

Canny Edge Detection

- Staged Algorithm
- Invented by John F. Canny in 1986

Stage 1: Noise Reduction

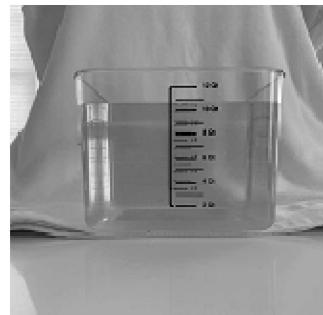
A noisy image can be problematic due to the reliance of this algorithm on gradient calculations in subsequent steps. So, first stage is to filter the image. The most common filter is a 5x5 Gaussian filter like the one we implemented in the homework earlier. Python has this functionality built in to several of its libraries. We will use OpenCV here because it has tools we will use later as well.

In [323]:

```
1  '''Import Libraries'''
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import matplotlib.image as mpimg
5  import cv2 as cv
6
7  inFile = 'test2.png'
8  vidFile = 'Math 7203 Mini Project 1 Video ( 1 min).mp4'
9  testPNGs = ['IMG02.png','IMG03.png','IMG04.png','IMG05.png',
10           'IMG06.png','IMG07.png']
11
12 '''Helper Functions For Displaying Images'''
13 def showAll(img_list):
14     '''Plot All the Images Passed in as a List of
15        ndarrays'''
16     N_ROWS = 2      # number of rows for display of pics
17     R_SIZE = 3      # row cells size
18     N_COLS = 3      # number of columns for display of pics
19     C_SIZE = 3      # columns cell size
20     # Display Images
21     plt.figure(figsize=(N_COLS*C_SIZE, N_ROWS*R_SIZE))
22     for i,img in enumerate(img_list):
23         plt.subplot(N_ROWS, N_COLS, i+1)
24         plt.imshow(img, cmap="gray",)
25         plt.axis('off')
26
27     plt.tight_layout()
28     plt.show()
29
```

In [324]:

```
1 '''Read in Files and Display'''
2 imgs = []          #list to hold raw images
3
4 # Read Images
5 for img in testPNGs:
6     imgs.append(cv.imread(img, cv.IMREAD_GRAYSCALE))
7
8 # Display images
9 showAll(imgs)
10
11 # display what format the image data is stored in
12 print('File Numeric Type: ', type(imgs[0][0,0]))
```



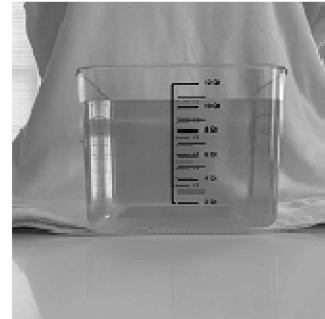
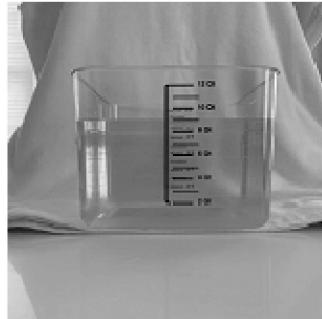
File Numeric Type: <class 'numpy.uint8'>

In [325]:

```

1  '''Filter images And Display Again'''
2  filtImgs = [] # list to hold filtered images
3
4  SIGMA = 0.25    # x-sigma, is copied to y-sigma if y-sigma is not specified
5  K_SIZE = (5,5)  # Kernel Window Size
6
7  # filter images
8  for img in imgs:
9      # Documentation for call from OpenCV Help
10     # ...GaussianBlur(src, ksize, sigmaX[, dst[, sigmaY[, borderType]]])
11     filtImgs.append(cv.GaussianBlur(img,K_SIZE,SIGMA,cv.BORDER_REPLICATE))
12
13 # Display filtered images
14 showAll(filtImgs)
15
16 # Display what format the image data is stored in
17 print('File Numeric Type: ', type(filtImgs[0][0,0]))

```



File Numeric Type: <class 'numpy.uint8'>

Stage 2: Intensity Gradient of Image

The edge gradient is calculated for the image through use of the Sobel operator and the resulting vectors are stored in angle and magnitude form. The angle is typically rounded to one of a few selected values to make the following steps easier. For our purposes, we will round to either 0, 45, 90, 135 degrees adjusting angles in the 3rd and 4th quadrant in to the 1st or 2nd.

