Assignment 3

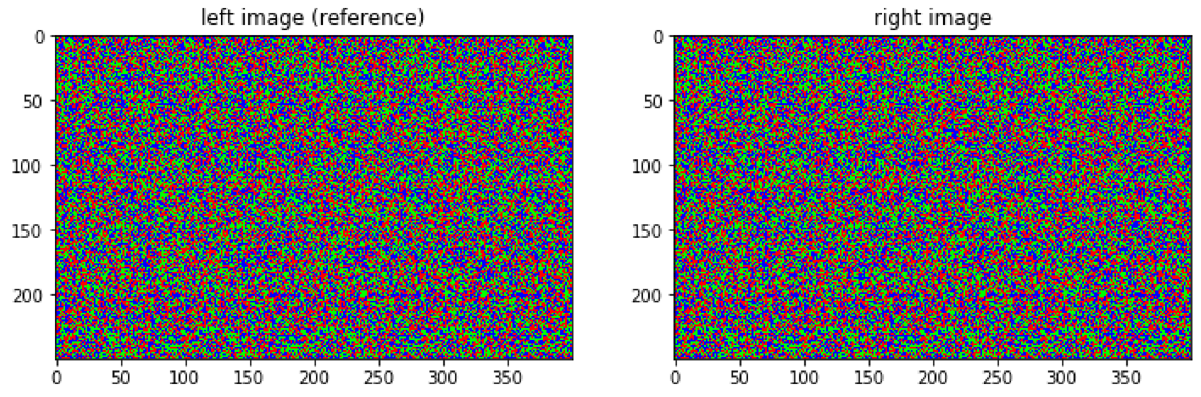
Part I: Windows-based stereo

$\longleftarrow 70\ pts$

In [1]:
```
%matplotlib inline
```

In [10]:
```
import numpy as np
from numpy import linalg as LA
import matplotlib.pyplot as plt
import matplotlib.image as image
from matplotlib.colors import LogNorm
from skimage import img_as_ubyte
from skimage.color import rgb2grey
```

In [3]:
```
# you should use this random dot stereo pair for code developing/testing
in Probelms 1-5
im_left = image.imread("images/stereo_pairs/rds_left.gif")
im_right = image.imread("images/stereo_pairs/rds_right.gif")

fig = plt.figure(figsize = (12, 5))
plt.subplot(121)
plt.title("left image (reference)")
plt.imshow(im_left)
plt.subplot(122)
plt.title("right image")
plt.imshow(im_right)

# the range of disparities for this random dot stereo pair
d_min = 0
d_max = 2
```



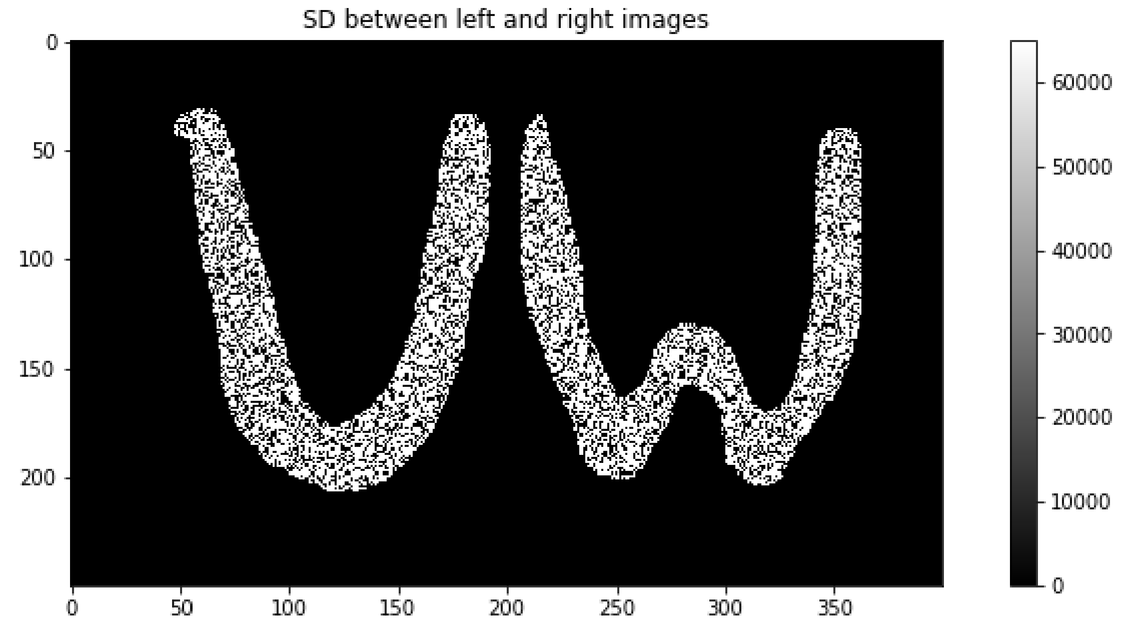
Problem 1: compute and visualize (as an image) an array of "squared differences" between RGB pixel values in the left and the right images. Treat each RGB pixel value as 3-vector and interprete "squared difference" as squared L2 norm of the difference between the corresponding vectors.

