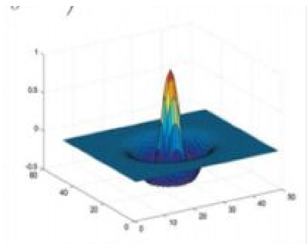
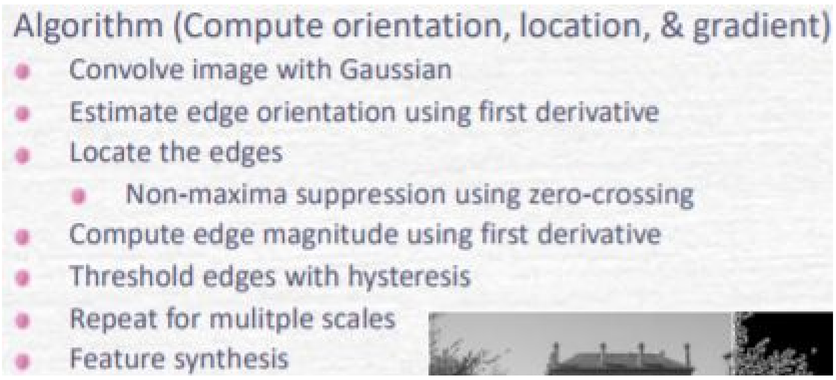
Edges

- **Edge Detection**
    - A segmentation technique
        - **Analysis of discontinuities in an image**
    - Alternative segmentation to region based processing
    - **What is an edge?**
        - Abrupt change in brightness
        - Edges have **magnitude (gradient) and direction (orientation)**
        - **Profiles**
            - Step / Real / Noisy
    - **1st derivative edge detection**
        - Calculus: derivative is the rate of change in two directions
        - **Vector variable:**
            - **Gradient Magnitude** $\nabla f(i,j) = \sqrt{\left(\frac{\delta f(i,j)}{\delta i}\right)^2 + \left(\frac{\delta f(i,j)}{\delta j}\right)^2}$
                - **Or** $\nabla f(i,j) = \left|\frac{\delta f(i,j)}{\delta i}\right| + \left|\frac{\delta f(i,j)}{\delta j}\right|$
            - **Orientation (0° is East)** $\phi(i,j) = \arctan\left(\frac{\delta f(i,j)}{\delta j}, \frac{\delta f(i,j)}{\delta i}\right)$
        - Derivatives work on continuous functions
            - Map every point in the input image to the output
        - **Discrete domain**
            - Differences
            - Orthogonal
        - **Roberts Edge Detector**
            - **Uses two partial derivatives to compare diagonal differences between pixels**
                - $\delta_1(i,j) = f(i,j) - f(i+1,j+1)$
                  $\delta_2(i,j) = f(i,j+1) - f(i+1,j)$
            - **Uses two convolution masks, which are moved across the entire image to compute the function**
                - $h_1(i,j) = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad h_2(i,j) = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$
            - **Uses RMS to compute the gradient and inverse tan to get the angle**
            - Works well on binary
            - Partial derivs are sensitive to noise
            - Non-binary images change gradually, roberts only considers adjacency
            - Edges are a half pixel out due to being placed at the halfway point between two partial derivs
        - **Edge detectors need to…**

        - Cross at a single middle point, ideally at the centre of a pixel
        - Evaluate points that aren't too close together, and handle noise
    - **Compass Edge Detectors**
        - **Partial derivatives defined for a number of orientations** (typically 8)
            - **Only really need two orthogonal ones** (regularly taken as h1 and h3 below)
            - Gives positive and negative values
        - **Prewitt**
            - $$h_1(i,j) = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \quad h_2(i,j) = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix} \quad h_3(i,j) = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad h_4(i,j) = \begin{bmatrix} -1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$ $$h_5(i,j) = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad h_6(i,j) = \begin{bmatrix} 0 & -1 & -1 \\ 1 & 0 & -1 \\ 1 & 1 & 0 \end{bmatrix} \quad h_7(i,j) = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \quad h_8(i,j) = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$
        - **Sobel**
            - $$h_1(i,j) = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad h_3(i,j) = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$
        - **These masks essentially implement smoothing, account for slightly more distant pixels, and have a real centre**
    - **Thresholding**
        - **Simple thresholding** - will either get too many or too few points
        - **Non-maxima suppression**
            - Use gradient and orientation information to identify central edge points
            - Orientations are quantised into 8 values
            - **Uses edge information to compare an edge pixel to the pixel ahead and behind the current one**
                - Suppresses the non-maximum pixels
            - Quantise edge orientations
              For all points (i,j)
                - Look at the 2 points orthogonal to edge
                - if gradient(i,j) < gradient(either of these 2 points)
                    - output(i,j) = 0
                    - else output(i,j) = gradient(i,j)
- **2nd derivative edge detection**
    - **Laplace operator**
        - $$h(i,j) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad h(i,j) = \begin{bmatrix} 2 & -1 & 2 \\ -1 & -4 & -1 \\ 2 & -1 & 2 \end{bmatrix}$$
        - **One or the other**
        - **Note high weighting of centre pixel -** very susceptible to noise
            - As a result - typically preceded by smoothing
    - **2nd derivative edge detectors find the edges without their orientation**