Sobel Operator

Let I_{mg} be the matrix representing the image file and SO be the Sobel Operator, then

$$SO = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

$$\nabla_x = SO * I_{mg}$$

and,

$$\nabla_y = SO^T * I_{mg}$$

where, $*$ denotes convolution

Intensity Gradient Calculation

Let the Edge Gradient Magnitude = M and the Angle = Θ then

$$M = \sqrt{\nabla_x^2 + \nabla_y^2}$$

$$\Theta = \tan^{-1}\left(\frac{\nabla_y}{\nabla_x}\right)$$

Example Calculation No Edge

Example Calculation With Edge

In [326]:

```
1  '''Define function for the Sobel Operator'''
2
3  def sobel(A):
4      '''Applies 3x3 Sobel Operator to Image Array and
5          returns Magnitude and Angle Arrays'''
6
7      # Sobel Operator
8      Gx = np.matrix([[-1,0,1],[-2,0,2],[-1,0,1]])
9      Gy = Gx.T
10
11     # Get Dimensions of kernel and image
12     kr,kc = Gx.shape
13     r,c = A.shape
14
15     # Dimension Output Arrays
16     mag = np.zeros((r,c))
17     angle = np.zeros((r,c))
18
19
20     # Apply Sobel via Convolution
21     o_r = kr//2 # offset for rows to center
22     o_c = kc//2 # offset for columns to center
23     for i in range(r - kr):
24         for j in range(c - kc):
25             # Perform X and Y Gradient Calcs on Slices of Image
26             S1 = np.sum(np.multiply(Gx,A[i:i+kr,j:j+kc]))
27             S2 = np.sum(np.multiply(Gy,A[i:i+kr,j:j+kc]))
28
29             # Calculate Magnitude and Angle for each pixel
30             mag[i+ o_r, j+o_c] = np.sqrt(S1*S1+S2*S2)
31             angle[i+ o_r, j+o_c] = np.arctan2(S2,S1)
32
33     return mag, angle
34
```

In [327]:

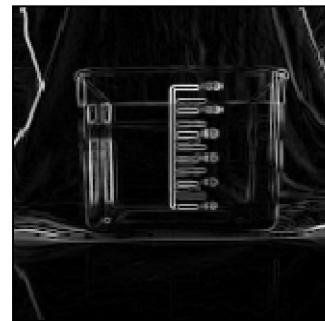
```

1  '''Process Images with Sobel Operator and Display'''
2  mags = []      # list to hold gradient magnitudes
3  angs = []      # list to hold gradient angles
4
5  # Process Filtered Images
6  N = len(filtImgs)  # Get number of images to process
7  for i,img in enumerate(filtImgs):
8      print('Processing {:d} of {:d} images....'.format(i+1,N))
9      m,a = sobel(img)
10     mags.append(m)
11     angs.append(a)
12
13 print('Processing Complete!')
14
15 # Display Magnitude Images
16 print('===== Magnitudes =====')
17 showAll(mags)
18
19 # Display Angle Images
20 print('===== Angles =====')
21 showAll(angs)

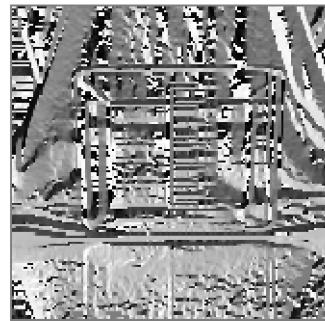
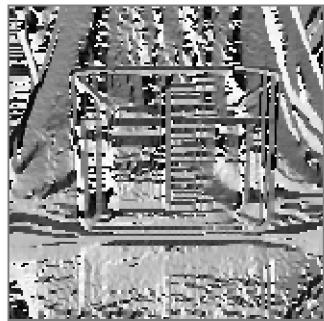
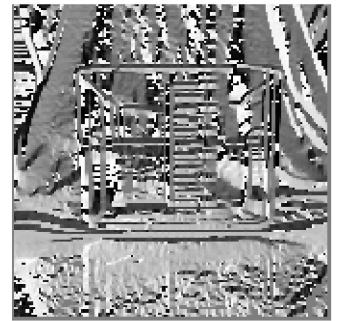
```

