# 1. FEATURE-BASED PANORAMIC IMAGE STITCHING

## Algorithm

1. Load Images
2. Register Image Pairs

- Start by detecting and matching features between I(n) and I(n-1)

```
% Initialize features for I(1)
grayImage = rgb2gray(I);
points = detectSURFFeatures(grayImage);
[features, points] = extractFeatures(grayImage, points);
indexPairs = matchFeatures(features, featuresPrevious,
'Unique', true);
```

- From them, calculate the geometric and mapping transformation

3. Initialize Panorama

```
tforms(n) = estimateGeometricTransform(matchedPoints,
   matchedPointsPrev, 'projective', 'Confidence', 99.9,
   'MaxNumTrials', 2000);
 tforms(n).T = tforms(n).T * tforms(n-1).T;
```

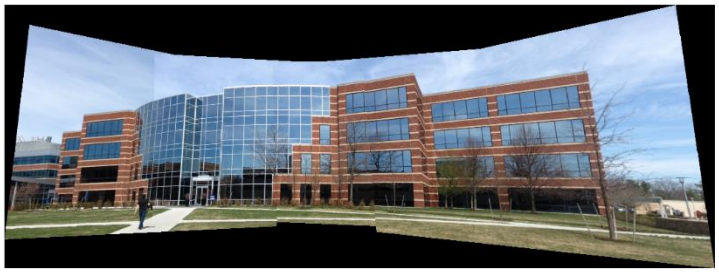
- First find max, min limits and calculate the size of panorama.

4. Create Panorama

- imwarp is used to map images into the blank panorama and vision.AlphaBlender is used for overlaying

```
  warpedImage = imwarp(I, tforms(i), 'OutputView',
  panoramaView);
  mask = imwarp(true(size(I,1),size(I,2)), tforms(i),
  'OutputView', panoramaView);
  panorama = step(blender, panorama, warpedImage, mask);
```

## Results



## Interpretations

This is the basics of stitching images into panorama. We use SURF features to compute necessary transformation. Lens distorted images can create problems and they should be camera calibrated.

# 2. EVALUATING THE ACCURACY OF A SINGLE CAMERA CALIBRATION

## Algorithm

1. Calibrating the Camera
   a. Detect calibration pattern from images
   b. Generate world coordinates

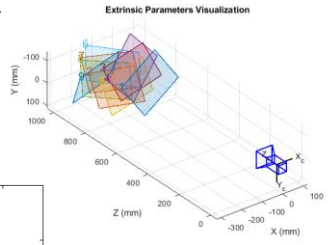
```
squareSize = 29; % mm
worldPoints = generateCheckerboardPoints(boardSize,
squareSize);
```

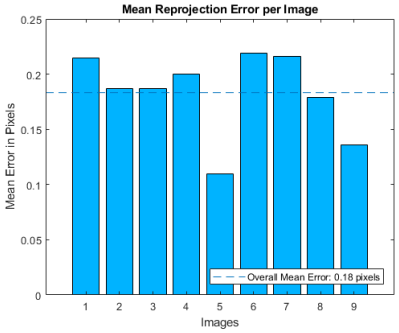
   c. Calibrate the camera

```
I = readimage(images, 1);
imageSize = [size(I, 1), size(I, 2)];
[params, ~, estimationErrors] =
estimateCameraParameters(imagePoints, worldPoints,
'ImageSize', imageSize);
```

2. Plot the calibration pattern in camera's coordinates or location of camera in pattern's coordinates. `showExtrinsics` function is used for this.
3. Mean error in reprojection is found with. `showReprojectionErrors` function.
4. `estimateCameraParameters` function gives estimation errors which show the uncertainty in each estimated parameter. Using the standard error (sigma) of this, we can calculate confidence intervals by simple statist

## Results




## Interpretations

Camera calibration is a vital preprocessing step in any machine vision application. This removes the intrinsic error introduced by camera. Here we calculate camera parameters. The average gap between a world point and its's reprojection is Mean Error in Reprojection. This is a good qualitative measure of accuracy in those parameters.

