

# Visual-SLAM

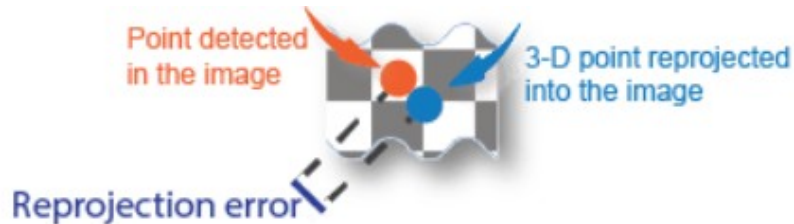
(Simultaneous Localization And  
Mapping)



# Implement Visual SLAM in MATLAB

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 visual SLAM include augmented reality, robotics, and autonomous driving.

Visual SLAM algorithms are broadly classified into two categories, depending on how they estimate the camera motion. The indirect, feature-based method uses feature points of images to minimize the **reprojection error**. The direct method uses the overall brightness of images to minimize the photometric error. The feature-based visual SLAM workflow consists of map initialization, tracking, local mapping, loop detection, and drift correction.



# Terms Used in Visual SLAM

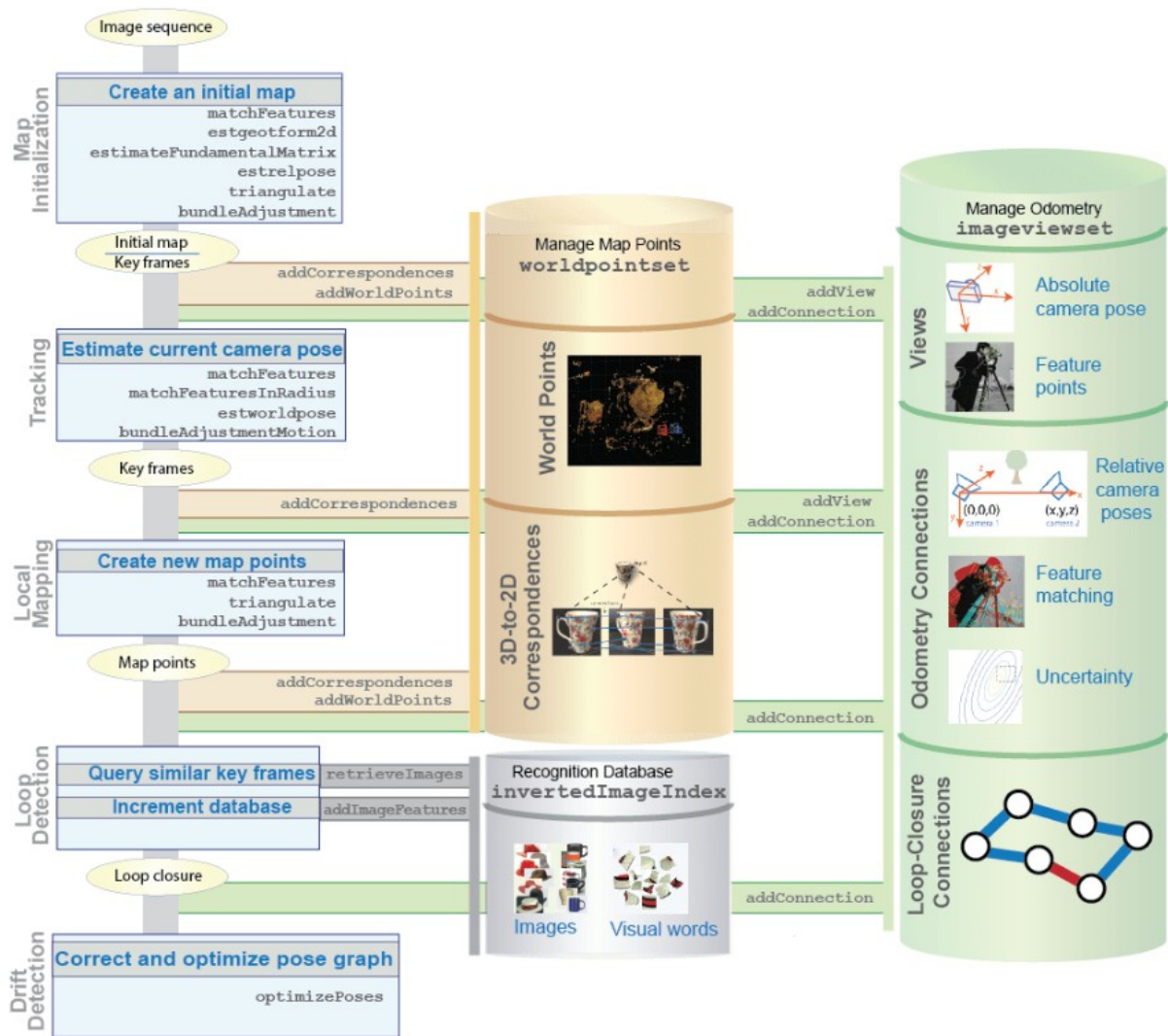
Visual SLAM literature uses these common terms:

- **Key Frames** — A subset of video frames that contain cues for localization and tracking. Two consecutive key frames usually indicate a large visual change caused by a camera movement.
- **Map Points** — A list of 3-D world points that represent the map of the environment reconstructed from the key frames.
- **Covisibility Graph** — A graph with key frames 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.
- **Recognition Database** — A database that stores the visual word-to-image mapping based on the input bag of features. Determine whether a place has been visited in the past by searching the database for an image that is visually similar to the query image.

# Typical Feature-based Visual SLAM Workflow

To construct a feature-based visual SLAM pipeline on a sequence of images, follow these steps:

1. **Initialize Map** — Initialize the map of 3-D points from two image frames. Compute the 3-D points and relative camera pose by using triangulation based on 2-D feature correspondences.
2. **Track Features** — For each new frame, estimate the camera pose by matching features in the current frame to features in the last key frame.
3. **Create Local Map** — If you identify the current frame as a key frame, create a new 3-D map of points. Use bundle adjustment to refine the camera pose and 3-D points.
4. **Detect Loops** — Detect loops for each key frame by comparing the current frame to all previous key frames using the bag-of-features approach.
5. **Correct Drift** — Optimize the pose graph to correct the drift in the camera poses of all the key frames.



# Key Frame and Map Data Management

Use the view set, point set, and transformation objects to manage key frames and map data.

- Use the **imageviewset** object to manage data associated with the odometry and mapping process. The object contains data as a set of views and pairwise connections between views. The object can also be used to build and update a pose graph.
  - Each view consists of the absolute camera pose and the feature points extracted from the image. Each view, with its unique identifier (view ID), within the view set forms a node of the pose graph.
  - Each connection stores information that links one view to another view. The connection includes the indices of matched features between the views, the relative transformation between the connected views, and the uncertainty in computing the measurement. Each connection forms an edge in the pose graph.
- Use the **worldpointset** object to store correspondences between 3-D map points and 2-D image points across camera views.
  - The WorldPoints property of worldpointset stores the 3-D locations of map points.
  - The Correspondence property of worldpointset stores the view IDs of the key frames that observe the map points.

## Map Initialization

To initialize mapping, you must match features between two images, estimate the relative camera pose, and triangulate initial 3-D world points. This workflow commonly uses the Speeded-Up Robust Features (SURF) and Oriented FAST and Rotated BRIEF (ORB) features point features. The map initialization workflow consists of a detecting, extracting, and matching features, and then finding a relative camera pose estimate, finding the 3-D locations of matched features, and refining the initial map. Finally, store the resulting key frames and mapped points in an image view set and a world point set, respectively.

Workflow	Function	Description
1. Detect	<code>detectSURFFeatures</code>	Detect SURF features and return a <code>SURFPoints</code> object.
	<code>detectORBFeatures</code>	Detect ORB features and return an <code>ORBPoints</code> object.
	<code>detectSIFTFeatures</code>	Detect SIFT features and return a <code>SIFTPoints</code> object.
2. Extract	<code>extractFeatures</code>	Extract feature vectors and their corresponding locations in a binary or intensity image.
3. Match	<code>matchFeatures</code>	Obtain the indices of the matching features between two feature sets.
4. Estimate relative camera pose from matched feature points	<code>estgeotform2d</code>	Compute a homography from matching point pairs.
	<code>estimateFundamentalMatrix</code>	Estimate the fundamental matrix from matching point pairs.
	<code>estrelpose</code>	Compute the relative camera poses, represented as a <code>rigidtform3d</code> object, based on a homography or a fundamental matrix. The location can only be computed up to scale, so the distance between two cameras is set to 1.
5. Find 3-D locations of the matched feature points	<code>triangulate</code>	Find the 3-D locations of matching pairs of undistorted image points.
6. Refine initial map	<code>bundleAdjustment</code>	Refine 3-D map points and camera poses that minimize reprojection errors.
7. Manage data for initial map and key frames	<code>addView</code>	Add the two views formed by the feature points and their absolute poses to the <code>imageviewset</code> object.
	<code>addConnection</code>	Add the odometry edge defined by the connection between successive key views, formed by the relative pose transformation between the cameras, to the <code>imageviewset</code> object.
	<code>addWorldPoints</code>	Add the initial map points to the <code>worldpointset</code> object.
	<code>addCorrespondences</code>	Add the 3-D to 2-D projection correspondences between the key frames and the map points to the <code>worldpointset</code> object.



# Tracking

The tracking workflow uses every frame to determine when to insert a new key frame. Use these steps and functions for the tracking workflow.

Workflow	Function	Description
Match extracted features	<code>matchFeatures</code>	Match extracted features from the current frame with features in the last key frame that have known 3-D locations.
Estimate camera pose	<code>estworldpose</code>	Estimate the current camera pose.
Project map points	<code>world2img</code>	Project the map points observed by the last key frame into the current frame.
Search for feature correspondences	<code>matchFeaturesInRadius</code>	Search for feature correspondences within spatial constraints.
Refine camera pose	<code>bundleAdjustmentMotion</code>	Refine the camera pose with 3-D to 2-D correspondence by performing a motion-only bundle adjustment.
Identify local map points	<code>findWorldPointsInView</code> <code>findWorldPointsInTracks</code>	Identify points in the view and points that correspond to point tracks.
Search for more feature correspondences	<code>matchFeaturesInRadius</code>	Search for more feature correspondences in the current frame, which contains projected local map points.
Refine camera pose	<code>bundleAdjustmentMotion</code>	Refine the camera pose with 3-D to 2-D correspondence by performing a motion-only bundle adjustment.
Store new key frame	<code>addView</code> <code>addConnection</code>	If you determine that the current frame is a new key frame, add it and its connections to covisible key frames to the <code>imageviewset</code> .

## Tracking

Feature matching is critical in the tracking workflow. Use the `matchFeaturesInRadius` function to return more putative matches when an estimation of the positions of matched feature points is available. The two match feature functions used in the workflow are:

- `matchFeatures` — Returns the indices of the matching features in the two input feature sets.
- `matchFeaturesInRadius` — Returns the indices of the matching features, which satisfy spatial constraints, in the two input feature sets.

To get a greater number of matched feature pairs, increase the values for the `MatchThreshold` and `MaxRatio` name-value arguments of the `matchFeatures` and `matchFeaturesInRadius` functions. The outliers pairs can be discarded after performing bundle adjustment in the local mapping step.

