);
    matchedImagePoints = currPoints.Location(featureIdx, :);
    currPose = bundleAdjustmentMotion(matchedWorldPoints, matchedImagePoints, ...
        currPose, intrinsics, 'PointsUndistorted', true, 'AbsoluteTolerance', 1e-7,...
        'RelativeTolerance', 1e-15, 'MaxIteration', 20);
    end
```

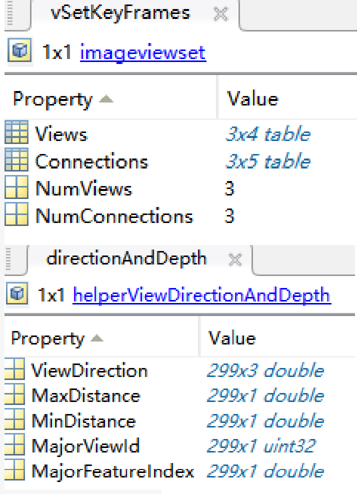
```
function [localKeyFrameIds, currPose, mapPointIdx, featureIdx, isKeyFrame] = ...
  helperTrackLocalMap(mapPoints, directionAndDepth, vSetKeyFrames, mapPointIdx, ...
  featureIdx, currPose, currFeatures, currPoints, intrinsics, scaleFactor, numLevels, ...
  newKeyFrameAdded, lastKeyFrameIndex, currFrameIndex, numSkipFrames, numPointsKeyFrame)
helperTrackLocalMap Refine camera pose by tracking the local map

  Inputs
  ------
  mapPoints         - A worldpointset object storing map points
  directionAndDepth - A helperDirectionAndDepth object of map point attributes
  vSetKeyFrames     - An imageviewset storing key frames
  mapPointsIndices  - Indices of map points observed in the current frame
  featureIndices    - Indices of features in the current frame
                      corresponding to map points denoted by mapPointsIndices
  currPose          - Current camera pose
  currFeatures      - ORB Features in the current frame
  currPoints        - Feature points in the current frame
  intrinsics        - Camera intrinsics
  scaleFactor       - scale factor of features
  numLevels         - number of levels in feature exatraction
  newKeyFrameAdded  - A boolean scalar indicating if a new key frame is
                      added recently
  lastKeyFrameIndex - Frame index of the last key frame
  currFrameIndex    - Frame index of the current frame
  numSkipFrames     - Largest number of frames to skip
  numPointsKeyFrame - Minimum number of points tracked by a key frame

  Outputs
  -------
  localKeyFrameIds  - ViewIds of the local key frames
  currPose          - Refined camera pose of the current frame
  mapPointIdx       - Indices of map points observed in the current frame
  featureIdx        - Indices of features in the current frame corresponding
                      to mapPointIdx
  isKeyFrame        - A boolean scalar indicating if the current frame is
                      a key frame
```

Local Mapping

Local mapping is performed for every key frame. When a new key frame is determined, add it to the key frames and update the attributes of the map points observed by the new key frame. To ensure that mapPointSet contains as few outliers as possible, a valid map point must be observed in at least 3 key frames. New map points are created by triangulating ORB feature points in the current key frame and its connected key frames. For each unmatched feature point in the current key frame, search for a match with other unmatched points in the connected key frames using matchFeatures. The local bundle adjustment refines the pose of the current key frame, the poses of connected key frames, and all the map points observed in these key frames.

Local Mapping

```
% Add the new key frame
[mapPointSet, vSetKeyFrames] = helperAddNewKeyFrame(mapPointSet, vSetKeyFrames, ...
    currPose, currFeatures, currPoints, mapPointsIdx, featureIdx, localKeyFrameIds);

% Remove outlier map points that are observed in fewer than 3 key frames
[mapPointSet, directionAndDepth, mapPointsIdx] = helperCullRecentMapPoints(mapPointSet, ...
    directionAndDepth, mapPointsIdx, newPointIdx);

% Create new map points by triangulation
minNumMatches = 20;
minParallax   = 3;
[mapPointSet, vSetKeyFrames, newPointIdx] = helperCreateNewMapPoints(mapPointSet, vSetKeyFrames, ...
    currKeyFrameId, intrinsics, scaleFactor, minNumMatches, minParallax);

% Update view direction and depth
directionAndDepth = update(directionAndDepth, mapPointSet, vSetKeyFrames.Views, ...
    [mapPointsIdx; newPointIdx], true);

% Local bundle adjustment
[mapPointSet, directionAndDepth, vSetKeyFrames, newPointIdx] = helperLocalBundleAdjustment( ...
    mapPointSet, directionAndDepth, vSetKeyFrames, ...
    currKeyFrameId, intrinsics, newPointIdx);

% Visualize 3D world points and camera trajectory
updatePlot(mapPlot, vSetKeyFrames, mapPointSet);
```



vSetKeyFrames

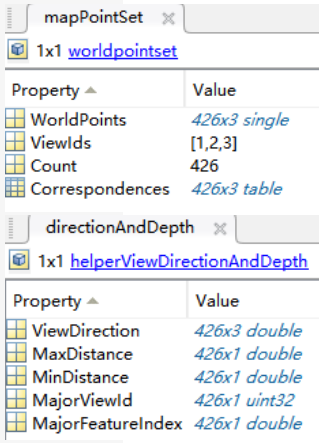
1x1 imageviewset

Property	Value
Views	3x4 table
Connections	3x5 table
NumViews	3
NumConnections	3

directionAndDepth

1x1 helperViewDirectionAndDepth

Property	Value
ViewDirection	299x3 double
MaxDistance	299x1 double
MinDistance	299x1 double
MajorViewId	299x1 uint32
MajorFeatureIndex	299x1 double



mapPointSet

1x1 worldpointset

Property	Value
WorldPoints	426x3 single
ViewIds	[1,2,3]
Count	426
Correspondences	426x3 table

directionAndDepth

1x1 helperViewDirectionAndDepth

Property	Value
ViewDirection	426x3 double
MaxDistance	426x1 double
MinDistance	426x1 double
MajorViewId	426x1 uint32
MajorFeatureIndex	426x1 double