# 3.
# MEASURING PLANAR OBJECTS WITH A CALIBRATED CAMERA

## Algorithm

1. Calibrating the Camera
2. Undistort the Image

UndistortImage function is used with the obtained camera parameters.

```
[im, newOrigin] = undistortImage(imOrig, cameraParams,
'OutputView', 'full');
figure; imshow(im, 'InitialMagnification',
magnification);
```

3. Segmentation

In this example, we use HSV Thresholding.

```
imHSV = rgb2hsv(im);
saturation = imHSV(:, :, 2);
t = graythresh(saturation);
imCoin = (saturation > t);
```

4. Detect Coins

a. Find the connected components by blobAnalysis.
b. Sort all connected components by area and get the two largest (we assume these are coins)
c. Adjust for shift caused by undistortImage

```
blobAnalysis = vision.BlobAnalysis('AreaOutputPort',
true,'CentroidOutputPort', false,
'BoundingBoxOutputPort','MinimumBlobArea', 200,
'ExcludeBorderBlobs', true);
[areas, boxes] = step(blobAnalysis, imCoin);
[~, idx] = sort(areas, 'Descend');
boxes = double(boxes(idx(1:2), :));
boxes(:, 1:2) = bsxfun(@plus, boxes(:, 1:2),
newOrigin);
```

5. Calculate Extrinsic

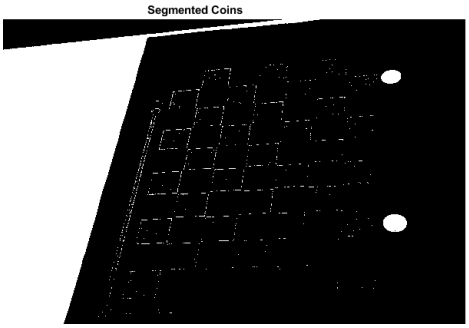
(as in 2)

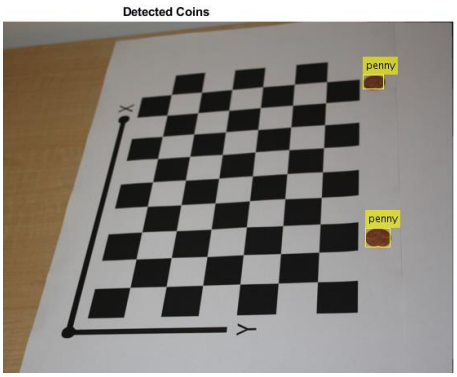
6. Measure a Coin

We measure the width of the bounding box and find distance in mm.

```
% Get the top-left and the top-right corners.
box1 = double(boxes(1, :));
imagePoints1 = [box1(1:2); ...
                box1(1) + box1(3), box1(2)];
% Get the world coordinates of the corners
worldPoints1 = pointsToWorld(cameraParams, R, t,
imagePoints1);
% Compute the diameter of the coin in millimeters.
d = worldPoints1(2, :) - worldPoints1(1, :);
diameterInMillimeters = hypot(d(1), d(2));
fprintf('Measured diameter of one penny = %0.2f mm\n',
diameterInMillimeters);
```

## Results




## Interpretations

Here we measuring size of a coin after calibrating a camera. We are able to use saturation component of hue, saturation, value to segment them.

# 5.
# STEREO CALIBRATION AND SCENE RECONSTRUCTION

## Algorithm

1. Specify Calibration Images

Images stored in two arrays.

```
numImagePairs = 10;
imageFiles1 = cell(numImagePairs, 1);
imageFiles2 = cell(numImagePairs, 1);
imageDir  =  fullfile(matlabroot,  'toolbox',  'vision',
'visiondemos', 'calibration', 'stereoWebcams');

for i = 1:numImagePairs
    imageFiles1{i}=fullfile(imageDir,
sprintf('left%02d.png', i));
    imageFiles2{i}          =          fullfile(imageDir,
sprintf('right%02d.png', i));
end
```

2. Try Detecting Checkerboards

Here, `detectCheckerboardPoints` function fails due to the texture of the wall.

```
% Try to detect the checkerboard
im = imread(imageFiles1{1});
imagePoints = detectCheckerboardPoints(im);

% Display the image with the incorrectly detected
checkerboard
figure; imshow(im, 'InitialMagnification', 50);
hold on;
plot(imagePoints(:, 1), imagePoints(:, 2), '*-g');
title('Failed Checkerboard Detection');
```