# Local Mapping

Perform local mapping for every key frame. Follow these steps to create new map

Workflow	Function	Description
Connect key frames	<code>connectedViews</code>	Find the covisible key frames of the current key frame.
Search for matches in connected key frames	<code>matchFeatures</code>	For each unmatched feature point in the current key frame, use the <code>matchFeatures</code> function to search for a match with other unmatched points in the covisible key frames.
Compute location for new matches	<code>triangulate</code>	Compute the 3-D locations of the matched feature points.
Store new map points	<code>addWorldPoints</code>	Add the new map points to the <code>worldpointset</code> object.
Store 3-D to 2-D correspondences	<code>addCorrespondences</code>	Add new 3-D to 2-D correspondences to the <code>worldpointset</code> object.
Update odometry connection	<code>updateConnection</code>	Update the connection between the current key frame and its covisible frames with more feature matches.
Store representative view of 3-D points	<code>updateRepresentativeView</code>	Update representative view ID and corresponding feature index.
Store distance limits and viewing direction of 3-D points	<code>updateLimitsAndDirection</code>	Update distance limits and mean viewing direction.
Refine pose	<code>bundleAdjustment</code>	<p>Refine the pose of the current key frame, the poses of covisible key frames, and all the map points observed in these key frames. For improved performance, only include strongly connected, covisible key frames in the refinement process.</p> <p>Use the <code>minNumMatches</code> argument of the <code>connectedViews</code> function to select strongly-connected covisible key frames.</p>
Remove outliers	<code>removeWorldPoints</code>	Remove outlier map points with large reprojection errors from the <code>worldpointset</code> object. The associated 3-D to 2-D correspondences are removed automatically.

# Local Mapping

This table compares the camera poses, map points, and number of cameras for each of the bundle adjustment functions used in 3-D reconstruction.

Function	Camera Poses	Map Points	Number of Cameras
<code>bundleAdjustment</code>	Optimized	Optimized	Multiple
<code>bundleAdjustmentMotion</code>	Optimized	Fixed	One
<code>bundleAdjustmentStructure</code>	Fixed	Optimized	Multiple

# Loop Detection

Due to an accumulation of errors, using visual odometry alone can lead to drift. These errors can result in severe inaccuracies over long distances. Using graph-based SLAM helps to correct the drift. To do this, detect loop closures by finding a previously visited location. A common approach is to use this bag-of-features workflow:

Workflow	Function	Description
Construct bag of visual words	<a href="#">bagOfFeatures</a>	Construct a bag of visual words for place recognition.
Create recognition database	<a href="#">indexImages</a>	Create a recognition database, <a href="#">invertedImageIndex</a> , to map visual words to images.
Identify loop closure candidates	<a href="#">retrieveImages</a>	Search for images that are similar to the current key frame. Identify consecutive images as loop closure candidates if they are similar to the current frame. Otherwise, add the current key frame to the recognition database.
Compute relative camera pose for loop closure candidates	<a href="#">estGeotform3d</a>	Compute the relative camera pose between the candidate key frame and the current key frame, for each loop closure candidate
Close loop	<a href="#">addConnection</a>	Close the loop by adding a loop closure edge with the relative camera pose to the <a href="#">imageViewset</a> object.

## Drift Correction

The `imageviewset` object internally updates the pose graph as views and connections are added. To minimize drift, perform pose graph optimization by using the `optimizePoses` function, once sufficient loop closures are added. The `optimizePoses` function returns an `imageviewset` object with the optimized absolute pose transformations for each view.

You can use the `createPoseGraph` function to return the pose graph as a MATLAB<sup>®</sup> digraph object. You can use graph algorithms in MATLAB to inspect, view, or modify the pose graph. Use the `optimizePoseGraph` (Navigation Toolbox) function from Navigation Toolbox<sup>™</sup> to optimize the modified pose graph, and then use the `updateView` function to update the camera poses in the view set.

# Visualization

To develop the visual SLAM system, you can use the following visualization functions.

Function	Description
<code>imshow</code>	Display an image
<code>showMatchedFeatures</code>	Display matched feature points in two images
<code>plot</code>	Plot image view set views and connections
<code>plotCamera</code>	Plot a camera in 3-D coordinates
<code>pcshow</code>	Plot 3-D point cloud
<code>pcplayer</code>	Visualize streaming 3-D point cloud data

# 视觉 SLAM 示例

- Rotations, Orientation, and Quaternions
- Monocular Visual Odometry



# Rotations, Orientation, and Quaternions

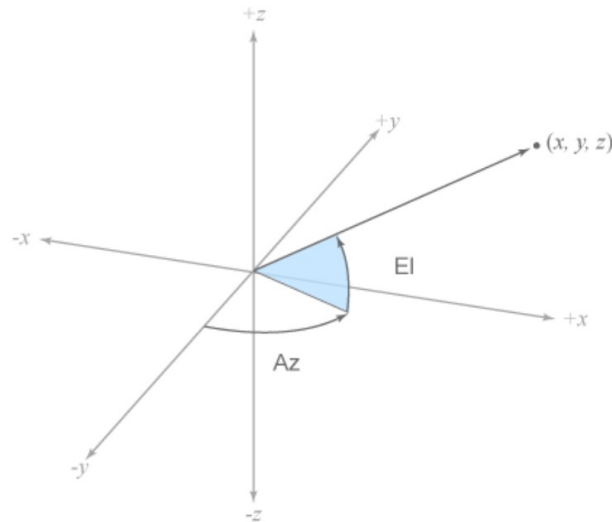
This example reviews concepts in three-dimensional rotations and how quaternions are used to describe orientation and rotations. Quaternions are a skew field of hypercomplex numbers. They have found applications in aerospace, computer graphics, and virtual reality. In MATLAB®, quaternion mathematics can be represented by manipulating the quaternion class.

The HelperDrawRotation class is used to illustrate several portions of this example.

```
dr = HelperDrawRotation;
```

## Method Summary

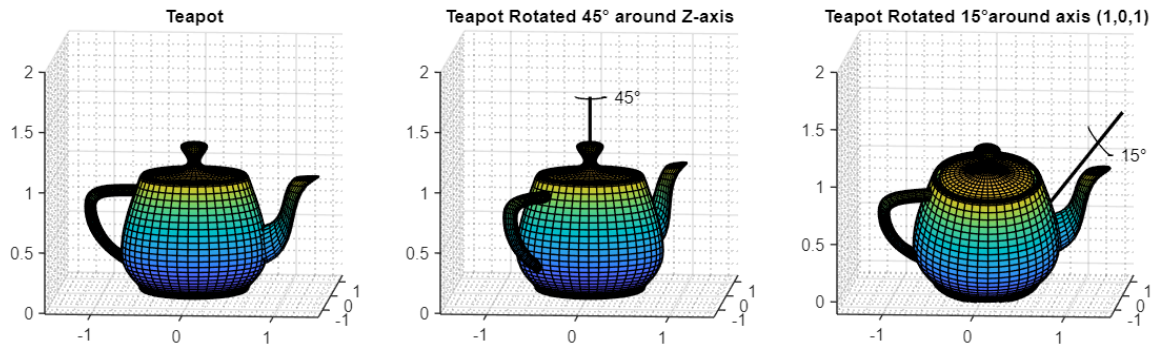
Static	<a href="#">draw2DFrameRotation</a>	2D Frame rotation example
Static	<a href="#">draw2DPointRotation</a>	2D Point rotation example
Static	<a href="#">draw3DOrientation</a>	3D Orientation example
Static	<a href="#">drawEulerRotation</a>	Draw Frame rotation using Euler angles
Static	<a href="#">drawTeapotRotations</a>	Teapot Rotation example



# Rotations in Three Dimensions

All rotations in 3-D can be defined by an axis of rotation and an angle of rotation about that axis. Consider the 3-D image of a teapot in the leftmost plot. The teapot is rotated by 45 degrees around the Z-axis in the second plot. A more complex rotation of 15 degrees around the axis  $[1\ 0\ 1]$  is shown in the third plot. Quaternions encapsulate the axis and angle of rotation and have an algebra for manipulating these rotations. The quaternion class, and this example, use the "right-hand rule" convention to define rotations. That is, positive rotations are clockwise around the axis of rotation when viewed from the origin.

```
dr.drawTeapotRotations;
```

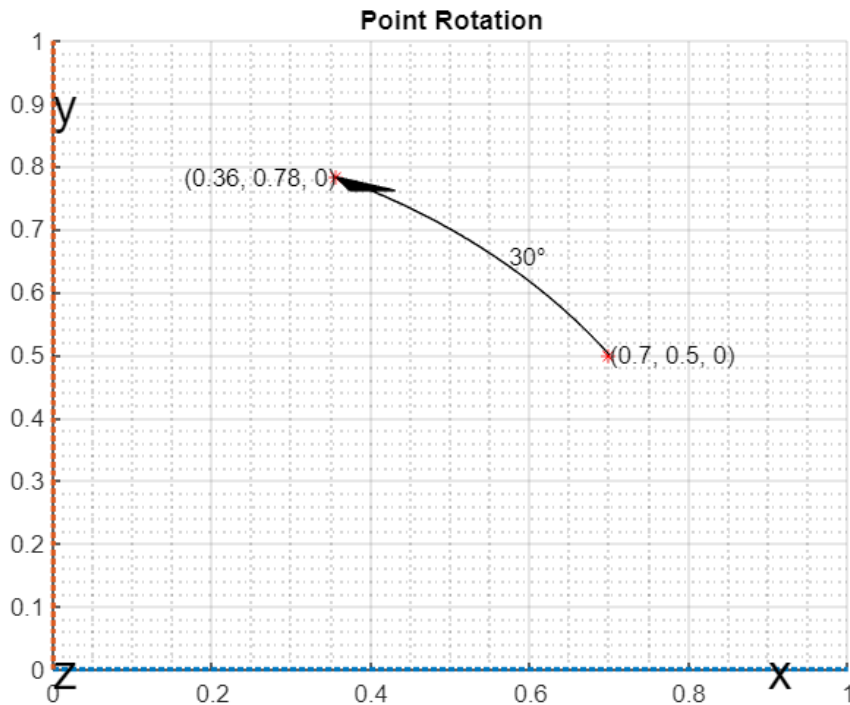


## Point Rotation

The vertices of the teapot were rotated about the axis of rotation in the reference frame. Consider a point  $(0.7, 0.5)$  rotated 30 degrees about the Z-axis.

figure;

```
dr.draw2DPointRotation(gca);
```

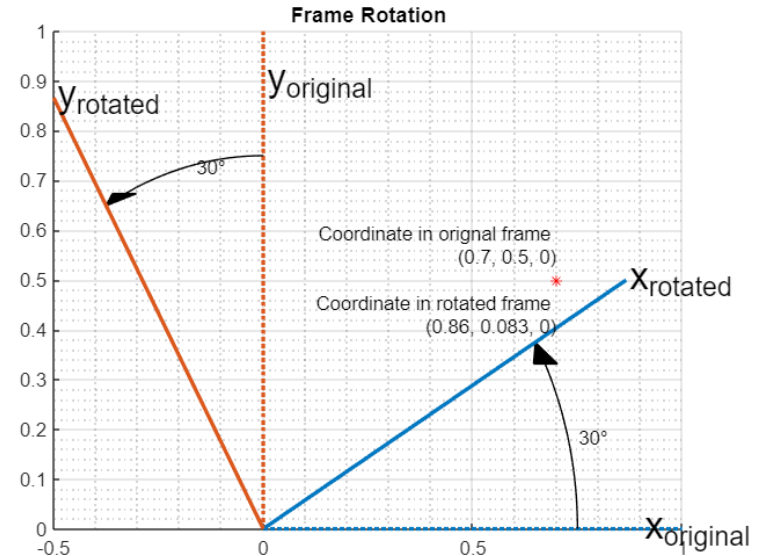


# Frame Rotation

Frame rotation is, in some sense, the opposite of point rotation. In frame rotation, the points of the object stay fixed, but the frame of reference is rotated. Again, consider the point  $(0.7, 0.5)$ . Now the reference frame is rotated by 30 degrees around the Z-axis. Note that while the point  $(0.7, 0.5)$  stays fixed, it has different coordinates in the new, rotated frame of reference.

figure;

```
dr.draw2DFrameRotation(gca);
```



## Orientation

Orientation refers to the angular displacement of an object relative to a frame of reference. Typically, orientation is described by the rotation that causes this angular displacement from a starting orientation. In this example, orientation is defined as the rotation that takes a quantity in a parent reference frame to a child reference frame. Orientation is usually given as a quaternion, rotation matrix, set of Euler angles, or rotation vector. It is useful to think about orientation as a frame rotation: the child reference frame is rotated relative to the parent frame.