HINT (important here and later): convert R, G, B values to floats to avoid severe "overflow" bugs while adding small-range types (one-byte for *char*) or subtracting unsigned types. Note that *imshow* function can display (as an image) any 2D array of floats.

In [14]:
```
SD = np.zeros((np.shape(im_left)[0],np.shape(im_left)[1]))
diffR = (im_left[:,:,0]-im_right[:,:,0]).flatten()
diffG = (im_left[:,:,1]-im_right[:,:,1]).flatten()
diffB = (im_left[:,:,2]-im_right[:,:,2]).flatten()

norm = LA.norm(np.vstack([diffR, diffG, diffB]), axis=0)
squared_diff = norm**2
SD = squared_diff.reshape((np.shape(im_left)[0],np.shape(im_left)[1]))

fig = plt.figure(figsize = (12, 5))
plt.title("SD between left and right images")
plt.imshow(SD, cmap = "gray")
plt.colorbar()
```

```
(100000,)
```

Out[14]:
```
<matplotlib.colorbar.Colorbar at 0x7f8468256b50>
```



Problem 2: write function for computing squared differences between RGB pixel values in the reference (left) image and the "shifted" right image for ALL shifts/disparities Δ in the range $\Delta \in [d_{min}, d_{max}]$. You should think about the correct direction of the shift. The output should be array SD such that $SD[i]$ is an image of Squared Differences for shift $\Delta = d_{min} + i$ for any $i \in [0, d_{max} - d_{min}]$.

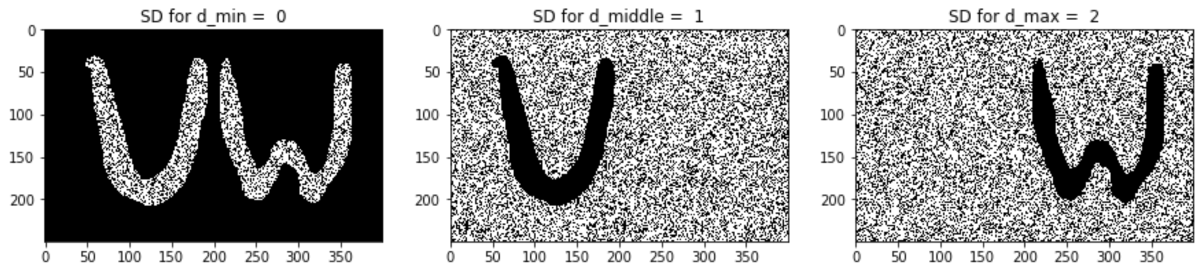
In [19]:
```
def SD_array(imageL, imageR, d_minimum, d_maximum):
    # initialization of the array of "squared differences" for different
shifts
    SD = np.zeros((1+d_maximum-d_minimum,np.shape(imageL)[0],np.shape(im
ageL)[1]))

    for i in range(1+d_maximum-d_minimum):
        delta = d_minimum + i
        SD2 = np.zeros((np.shape(im_left)[0],np.shape(im_left)[1]))

        diffR = (imageL[:,:,0] - np.roll(imageR[:,:,0], delta, axis=1)).
flatten()
        diffG = (imageL[:,:,1] - np.roll(imageR[:,:,1], delta, axis=1)).
flatten()
        diffB = (imageL[:,:,2] - np.roll(imageR[:,:,2], delta, axis=1)).
flatten()

        norm = LA.norm(np.vstack([diffR, diffG, diffB]), axis=0)
        squared_diff = norm**2
        SD2 = squared_diff.reshape((np.shape(im_left)[0],np.shape(im_lef
t)[1]))
        SD[i] = SD2

    return SD
```

Use SD_array function to compute SD images for the random dot stereo pair. Visualize such squared difference images for $\Delta = d_{min}$, $\Delta = d_{mid} \approx \frac{d_{min}+d_{max}}{2}$, and $\Delta = d_{max}$. Note that the first image should be identical to the one in Problem 1. (fully implemented)

In [20]:
```
SD = SD_array(im_left, im_right, d_min, d_max)
print(np.shape(im_left))
print(np.shape(im_right))
print(np.shape(SD))

fig = plt.figure(figsize = (15, 4))
plt.subplot(131)
plt.title('SD for d_min = {:>2d}'.format(d_min))
plt.imshow(SD[0], cmap = "gray")
plt.subplot(132)
d_middle = round((d_min+d_max)/2)
plt.title('SD for d_middle = {:>2d}'.format(d_middle))
plt.imshow(SD[d_middle-d_min], cmap = "gray")
plt.subplot(133)
plt.title('SD for d_max = {:>2d}'.format(d_max))
plt.imshow(SD[d_max-d_min], cmap = "gray")
#plt.colorbar(cax=plt.axes([0.91, 0.25, 0.01, 0.5]))
```

```
(250, 400, 4)
(250, 400, 4)
(3, 250, 400)
```