```
function [mapPoints, vSetKeyFrames] = helperAddNewKeyFrame(mapPoints, vSetKeyFrames,...
    cameraPose, currFeatures, currPoints, mapPointsIndices, featureIndices, keyFramesIndices)
%helperAddNewKeyFrame add key frames to the key frame set

viewId = vSetKeyFrames.Views.ViewId(end)+1;

vSetKeyFrames = addView(vSetKeyFrames, viewId, cameraPose,...
    'Features', currFeatures.Features, ...
    'Points', currPoints);

viewsAbsPoses = vSetKeyFrames.Views.AbsolutePose;

for i = 1:numel(keyFramesIndices)
    localKeyFrameId = keyFramesIndices(i);
    [index3d, index2d] = findWorldPointsInView(mapPoints, localKeyFrameId);
    [~, ia, ib] = intersect(index3d, mapPointsIndices, 'stable');

    prePose   = viewsAbsPoses(localKeyFrameId);
    relPose = rigid3d(cameraPose.Rotation*prePose.Rotation', ...
        (cameraPose.Translation-prePose.Translation)*prePose.Rotation');

    if numel(ia) > 5
        vSetKeyFrames = addConnection(vSetKeyFrames, localKeyFrameId, viewId, relPose, ...
            'Matches', [index2d(ia),featureIndices(ib)]);
    end
end

mapPoints = addCorrespondences(mapPoints, viewId, mapPointsIndices, ...
    featureIndices);
end
```

`helperCullRecentMapPoints` cull recently added map points.

```
function [mapPointSet, directionAndDepth, mapPointsIdx] = helperCullRecentMapPoints(mapPointSet, directionAndDepth, mapPointsIdx, newPointIdx)
outlierIdx    = setdiff(newPointIdx, mapPointsIdx);
if ~isempty(outlierIdx)
    mapPointSet   = removeWorldPoints(mapPointSet, outlierIdx);
    directionAndDepth = remove(directionAndDepth, outlierIdx);
    mapPointsIdx  = mapPointsIdx - arrayfun(@(x) nnz(x>outlierIdx), mapPointsIdx);
end
end
```

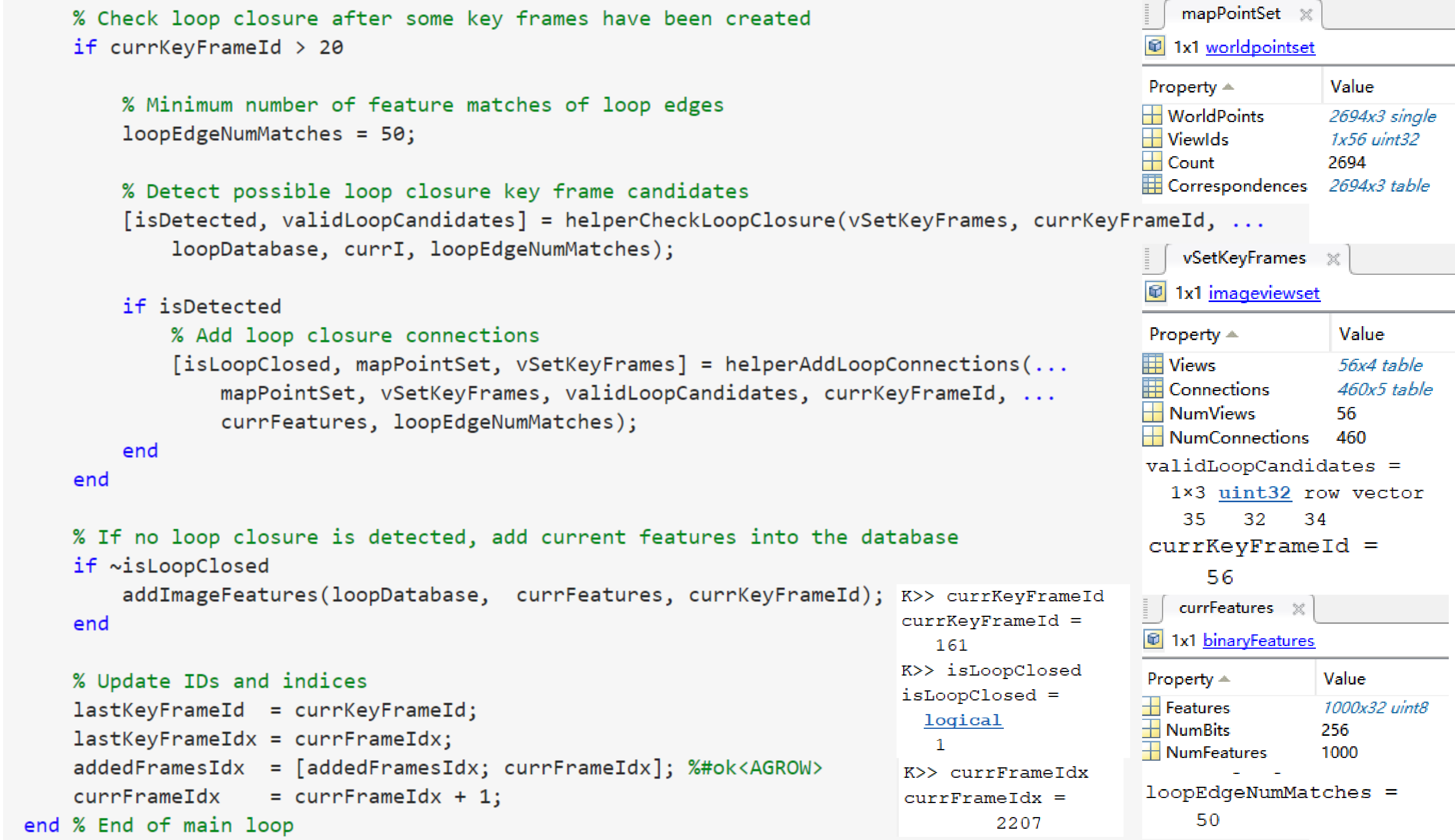
```
function [mapPoints, vSetKeyFrames, recentPointIdx] = helperCreateNewMapPoints(...
    mapPoints, vSetKeyFrames, currKeyFrameId, intrinsics, scaleFactor, minNumMatches, minParallax)
%helperCreateNewMapPoints creates new map points by triangulating matched
%   feature points in the current key frame and the connected key frames.
```