Processing 1 of 6 images....
 Processing 2 of 6 images....
 Processing 3 of 6 images....
 Processing 4 of 6 images....
 Processing 5 of 6 images....
 Processing 6 of 6 images....
 Processing Complete!

===== Magnitudes =====



===== Angles =====



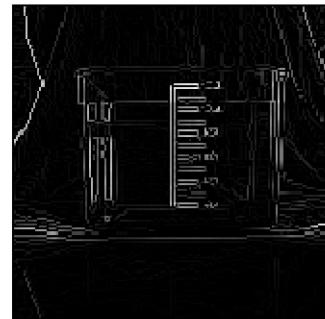
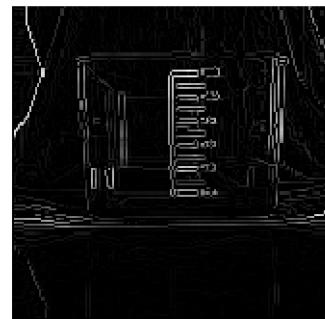
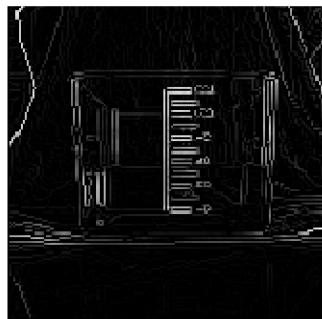
Stage 3: Non-maximum Suppression

need to write up...make pictures

```
In [328]: 1 def nonMaxSupression(m,a):
2     '''Round angle in radians to either 0°, 45°, 90°,
3         or 135° and check neighbors in direction of gradient'''
4
5     BP1 = 0.3927          # 22.5°
6     BP2 = 1.1781          # 67.5°
7     BP3 = 1.9635          # 112.5°
8     BP4 = 2.7489          # 157.5°
9
10    s = m.copy()          # copy array
11    r,c = s.shape         # get rows and columns
12
13    #Loop through image skipping edge pixel
14    for i in range(1,r-1):
15        # i for rows
16        for j in range(1,c-1):
17            # j for cols
18
19            a1 = np.abs(a[i,j]) # reflect to q 1&2
20            # classify angles
21            if (0<= a1 <BP1) or (BP4<= a1 <=np.pi):
22                #closest to x-axis check east to west
23                s[i,j] = s[i,j] if \
24                    (s[i,j-1] < s[i,j] and s[i,j+1] < s[i,j]) else 0
25            elif (BP1<= a1 <BP2):
26                # closest to 45° check SW and NE
27                s[i,j] = s[i,j] if \
28                    (s[i+1,j-1] < s[i,j] and s[i-1,j+1] < s[i,j]) else 0
29            elif (BP2<= a1 <BP3):
30                # closest to 90° North and South
31                s[i,j] = s[i,j] if \
32                    (s[i+1,j] < s[i,j] and s[i-1,j] < s[i,j]) else 0
33            else:
34                # closest to 135° NW and SE
35                s[i,j] = s[i,j] if \
36                    (s[i-1,j-1] < s[i,j] and s[i+1,j+1] < s[i,j]) else 0
37
38    return s
```

```
In [329]: 1 '''Process Images with Non Maximum Supression and Display'''
2 nms = []
3
4 # Process Images
5 N = len(mags) # Get number of images to process
6
7 for k in range(N):
8     print('Processing {:d} of {:d} images....'.format(k+1,N))
9     nms.append(nonMaxSupression(mags[k],angs[k]))
10
11 print('Processing Complete!')
12
13 # Display Non-Max Suppressed Images
14 showAll(nms)
15
```

Processing 1 of 6 images....
Processing 2 of 6 images....
Processing 3 of 6 images....
Processing 4 of 6 images....
Processing 5 of 6 images....
Processing 6 of 6 images....
Processing Complete!