Out[20]:
```
<matplotlib.image.AxesImage at 0x7f84490957d0>
```



Problem 3: write function to compute an "integral image" for any given "scalar" image

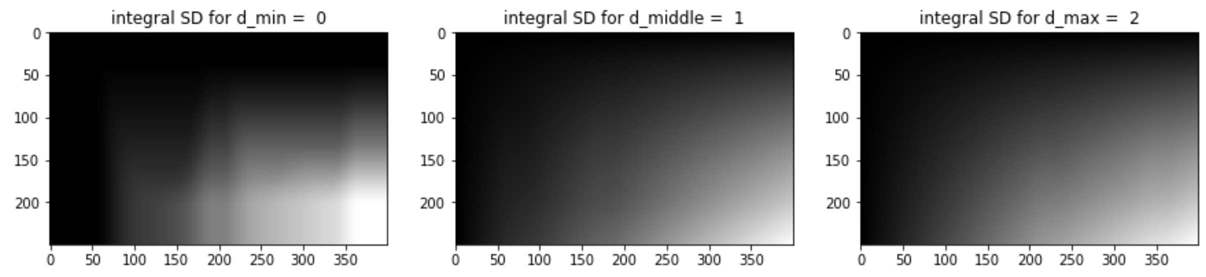
In [23]:
```
# Function integral_image can be applied to any scalar 2D array/image.
# This function should return a double/float64 (precision) array/image o
f the same size.
# NOTE: it is safer to explicitly specify double/float64 precision for i
ntegral images since
# later we will be adding/subtracting ("differenting") their values in n
earby pixels .

def integral_image(img):
    int_img = np.zeros((img.shape))
    for i in range(img.shape[0]):
        s = 0 # sum
        for j in range(img.shape[1]):
            if i == 0 and j == 0:
                int_img[i][j] = float(img[i][j])
                s = float(img[i][j])
            elif i == 0:
                int_img[i][j] = s + float(img[i][j])
                s = s + float(img[i][j])
            elif j == 0:
                s = float(img[i][j])
                int_img[i][j] = int_img[i-1][j] + s
            else:
                int_img[i][j] = int_img[i-1][j] + float(img[i][j]) + s
                s = s + float(img[i][j])
    return int_img
```

apply intergal_image function to the "squared differences" (SD) for each disparity (fully implemented)

In [24]:
```
integral_SD = np.zeros(np.shape(SD))
print(np.shape(integral_SD),np.shape(SD))

for Delta in range(1+d_max-d_min):
    integral_SD[Delta] = integral_image(SD[Delta])

fig = plt.figure(figsize = (15, 4))
plt.subplot(131)
plt.title('integral SD for d_min = {:>2d}'.format(d_min))
plt.imshow(integral_SD[0], cmap = "gray")
plt.subplot(132)
d_middle = round((d_min+d_max)/2)
plt.title('integral SD for d_middle = {:>2d}'.format(d_middle))
plt.imshow(integral_SD[d_middle-d_min], cmap = "gray")
plt.subplot(133)
plt.title('integral SD for d_max = {:>2d}'.format(d_max))
plt.imshow(integral_SD[d_max-d_min], cmap = "gray")
#plt.colorbar(cax=plt.axes([0.91, 0.2, 0.01, 0.6]))
```

```
(3, 250, 400) (3, 250, 400)
```