```
function [mapPoints, directionAndDepth, vSetKeyFrames, newPointIdx] = helperLocalBundleAdjustment(mapPoints, .
    directionAndDepth, vSetKeyFrames, currKeyFrameId, intrinsics, newPointIdx)
%helperLocalBundleAdjustment refine the pose of the current key frame and
%   the map of the surrounding scene.

% Connected key frames of the current key frame
covisViews    = connectedViews(vSetKeyFrames, currKeyFrameId);
covisViewsIds = covisViews.ViewId;

% Identify the fixed key frames that are connected to the connected
% key frames of the current key frame
fixedViewIds  = [];
for i = 1:numel(covisViewsIds)
    if numel(fixedViewIds) > 10
        break
    end
    tempViews = connectedViews(vSetKeyFrames, covisViewsIds(i));
    tempId    = tempViews.ViewId;

    for j = 1:numel(tempId)
        if ~ismember(tempId(j), [fixedViewIds; currKeyFrameId; covisViewsIds])
            fixedViewIds = [fixedViewIds; tempId(j)]; %#ok<AGROW>
            if numel(fixedViewIds) > 10
                break
            end
        end
    end
end
```

```
% Always fix the first two key frames
fixedViewIds = [fixedViewIds; intersect(covisViewsIds, [1 2])];

refinedKeyFrameIds = [unique([fixedViewIds; covisViewsIds]); currKeyFrameId];

% Refine local key frames and map points
[mapPoints, vSetKeyFrames, mapPointIdx, reprojectionErrors] = bundleAdjustment(...
    mapPoints, vSetKeyFrames, refinedKeyFrameIds, intrinsics, 'FixedViewIDs', fixedViewIds, ...
    'PointsUndistorted', true, 'AbsoluteTolerance', 1e-7,...
    'RelativeTolerance', 1e-16, 'Solver', 'preconditioned-conjugate-gradient', ...
    'MaxIteration', 10);

maxError   = 6;
isInlier   = reprojectionErrors < maxError;
inlierIdx  = mapPointIdx(isInlier);
outlierIdx = mapPointIdx(~isInlier);

directionAndDepth = update(directionAndDepth, mapPoints, vSetKeyFrames.Views, inlierIdx, false);

newPointIdx = setdiff(newPointIdx, outlierIdx);

% Update map points and key frames
if ~isempty(outlierIdx)
    mapPoints = removeWorldPoints(mapPoints, outlierIdx);
    directionAndDepth = remove(directionAndDepth, outlierIdx);
end

newPointIdx = newPointIdx - arrayfun(@(x) nnz(x>outlierIdx), newPointIdx);
end
```

Loop Closure

The loop closure detection step takes the current key frame processed by the local mapping process and tries to detect and close the loop. Loop candidates are identified by querying images in the database that are visually similar to the current key frame using evaluateImageRetrieval. A candidate key frame is valid if it is not connected to the last key frame and three of its neighbor key frames are loop candidates.
When a valid loop candidate is found, use estimateGeometricTransform3D to compute the relative pose between the loop candidate frame and the current key frame. The relative pose represents a 3-D similarity transformation stored in an affine3d object. Then add the loop connection with the relative pose and update mapPointSet and vSetKeyFrames.



```
    % Check loop closure after some key frames have been created
    if currKeyFrameId > 20

        % Minimum number of feature matches of loop edges
        loopEdgeNumMatches = 50;

        % Detect possible loop closure key frame candidates
        [isDetected, validLoopCandidates] = helperCheckLoopClosure(vSetKeyFrames, currKeyFrameId, ...
            loopDatabase, currI, loopEdgeNumMatches);

        if isDetected
            % Add loop closure connections
            [isLoopClosed, mapPointSet, vSetKeyFrames] = helperAddLoopConnections(...
                mapPointSet, vSetKeyFrames, validLoopCandidates, currKeyFrameId, ...
                currFeatures, loopEdgeNumMatches);
        end
    end

    % If no loop closure is detected, add current features into the database
    if ~isLoopClosed
        addImageFeatures(loopDatabase,  currFeatures, currKeyFrameId);
    end

    % Update IDs and indices
    lastKeyFrameId  = currKeyFrameId;
    lastKeyFrameIdx = currFrameIdx;
    addedFramesIdx  = [addedFramesIdx; currFrameIdx]; %#ok<AGROW>
    currFrameIdx    = currFrameIdx + 1;
end % End of main loop
```