Stage 4: Double Threshold

The continuous nature of leaves many blurred and weak edge boundaries. The double threshold sharpens by applying two thresholds:

- High Threshold-gradient magnitudes above this threshold are encoded as 'strong' edge pixels
- Low Threshold-gradient magnitudes below are suppressed.
- Between High and Low-gradient magnitudes in this range are encoded as 'weak' edge pixels

In [330]:

```
1  '''Double Threshold Function'''
2  def dblThreshold(A,H,L):
3      '''Suppress Values in image, A, below L and highlight
4          pixel values above H.  Mark those in between as Weak'''
5      STRONG = 255
6      WEAK = 128
7
8      # store shape of A
9      r,c = A.shape
10     B = np.zeros(r*c) # create vector with same num pixels
11
12     #flatten and cycle through image
13     for i,pixel in enumerate(A.reshape(1,-1)[0,:]):
14         if pixel > H:
15             B[i] = STRONG
16         if pixel < H and L < pixel:
17             B[i] = WEAK
18         # Array was init to zero so
19         # no need to check for suppression conditions
20     return B.reshape(r,c)
```

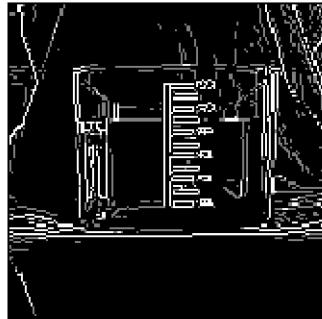
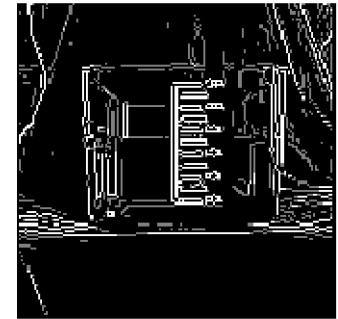
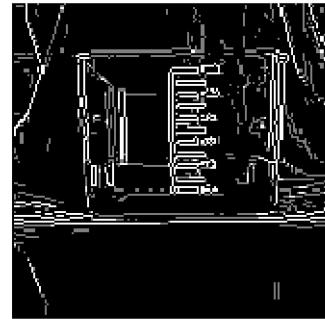
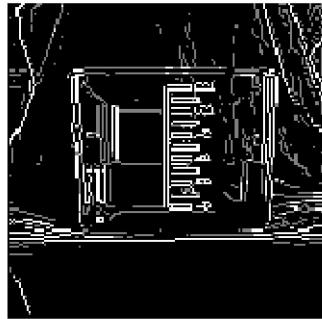
In [332]:

```

1  '''Process Images with Double Threshold and Display'''
2  dblTh = []
3
4  HIGH = 110
5  LOW = 45
6
7  # Process Images
8  N = len(nms)  # Get number of images to process
9
10 for k in range(N):
11     print('Processing {:d} of {:d} images....'.format(k+1,N))
12     dblTh.append(dblThreshold(nms[k],HIGH,LOW))
13
14 print('Processing Complete!')
15
16 # Display Non-Max Suppressed Images
17 showAll(dblTh)
18

```

Processing 1 of 6 images....
 Processing 2 of 6 images....
 Processing 3 of 6 images....
 Processing 4 of 6 images....
 Processing 5 of 6 images....
 Processing 6 of 6 images....
 Processing Complete!