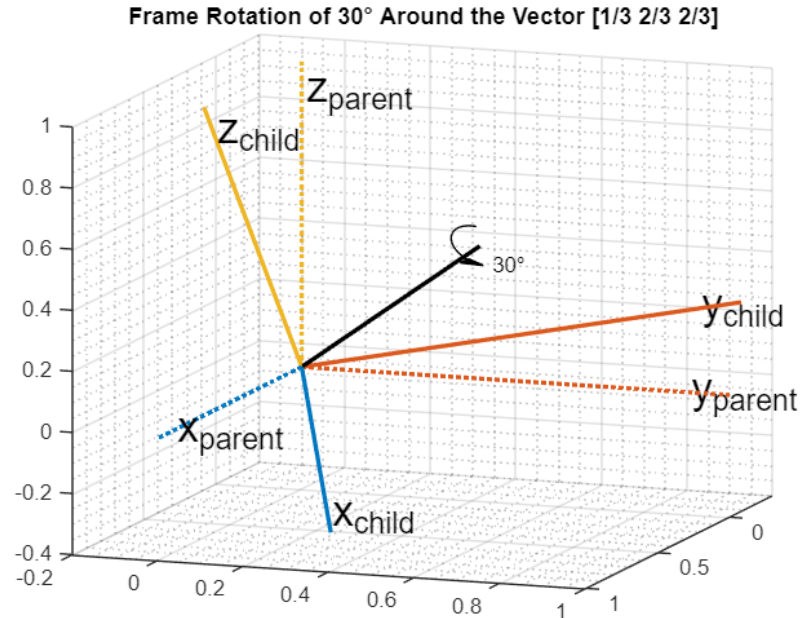
Consider an example where the child reference frame is rotated 30 degrees around the vector  $[1/3 \ 2/3 \ 2/3]$ .

# Orientation

Consider an example where the child reference frame is rotated 30 degrees around the vector  $[1/3 \ 2/3 \ 2/3]$ .

figure;

```
dr.draw3DOrientation(gca, [1/3 2/3 2/3], 30);
```



# Quaternions

Quaternions are numbers of the form

$$a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$$

where

$$i^2 = j^2 = k^2 = ijk = -1$$

and  $a$ ,  $b$ ,  $c$  and  $d$  are real numbers. In the rest of this example, the four numbers  $a$ ,  $b$ ,  $c$  and  $d$  are referred to as the parts of the quaternion.

# Quaternions for Rotations and Orientation

The axis and the angle of rotation are encapsulated in the quaternion parts. For a unit vector axis of rotation  $[x, y, z]$ , and rotation angle  $\alpha$ , the quaternion describing this rotation is

$$\cos\left(\frac{\alpha}{2}\right) + \sin\left(\frac{\alpha}{2}\right)(x\mathbf{i} + y\mathbf{j} + z\mathbf{k})$$

Note that to describe a rotation using a quaternion, the quaternion must be a *unit quaternion*. A unit quaternion has a norm of 1, where the norm is defined as

$$\text{norm}(q) = \sqrt{a^2 + b^2 + c^2 + d^2}$$



# Quaternions for Rotations and Orientation

There are a variety of ways to construct a quaternion in MATLAB, for example:

`q1 = quaternion(1,2,3,4)`

`q1 = quaternion`  
 $1 + 2i + 3j + 4k$

Arrays of quaternions can be made in the same way:

`quaternion([1 10; -1 1], [2 20; -2 2], [3 30; -3 3], [4 40; -4 4])`

Arrays with four columns can also be created with quaternions, with each column representing a quaternion

`qmgk = quaternion(magic(4))`

`ans = 2x2 quaternion array`

$1 + 2i + 3j + 4k$	$10 + 20i + 30j + 40k$
$-1 - 2i - 3j - 4k$	$1 + 2i + 3j + 4k$

`qmgk = 4x1 quaternion array`

$16 + 2i + 3j + 13k$
$5 + 11i + 10j + 8k$
$9 + 7i + 6j + 12k$
$4 + 14i + 15j + 1k$

Quaternions can be in

`qmgk(3)`

`reshape(qmgk,2,2)`

`[q1; q1]`

`ans = quaternion`

$9 + 7i + 6j + 12k$

`ans = 2x2 quaternion array`

$16 + 2i + 3j + 13k$

$5 + 11i + 10j + 8k$

`ans = 2x1 quaternion array`

$1 + 2i + 3j + 4k$

$1 + 2i + 3j + 4k$

like any other array:

$9 + 7i + 6j + 12k$

$4 + 14i + 15j + 1k$

# Quaternion Math

Quaternions have well-defined arithmetic operations. Addition and subtraction are similar to complex numbers: parts are added/subtracted independently. Multiplication is more complicated because of the earlier equation:

$$i^2 = j^2 = k^2 = ijk = -1$$

This means that multiplication of quaternions is not commutative. That is,  $pq \neq qp$  for quaternions  $p$  and  $q$ . However, every quaternion has a multiplicative inverse, so quaternions can be divided. Arrays of the `quaternion` class can be added, subtracted, multiplied, and divided in MATLAB.

```
q = quaternion(1,2,3,4);  
p = quaternion(-5,6,-7,8);
```

# Quaternion Math

Addition

```
ans = quaternion
      -4 + 8i - 4j + 12k
```

$p + q$

Subtraction

```
ans = quaternion
      -6 + 4i - 10j + 4k
```

$p - q$

Multiplication

```
ans = quaternion
     -28 - 56i - 30j + 20k
```

$p * q$

Multiplication in the reverse order (note the different

```
ans = quaternion
     -28 + 48i - 14j - 44k
```

$q * p$

Right division of p by q is equivalent to

$p ./ q$

```
0.6 + 2.2667i + 0.53333j - 0.13333k
```

Left division of q by p is equivalent to

$p .\ q$

```
ans = quaternion
0.10345 + 0.2069i + 0j - 0.34483k
```

The conjugate of a quaternion is formed by negating each of the non-real parts, similar to conjugation for a complex number

$\text{conj}(p)$

```
ans = quaternion
     -5 - 6i + 7j - 8k
```

Quaternions can be normalized in MATLAB

$\text{pnormed} = \text{normalize}(p)$

```
pnormed = quaternion
     -0.37905 + 0.45486i - 0.53067j + 0.60648k
```

$\text{norm}(\text{pnormed})$

```
ans = 1
```

```
q = quaternion(1,2,3,4);
p = quaternion(-5,6,-7,8);
```

# Point and Frame Rotations with Quaternions

Quaternions can be used to rotate points in a static frame of reference, or to rotate the frame of reference itself. The `rotatepoint` function rotates a point  $v = (v_x, v_y, v_z)$  using a quaternion  $q$  through the following equation:

$$pv_{quat}p^*$$

where  $v_{quat}$  is

$$v_{quat} = 0 + v_x\mathbf{i} + v_y\mathbf{j} + v_z\mathbf{k}$$

and  $p^*$  indicates quaternion conjugation. Note the above quaternion multiplication results in a quaternion with the real part,  $a$ , equal to 0. The  $b$ ,  $c$ , and  $d$  parts of the result form the rotated point  $(b, c, d)$ .

## Point and Frame Rotations with Quaternions

Consider the example of point rotation from above. The point (0.7, 0.5) was rotated 30 degrees around the Z-axis. In three dimensions this point has a 0 Z-coordinate. Using the axis-angle formulation, a quaternion can be constructed using  $[0 \ 0 \ 1]$  as the axis of rotation.

```
ang = deg2rad(30);  
q = quaternion(cos(ang/2), 0, 0, sin(ang/2));  
pt = [0.7, 0.5, 0]; Z-coordinate is 0 in the X-Y plane  
ptrot = rotatepoint(q, pt)
```

```
ptrot = 1x3  
    0.3562    0.7830    0
```

# Point and Frame Rotations with Quaternions

Similarly, the `rotateframe` function takes a quaternion  $q$  and point  $v$  to compute

$$P^* v_{quat} P$$

Again the above quaternion multiplication results in a quaternion with 0 real part. The  $(b, c, d)$  parts of the result form the coordinate of the point  $v$  in the new, rotated reference frame.

Using the `quaternion` class:

```
ptframerot = rotateframe(q, pt)
```

```
ptframerot = 1x3  
    0.8562    0.0830    0
```

## Point and Frame Rotations with Quaternions

A quaternion and its conjugate have opposite effects because of the symmetry in the point and frame rotation equations. Rotating by the conjugate "undoes" the rotation.

`rotateframe(conj(q), ptframerot)`

```
ans = 1x3  
      0.7000    0.5000         0
```

Because of the symmetry of the equations, this code performs the same rotation.

`rotatepoint(q, ptframerot)`

```
ans = 1x3  
      0.7000    0.5000         0
```

## Other Rotation Representations

Often rotations and orientations are described using alternate means: Euler angles, rotation matrices, and/or rotation vectors. All of these interoperate with quaternions in MATLAB.