```
K>> currKeyFrameId
currKeyFrameId =
   161
K>> isLoopClosed
isLoopClosed =
  logical
   1
K>> currFrameIdx
currFrameIdx =
        2207
```

mapPointSet

1x1 worldpointset

Property	Value
WorldPoints	2694x3 single
ViewIds	1x56 uint32
Count	2694
Correspondences	2694x3 table

vSetKeyFrames

1x1 imageviewset

Property	Value
Views	56x4 table
Connections	460x5 table
NumViews	56
NumConnections	460

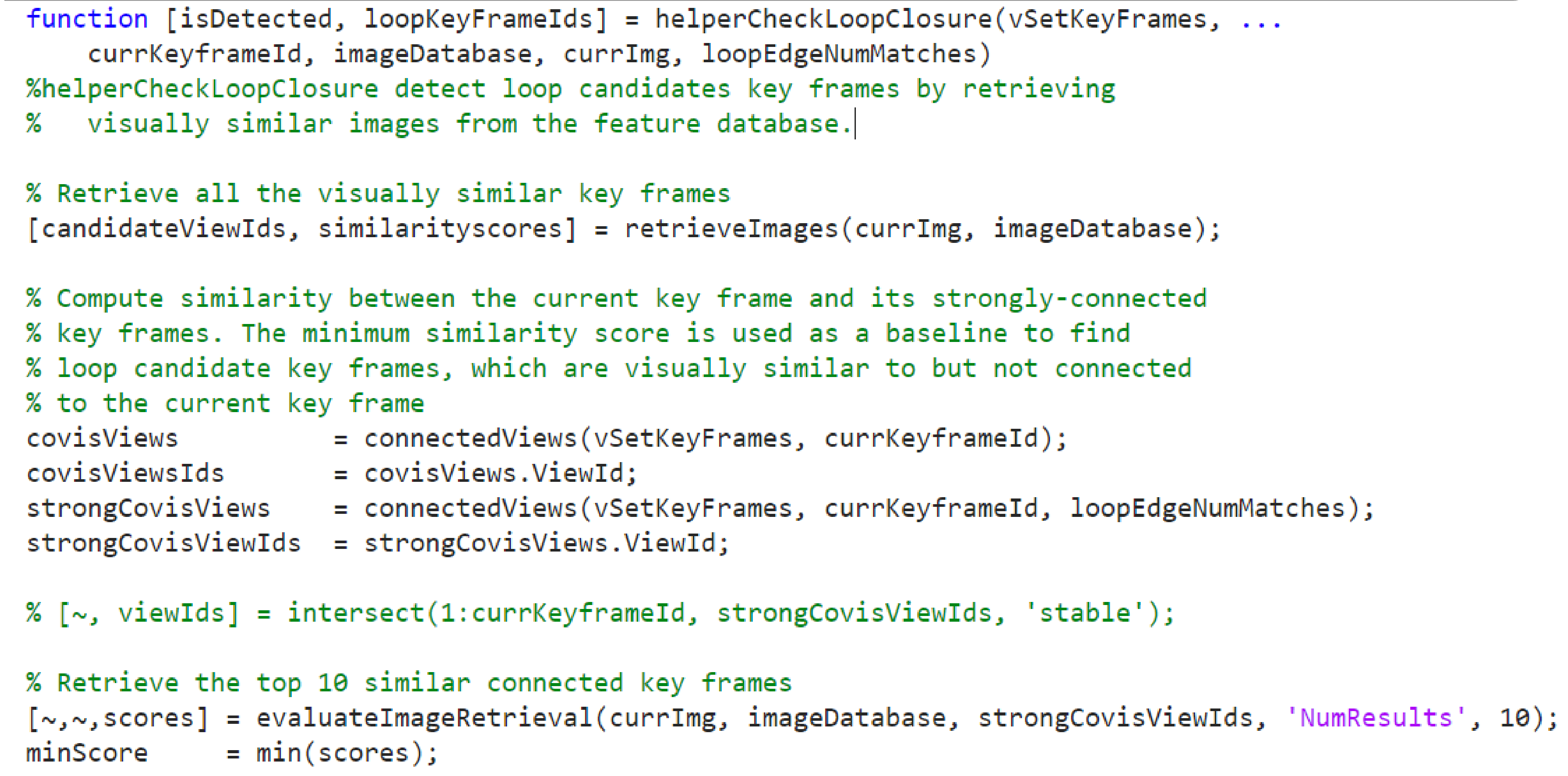
```
validLoopCandidates =
  1×3 uint32 row vector
   35   32   34
currKeyFrameId =
    56
```

currFeatures

1x1 binaryFeatures

Property	Value
Features	1000x32 uint8
NumBits	256
NumFeatures	1000