Stage 5: Hysteresis Thresholding

Hysteresis thresholding seeks to classify the weak pixels based on context. If as we sweep the image, we encounter a strong pixel; we assume that everything between that and the next suppressed pixel is a strong edge. Similarly, if we encounter a suppressed pixel, we assume everything between it and the next strong pixel is also suppressed.

In [333]:

```
1  '''Define Hysteresis Thresholding Function'''
2  def hystTresh(A):
3      '''Classify pixels as suppressed or strong based on neighbors'''
4      TOL = 25                      # tolerance
5      STRONG = 255                  # Strong pixel
6      WEAK = 128                    # Weak pixel
7      SUPRESSED = 0                # Suppressed pixel
8      WIN = 3                       # Window Size of neighbors to check
9
10     r,c = A.shape               # Get Shape of A
11     B = A.copy()                # Copy A
12
13     for i in range(1,r-1):
14         # rows Loop
15
16         for j in range(1,c-1):
17             # cols Loop
18             # check if pixel is weak
19             if B[i,j] < WEAK + TOL and B[i,j] > WEAK - TOL:
20                 #is weak so check neighbors
21                 if np.max(B[i-1:i+WIN-1,j-1:j+WIN-1]) > STRONG - TOL:
22                     B[i,j] = STRONG
23                 else:
24                     B[i,j] = SUPRESSED
25
26     return B
```

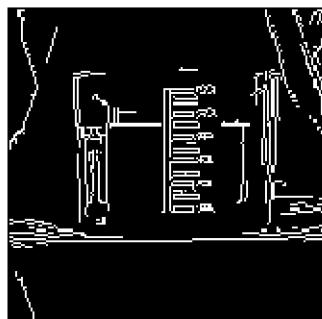
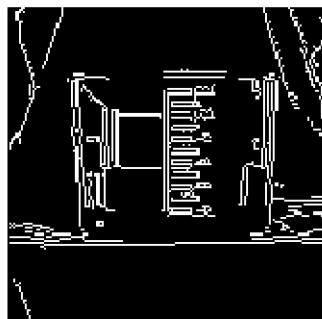
In [334]:

```

1  '''Process Images with Double Threshold and Display'''
2  hys = []
3
4  # Process Images
5  N = len(db1Th)  # Get number of images to process
6
7  for k in range(N):
8      print('Processing {:d} of {:d} images....'.format(k+1,N))
9      hys.append(hystTresh(db1Th[k]))
10
11 print('Processing Complete!')
12
13 # Display Non-Max Suppressed Images
14 showAll(hys)

```

Processing 1 of 6 images....
 Processing 2 of 6 images....
 Processing 3 of 6 images....
 Processing 4 of 6 images....
 Processing 5 of 6 images....
 Processing 6 of 6 images....
 Processing Complete!



Altered Algorithm For Finding Level

- Window the image to create a smaller area to process both for speed and to focus on the area where the level is easiest to identify
- Eliminate all edges that don't have a gradient angle pointing up (90°) (alter the non-maximum suppression function)

In [341]:

```

1  '''Water Level Finding Funcitons'''
2
3  # altered non-maximum supression
4  def nonHorzSupression(m,a):
5      '''Round angle in radians to either 0 or 90°,
6      and check neighbors in direction of gradient'''
7
8      BP1 = 0.3927          # 22.5°
9      BP2 = 1.1781          # 67.5°
10     BP3 = 1.9635          # 112.5°
11     BP4 = 2.7489          # 157.5°
12
13     s = m.copy()          # copy array
14     r,c = s.shape         # get rows and columns
15
16     #Loop through image skipping edge pixel
17     for i in range(1,r-1):
18         # i for rows
19         for j in range(1,c-1):
20             # j for cols
21
22             a1 = np.abs(a[i,j]) # reflect to q 1&2
23             # classify angles
24             if (BP2<= a1 <BP3):
25                 # closest to 90° North and South
26                 s[i,j] = s[i,j] if \
27                     (s[i+1,j] < s[i,j] and s[i-1,j] < s[i,j]) else 0
28             else:
29                 # not a horizontal line
30                 s[i,j] = 0
31
32     return s
33
34     # image windowing fucntion
35     def window(A,dim):
36         '''Takes in image, x,y coordinates for start of window and
37             the height (H) and width (W) of the window. dim = (x,y,H,W)
38             Returns pixels in window'''
39         x,y,H,W = dim # break out dimensions for clarity
40         return A[y:y+H,x:x+W]