Out[24]:
```
<matplotlib.image.AxesImage at 0x7f8448ca9850>
```



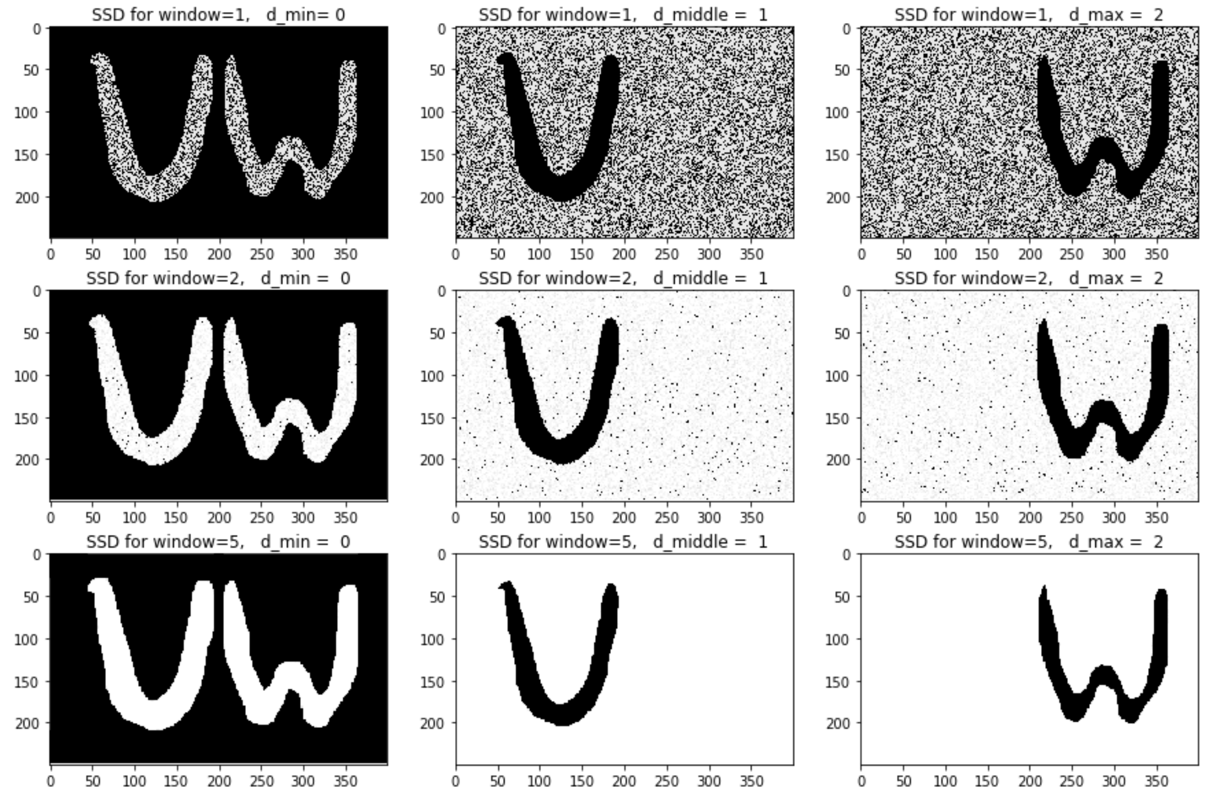
Problem 4: write function that sums the elements of the input image within fixed-size windows around image pixels. Note that this function should work for any (odd or even) values of parameter $\text{window\_width} \in \{1, 2, 3, 4, 5, \ldots\}$ according to the windows illustration below:

In [31]:
```
# function windSum can be applied to any scalar 2D array/image. It shoul
d return an array/image where the value of
# each element (pixel p) is the "sum" of the values in the input array/i
mage within a window around element p.
# The return image should be of the same size/type and have its margins
 (around half-window width) filled with INFTY.
# NOTE: you should use function integral_image implemented earlier.
# HINT: you should use standard np.roll function to avoid double or trip
le for loops.
INFTY = np.inf
import math

def windSum(img, window_width):
    pos = math.floor(window_width/2)
    if pos:
        int_img = integral_image(img)
        bl = np.roll(int_img, window_width, axis=1)
        bl[:,window_width - 1:window_width] = 0
        ur = np.roll(int_img, window_width,axis=0)
        ur[window_width - 1:window_width,:] = 0
        tl = np.roll(np.roll(int_img, window_width, axis=0), window_widt
h, axis=1)
        tl[:,window_width-1:window_width] = 0
        tl[window_width-1:window_width,:] = 0

        win = int_img - bl - ur + tl
        win[:pos,:] = INFTY
        win[:,:pos] = INFTY
        win = np.roll(np.roll(win, -pos, axis=1), -pos, axis=0)

        return win
    return img
```

Compute SSD images for windows of different widths and for different disparities by applying windSum **function to the "squared differences" SD. Note that the results for windows of width 1 (the first row of the figure below) should look identical (except for the narrow "margin") to the results in Problem 2. (fully implemented)**