```
loopEdgeNumMatches =
    50
```



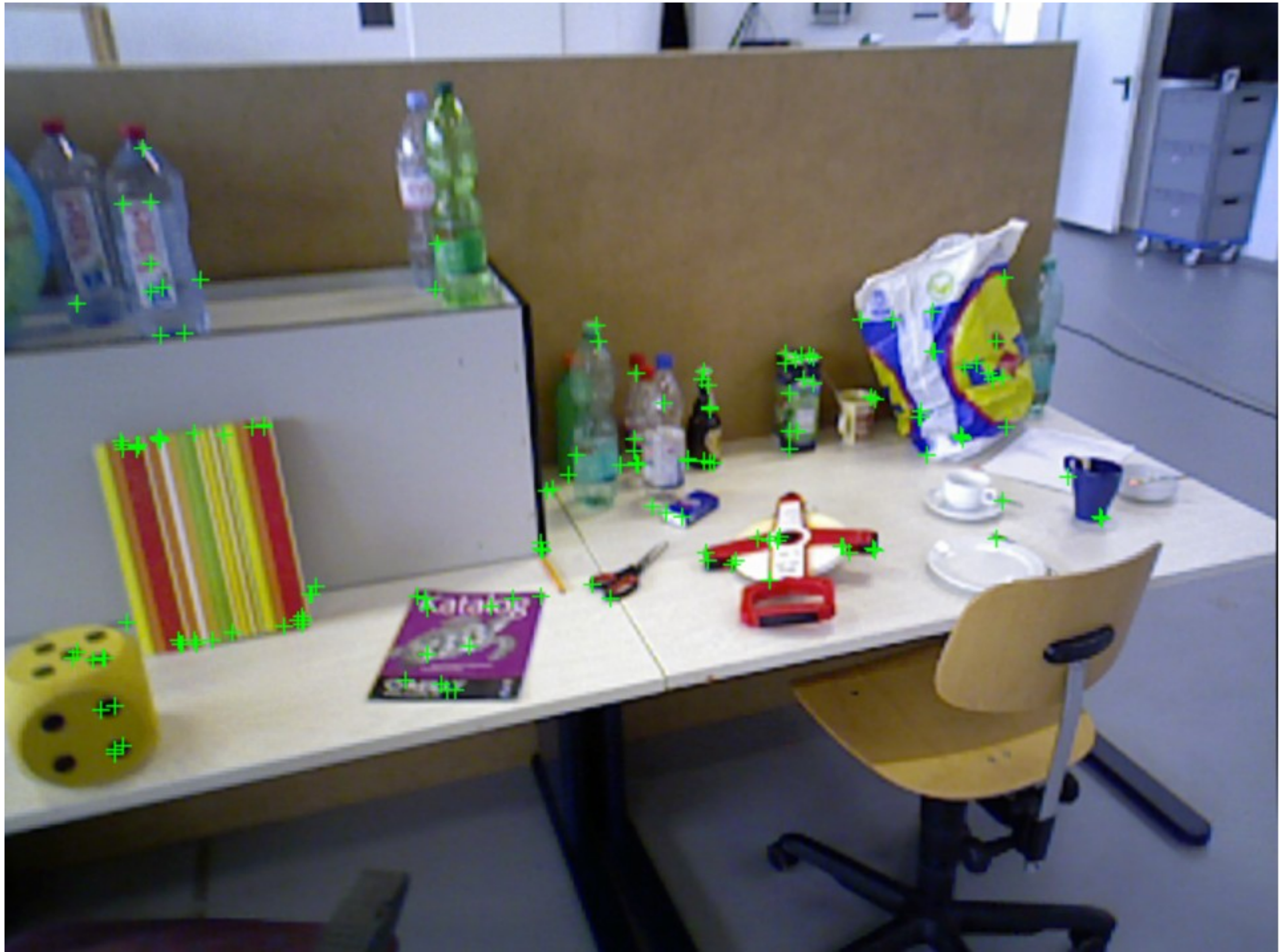
```
function [isDetected, loopKeyFrameIds] = helperCheckLoopClosure(vSetKeyFrames, ...
    currKeyframeId, imageDatabase, currImg, loopEdgeNumMatches)
%helperCheckLoopClosure detect loop candidates key frames by retrieving
%   visually similar images from the feature database.

% Retrieve all the visually similar key frames
[candidateViewIds, similarityscores] = retrieveImages(currImg, imageDatabase);

% Compute similarity between the current key frame and its strongly-connected
% key frames. The minimum similarity score is used as a baseline to find
% loop candidate key frames, which are visually similar to but not connected
% to the current key frame
covisViews           = connectedViews(vSetKeyFrames, currKeyframeId);
covisViewsIds        = covisViews.ViewId;
strongCovisViews     = connectedViews(vSetKeyFrames, currKeyframeId, loopEdgeNumMatches);
strongCovisViewIds   = strongCovisViews.ViewId;

% [~, viewIds] = intersect(1:currKeyframeId, strongCovisViewIds, 'stable');

% Retrieve the top 10 similar connected key frames
[~,~,scores] = evaluateImageRetrieval(currImg, imageDatabase, strongCovisViewIds, 'NumResults', 10);
minScore     = min(scores);
```

```
[loopKeyFrameIds,ia] = setdiff(candidateViewIds, covisViewsIds, 'stable');

% Scores of non-connected key frames
candidateScores  = similarityscores(ia); % Descending

if ~isempty(ia)
    bestScore       = candidateScores(1);

    % Score must be higher than the 75% of the best score
    isValid         = candidateScores > max(bestScore*0.75, minScore);