    - Location and gradient detection
    - **Gradient magnitude = slope of zero crossing**
        - This is very expensive to compute, so we generally opt for the magnitude from first derivative
    - **Why use 2nd derivative over first?**
        - **Edge location accuracy is substantially higher**
- **Marr-Hildreth edge detection**
    - 2nd derivative zero crossings
    - Requires image smoothing
        - Smoothing filter must be: smooth and band limited, spatially localised
            - Spatial localisation prevents the filter from moving the edges
    - Optimal solution is the **Gaussian filter**
        - $$G(i,j) = e^{-\frac{i^2+j^2}{2\sigma^2}}$$
        - σ^2 **= width of gaussian width**
    - **Laplacian of Gaussian**
        - Combine **Gaussian smoothing with laplacian operator**
            - Instead of applying gaussian filtering and then laplacian operator, we can combine them
        - Change the ordering
        - $$h(i,j) = c\left(\frac{i^2+j^2-\sigma^2}{\sigma^4}\right)e^{-\frac{i^2+j^2}{2\sigma^2}}$$
            - 
            - **Mexican hat filter**
                - Positive in centre, goes negative and zeros out
        - **Pros**
            - Takes into account a larger area
            - Guarantees closed loops of edges
        - **Cons**
            - Can miss edges if heavy nucleation due to large area
            - Too much smoothing (losing corners)
            - Can be very large (depending on sigma) so hard to compute
        - **Computation tactics**

                - Break 2 laplacian of gaussian into 4 1D filters
                - Or difference of gaussians (smooth an image with 2 different sigmas and analyse inter-image difference)
        - **Finding zero crossings**
            - Can't look for zeros (non-continuous function)
            - If sign differs between two points then we know there has been a crossing
                - Mark latter point as zero crossing
                - **Defeats the point of 2nd deriv (high accuracy)**
    - **Multi-Scale Edge Detection**
        - Images use local pixels (from a sized neighbourhood)
            - How should we pick the size? Really depends on the objects we're looking for in the image
        - **We can avoid this near paradox problem by processing at multiple scales simultaneously**
            - Observe the differences that occur during processing at multiple scales and use this knowledge that we wouldn't have otherwise
        - **Use multiple gaussians on one image to obtain a sense of scale**
            - **Common discontinuities suggest edges**
    - **Canny Edge Detection**
        - Combines first and second derivative
            - **Can obtain magnitude and orientation**
        - **Optimises three criteria:**
            - Detection - shouldn't miss edges
            - Localisation - distance between located and actual edges should be minimal
            - One response - minimises responses to each edge
        - 

- **Multispectral edge detection**
    - I.e. colour edge detection
    - Difficult - different colours can have similar greyscales, so edges can be missed easily
    - **3 main methods to approach it**

- **Vector methods**
    - Colours treated as vectors
    - Calculations of median vector and vector distance can be used to compute magnitude and orientation
    - Each colour is a single entity and not 3 dimensions
- **Multidimensional gradient methods**
    - Gradient and orientation are computed using data from all three channels - complex
- **Output fusion methods**
    - Separate computation of gradient and orientation for all channels and then combined into one using a weighted sum of sorts

- **Image sharpening**
    - Make image edges steeper
    - Done by subtracting a multiple e.g. 0.3 of the Laplacian from the image

- **Contour Segmentation**
    - Edge images need to be more explicitly represented to prove useful
        - Need to definitively say which points are edges and use e.g. graph traversal to record information about them
    - **Basic edge data representations**
        - **Boundary Chain Codes**
            - Each chain contains a start point, and list of orientations
                - Start point and then a sequence of values (0-7) referring to the 8 possible orientations
            - Essentially synthesises edges to a shape
                - **Heavily dependent on orientation and scale**
                - **Somewhat position dependent**
                - **Can use smoothing to reduce boundary noise, but hard to get a consistent shape**
        - **Directed Graph**
            - Represent pixels as nodes joined by oriented arcs
                - Add all edge pixels to the graph if their magnitude exceeds some threshold T
                - Then look at orientation of each of these nodes, and determine which nodes are connected
                    - Consider pixels bordering the current one, and those within a $\pm 45°$ of the current node
                - The connection between nodes is extended to neighbouring nodes
            - **Algorithm:**

- Algorithm:
  1. Search for the strongest (unused) node in the graph. If none go to step 5.
  2. Expand all the edges in front of the specified edge
  3. Expand all the edges behind of the specified edge
  4. If the edge chain consists of > 3 pixels store it. Go to step 1.
  5. Modify edge chains to fill small breaks and to remove duplicate contours.
  6. Repeat step 5 until stable.