3. Detect Checkerboards

We mask the wall using `imtool` or `imrect`

```
images1 = cast([], 'uint8');
images2 = cast([], 'uint8');
for i = 1:numel(imageFiles1)
    im = imread(imageFiles1{i});
    im(3:700, 1247:end, :) = 0;
    images1(:, :, :, i) = im;
    im = imread(imageFiles2{i});
    im(1:700, 1198:end, :) = 0;
    images2(:, :, :, i) = im;
end
[imagePoints,boardSize]=
detectCheckerboardPoints(images1, images2);
```

4. Calibrate the Stereo Camera System as in (2)

5. Rectify a Pair of Images

```
% Rectify the images.
[J1, J2] = rectifyStereoImages(I1, I2, stereoParams);
```

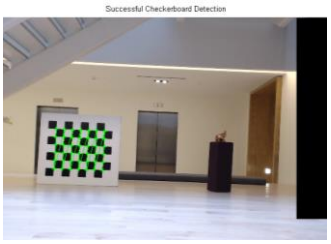
6. Compute Disparity, Reconstruct 3D

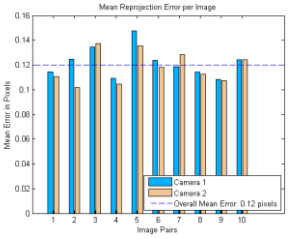
Follow steps of (6).

7. Visualize 3D Scene

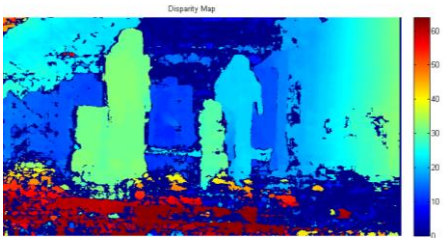
```
% Reduce the number of colors in the image to 128.
[reducedColorImage, reducedColorMap] = rgb2ind(J1, 128);
% Plot the 3D points of each color.
hFig = figure; hold on;
set(hFig, 'Position', [1 1 840   630]);
hAxes = gca;
X = pointCloud(:, :, 1);
Y = pointCloud(:, :, 2);
Z = pointCloud(:, :, 3);
for i = 1:size(reducedColorMap, 1)
    % Find the pixels of the current color.
    x = X(reducedColorImage == i-1);
    y = Y(reducedColorImage == i-1);
    z = Z(reducedColorImage == i-1);
    if isempty(x)
        continue;
    end
    % Eliminate invalid values.
    idx = isfinite(x);
    x = x(idx);
    y = y(idx);
    z = z(idx);
```

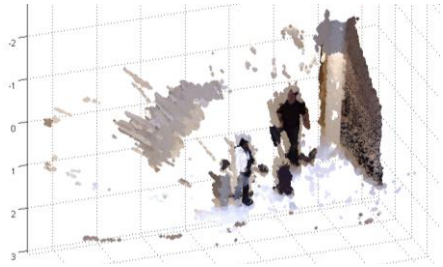
## Results







## Interpretations

Here, we estimate the intrinsic parameters and the translation and rotation of the two cameras. A stereo pair of images can be rectified after this and that is used to compute the disparity map which in turn is used to reconstruct the 3-D scene.

Cameras are calibrated by multiple pairs of images on plane surface of a known size calibration pattern (like checkerboard) at equal distance taken from different angles. These images should be stored in lossless formats (PNG) to retain accuracy. Orientation of the checkerboards are detected by `detectCheckerboardPoints` function.

# 4.
# UNCALIBRATED STEREO IMAGE RECTIFICATION

## Algorithm

### 1. Read Stereo Image Pair in Grayscale

Two color images of the same scene, taken from different positions are converted to grayscale. Then a color composite of pixel-wise differences is generated.

```
imshow(stereoAnaglyph(I1,I2));
title('Composite Image (Red - Left Image, Cyan - Right Image)');
```

### 2. Collect Interest Points

We collect points of interest from both images, and then choose potential matches between them. `detectSURFFeatures` function is used. Then we visualize the location and scale of the thirty strongest SURF features in imaged with using `plot(selectStrongest(blobs1, 30));`