Euler angles are frequently used because they are easy to interpret. Consider a frame of reference rotated by 30 degrees around the Z-axis, then 20 degrees around the Y-axis, and then -50 degrees around the X-axis. Note here, and throughout, the rotations around each axis are intrinsic: each subsequent rotation is around the newly created set of axes. In other words, the second rotation is around the "new" Y-axis created by the first rotation, not around the original Y-axis.

figure;

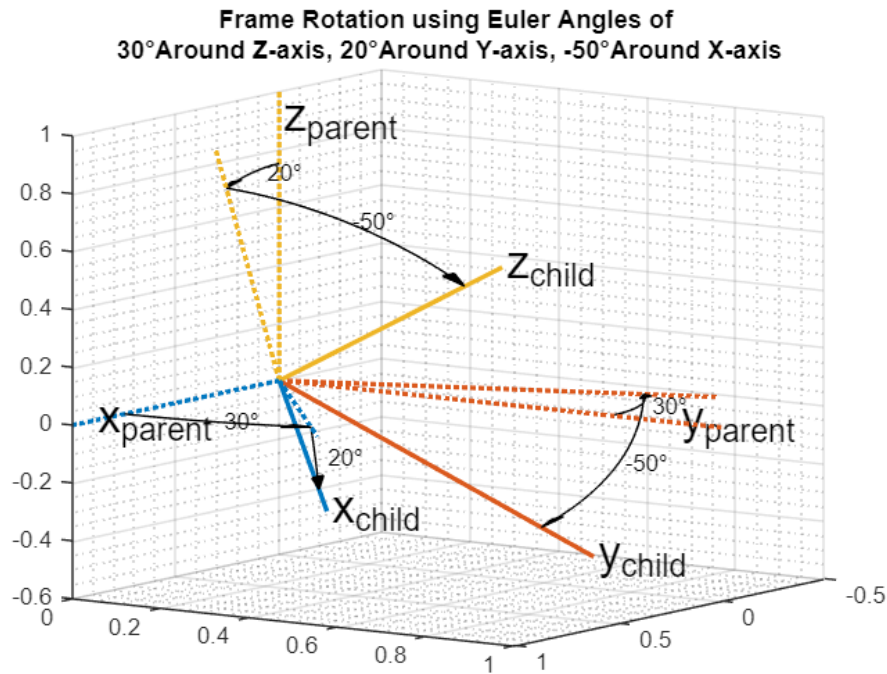
```
euld = [30 20 -50];
```

```
dr.drawEulerRotation(gca, euld);
```



# Other Rotation Representations

```
figure;  
euld = [30 20 -50];  
dr.drawEulerRotation(gca, euld);
```



## Other Rotation Representations

To build a quaternion from these Euler angles for the purpose of frame rotation, use the quaternion constructor. Since the order of rotations is around the Z-axis first, then around the new Y-axis, and finally around the new X-axis, use the 'ZYX' flag.

```
qeul = quaternion(deg2rad(euld), 'euler', 'ZYX', 'frame')
```

```
qeul = quaternion  
      0.84313 - 0.44275i + 0.044296j + 0.30189k
```

The 'euler' flag indicates that the first argument is in radians. If the argument is in degrees, use the 'eulerd' flag.

```
qeuld = quaternion(euld, 'eulerd', 'ZYX', 'frame')
```

```
qeuld = quaternion  
      0.84313 - 0.44275i + 0.044296j + 0.30189k
```

## Other Rotation Representations

To convert back to Euler angles:  
`rad2deg(euler(qeul, 'ZYX', 'frame'))`

```
ans = 1x3
      30.0000    20.0000   -50.0000
```

Equivalently, the `eulerd` method can be used.  
`eulerd(qeul, 'ZYX', 'frame')`

```
ans = 1x3
      30.0000    20.0000   -50.0000
```

Alternatively, this same rotation can be represented as a rotation matrix:  
`rmat = rotmat(qeul, 'frame')`

```
rmat = 3x3
        0.8138    0.4698   -0.3420
       -0.5483    0.4257   -0.7198
       -0.1926    0.7733    0.6040
```

The conversion back to quaternions is similar:  
`quaternion(rmat, 'rotmat', 'frame')`

```
ans = quaternion
      0.84313 - 0.44275i + 0.044296j + 0.30189k
```

## Other Rotation Representations

Just as a quaternion can be used for either point or frame rotation, it can be converted to a rotation matrix (or set of Euler angles) specifically for point or frame rotation. The rotation matrix for point rotation is the transpose of the matrix for frame rotation. To convert between rotation representations, it is necessary to specify 'point' or 'frame'.

The rotation matrix for the point rotation section of this example is:

`rotmatPoint = rotmat(q, 'point')`

```
rotmatPoint = 3x3
    0.8660    -0.5000         0
    0.5000     0.8660         0
         0         0     1.0000
```

缩放矩阵	旋转矩阵	平移矩阵
$S = \begin{bmatrix} s_x & & & \\ & s_y & & \\ & & s_z & \\ & & & 1 \end{bmatrix}$	$R_x = \begin{bmatrix} \cos \theta & -\sin \theta & & \\ \sin \theta & \cos \theta & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$ $R_y = \begin{bmatrix} \cos \theta & -\sin \theta & & \\ & 1 & & \\ \sin \theta & \cos \theta & & \\ & & & 1 \end{bmatrix}$ $R_z = \begin{bmatrix} 1 & & & \\ & \cos \theta & -\sin \theta & \\ & \sin \theta & \cos \theta & \\ & & & 1 \end{bmatrix}$	$T = \begin{bmatrix} 1 & & & t_x \\ & 1 & & t_y \\ & & 1 & t_z \\ & & & 1 \end{bmatrix}$

## Other Rotation Representations

To find the location of the rotated point, right-multiply rotmatPoint by the transposed array pt.

rotmatPoint \* (pt')

```
ans = 3x1
    0.3562
    0.7830
         0
```

The rotation matrix for the frame rotation section of this example is:

rotmatFrame = rotmat(q, 'frame')

```
rotmatFrame = 3x3
    0.8660    0.5000         0
   -0.5000    0.8660         0
         0         0    1.0000
```

To find the location of the point in the rotated reference frame, right-multiply rotmatFrame by the transposed array pt.

rotmatFrame \* (pt')

```
ans = 3x1
    0.8562
    0.0830
         0
```

## Other Rotation Representations

A rotation vector is an alternate, compact rotation encapsulation. A rotation vector is simply a three-element vector that represents the unit length axis of rotation scaled-up by the angle of rotation in radians. There is no frame-ness or point-ness associated with a rotation vector. To convert to a rotation vector:

`rv = rotvec(qeul)`

```
rv = 1x3  
    -0.9349    0.0935    0.6375
```

To convert to a quaternion:  
`quaternion(rv, 'rotvec')`

```
ans = quaternion  
    0.84313 - 0.44275i + 0.044296j + 0.30189k
```

## Distance

One advantage of quaternions over Euler angles is the lack of discontinuities. Euler angles have discontinuities that vary depending on the convention being used. The `dist` function compares the effect of rotation by two different quaternions. The result is a number in the range of 0 to  $\pi$ . Consider two quaternions constructed from Euler angles:

```
eul1 = [0, 10, 0];
```

```
eul2 = [0, 15, 0];
```

```
qdist1 = quaternion(deg2rad(eul1), 'euler', 'ZYX', 'frame');
```

```
qdist2 = quaternion(deg2rad(eul2), 'euler', 'ZYX', 'frame');
```

## Distance

Subtracting the Euler angles, you can see there is no rotation around the Z-axis or X-axis.

eul2 - eul1

```
ans = 1x3
      0      5      0
```

The difference between these two rotations is five degrees around the Y-axis. The dist shows the difference as well.

rad2deg(dist(qdist1, qdist2))

```
ans = 5.0000
```

For Euler angles such as eul1 and eul2, computing angular distance is trivial. A more complex example, which spans an Euler angle discontinuity, is:

eul3 = [0, 89, 0];

eul4 = [180, 89, 180];

qdist3 = quaternion(deg2rad(eul3), 'euler', 'ZYX', 'frame');

qdist4 = quaternion(deg2rad(eul4), 'euler', 'ZYX', 'frame');



## Distance

Though eul3 and eul4 represent nearly the same orientation, simple Euler angle subtraction gives the impression that these two orientations are very far apart.

euldiff = eul4 - eul3

```
euldiff = 1x3
        180      0    180
```

Using the dist function on the quaternions shows that there is only a two-degree difference in these rotations:

euldist = rad2deg(dist(qdist3, qdist4))

```
euldist = 2.0000
```

A quaternion and its negative represent the same rotation. This is not obvious from subtracting quaternions, but the dist function makes it clear.

qpos = quaternion(-cos(pi/4), 0, 0, sin(pi/4))

qneg = -qpos

qdiff = qpos - qneg

dist(qpos, qneg)

```
qpos = quaternion
      -0.70711 +      0i +      0j + 0.70711k
qneg = quaternion
      0.70711 +      0i +      0j - 0.70711k
qdiff = quaternion
      -1.4142 +      0i +      0j + 1.4142k
```

```
ans = 0
```

# Monocular Visual Odometry

Visual odometry is the process of determining the location and orientation of a camera by analyzing a sequence of images. Visual odometry is used in a variety of applications, such as mobile robots, self-driving cars, and unmanned aerial vehicles. This example shows you how to estimate the trajectory of a single calibrated camera from a sequence of images.

# Overview

This example shows how to estimate the trajectory of a calibrated camera from a sequence of 2-D views. This example uses images from the New Tsukuba Stereo Dataset created at Tsukuba University's CVLAB. (<https://cvlab.cs.tsukuba.ac.jp>). The dataset consists of synthetic images, generated using computer graphics, and includes the ground truth camera poses.

Without additional information, the trajectory of a monocular camera can only be recovered up to an unknown scale factor. Monocular visual odometry systems used on mobile robots or autonomous vehicles typically obtain the scale factor from another sensor (e.g. wheel odometer or GPS), or from an object of a known size in the scene. This example computes the scale factor from the ground truth.

# Overview

The example is divided into three parts:

- 1. Estimating the pose of the second view relative to the first view.** Estimate the pose of the second view by estimating the essential matrix and decomposing it into camera location and orientation.
- 2. Bootstrapping estimating camera trajectory using global bundle adjustment.** Eliminate outliers using the epipolar constraint. Find 3D-to-2D correspondences between points triangulated from the previous two views and the current view. Compute the world camera pose for the current view by solving the perspective-n-point (PnP) problem. Estimating the camera poses inevitably results in errors, which accumulate over time. This effect is called the drift. To reduce the drift, the example refines all the poses estimated so far using bundle adjustment.
- 3. Estimating remaining camera trajectory using windowed bundle adjustment.** With each new view the time it takes to refine all the poses increases. Windowed bundle adjustment is a way to reduce computation time by only optimizing the last  $n$  views, rather than the entire trajectory. Computation time is further reduced by not calling bundle adjustment for every view.

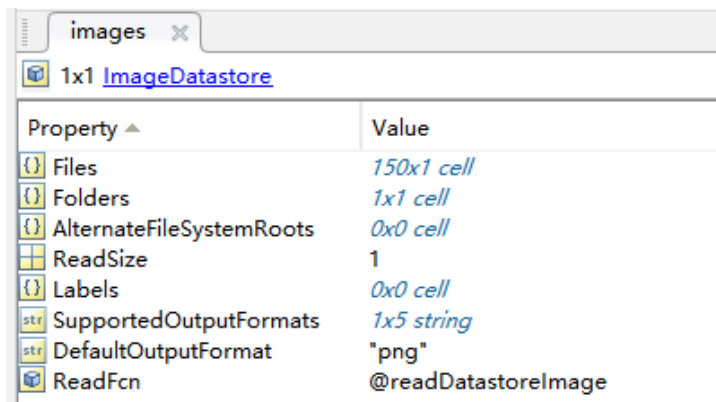
# Read Input Image Sequence and Ground Truth

This example uses images from the New Tsukuba Stereo Dataset created at Tsukuba University's CVLAB.

```
images = imageDatastore(fullfile(toolboxdir('vision'), 'visiondata', 'NewTsukuba'));
```

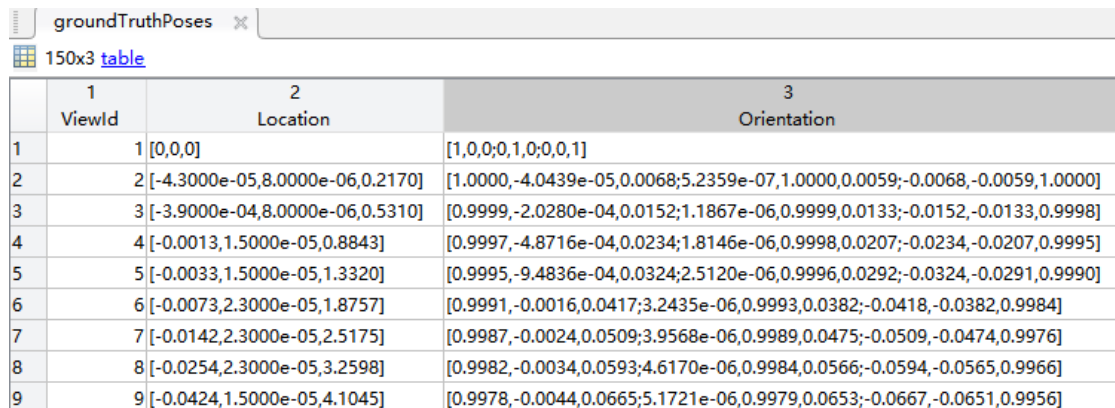
Load ground truth camera poses.

```
load(fullfile(toolboxdir('vision'), 'visiondata', 'visualOdometryGroundTruth.mat'));
```