- **Border Detection**
  - A search for the optimal path from source to destination, or a search for the best representation of all edge points in the image
    - **In the first case we require a lot of a priori knowledge, so the source and destination need to be at least roughly known beforehand**
    - In the latter case we are trying to extract a general representation of all the edge contours in the scene
  - Uses directed graph traversal, finding the best path from the start to end
    - Requires some defensive labelling to prevent infinite loops caused by circles in the graph
  - **Requires a cost function**
    - **Strength based:** the cost of adding a new node is the difference between the magnitude of the strongest edge points in the image, and the one being added
    - **Border curvature:** the cost of adding a node to the graph is the absolute change in orientation (**may be better to use relative positions and orientations and create an inverse of continuity**)
    - **Distance to an expected border:** useful so long as an approximate border is known
    - **Distance to the destination:** euclidean distance from the given node to the destination
- **Line segment extraction**
  - Particularly with man made objects we can summarise large amounts of information (depending on the length of the line) by only noting the start and end points of the segments
  - **Recursive boundary splitting**
    - Splits a contour into multiple line segments using the data of the line to drive it
    - **Using the curve an a line connecting the start and end, split the line at the point which is as far away from the straight line as possible**
      - Recursively do this until the distance to the curve is less than a threshold
  - **Divide and conquer**

        - Given a tolerance value, if the distance to the curve is greater than it, split the line in the middle - repeat until all line segments are less than the threshold
    - **While both of these succeed in closing in on the curve, there is clearly better ways to approximate/extract information**
        - **Curved segments**
            - Introduces a lot of questions (so we generally stick to straight lines)
                - Where does one curve stop and another start? What order of polynomial is to be considered?
            - **Understanding curves is still vital for feature detection**
                - Compare a point x on the contour, to the point n behind it, and the point n ahead of it
                    - **Large value approaching corners**
                    - **Small value following corners**
- **Hough Transform**
    - Direct transformation from image space to the probability space of the existence of some feature
        - Lines / Circles / Generalised Shapes
        - **Can even detect partial objects**
    - Expensive computation, reductions available:
        - **Scaled processing:** find local maxima at low res, restrict probability space searches at higher res to these areas
        - **Use of edge orientation information:** use edge orientation information to direct hough transform's search
    - **Hough for Circles**
        - **Equation for a circle:** $(i-a)^2 + (j-b)^2 = r^2$
            - Assumes constant r, where (a,b) is the circle centre
            - When r is unknown we create a 3D probability space (a,b,r)
        - We transform from image space (x,y) to Hough space (a,b)
        -
            - Initialise accumulator to 0
            - For every edge point
                - Increment cells in accumulator corresponding to all possible circle centers
            - Search for Maximums
        - **Can find partial circles so technically could find a circle out of the image by r, so hough space should be 2*r larger than the source image in both i and j**
        - As all points that are found to be one the circle are used as evidence, the centre is found to sub pixel accuracy (so hough space requires higher resolution)

- **Hough for Lines**
    - Can't use standard line equation for this, as cannot account for vertical lines
    - $r = i.\cos\theta + j.\sin\theta$
        - **r is the distance to the origin, θ is the angle to the I-axis**
    - Transform image from (x,y) to Hough probability space (r,θ)
    - Increment each point in Hough space which corresponds to all possible lines through the point
    - **Local maximums represent high probability lines**
    - **Can half our computation costs by only considering 180° but must be careful as the Hough space will be wrapped on itself**
    - **Can reduce the range of r values by moving the origin of the image to the centre**
- **Generalised Hough**
    - Define the arbitrary shape in terms of distance and angles from some reference x^R
        - **Distance r, orientation Φ, orientation α of line from x^R through edge point**
    - **Training:** build up an R-table, for every Φ store (r,α) pairs
    - **Recognition**
        - Create an accumulator for x^R
        - For every edge point
            - determine its orientation
            - Select a list of (r,α) pairs from the R-table
            - Determine the position of x^R for each (Φ,r,α) and increment appropriately
        - Search for maximums

- **Least Squared Error**
    - Linear fit which best matches data
        - Minimum error
    - **"For a straight line it minimises the sum of the vertical residuals"**
    - Compute slope m, and then the intercept c

Given (x_i, y_i) where $i = 1..N$

$$\mu_x = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad \mu_y = \frac{1}{N}\sum_{i=1}^{N} y_i$$

$$\sigma_x = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_x)^2} \qquad \sigma_y = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(y_i - \mu_y\right)^2}$$

Pearson's correlation coefficient: $\rho_{xy} = \frac{cov_{xy}}{\sigma_x\sigma_y} = \frac{\sum_{i=1}^{N}(x_i-\mu_x)(y_i-\mu_y)}{\sigma_x\sigma_y}$

-

$$m = \rho_{xy} \frac{\sigma_y}{\sigma_x} \qquad c = \mu_y - m.\mu_x$$

  -
  - **Assuming:**
    - Line is not vertical
    - Distribution is normal
    - That the points which should be included are known
    - No significant outliers
- **Random Sample Consensus (RANSAC)**
  - Uses the minimum number of data points (m) to determine the model
    - For a straight line m = 2
  - **Algorithm:**
    - Randomly select the m data points from the N available (where each data point is a coordinate pair)
    - Determine the model using the points
    - Determine how many points are with the tolerance of the model - **the consensus set**
    - If the set size is below a threshold go back to step 1
    - If the set is big enough, re-compute the model using the consensus set as input