In [32]:
```
SSD1 = np.zeros(np.shape(SD))
SSD2 = np.zeros(np.shape(SD))
SSD5 = np.zeros(np.shape(SD))

for Delta in range(1+d_max-d_min):
    SSD1[Delta] = windSum(SD[Delta],1)
    SSD2[Delta] = windSum(SD[Delta],2)
    SSD5[Delta] = windSum(SD[Delta],5)

d_middle = round((d_min+d_max)/2)

fig = plt.figure(figsize = (15, 10))
plt.subplot(331)
plt.title('SSD for window=1,   d_min={:>2d}'.format(d_min))
plt.imshow(1+SSD1[0], cmap = "gray", norm=LogNorm(vmin=1, vmax=200000))
plt.subplot(332)
plt.title('SSD for window=1,   d_middle = {:>2d}'.format(d_middle))
plt.imshow(1+SSD1[d_middle-d_min], cmap = "gray", norm=LogNorm(vmin=1, v
max=200000))
plt.subplot(333)
plt.title('SSD for window=1,   d_max = {:>2d}'.format(d_max))
plt.imshow(1+SSD1[d_max-d_min], cmap = "gray", norm=LogNorm(vmin=1, vmax
=200000))
plt.subplot(334)
plt.title('SSD for window=2,   d_min = {:>2d}'.format(d_min))
plt.imshow(1+SSD2[0], cmap = "gray", norm=LogNorm(vmin=1, vmax=200000))
plt.subplot(335)
plt.title('SSD for window=2,   d_middle = {:>2d}'.format(d_middle))
plt.imshow(1+SSD2[d_middle-d_min], cmap = "gray", norm=LogNorm(vmin=1, v
max=200000))
plt.subplot(336)
plt.title('SSD for window=2,   d_max = {:>2d}'.format(d_max))
plt.imshow(1+SSD2[d_max-d_min], cmap = "gray", norm=LogNorm(vmin=1, vmax
=200000))
plt.subplot(337)
plt.title('SSD for window=5,   d_min = {:>2d}'.format(d_min))
plt.imshow(1+SSD5[0], cmap = "gray", norm=LogNorm(vmin=1, vmax=200000))
plt.subplot(338)
plt.title('SSD for window=5,   d_middle = {:>2d}'.format(d_middle))
plt.imshow(1+SSD5[d_middle-d_min], cmap = "gray", norm=LogNorm(vmin=1, v
max=200000))
plt.subplot(339)
plt.title('SSD for window=5,   d_max = {:>2d}'.format(d_max))
plt.imshow(1+SSD5[d_max-d_min], cmap = "gray", norm=LogNorm(vmin=1, vmax
=200000))
```

```
Out[32]: <matplotlib.image.AxesImage at 0x7f8408de5f50>
```



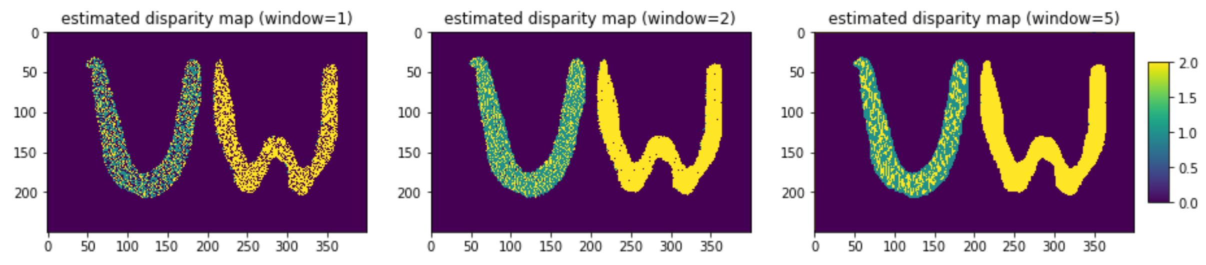
Problem 5: write code for function computing "disparity map" from SSD arrays (as above) for each disparity in the range specified by integers d_{min}, d_{max}. It should return a disparity map (image). At each pixel, disparity map image should have disparity value corresponding to the minimum SSD at this pixel. For pixels at the margins, the disparity map should be set to zero. HINT: margin pixels are characterized by $INFTY$ values of SSD.

```
In [35]: # You should use functions np.where (pointwise "if, then, else" operatio
n) and np.minimum (pointwise "minimum" operation)
# These functions will help to avoid double loops for traversing the pix
els.
# WARNING: there will be a deducton for double-loops traversing pixels,
 but single loop to traverse disparities is OK.

def SSDtoDmap(SSD_array, d_minimum, d_maximum):

    dMap = np.full(np.shape(SD[0]),d_minimum)

    for i in range(1,1+d_maximum-d_minimum):
        dMap = np.where(SSD_array[i] < SSD_array[0], i, dMap)
    return dMap
```

Compute and show disparity map (fully implemented)

In [36]:
```
dMap1 = SSDtoDmap(SSD1,d_min,d_max)
dMap2 = SSDtoDmap(SSD2,d_min,d_max)
dMap5 = SSDtoDmap(SSD5,d_min,d_max)

fig = plt.figure(figsize = (15, 3))
plt.subplot(131)
plt.title("estimated disparity map (window=1)")
plt.imshow(dMap1, vmin = 0, vmax = d_max)
plt.subplot(132)
plt.title("estimated disparity map (window=2)")
plt.imshow(dMap2, vmin = 0, vmax = d_max)
plt.subplot(133)
plt.title("estimated disparity map (window=5)")
plt.imshow(dMap5, vmin = 0, vmax = d_max)
plt.colorbar(cax=plt.axes([0.91, 0.25, 0.015, 0.5]))
```