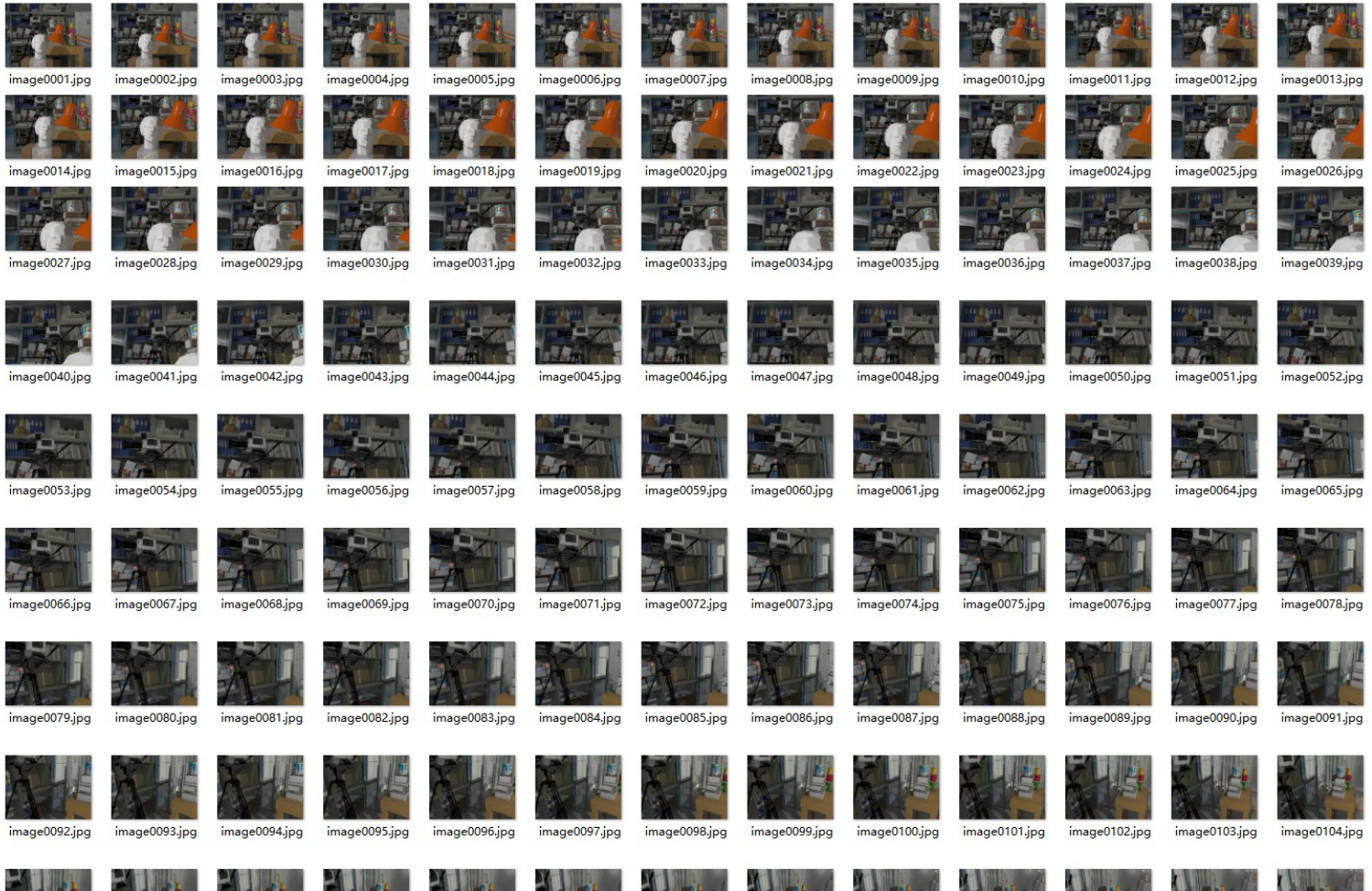
The screenshot shows the MATLAB Variable Inspector for the variable 'images'. It is a 1x1 ImageDatastore. The properties listed are:

Property	Value
Files	150x1 cell
Folders	1x1 cell
AlternateFileSystemRoots	0x0 cell
ReadSize	1
Labels	0x0 cell
SupportedOutputFormats	1x5 string
DefaultOutputFormat	'png'
ReadFcn	@readDatastoreImage



The screenshot shows the MATLAB Variable Inspector for the variable 'groundTruthPoses'. It is a 150x3 table. The columns are:

	1 ViewId	2 Location	3 Orientation
1	1	[0,0,0]	[1,0,0;0,1,0;0,0,1]
2	2	[-4.3000e-05,8.0000e-06,0.2170]	[1.0000,-4.0439e-05,0.0068;5.2359e-07,1.0000,0.0059;-0.0068,-0.0059,1.0000]
3	3	[-3.9000e-04,8.0000e-06,0.5310]	[0.9999,-2.0280e-04,0.0152;1.1867e-06,0.9999,0.0133;-0.0152,-0.0133,0.9998]
4	4	[-0.0013,1.5000e-05,0.8843]	[0.9997,-4.8716e-04,0.0234;1.8146e-06,0.9998,0.0207;-0.0234,-0.0207,0.9995]
5	5	[-0.0033,1.5000e-05,1.3320]	[0.9995,-9.4836e-04,0.0324;2.5120e-06,0.9996,0.0292;-0.0324,-0.0291,0.9990]
6	6	[-0.0073,2.3000e-05,1.8757]	[0.9991,-0.0016,0.0417;3.2435e-06,0.9993,0.0382;-0.0418,-0.0382,0.9984]
7	7	[-0.0142,2.3000e-05,2.5175]	[0.9987,-0.0024,0.0509;3.9568e-06,0.9989,0.0475;-0.0509,-0.0474,0.9976]
8	8	[-0.0254,2.3000e-05,3.2598]	[0.9982,-0.0034,0.0593;4.6170e-06,0.9984,0.0566;-0.0594,-0.0565,0.9966]
9	9	[-0.0424,1.5000e-05,4.1045]	[0.9978,-0.0044,0.0665;5.1721e-06,0.9979,0.0653;-0.0667,-0.0651,0.9956]



# Create a View Set Containing the First View of the Sequence

Use an `imageviewset` object to store and manage the image points and the camera pose associated with each view, as well as point matches between pairs of views. Once you populate an `imageviewset` object, you can use it to find point tracks across multiple views and retrieve the camera poses to be used by `triangulateMultiview` and `bundleAdjustment` functions.

Create an empty `imageviewset` object to manage the data associated with each view.

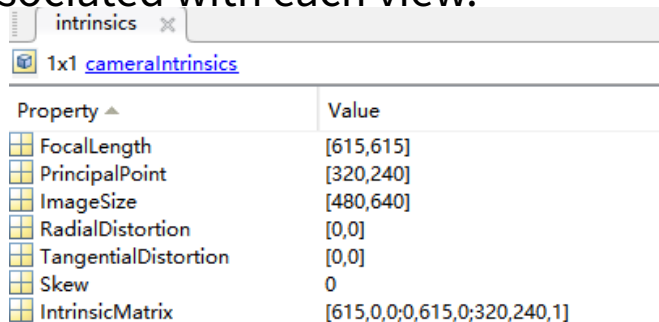
```
vSet = imageviewset;
```

Read and display the first image.

```
lrgb = readimage(images, 1);  
player = vision.VideoPlayer('Position', [20, 400, 650, 510]);  
step(player, lrgb);
```

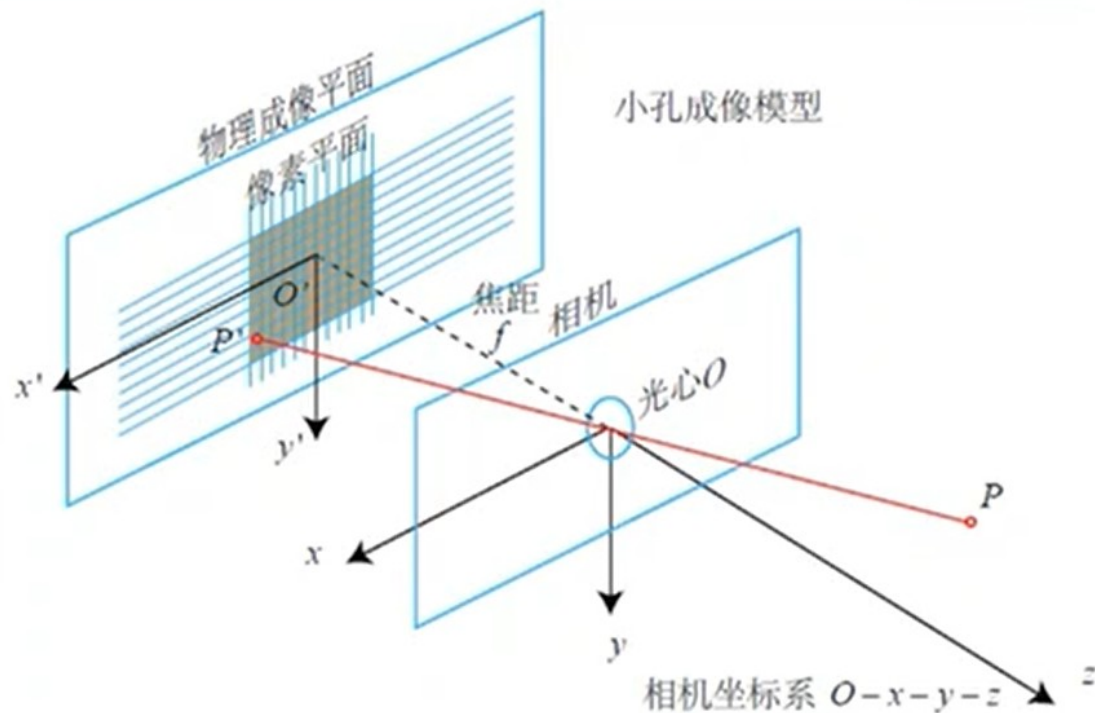
Create the camera intrinsics object using camera intrinsics from the New Tsukuba dataset.

```
focalLength = [615 615];    specified in units of pixels  
principalPoint = [320 240]; in pixels [x, y]  
imageSize = size(lrgb,[1,2]); in pixels [mrows, ncols]  
intrinsics = cameraIntrinsics(focalLength, principalPoint, imageSize);
```



Property ▲	Value
FocalLength	[615,615]
PrincipalPoint	[320,240]
ImageSize	[480,640]
RadialDistortion	[0,0]
TangentialDistortion	[0,0]
Skew	0
IntrinsicMatrix	[615,0,0;0,615,0;320,240,1]