    loopKeyFrameIds = loopKeyFrameIds(isValid);
else
    loopKeyFrameIds = [];
end

% Loop candidates need to be consecutively detected
minNumCandidates = 3; % At least 3 candidates are found
if size(loopKeyFrameIds,1) >= minNumCandidates
    groups = nchoosek(loopKeyFrameIds, minNumCandidates);
    consecutiveGroups = groups(max(groups,[],2) - min(groups,[],2) < 4, :);
    if ~isempty(consecutiveGroups) % Consecutive candidates are found
        loopKeyFrameIds = consecutiveGroups(1,:);
        isDetected = true;
    else
        isDetected = false;
    end
else
    isDetected = false;
end
end
```

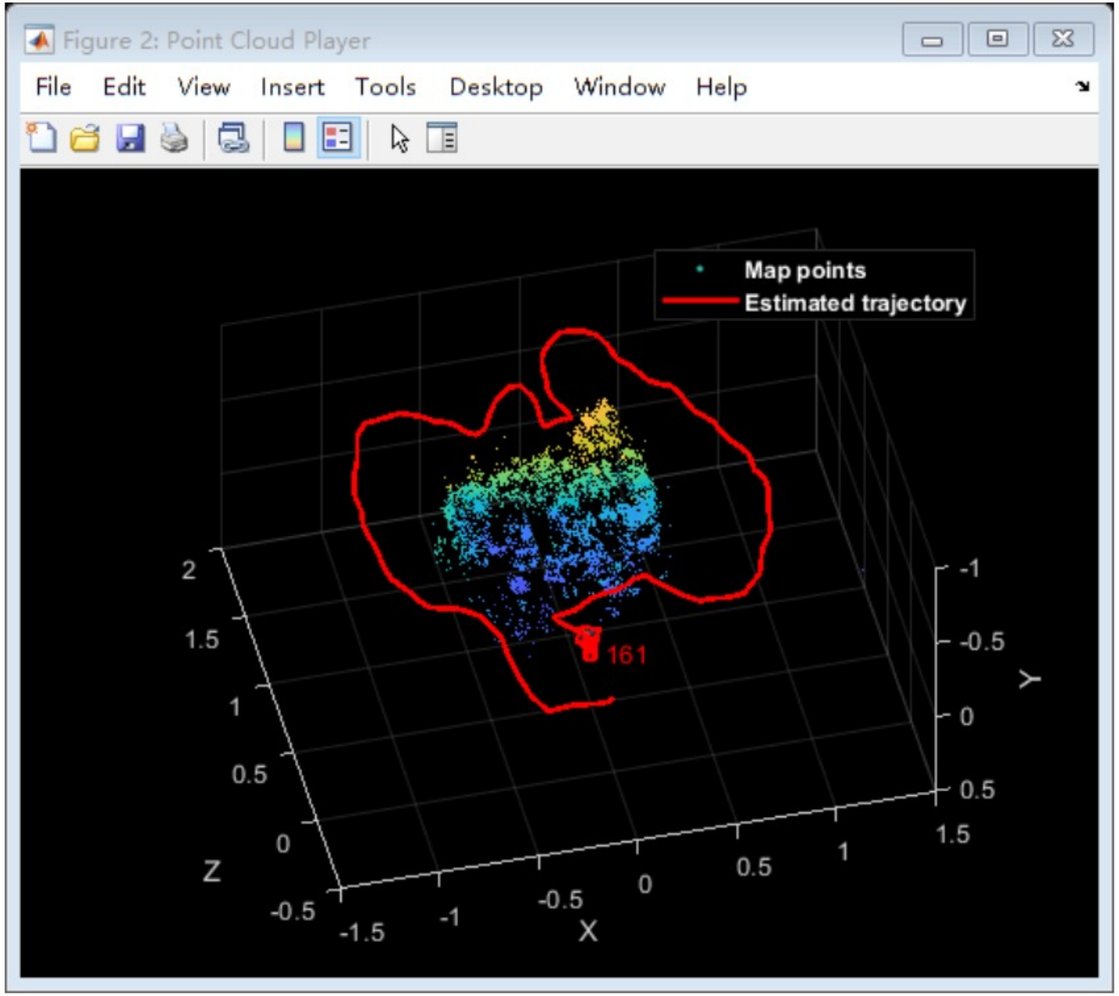
```matlab
function [isLoopClosed, mapPoints, vSetKeyFrames] = helperAddLoopConnections(...
    mapPoints, vSetKeyFrames, loopCandidates, currKeyFrameId, currFeatures, ...
    loopEdgeNumMatches)
%helperAddLoopConnections add connections between the current key frame and
%   the valid loop candidate key frames. A loop candidate is valid if it has
%   enough covisible map points with the current key frame.

loopClosureEdge = [];

numCandidates   = size(loopCandidates,1);
[index3d1, index2d1] = findWorldPointsInView(mapPoints, currKeyFrameId);
validFeatures1  = currFeatures.Features(index2d1, :);

for k = 1 : numCandidates
    [index3d2, index2d2] = findWorldPointsInView(mapPoints, loopCandidates(k));
    allFeatures2   = vSetKeyFrames.Views.Features{loopCandidates(k)};
    validFeatures2 = allFeatures2(index2d2, :);

    indexPairs = matchFeatures(binaryFeatures(validFeatures1), binaryFeatures(validFeatures2), ...
        'Unique', true, 'MaxRatio', 0.9, 'MatchThreshold', 40);

    % Check if all the candidate key frames have strong connection with the
    % current keyframe
    if size(indexPairs, 1) < loopEdgeNumMatches
        continue
    end
```

```
        % Estimate the relative pose of the current key frame with respect to the
        % loop candidate keyframe with the highest similarity score

        worldPoints1 = mapPoints.WorldPoints(index3d1(indexPairs(:, 1)), :);
        worldPoints2 = mapPoints.WorldPoints(index3d2(indexPairs(:, 2)), :);

        pose1 = vSetKeyFrames.Views.AbsolutePose(end);
        pose2 = vSetKeyFrames.Views.AbsolutePose(loopCandidates(k));
        [rotation1, translation1] = cameraPoseToExtrinsics(pose1.Rotation, pose1.Translation);
        [rotation2, translation2] = cameraPoseToExtrinsics(pose2.Rotation, pose2.Translation);

        worldPoints1InCamera1 = worldPoints1 * rotation1 + translation1;
        worldPoints2InCamera2 = worldPoints2 * rotation2 + translation2;

        w = warning('off','all');
        [tform, inlierIndex] = estimateGeometricTransform3D(...
            worldPoints1InCamera1, worldPoints2InCamera2, 'similarity', 'MaxDistance', 0.1);
        warning(w);