```

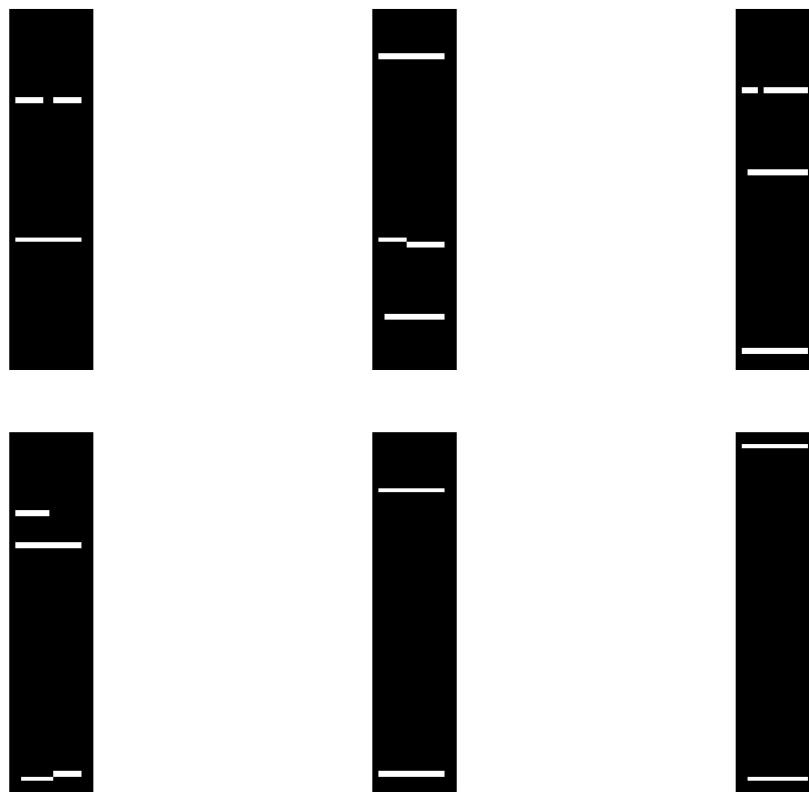
In [388]:

```

1 #Filter Constants
2 SIGMA = .5      # x-sigma, is copied to y-sigma if y-sigma is not specified
3 K_SIZE = (5,5)  # Kernel Window Size
4
5 # Window Constants
6 X,Y = 55,35
7 H,W = 65,15
8 d = (X,Y,H,W)
9
10 # Double Thresholding Constants
11 HIGH = 50
12 LOW = 10
13
14 winEdges = []
15
16 # Process Images
17 N = len(imgs)  # Get number of images to process
18
19 for k in range(N):
20     print('Processing {:d} of {:d} images....'.format(k+1,N))
21
22     # Window Images
23     temp = window(imgs[k],d)
24
25     # Stage 1 Filter
26     temp = cv.GaussianBlur(temp,K_SIZE,SIGMA,cv.BORDER_REPLICATE)
27
28     #Stage 2 Gradient
29     tempM,tempA = sobel(temp)
30
31     # Stage 3 Non Max Supress (modified)
32     temp = nonHorzSupression(tempM,tempA)
33
34     # Stage 4 Double Thresh
35     temp = dblThreshold(temp,HIGH,LOW)
36
37     # Stage 5 Hysteresis
38     temp = hystTresh(temp)
39
40     # Save Results
41     winEdges.append(temp)
42
43 print('Processing Complete!')
44
45 # Display Non-Max Suppressed Images
46 showAll(winEdges)
47
48

```

Processing 1 of 6 images....
 Processing 2 of 6 images....
 Processing 3 of 6 images....
 Processing 4 of 6 images....
 Processing 5 of 6 images....
 Processing 6 of 6 images....
 Processing Complete!