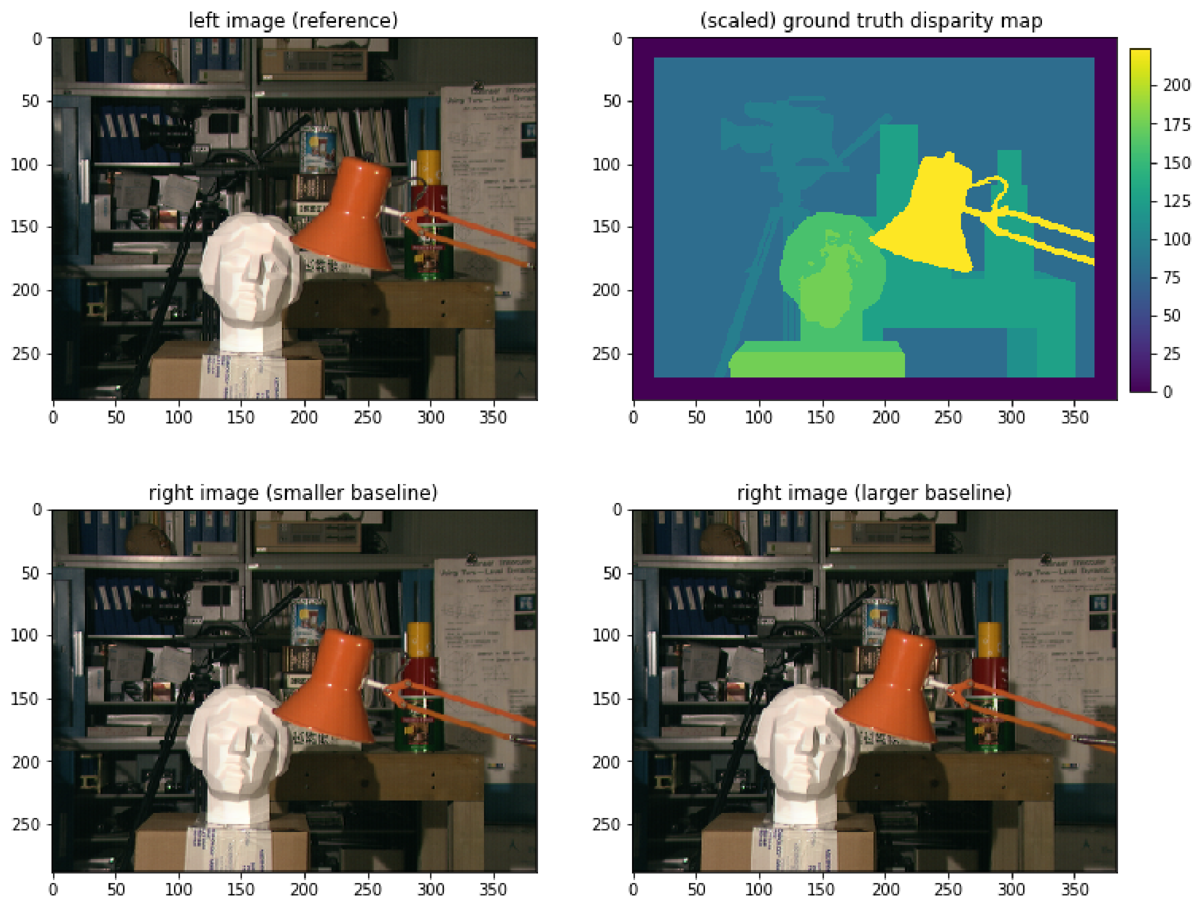
Out[36]:
```
<matplotlib.colorbar.Colorbar at 0x7f8468208410>
```



Problem 6: test your code on a real stereo pair with ground truth (Tsukuba)

In [37]:
```
# images/tsukuba subdirectory contains (a subset of) "Tsukuba" stereo im
ages that are probably
# the oldest stereo data with dense ground-truth produced at the Univers
ity of Tsukuba in 2001.
# The full Tsukuba dataset and many other stereo images with ground-trut
h disparity maps can be
# downloaded from well-known Middlebury repository  http://vision.middle
bury.edu/stereo/
im_left = image.imread("images/stereo_pairs/tsukuba/scene1.row3.col3.pp
m")
im_gt = image.imread("images/stereo_pairs/tsukuba/truedisp.row3.col3.pg
m")
im_right = image.imread("images/stereo_pairs/tsukuba/scene1.row3.col4.pp
m")
im_right2 = image.imread("images/stereo_pairs/tsukuba/scene1.row3.col5.p
pm")

fig = plt.figure(figsize = (12, 10))
plt.subplot(221)
plt.title("left image (reference)")
plt.imshow(im_left)
plt.subplot(222)
plt.title("(scaled) ground truth disparity map ")
plt.imshow(im_gt)
plt.colorbar(cax=plt.axes([0.91, 0.557, 0.015, 0.3]))
plt.subplot(223)
plt.title("right image (smaller baseline)")
plt.imshow(im_right)
plt.subplot(224)
plt.title("right image (larger baseline)")
plt.imshow(im_right2)
```

```
Out[37]: <matplotlib.image.AxesImage at 0x7f8478bd0dd0>
```


left image (reference)
(scaled) ground truth disparity map
0
25
50
75
100
125
150
175
200
right image (smaller baseline)
right image (larger baseline)

Note that the integer-valued ground truth image above represents scaled disparity values for the pixels in the reference (left) mage. The scale w.r.t. the smaller baseline right image (im_right) is 16 and for the larger baseline image (im_right2) is 8. Below, you should use the smaller-baseline right image (im_right).