        % Add connection between the current key frame and the loop key frame
        matches = uint32([index2d2(indexPairs(inlierIndex, 2)), index2d1(indexPairs(inlierIndex, 1))]);
        vSetKeyFrames = addConnection(vSetKeyFrames, loopCandidates(k), currKeyFrameId, tform, 'Matches', matches);
        disp(['Loop edge added between keyframe: ', num2str(loopCandidates(k)), ' and ', num2str(currKeyFrameId)]);

        % Fuse co-visible map points
        matchedIndex3d1 = index3d1(indexPairs(inlierIndex, 1));
        matchedIndex3d2 = index3d2(indexPairs(inlierIndex, 2));
        mapPoints = updateWorldPoints(mapPoints, matchedIndex3d1, mapPoints.WorldPoints(matchedIndex3d2, :));

        loopClosureEdge = [loopClosureEdge; loopCandidates(k), currKeyFrameId];
    end
    isLoopClosed = ~isempty(loopClosureEdge);
    end
```

Loop Closure



Loop Closure

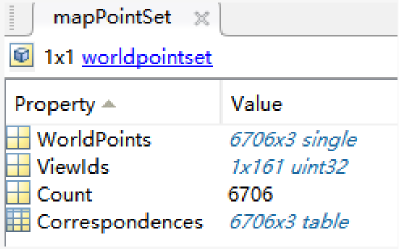


```
Loop edge added between keyframe: 5 and 161
```

Loop Closure

Finally, a similarity pose graph optimization is performed over the essential graph in vSetKeyFrames to correct the drift of camera poses. The essential graph is created internally by removing connections with fewer than minNumMatches matches in the covisibility graph. After similarity pose graph optimization, update the 3-D locations of the map points using the optimized poses and the associated scales

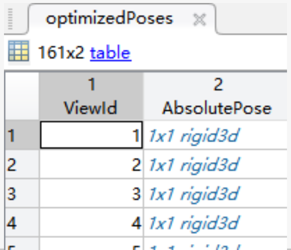
```
% Optimize the poses
minNumMatches      = 30;
[vSetKeyFramesOptim, poseScales] = optimizePoses(vSetKeyFrames, minNumMatches, 'Tolerance', 1e-16);

% Update map points after optimizing the poses
mapPointSet = helperUpdateGlobalMap(mapPointSet, directionAndDepth, ...
    vSetKeyFrames, vSetKeyFramesOptim, poseScales);

updatePlot(mapPlot, vSetKeyFrames, mapPointSet);

% Plot the optimized camera trajectory
optimizedPoses  = poses(vSetKeyFramesOptim);
plotOptimizedTrajectory(mapPlot, optimizedPoses)

% Update legend
showLegend(mapPlot);
```



mapPointSet

1x1 worldpointset

Property	Value
WorldPoints	6706x3 single
ViewIds	1x161 uint32
Count	6706
Correspondences	6706x3 table



optimizedPoses

161x2 table

	1 ViewId	2 AbsolutePose
1	1	1x1 rigid3d
2	2	1x1 rigid3d
3	3	1x1 rigid3d
4	4	1x1 rigid3d

helperUpdateGlobalMap

helperUpdateGlobalMap update 3-D locations of map points after pose graph optimization

```
function [mapPointSet, directionAndDepth] = helperUpdateGlobalMap(...
    mapPointSet, directionAndDepth, vSetKeyFrames, vSetKeyFramesOptim, poseScales)
%helperUpdateGlobalMap update map points after pose graph optimization
posesOld     = vSetKeyFrames.Views.AbsolutePose;
posesNew     = vSetKeyFramesOptim.Views.AbsolutePose;
positionsOld = mapPointSet.WorldPoints;
positionsNew = positionsOld;
indices     = 1:mapPointSet.Count;