Finding the Right Line

Adhoc Algorithm

- Sum up processed windowed segments along the x (rows) axis
- Since we expect the water to be somewhere in the middle, we build a normal gaussian distribution that spans the height of the windowed segment and then convolve it with the flattened sum matrix
- Argmax of the resulting array should give us the index of the most likely spot for the waterline

In [389]:

```
1 def normalDist(L):
2     '''return an array of weights corresponding to the
3         length of L centered at L/2'''
4     mu = L/2.
5     sig = mu/3
6     z = (np.linspace(0,L,L) - mu)/sig
7     return 1/sig/np.sqrt(2*np.pi)*np.exp(-0.5*z*z)
8
```

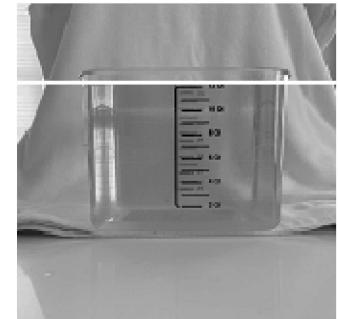
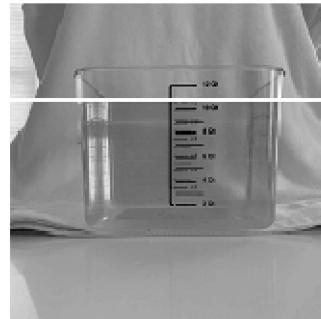
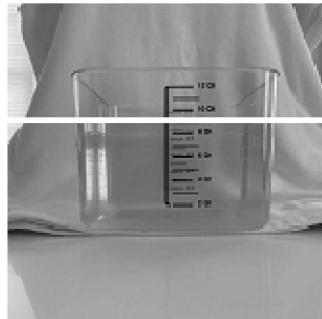
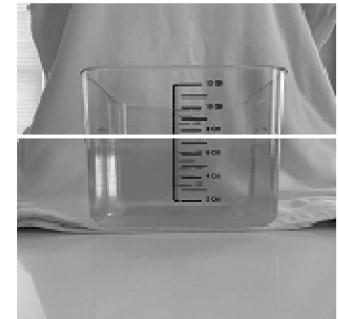
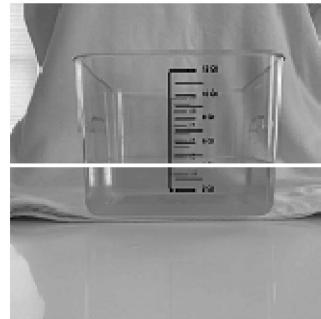
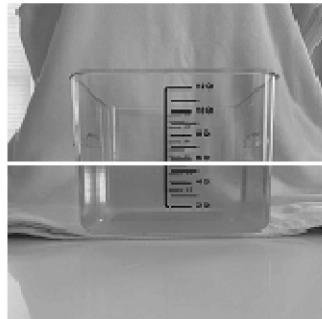
In [390]:

```

1  '''Draw Line at Most likely Water level'''
2  #Constants
3  LW = 3 # line width to mark up original image
4
5  finalImg = []
6
7  probCurve = normalDist(H)
8
9  # Process Images
10 N = len(imgs) # Get number of images to process
11
12 for k in range(N):
13     print('Processing {:d} of {:d} images....'.format(k+1,N))
14     Water = np.argmax(np.multiply(np.sum(winEdges[k],axis=1),probCurve))
15     A = imgs[k].copy()
16     # Draw Line across image
17     A[Y+Water-LW//2:Y+Water+LW//2,:] = 255
18     finalImg.append(A)
19
20 print('Processing Complete!')
21
22 # Display Non-Max Suppressed Images
23 showAll(finalImg)

```

Processing 1 of 6 images....
 Processing 2 of 6 images....
 Processing 3 of 6 images....
 Processing 4 of 6 images....
 Processing 5 of 6 images....
 Processing 6 of 6 images....
 Processing Complete!