### Step 3: Putative Point Correspondences are Found

`extractFeatures` and `matchFeatures` functions and Sum of absolute differences (SAD) metric is used.

```
indexPairs = matchFeatures(features1, features2,
'Metric', 'SAD', ...
  'MatchThreshold', 5);

matchedPoints1 = validBlobs1(indexPairs(:,1),:);
matchedPoints2 = validBlobs2(indexPairs(:,2),:);

figure;
showMatchedFeatures(I1, I2, matchedPoints1,
matchedPoints2);
```

### 4. Outliers are removed Using Epipolar Constraint

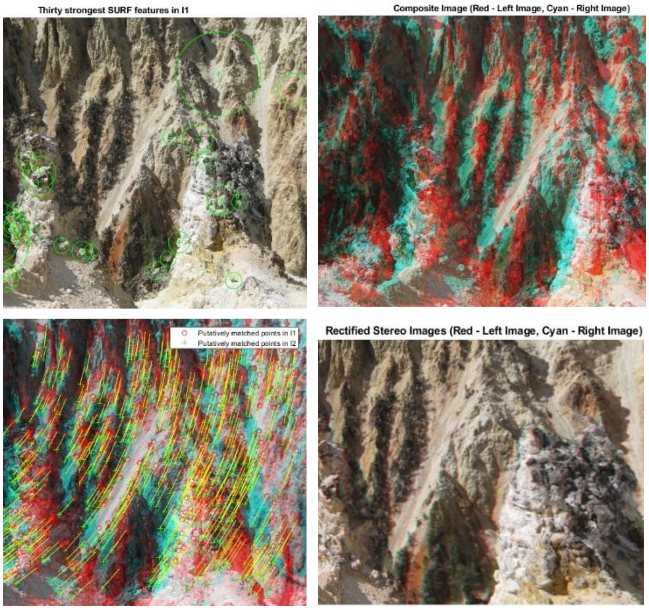
`estimateFundamentalMatrix` function is used to compute the fundamental matrix and find the inliers that meet the epipolar constraint.

```
[fMatrix, epipolarInliers, status] =
estimateFundamentalMatrix(...
  matchedPoints1, matchedPoints2, 'Method', 'RANSAC',
  'NumTrials', 10000, 'DistanceThreshold', 0.1,
'Confidence', 99.99);

inlierPoints1 = matchedPoints1(epipolarInliers, :);
inlierPoints2 = matchedPoints2(epipolarInliers, :);
```

### Step 5: Rectification

`estimateUncalibratedRectification` function is used to compute the rectification transformations and `rectifyStereoImages` is used to apply them.

## Results




## Interpretations

Rectification can be useful for stereo vision, since the problem of 2-D stereo correspondence is reduced to a 1-D problem.

In color composite, a visible offset is present between the images in orientation and position. We align them by rectification. Not all of the detected features can be matched because they were not detected in both images or because some of them were not present in one of the images due to camera motion.

Outliers seen when matching points on top of the composite image can be removed by the fact that correctly matched point must lie on the epipolar line determined by its corresponding point. Rectified images can be shown as a stereo anaglyph that can be seen in 3D with red-cyan stereo glasses.

**Note**: We may use the `cvexRectifyStereoImages` function, which contains additional logic to automatically adjust the rectification parameters in a generalized algorithm

# 6.
# DEPTH ESTIMATION FROM STEREO VIDEO

## Algorithm

### 1. Load the Parameters of the Stereo Camera

Loading the `stereoParameters` object gives parameters of calibrated camera.

### 2. Create Video File Readers and the Video Player

Objects created for reading and displaying the video.

```
videoFileLeft = 'handshake_left.avi';
videoFileRight = 'handshake_right.avi';

readerLeft = vision.VideoFileReader(videoFileLeft,
'VideoOutputDataType', 'uint8');
readerRight = vision.VideoFileReader(videoFileRight,
'VideoOutputDataType', 'uint8');
player = vision.DeployableVideoPlayer('Location', [20,
400]);
```

### 3. Read and Rectify Video Frames

`rectifyStereoImages` is used as before between cameras.