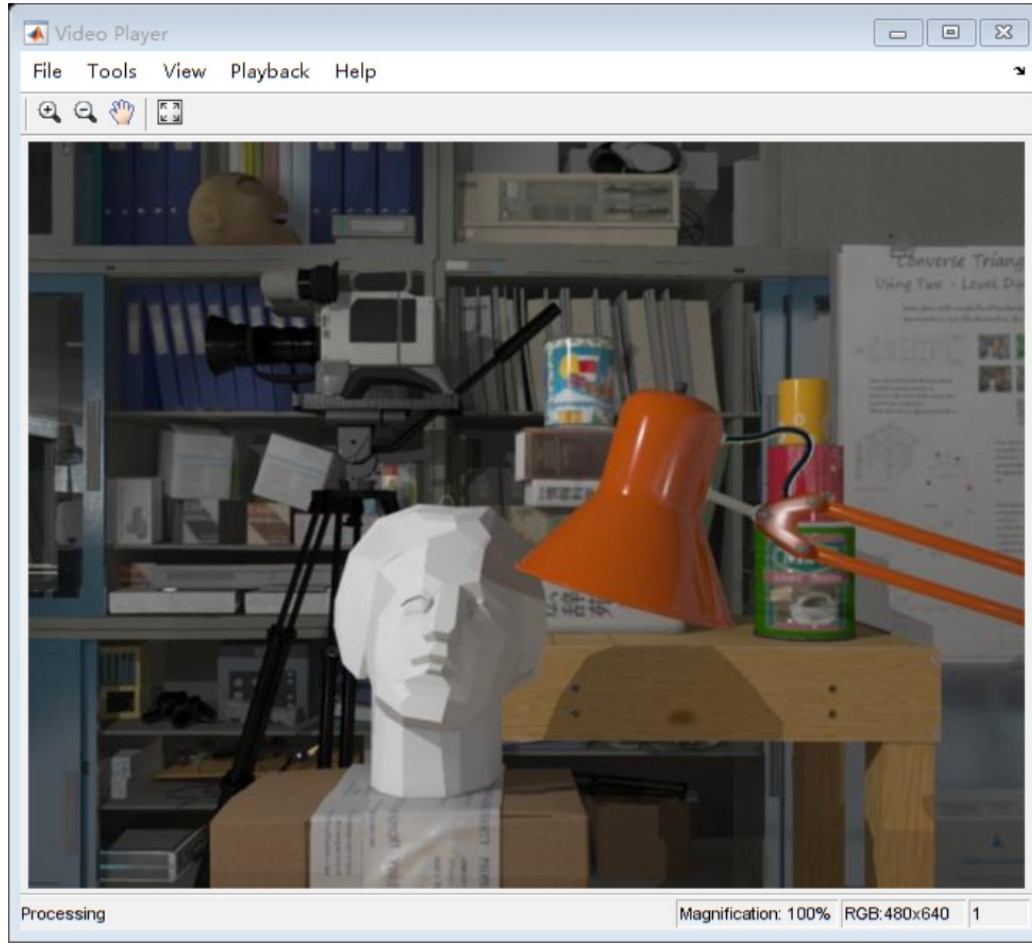
# Create a View Set Containing the First View of the Sequence



intrinsics	
1x1 <a href="#">cameraIntrinsics</a>	
Property	Value
FocalLength	[615,615]
PrincipalPoint	[320,240]
ImageSize	[480,640]
RadialDistortion	[0,0]
TangentialDistortion	[0,0]
Skew	0
IntrinsicMatrix	[615,0,0;0,615,0;320,240,1]



# Create a View Set Containing the First View of the Sequence



# Create a View Set Containing the First View of the Sequence

Convert to gray scale and undistort. In this example, undistortion has no effect, because the images are synthetic, with no lens distortion. However, for real images, undistortion is necessary.

```
prevI = undistortImage(im2gray(Irgb), intrinsics);
```

Detect features.

```
prevPoints = detectSURFFeatures(prevI, 'MetricThreshold', 500);
```

Select a subset of features, uniformly distributed throughout the image.

```
numPoints = 200;
```

```
prevPoints = selectUniform(prevPoints, numPoints, size(prevI));
```

Extract features. Using 'Upright' features improves matching quality if the camera motion involves little or no in-plane rotation.

```
prevFeatures = extractFeatures(prevI, prevPoints, 'Upright', true);
```

Add the first view. Place the camera with the first view at the origin, oriented along the Z-axis.

```
viewId = 1;
```

```
vSet = addView(vSet, viewId, rigid3d(eye(3), [0 0 0]), 'Points', prevPoints);
```

vSet.Views			
vSet.Views			
1	2	3	4
ViewId	AbsolutePose	Features	Points
1	<i>1x1 rigid3d</i>	<i>[]</i>	<i>200x1 SURFPoints</i>

```

% Setup axes.
figure
axis([-220, 50, -140, 20, -50, 300]);

% Set Y-axis to be vertical pointing down.
view(gca, 3);
set(gca, 'CameraUpVector', [0, -1, 0]);
camorbit(gca, -120, 0, 'data', [0, 1, 0]);

grid on
xlabel('X (cm)');
ylabel('Y (cm)');
zlabel('Z (cm)');
hold on

% Plot estimated camera pose.
cameraSize = 7;
camPose = poses(vSet);
camEstimated = plotCamera(camPose, 'Size', cameraSize, ...
    'Color', 'g', 'Opacity', 0);

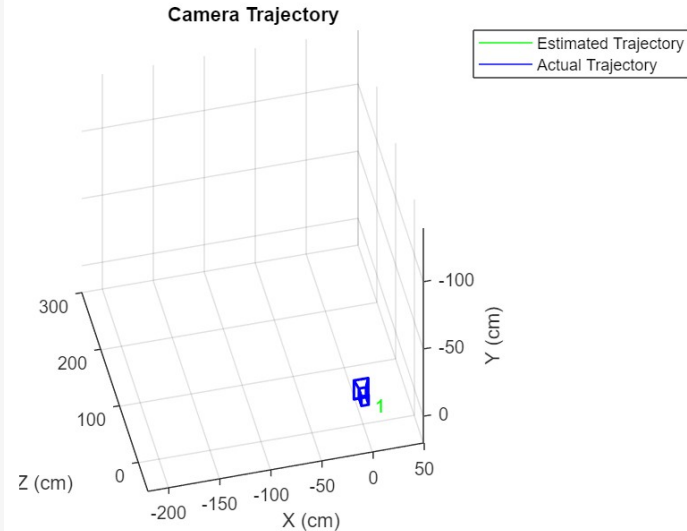
% Plot actual camera pose.
camActual = plotCamera('Size', cameraSize, 'AbsolutePose', ...
    rigid3d(groundTruthPoses.Orientation{1}, groundTruthPoses.Location{1}), ...
    'Color', 'b', 'Opacity', 0);

% Initialize camera trajectories.
trajectoryEstimated = plot3(0, 0, 0, 'g-');
trajectoryActual = plot3(0, 0, 0, 'b-');

legend('Estimated Trajectory', 'Actual Trajectory');
title('Camera Trajectory');

```

Create two graphical camera objects representing the estimated and the actual camera poses based on ground truth data from the New Tsukuba dataset.



# Estimate the Pose of the Second View

```
% Read and display the image.
viewId = 2;
Irgb = readimage(images, viewId);|
step(player, Irgb);

% Convert to gray scale and undistort.
I = undistortImage(im2gray(Irgb), intrinsics);

% Match features between the previous and the current image.
[currPoints, currFeatures, indexPairs] = helperDetectAndMatchFeatures(...
    prevFeatures, I);

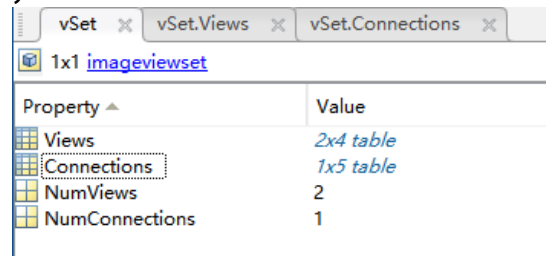
% Estimate the pose of the current view relative to the previous view.
[orient, loc, inlierIdx] = helperEstimateRelativePose(...
    prevPoints(indexPairs(:,1)), currPoints(indexPairs(:,2)), intrinsics);

% Exclude epipolar outliers.
indexPairs = indexPairs(inlierIdx, :);

% Add the current view to the view set.
vSet = addView(vSet, viewId, rigid3d(orient, loc), 'Points', currPoints);

% Store the point matches between the previous and the current views.
vSet = addConnection(vSet, viewId-1, viewId, 'Matches', indexPairs);
```

Detect and extract features from the second view, and match them to the first view using `helperDetectAndMatchFeatures`. Estimate the pose of the second view relative to the first view using `helperEstimateRelativePose`, and add it to the



vSet	
1x1 imageviewset	
Property	Value
Views	2x4 table
Connections	1x5 table
NumViews	2
NumConnections	1

# helperDetectAndMatchFeatures

Detect, extract, and match features

[currPoints, currFeatures, indexPairs] = helperDetectAndMatchFeatures(prevFeatures, I) detects and extract features from image I and matchesthem to prevFeatures. prevFeatures is an M-by-N matrix fof SURF

descriptors. I is a grayscale image. currPoints are the SURF points detected in image I, and currFeatures are the corresponding SURF descriptors. indexPairs is an M-by-2 matrix containing the indices of matches between prevFeatures and currFeatures.

```
function [currPoints, currFeatures, indexPairs] = helperDetectAndMatchFeatures(prevFeatures, I)
numPoints = 150;
% Detect and extract features from the current image.
currPoints = detectSURFFeatures(I, 'MetricThreshold', 500);
currPoints = selectUniform(currPoints, numPoints, size(I));
currFeatures = extractFeatures(I, currPoints, 'Upright', true);

% Match features between the previous and current image.
indexPairs = matchFeatures(prevFeatures, currFeatures, 'Unique', true);
```

# helperEstimateRelativePose

Robustly estimate relative camera pose

[orientation, location, inlierIdx] = helperEstimateRelativePose(matchedPoints1, matchedPoints2, cameraParams) returns the pose of camera 2 in camera 1's coordinate system. The function calls estimateEssentialMatrix and cameraPose functions in a loop, until a reliable camera pose is obtained.

## Inputs:

matchedPoints1 - points from image 1 specified as an M-by-2 matrix of [x,y] coordinates, or as any of the point feature types

matchedPoints2 - points from image 2 specified as an M-by-2 matrix of [x,y] coordinates, or as any of the point feature types

cameraParams - cameraParameters object

## Outputs:

orientation - the orientation of camera 2 relative to camera 1 specified as a 3-by-3 rotation matrix

location - the location of camera 2 in camera 1's coordinate system specified as a 3-element vector

inlierIdx - the indices of the inlier points from estimating the fundamental matrix

```

function [orientation, location, inlierIdx] = ...
    helperEstimateRelativePose(matchedPoints1, matchedPoints2, cameraParams)

if ~isnumeric(matchedPoints1)
    matchedPoints1 = matchedPoints1.Location;
end
if ~isnumeric(matchedPoints2)
    matchedPoints2 = matchedPoints2.Location;
end

for i = 1:100
    % Estimate the essential matrix.
    [E, inlierIdx] = estimateEssentialMatrix(matchedPoints1, matchedPoints2, cameraParams);

    % Make sure we get enough inliers
    if sum(inlierIdx) / numel(inlierIdx) < .3
        continue;
    end

    % Get the epipolar inliers.
    inlierPoints1 = matchedPoints1(inlierIdx, :);
    inlierPoints2 = matchedPoints2(inlierIdx, :);

    % Compute the camera pose from the fundamental matrix. Use half of the
    % points to reduce computation.
    [orientation, location, validPointFraction] = ...
        relativeCameraPose(E, cameraParams, inlierPoints1(1:2:end, :), inlierPoints2(1:2:end, :));

    % validPointFraction is the fraction of inlier points that project in
    % front of both cameras. If the this fraction is too small, then the
    % fundamental matrix is likely to be incorrect.
    if validPointFraction > .8
        return;
    end
end

% After 100 attempts validPointFraction is still too low.
error('Unable to compute the Essential matrix');

```

# Estimate the Pose of the Second View

The location of the second view relative to the first view can only be recovered up to an unknown scale factor. Compute the scale factor from the ground truth using `helperNormalizeViewSet`, simulating an external sensor, which would be used in a typical monocular visual odometry system.

```
vSet = helperNormalizeViewSet(vSet, groundTruthPoses);
```

Update camera trajectory plots using `helperUpdateCameraPlots` and `helperUpdateCameraTrajectories`.