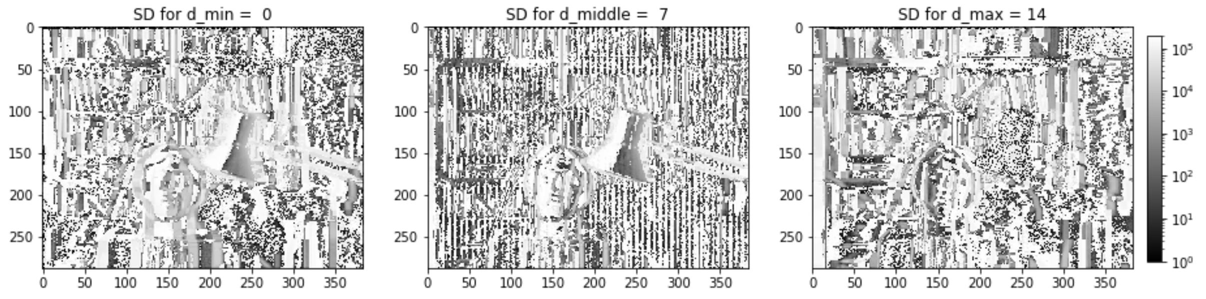
Problem 6a: Using ground truth disparity map, estimate the range of disparity values between pixels in the left image (im_left) and the right image (im_right) .

```
In [43]: # Solution: use standard functions to find min and max values in the gro
         und truth disparity map.
         # You should ignore 0-valued margin!

         d_min = 0 # change me
         d_max = 14 # change me
```

Compute squared differences using $\mathrm{SD\_array}$ function and visualize the results using logarithmic scale. Note that linear scale would make it hard to see smaller squared differences since there are many very large ones. (fully implemented)

In [44]:
```
SD = SD_array(im_left, im_right, d_min, d_max)

fig = plt.figure(figsize = (15, 4))
plt.subplot(131)
plt.title('SD for d_min = {:>2d}'.format(d_min))
plt.imshow(im_left)
plt.imshow(1+SD[0], cmap = "gray", norm=LogNorm(vmin=1, vmax=200000))
plt.subplot(132)
d_middle = round((d_min+d_max)/2)
plt.title('SD for d_middle = {:>2d}'.format(d_middle))
plt.imshow(1+SD[d_middle-d_min], cmap = "gray", norm=LogNorm(vmin=1, vma
x=200000))
plt.subplot(133)
plt.title('SD for d_max = {:>2d}'.format(d_max))
plt.imshow(1+SD[d_max-d_min], cmap = "gray", norm=LogNorm(vmin=1, vmax=2
00000))
plt.colorbar(cax=plt.axes([0.91, 0.2, 0.01, 0.6]))
```

Out[44]:
```
<matplotlib.colorbar.Colorbar at 0x7f849a809310>
```



Problem 6b: Explain the differences you observe above:

answer:

For d_min = 0, we see that things in the background are more obvious because things far away have a smaller disparity value. For d_middle = 7, we see that things in the middle are more obvious such as the sculpture. Similarly, for d_max = 14 things in the front such as the lamp are darkest because of the large disparity.

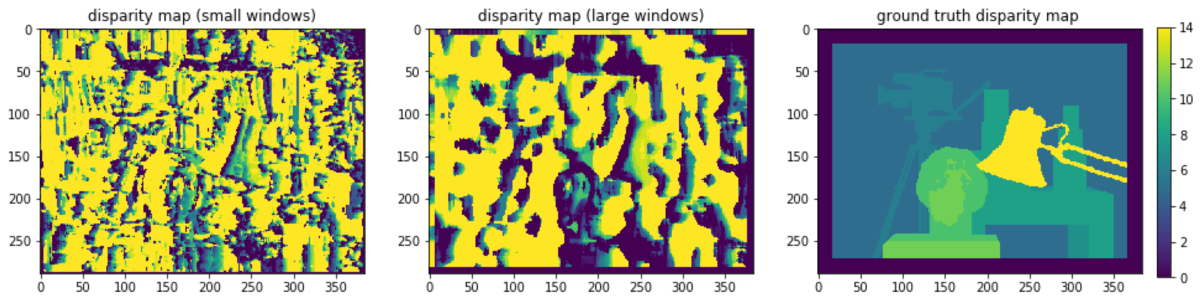
Problem 6c: Write function Dmap_Windows that returns disparith map from a given stereo pair (left and right image), specified disparity range, and window size. Your implementation should combine functions implemented and debugged earlier (SD_array, windSum, and SSDtoDmap).

```
In [47]: def Dmap_Windows(imageL, imageR, d_minimum, d_maximum, window_width):
             SD = SD_array(imageL, imageR, d_minimum, d_maximum)
             SSD = np.zeros(np.shape(SD))
             for i in range(1+d_maximum-d_minimum):
                 SSD[i] = windSum(SD[i], window_width)
             dMap = SSDtoDmap(SSD, d_minimum, d_maximum)
             return dMap
```