### 4. Disparity Computation

Disparity is the distance for each pixel in the left image to the corresponding pixel in the right image.

```
frameLeftGray  = rgb2gray(frameLeftRect);
frameRightGray = rgb2gray(frameRightRect);

disparityMap = disparity(frameLeftGray,
frameRightGray);
```

### 3. Reconstruct the 3-D Scene

Reconstruct and view the 3-D point cloud using the disparity map.

```
points3D = reconstructScene(disparityMap,
stereoParams);

points3D = points3D ./ 1000;
ptCloud = pointCloud(points3D, 'Color', frameLeftRect);

player3D = pcplayer([-3, 3], [-3, 3], [0, 8],
'VerticalAxis', 'y', ...
    'VerticalAxisDir', 'down');
view(player3D, ptCloud);
```

### 4. Detect People in the Left Image

`vision.PeopleDetector` system object is used.

```
peopleDetector = vision.PeopleDetector('MinSize', [166
83]);
bboxes = peopleDetector.step(frameLeftGray);
```

### 4. Determine Distance of Each Person to the Camera

3-D world coordinates of the centroid of each detected person is found and used to compute the distance from the centroid to the camera in meters.

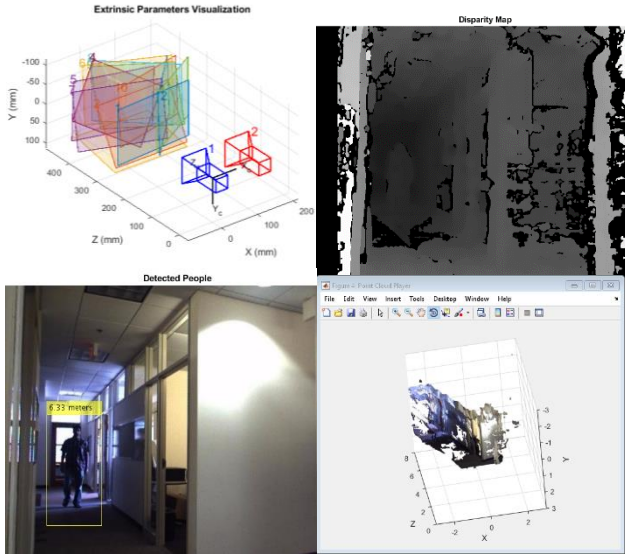
```
readerLeft = vision.VideoFileReader(videoFileLeft,
'VideoOutputDataType', 'uint8');
readerRight = centroidsIdx =
sub2ind(size(disparityMap), centroids(:, 2),
centroids(:, 1));
X = points3D(:, :, 1);
Y = points3D(:, :, 2);
Z = points3D(:, :, 3);
centroids3D = [X(centroidsIdx)'; Y(centroidsIdx)';
Z(centroidsIdx)'];
% Find the distances from the camera in meters.
dists = sqrt(sum(centroids3D .^ 2));
vision.VideoFileReader(videoFileRight,
'VideoOutputDataType', 'uint8');
player = vision.DeployableVideoPlayer('Location', [20,
400]);
```

### 5. Process the Rest of the Video

Above steps are repeated throughout the video with

```
while ~isDone(readerLeft) && ~isDone(readerRight)
```

## Results




## Interpretations

Here we people in video taken with a calibrated stereo camera and determine their distances from the camera. The frames from the left and the right cameras must be rectified as in (4) to reconstruct the 3-D scene. Calculated disparity is proportional to the distance of the corresponding world point from the camera. People detection object size is limited for speed.

# 7.
# STRUCTURE FROM MOTION FROM TWO VIEWS

## Algorithm

1. Read a Two Images
2. Load Camera Parameters
3. Remove Lens Distortion

`undistortImage` function is used. This process straightens the lines that are bent by the radial distortion of the lens.

4. Find Point Correspondences Between The Images

Detect good features to track. Reduce `'MinQuality'` to detect fewer points, which would be more uniformly distributed throughout the image.

```
% Detect feature points
imagePoints1 = detectMinEigenFeatures(rgb2gray(I1),
'MinQuality', 0.1);
% Create the point tracker
tracker = vision.PointTracker('MaxBidirectionalError',
1, 'NumPyramidLevels', 5);
% Initialize the point tracker
imagePoints1 = imagePoints1.Location;
initialize(tracker, imagePoints1, I1);
% Track the points
[imagePoints2, validIdx] = step(tracker, I2);
matchedPoints1 = imagePoints1(validIdx, :);
matchedPoints2 = imagePoints2(validIdx, :);
```

5. Estimate the Essential Matrix

Finding the inlier points that meet the epipolar constraint as before.