% Update world location of each map point based on the new absolute pose of
% the corresponding major view
for i = indices
    majorViewIds = directionAndDepth.MajorViewId(i);
    poseNew = posesNew(majorViewIds).T;
    poseNew(1:3, 1:3) = poseNew(1:3, 1:3) * poseScales(majorViewIds);
    tform = posesOld(majorViewIds).T \ poseNew;
    positionsNew(i, :) = positionsOld(i, :) * tform(1:3,1:3) + tform(4, 1:3);
end
mapPointSet = updateWorldPoints(mapPointSet, indices, positionsNew);
end
```

Compare with the Ground Truth

You can compare the optimized camera trajectory with the ground truth to evaluate the accuracy of ORB-SLAM. The downloaded data contains a groundtruth.txt file that stores the ground truth of camera pose of each frame. The data has been saved in the form of a MAT-file. You can also calculate the root-mean-square-error (RMSE) of trajectory estimates.

Load ground truth

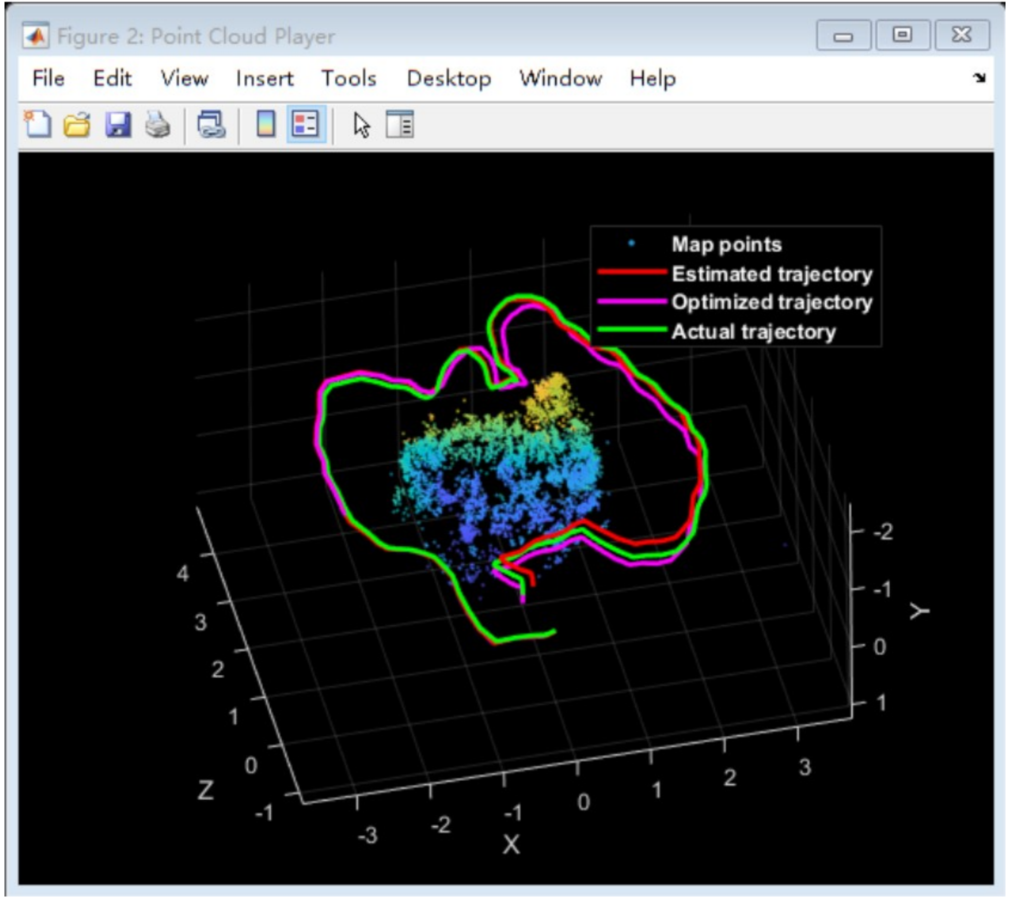
```
gTruthData = load('orbslamGroundTruth.mat');
gTruth     = gTruthData.gTruth;
```

Plot the actual camera trajectory

```
plotActualTrajectory(mapPlot, gTruth(addedFramesIdx), optimizedPoses);
```

Show legend

```
showLegend(mapPlot);
```



```
% Evaluate tracking accuracy
helperEstimateTrajectoryError(gTruth(addedFramesIdx), optimizedPoses);
```

```
Absolute RMSE for key frame trajectory (m): 0.17916
```

`helperEstimateTrajectoryError` calculate the tracking error.

```
function rmse = helperEstimateTrajectoryError(gTruth, cameraPoses)
locations       = vertcat(cameraPoses.AbsolutePose.Translation);
gLocations      = vertcat(gTruth.Translation);
scale           = median(vecnorm(gLocations, 2, 2))/ median(vecnorm(locations, 2, 2));
scaledLocations = locations * scale;

rmse = sqrt(mean( sum((scaledLocations - gLocations).^2, 2) ));
disp(['Absolute RMSE for key frame trajectory (m): ', num2str(rmse)]);
end
```

Compare with the Ground Truth

This concludes an overview of how to build a map of an indoor environment and estimate the trajectory of the camera using ORB-SLAM. You can test the visual SLAM pipeline with a different dataset by tuning the following parameters:

- numPoints: For image resolution of 480x640 pixels set numPoints to be 1000. For higher resolutions, such as 720 $\times$ 1280, set it to 2000. Larger values require more time in feature extraction.
- numSkipFrames: For frame rate of 30fps, set numSkipFrames to be 20. For a slower frame rate, set it to be a smaller value. Increasing numSkipFrames improves the tracking speed, but may result in tracking lost when the camera motion is fast.

Supporting Functions

Short helper functions are included below. Larger function are included in separate files.

- **helperAddLoopConnections** add connections between the current keyframe and the valid loop candidate.
- **helperAddNewKeyFrame** add key frames to the key frame set.
- **helperCheckLoopClosure** detect loop candidates key frames by retrieving visually similar images from the database.
- **helperCreateNewMapPoints** create new map points by triangulation.
- **helperLocalBundleAdjustment** refine the pose of the current key frame and the map of the surrrounding scene.
- **helperORBFeatureExtractorFunction** implements the ORB feature extraction used in bagOfFeatures.
- **helperTrackLastKeyFrame** estimate the current camera pose by tracking the last key frame.
- **helperTrackLocalMap** refine the current camera pose by tracking the local map.
- **helperViewDirectionAndDepth** store the mean view direction and the predicted depth of map points
- **helperVisualizeMatchedFeatures** show the matched features in a frame.
- **helperVisualizeMotionAndStructure** show map points and camera trajectory.
- **helperDetectAndExtractFeatures** detect and extract and ORB features from the image.

SLAM 示例代码

```
%Monocular Visual Simultaneous Localization and Mapping
openExample('vision/
MonocularVisualSimultaneousLocalizationAndMappingExample')
```