Compute and show disparity maps for Tsukuba using small and large windows. (fully implemented)

```
In [48]: dispMap_small = Dmap_Windows(im_left, im_right, d_min, d_max, 4)
         dispMap_large = Dmap_Windows(im_left, im_right, d_min, d_max, 15)

         fig = plt.figure(figsize = (16, 7))
         plt.subplot(131)
         plt.title("disparity map (small windows)")
         plt.imshow(dispMap_small, vmin = 0, vmax = d_max)
         plt.subplot(132)
         plt.title("disparity map (large windows)")
         plt.imshow(dispMap_large, vmin = 0, vmax = d_max)
         plt.subplot(133)
         plt.title("ground truth disparity map ")
         plt.imshow(im_gt/16, vmin = 0, vmax = d_max)
         plt.colorbar(cax=plt.axes([0.91, 0.3, 0.01, 0.4]))
```

```
Out[48]: <matplotlib.colorbar.Colorbar at 0x7f849a3c7410>
```



Part II: Scan-line stereo $\longleftarrow 30\ pts$

Problem 7(a): Program $Viterbi$ approach discussed in class and apply it to Tsukuba example. For the photo-consistency term of the loss function (objective function) you can use previously implemented SD_array

$$D_p(d) = |I_p - I_{p+d}|^2 \qquad \longleftarrow \qquad \text{SD\_array}[d][p]$$

that for every pixel p defines the cost for every possible disparity value $d \in \{d_{min}, \ldots, d_{max}\}$. The regularization term should be as discussed in class

$$V_{pq}(d_p, d_q) = w|d_p - d_q|$$

where you can select some good value for parameter w empirically (start from $w \approx 0$). Discuss the differences with the window-based stereo results above.

NOTE: You should implement $Viterbi$ optimization yourself - it was fully covered in class. Organize your code (e.g. add cells, introduce functions, write comments, ect) as part of your mark will depend on clarity. The main iteration of the forward pass in Viterbi (m^2-complexity operation for each pair of neighboring pixels on a scan-line) can be implemented as a separate function. You can avoid double for-loops using functions like *np.where* , *np.minimum* , *np.square* , *np.ogrid* or others similar general "vectorized" functions in numpy that alow to avoid multi-loops over matrix (image) elements (pixels).

```
In [ ]:
```

Problem 7(b): Test the case where the photoconsistency term $D_p(d)$ is computed by averaging SD in some small window of size h. That is, for each disparity d you should replace 2D array $\text{SD\_array}(d)$ in Problem 7(a) by

$$\text{SD\_array}(d) \qquad \longleftarrow \qquad windSum(SD(d), h).$$

Compare the results for different window sizes $h \in \{1, 3, 5\}$. Feel free to fine tune regularization parameter w for each case trying to obtain the better results.

NOTE 1: $h = 1$ should be equivalent to Problem 7(a) above.

NOTE 2: this version combines window-based stereo with regularization along scan-lines. The case when $w = 0$ should give the same results as in Problem 6(c).

```
In [ ]:
```

Problem 8 [optional, small bonus]: test the performance for quadratic and (robust) truncated-quadratic regularization terms

$$V_{pq}(d_p, d_q) = w|d_p - d_q|^2 \quad \text{and} \quad V_{pq}(d_p, d_q) = min\{w|d_p - d_q|^2, T\}.$$

Discuss the results.

```
In [ ]:
```

Problem 9 [optional, small bonus]: test the case where regularization weight $w = w_{pq}$ depends on a specific pair of neighboring points. That is,

$$V_{pq}(d_p, d_q) = w_{pq}|d_p - d_q|.$$

It is common to use local intensity contrast in the reference image to set such weights, for example

$$w_{pq} = w \exp \frac{-\|I_p - I_q\|^2}{2\sigma^2}$$

which weighs the overall regularization constant w by a Gaussian kernel (in RGB or grey-scale space). The latter makes it cheaper to draw large disparity jumps at "contrast edges" or "contrast boundaries" that are likely to happen at object boundaries. Note that bandwith parameter σ is important - it controls sensitivity to contrast. Discuss the results.

```
In [ ]:
```