6. Compute the Camera Pose

Location and orientation of the second camera relative to the first one is found.

```
[orient, loc] = relativeCameraPose(E, cameraParams,
inlierPoints1, inlierPoints2);
```

7. Reconstruct the 3-D Locations of Matched Points

More points got from first image with lower `'MinQuality'`. New points tracked into the second image and 3-D locations corresponding to the matched points found using the `triangulate` function.

```
% Detect dense feature points.
roi = [30, 30, size(I1, 2) - 30, size(I1, 1) - 30];
imagePoints1 = detectMinEigenFeatures(rgb2gray(I1), 'ROI', roi,
‘MinQuality', 0.001);
% Create and initialize the point tracker
tracker = vision.PointTracker('MaxBidirectionalError', 1,
'NumPyramidLevels', 5);
imagePoints1 = imagePoints1.Location;
initialize(tracker, imagePoints1, I1);
% Track the points
[imagePoints2, validIdx] = step(tracker, I2);
matchedPoints1 = imagePoints1(validIdx, :);
matchedPoints2 = imagePoints2(validIdx, :);
% Compute the camera matrices
camMatrix1 = cameraMatrix(cameraParams, eye(3), [0 0 0]);
% Compute extrinsics of the second camera
[R, t] = cameraPoseToExtrinsics(orient, loc);
camMatrix2 = cameraMatrix(cameraParams, R, t);
% Compute the 3-D points
points3D = triangulate(matchedPoints1, matchedPoints2, camMatrix1,
camMatrix2);
% Get the color of each reconstructed point
numPixels = size(I1, 1) * size(I1, 2);
allColors = reshape(I1, [numPixels, 3]);
colorIdx = sub2ind([size(I1, 1), size(I1, 2)], round(matchedPoints1(:,2)),
round(matchedPoints1(:, 1)));
color = allColors(colorIdx, :);
% Create the point cloud
ptCloud = pointCloud(points3D, 'Color', color);
```

8. Show the 3-D Point Cloud

`pcshow` function used to visualize the point cloud.

9. Fit a Sphere to the Point Cloud

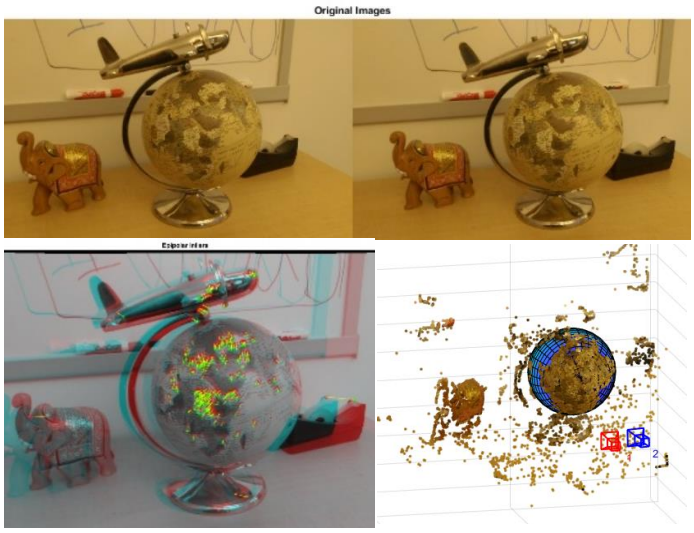
Find the globe in the point cloud by fitting a sphere to the 3-D points using the `pcfitsphere` function.

10.Metric Reconstruction

Result is scaled using the actual radius of the globe: 10cm.

```
scaleFactor = 10 / globe.Radius;
ptCloud = pointCloud(points3D * scaleFactor, 'Color',
color);
loc = loc * scaleFactor;
```

## Results




## Interpretation

Here, we use two views of a scene to create a 3D scene of unknown scale and use the size of one object there to solve it.

Lens distortion affects the accuracy of the reconstruction; hence it is removed. If the motion of the camera is not very large, then tracking using the KLT algorithm is a good way for point correspondences.