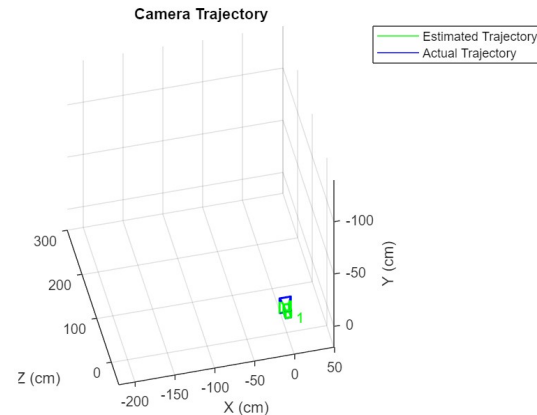
```
helperUpdateCameraPlots(viewId, camEstimated, camActual, poses(vSet), groundTruthPoses);
```

```
helperUpdateCameraTrajectories(viewId, trajectoryEstimated, trajectoryActual,...  
    poses(vSet), groundTruthPoses);
```

```
prevI = I;
```

```
prevFeatures = currFeatures;
```

```
prevPoints = currPoints;
```





# helperNormalizeViewSet

Translate and scale camera poses to align with ground truth

`vSet = helperNormalizedViewSet(vSet, groundTruth)` returns a view set with the camera poses translated to put the first camera at the origin looking along the Z axes, and scaled to match the scale of the ground truth. `vSet` is an `imageviewset` object. `groundTruth` is a table containing the actual camera poses.

```
function vSet = helperNormalizeViewSet(vSet, groundTruth)

camPoses = poses(vSet);

% Move the first camera to the origin.
locations = vertcat(camPoses.AbsolutePose.Translation);
locations = locations - locations(1, :);

locationsGT = cat(1, groundTruth.Location{1:height(camPoses)});
magnitudes = sqrt(sum(locations.^2, 2));
magnitudesGT = sqrt(sum(locationsGT.^2, 2));
scaleFactor = median(magnitudesGT(2:end) ./ magnitudes(2:end));

% Rotate the poses so that the first camera points along the Z-axis
R = camPoses.AbsolutePose(1).Rotation';
for i = 1:height(camPoses)
    % Scale the locations
    camPoses.AbsolutePose(i).Translation = camPoses.AbsolutePose(i).Translation * scaleFactor;
    camPoses.AbsolutePose(i).Rotation = camPoses.AbsolutePose(i).Rotation * R;
end

vSet = updateView(vSet, camPoses);
```

# helperUpdateCameraPlots

```
function helperUpdateCameraPlots(viewId, camEstimated, camActual, ...  
    posesEstimated, posesActual)  
  
% Move the estimated camera in the plot.  
camEstimated.AbsolutePose = posesEstimated.AbsolutePose(viewId);  
  
% Move the actual camera in the plot.  
camActual.AbsolutePose = rigid3d(posesActual.Orientation{viewId},...  
    posesActual.Location{viewId});
```

# helperUpdateCameraTrajectories

```
function helperUpdateCameraTrajectories(viewId, trajectoryEstimated, ...  
    trajectoryActual, posesEstimated, posesActual)  
  
% Plot the estimated trajectory.  
locations = vertcat(posesEstimated.AbsolutePose.Translation);  
set(trajectoryEstimated, 'XData', locations(:,1), 'YData', ...  
    locations(:,2), 'ZData', locations(:,3));  
  
% Plot the ground truth trajectory  
locationsActual = cat(1, posesActual.Location{1:viewId});  
set(trajectoryActual, 'XData', locationsActual(:,1), 'YData', ...  
    locationsActual(:,2), 'ZData', locationsActual(:,3));
```

# Bootstrap Estimating Camera Trajectory Using Global Bundle Adjustment

Find 3D-to-2D correspondences between world points triangulated from the previous two views and image points from the current view. Use `helperFindEpipolarInliers` to find the matches that satisfy the epipolar constraint, and then use `helperFind3Dto2DCorrespondences` to triangulate 3-D points from the previous two views and find the corresponding 2-D points in the current view.

Compute the world camera pose for the current view by solving the perspective-n-point (PnP) problem using `estimateWorldCameraPose`. For the first 15 views, use global bundle adjustment to refine the entire trajectory. Using global bundle adjustment for a limited number of views bootstraps estimating the rest of the camera trajectory, and it is not prohibitively expensive.

```

for viewId = 3:15
    % Read and display the next image
    Irgb = readimage(images, viewId);
    step(player, Irgb);

    % Convert to gray scale and undistort.
    I = undistortImage(im2gray(Irgb), intrinsics);

    % Match points between the previous and the current image.
    [currPoints, currFeatures, indexPairs] = helperDetectAndMatchFeatures(...
        prevFeatures, I);

    % Eliminate outliers from feature matches.
    inlierIdx = helperFindEpipolarInliers(prevPoints(indexPairs(:,1)),...
        currPoints(indexPairs(:, 2)), intrinsics);
    indexPairs = indexPairs(inlierIdx, :);

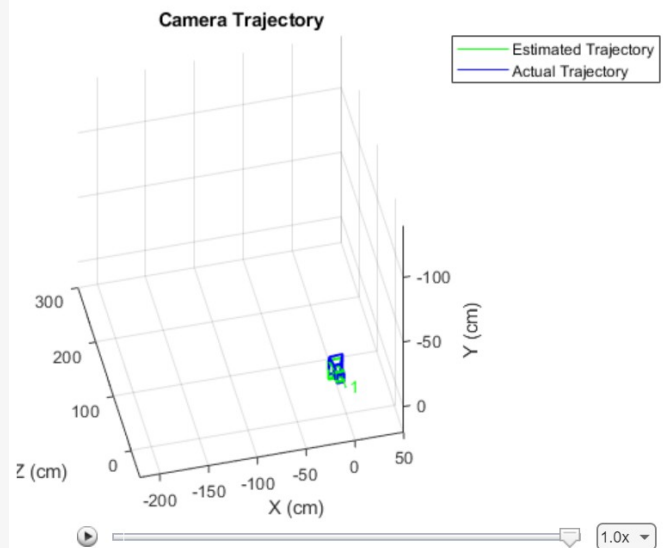
    % Triangulate points from the previous two views, and find the
    % corresponding points in the current view.
    [worldPoints, imagePoints] = helperFind3Dto2DCorrespondences(vSet,...
        intrinsics, indexPairs, currPoints);

    % Since RANSAC involves a stochastic process, it may sometimes not
    % reach the desired confidence level and exceed maximum number of
    % trials. Disable the warning when that happens since the outcomes are
    % still valid.
    warningstate = warning('off', 'vision:ransac:maxTrialsReached');

    % Estimate the world camera pose for the current view.
    [orient, loc] = estimateWorldCameraPose(imagePoints, worldPoints, intrinsics);

    % Restore the original warning state
    warning(warningstate)

```



## helperFindEpipolarInliers

Find epipolar inliers among matched image points

`inlierIdx = helperFindEpipolarInliers(matchedPoints1, matchedPoints2, cameraParams)`

returns indices of matched image points which satisfy the epipolar constraint.

`matchedPoints1` and `matchedPoints2` are sets of matched image points specified as M-by-2 matrices of [x,y] coordinates, or as any of the point feature types.

`cameraParams` is a `cameraParameters` object. `inlierIdx` is a logical index of the

```
function inlierIdx = helperFindEpipolarInliers(matchedPoints1, ...
    matchedPoints2, cameraParams)

% Use the inlierIdx output from helperEstimateRelativePose and ignore the
% orientation and rotation.
[~, ~, inlierIdx] = helperEstimateRelativePose(matchedPoints1, ...
    matchedPoints2, cameraParams);
```

## helperFind3Dto2DCorrespondences

helperFind3Dto2DCorrespondences triangulate points from last two views  
[worldPoints, imagePoints] = helperFind3Dto2DCorrespondences(vSet,  
cameraParams, matchIdx, currPoints) returns world points triangulated from the  
last two views in the view set, which are also visible in the current image.

**vSet** is a viewSet object, **cameraParams** is a cameraParameters object, and  
**matchIdx** is an M-by-2 matrix of matched indices between the points in the last  
view of vSet and the current image.

**worldPoints** is a three-column matrix containing the [x,y,z] coordinates of world  
points which are visible in the last two views of vSet and the current image.