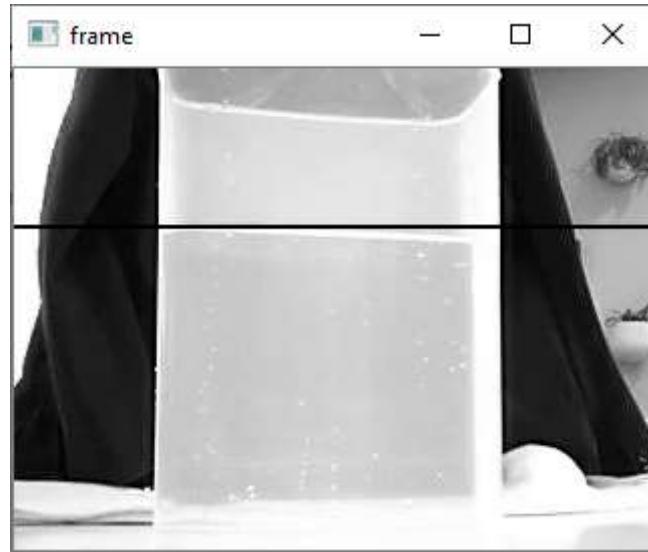
In [395]:

```

1  '''Create a waterline detection function'''
2  def detectLevel(A,win_dim,thresh_HL,filtSK):
3      # break out packed parameters for clarity
4      X,Y,H,W = win_dim
5      sigma,ksize = filtSK
6      high,low = thresh_HL
7
8      # Window Images
9      temp = window(A,win_dim)
10     # Stage 1 Filter
11     temp = cv.GaussianBlur(temp,ksize,sigma,cv.BORDER_REPLICATE)
12     #Stage 2 Gradient
13     tempM,tempA = sobel(temp)
14     # Stage 3 Non Max Supress (modified)
15     temp = nonHorzSupression(tempM,tempA)
16     # Stage 4 Double Thresh
17     temp = dblThreshold(temp,high,low)
18     # Stage 5 Hysteresis
19     temp = hystTresh(temp)
20     # Draw final line
21     Water = np.argmax(np.multiply(np.sum(temp, axis=1),normalDist(H)))
22     # Draw line across image
23     A[Y+Water-LW//2:Y+Water+LW//2,:] = 0
24
25     return A

```

Play Real Time Video of Water Draining from Different Vessel and Let Algorithm Track



In [398]:

```
1 #Filter Constants
2 SIGMA = .5      # x-sigma, is copied to y-sigma if y-sigma is not specified
3 K_SIZE = (5,5)  # Kernel Window Size
4
5 # Window Constants
6 X,Y = 130,0
7 H,W = 220,15
8 d = (X,Y,H,W)
9
10 # Double Thresholding Constants
11 HIGH = 50
12 LOW = 10
13
14 # Create a VideoCapture object and read from input file
15 # If the input is the camera, pass 0 instead of the video file name
16 # code modified from Help page
17 # https://docs.opencv.org/master/dd/d43/tutorial_py_video_display.html
18
19 cap = cv.VideoCapture(vidFile)
20 if not cap.isOpened():
21     print('Failed to Open',str(vidFile))
22 while cap.isOpened():
23     ret, frame = cap.read()
24     # if frame is read correctly ret is True
25     if not ret:
26         print("Can't receive frame (stream end?). Exiting ...")
27         break
28     gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
29     gray = detectLevel(gray,d,(HIGH,LOW),(SIGMA,K_SIZE))
30     cv.imshow('frame', gray)
31     if cv.waitKey(1) == ord('q'):
32         break
33 cap.release()
34 cv.destroyAllWindows()
```

Can't receive frame (stream end?). Exiting ...

In []:

1