# 8.
# 3-D POINT CLOUD REGISTRATION AND STITCHING

## Algorithm

### Registering Two Point Clouds

We preprocess the data by downsampling with a box grid filter and set the size of grid filter to be 10cm to filter noise. This divides the point cloud space into cubes and averages them into single output point.

```
gridSize = 0.1;
fixed = pcdownsample(ptCloudRef, 'gridAverage', gridSize);
moving = pcdownsample(ptCloudCurrent, 'gridAverage', gridSize);
```

ICP algorithm is used in aligning downsampled data. Using the first point cloud as the reference we apply the estimated transformation to the original second point cloud. Then the scene point cloud is merged with the aligned point cloud to process the overlapped points.

```
tform = pcregrigid(moving, fixed,
'Metric','pointToPlane','Extrapolate', true);
ptCloudAligned = pctransform(ptCloudCurrent,tform);
```

Now world scene with the registered data is created. Overlapped region is filtered using a 1.5cm box grid filter here.

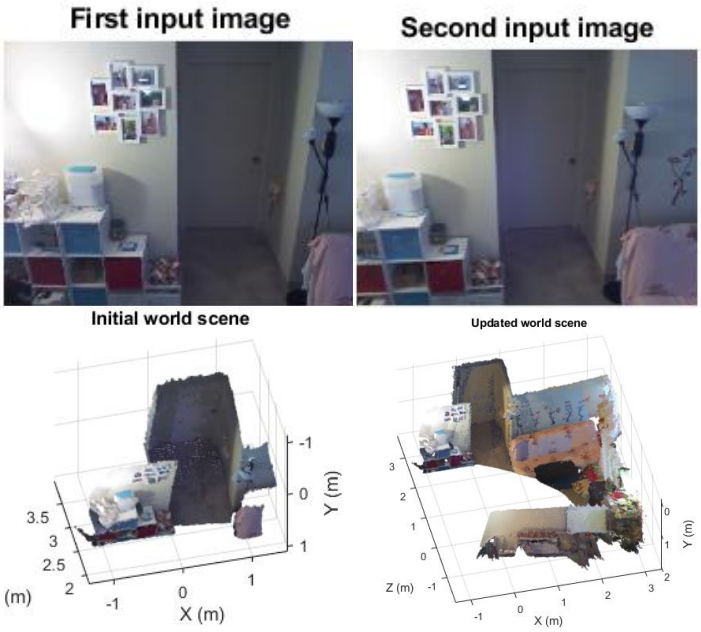
```
mergeSize = 0.015;
ptCloudScene = pcmerge(ptCloudRef, ptCloudAligned,
mergeSize);
```

### Stitch a Sequence of Point Clouds

If we need to compose a larger 3-D scene, we repeat the same procedure process a sequence of point clouds. We use the first point cloud to establish the reference coordinate system. Transform each point cloud to the reference coordinate system by successive multiplications of pairwise transformations.

```
% Store the transformation object that accumulates the
transformation.
accumTform = tform;

figure
hAxes = pcshow(ptCloudScene, 'VerticalAxis','Y',
'VerticalAxisDir', 'Down');
hScatter = hAxes.Children;

for i = 3:length(livingRoomData)
    ptCloudCurrent = livingRoomData{i};
    % Use previous moving point cloud as reference.
    fixed = moving;
    moving = pcdownsample(ptCloudCurrent,
'gridAverage', gridSize);
    % Apply ICP registration.
    tform = pcregrigid(moving, fixed,
'Metric','pointToPlane','Extrapolate', true);

    % Transform the current point cloud to the
reference coordinate system
    % defined by the first point cloud.
    accumTform = affine3d(tform.T * accumTform.T);
    ptCloudAligned = pctransform(ptCloudCurrent,
accumTform);

    % Update the world scene.
    ptCloudScene = pcmerge(ptCloudScene,
ptCloudAligned, mergeSize);

    % Visualize the world scene.
    hScatter.XData = ptCloudScene.Location(:,1);
    hScatter.YData = ptCloudScene.Location(:,2);
    hScatter.ZData = ptCloudScene.Location(:,3);
    hScatter.CData = ptCloudScene.Color;
    drawnow('limitrate')
end
```

## Results




## Interpretations

Here we use a set of point clouds taken with a Kinect to reconstruct a scene. ICP (Iterative Closest Point Algorithm) is used. This can be used in navigation, military and many other applications.

The quality of registration depends on data noise and initial settings of the ICP algorithm. Downsampling increases accuracy in addition to speed. Merge size can be increased to reduce the storage requirement of the resulting scene point cloud, and decrease the merge size to increase the scene resolution.