**idxTriplet** is a vector containing the indices of points in the current image  
corresponding to worldPoints.

```
function [worldPoints, imagePoints] = helperFind3Dto2DCorrespondences(vSet, ...  
    cameraParams, matchIdx, currPoints)
```

```
camPoses = poses(vSet);
```

```
% Compute the camera projection matrix for the next-to-the-last view.
```

```
loc1 = camPoses.AbsolutePose(end-1).Translation;  
orient1 = camPoses.AbsolutePose(end-1).Rotation;  
[R1, t1] = cameraPoseToExtrinsics(orient1, loc1);  
camMatrix1 = cameraMatrix(cameraParams, R1, t1);
```

```
% Compute the camera projection matrix for the last view.
```

```
loc2 = camPoses.AbsolutePose(end).Translation;  
orient2 = camPoses.AbsolutePose(end).Rotation;  
[R2, t2] = cameraPoseToExtrinsics(orient2, loc2);  
camMatrix2 = cameraMatrix(cameraParams, R2, t2);
```

```
% Find indices of points visible in all three views.
```

```
matchIdxPrev = vSet.Connections.Matches{end};  
[~, ia, ib] = intersect(matchIdxPrev(:, 2), matchIdx(:, 1));  
idx1 = matchIdxPrev(ia, 1);  
idx2 = matchIdxPrev(ia, 2);  
idxTriplet = matchIdx(ib, 2);
```

```
% Triangulate the points.
```

```
points1 = vSet.Views.Points{end-1};  
points2 = vSet.Views.Points{end};  
worldPoints = triangulate(points1(idx1,:), ...  
    points2(idx2,:), camMatrix1, camMatrix2);  
imagePoints = currPoints(idxTriplet).Location;
```



```

% Add the current view to the view set.
vSet = addView(vSet, viewId, rigid3d(orient, loc), 'Points', currPoints);

% Store the point matches between the previous and the current views.
vSet = addConnection(vSet, viewId-1, viewId, 'Matches', indexPairs);

tracks = findTracks(vSet); % Find point tracks spanning multiple views.
camPoses = poses(vSet);   % Get camera poses for all views.

% Triangulate initial locations for the 3-D world points.
xyzPoints = triangulateMultiview(tracks, camPoses, intrinsics);

% Refine camera poses using bundle adjustment.
[~, camPoses] = bundleAdjustment(xyzPoints, tracks, camPoses, ...
    intrinsics, 'PointsUndistorted', true, 'AbsoluteTolerance', 1e-12, ...
    'RelativeTolerance', 1e-12, 'MaxIterations', 200, 'FixedViewID', 1);

vSet = updateView(vSet, camPoses); % Update view set.

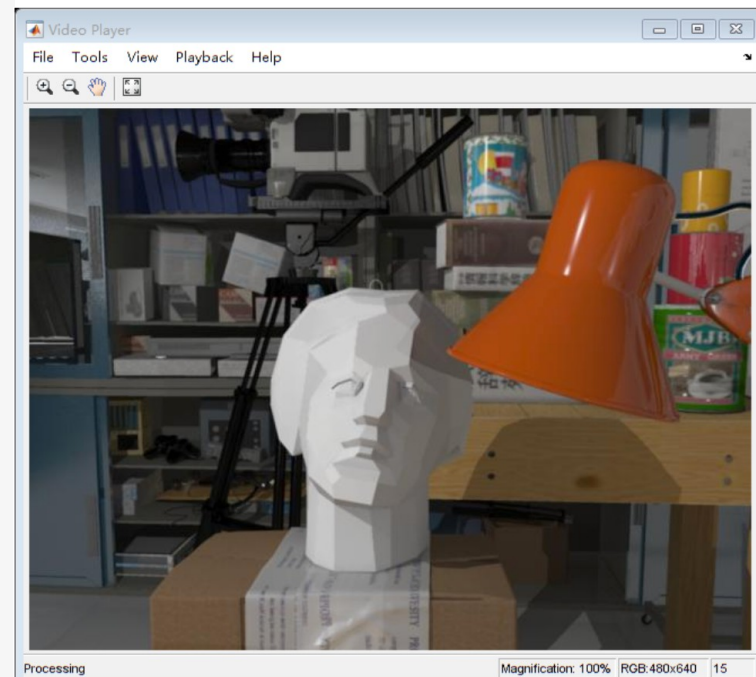
% Bundle adjustment can move the entire set of cameras. Normalize the
% view set to place the first camera at the origin looking along the
% Z-axis and adjust the scale to match that of the ground truth.
vSet = helperNormalizeViewSet(vSet, groundTruthPoses);

% Update camera trajectory plot.
helperUpdateCameraPlots(viewId, camEstimated, camActual, poses(vSet), ...
    groundTruthPoses);
helperUpdateCameraTrajectories(viewId, trajectoryEstimated, ...
    trajectoryActual, poses(vSet), groundTruthPoses);

prevI = I;
prevFeatures = currFeatures;
prevPoints = currPoints;

```

end



## Estimate Remaining Camera Trajectory Using Windowed Bundle Adjustment

Estimate the remaining camera trajectory by using windowed bundle adjustment to only refine the last 15 views, in order to limit the amount of computation. Furthermore, bundle adjustment does not have to be called for every view, because `estimateWorldCameraPose` computes the pose in the same units as the 3-D points. This section calls bundle adjustment for every 7th view. The window size and the frequency of calling bundle adjustment have been chosen experimentally.

```
for viewId = 16:numel(images.Files)
    % Read and display the next image
    Irgb = readimage(images, viewId);
    step(player, Irgb);

    % Convert to gray scale and undistort.
    I = undistortImage(im2gray(Irgb), intrinsics);

    % Match points between the previous and the current image.
    [currPoints, currFeatures, indexPairs] = helperDetectAndMatchFeatures(...
        prevFeatures, I);

    % Triangulate points from the previous two views, and find the
    % corresponding points in the current view.
    [worldPoints, imagePoints] = helperFind3Dto2DCorrespondences(vSet, ...
        intrinsics, indexPairs, currPoints);

    % Since RANSAC involves a stochastic process, it may sometimes not
    % reach the desired confidence level and exceed maximum number of
    % trials. Disable the warning when that happens since the outcomes are
    % still valid.
    warningstate = warning('off','vision:ransac:maxTrialsReached');

    % Estimate the world camera pose for the current view.
    [orient, loc] = estimateWorldCameraPose(imagePoints, worldPoints, intrinsics);

    % Restore the original warning state
    warning(warningstate)

    % Add the current view and connection to the view set.
    vSet = addView(vSet, viewId, rigid3d(orient, loc), 'Points', currPoints);
    vSet = addConnection(vSet, viewId-1, viewId, 'Matches', indexPairs);
```

```

% Refine estimated camera poses using windowed bundle adjustment. Run
% the optimization every 7th view.
if mod(viewId, 7) == 0
    % Find point tracks in the last 15 views and triangulate.
    windowSize = 15;
    startFrame = max(1, viewId - windowSize);
    tracks = findTracks(vSet, startFrame:viewId);
    camPoses = poses(vSet, startFrame:viewId);
    [xyzPoints, reprojErrors] = triangulateMultiview(tracks, camPoses, intrinsics);

    % Hold the first two poses fixed, to keep the same scale.
    fixedIds = [startFrame, startFrame+1];
    % Exclude points and tracks with high reprojection errors.
    idx = reprojErrors < 2;

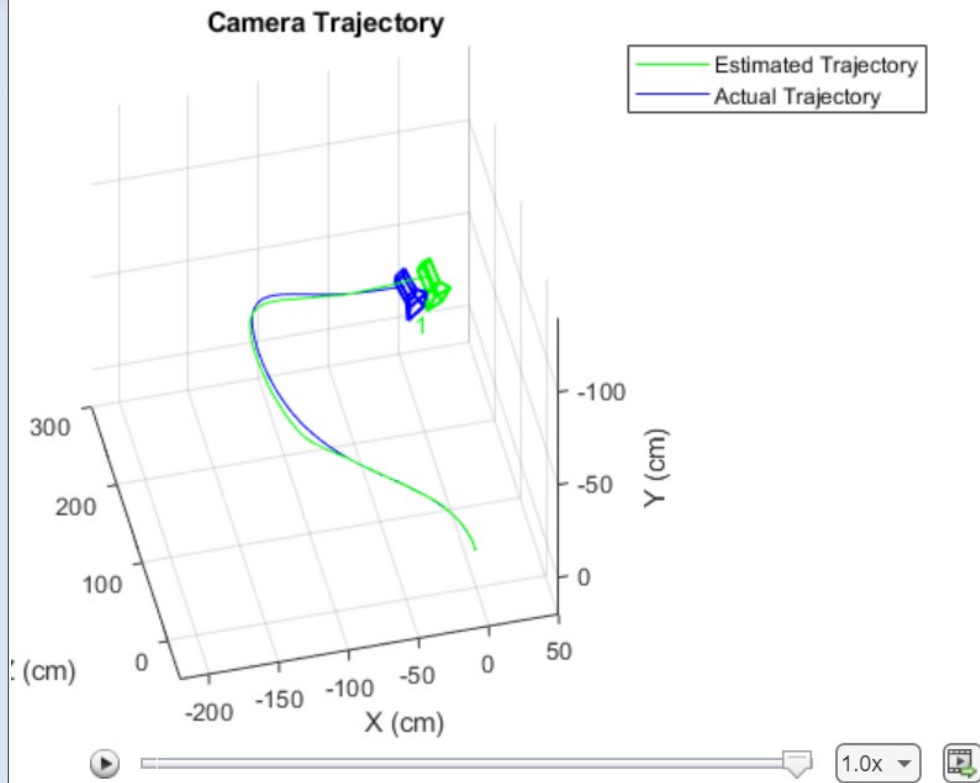
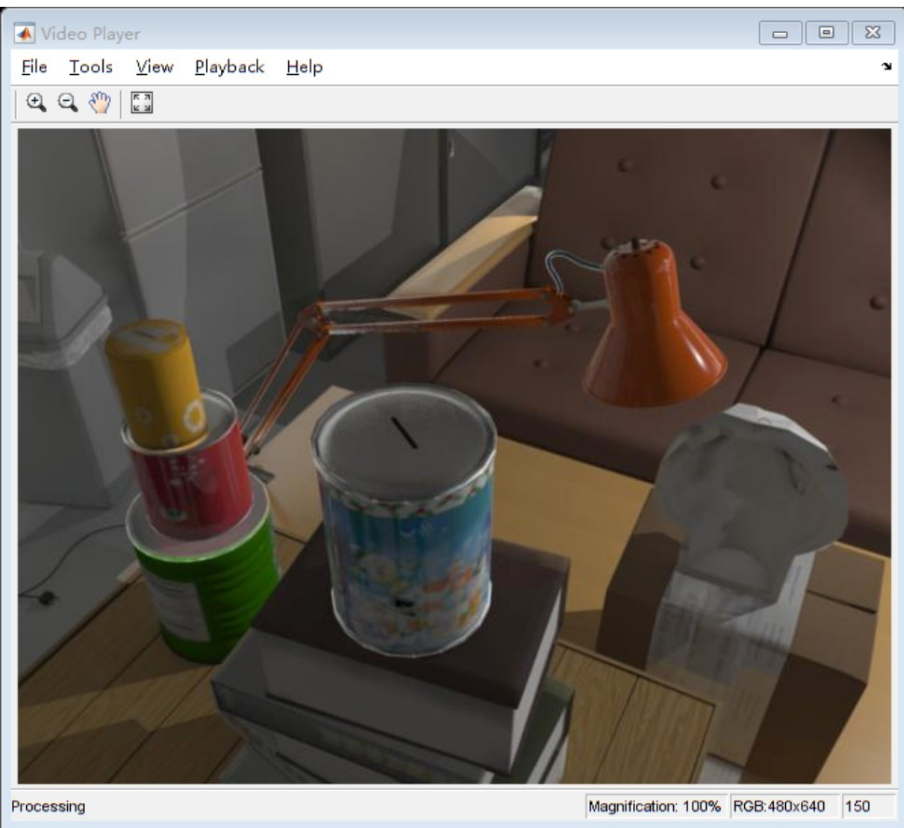
    [~, camPoses] = bundleAdjustment(xyzPoints(idx, :), tracks(idx), ...
        camPoses, intrinsics, 'FixedViewIDs', fixedIds, ...
        'PointsUndistorted', true, 'AbsoluteTolerance', 1e-12,...
        'RelativeTolerance', 1e-12, 'MaxIterations', 200);

    vSet = updateView(vSet, camPoses); % Update view set.
end

% Update camera trajectory plot.
helperUpdateCameraPlots(viewId, camEstimated, camActual, poses(vSet), ...
    groundTruthPoses);
helperUpdateCameraTrajectories(viewId, trajectoryEstimated, ...
    trajectoryActual, poses(vSet), groundTruthPoses);

prevI = I;
prevFeatures = currFeatures;
prevPoints = currPoints;
end
hold off

```



## Summary

This example showed how to estimate the trajectory of a calibrated monocular camera from a sequence of views. Notice that the estimated trajectory does not exactly match the ground truth. Despite the non-linear refinement of camera poses, errors in camera pose estimation accumulate, resulting in drift. In visual odometry systems this problem is typically addressed by fusing information from multiple sensors, and by performing loop closure.

# SLAM 示例代码

%Rotations, Orientation, and Quaternions

openExample('shared\_positioning/QuaternionExample')

%Monocular Visual Odometry

openExample('vision/VisualOdometryExample')