IVR Condensed Summary Notes For Quick In-Exam Strategic Fact Deployment

Maksymilian Mozolewski

December 9, 2020

Contents

Vision

Introduction to Vision

Computer vision

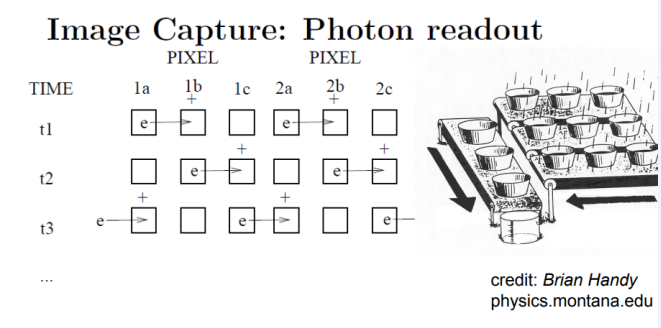
Processing data from any modality which uses the electromagnetic spectrum and produces an image

Image

Way of representing data in a picture-like format, with a direct correspondence to the scene being imaged

CCD Camera

Charged couple device, light falls on an array of MOS capacitors (which are rectangular and not square). The capacitors form a shift register and output either a line at a time or the whole array at one time (line vs frame transfer)



these "buckets" can overflow, resulting in over-saturation of the image

Frame grabber

Device which converts analog image signals to digital image signals. Essentially puts a discrete value on each pixel signal. 24bit color is usually required for robotics

Visual Erosion

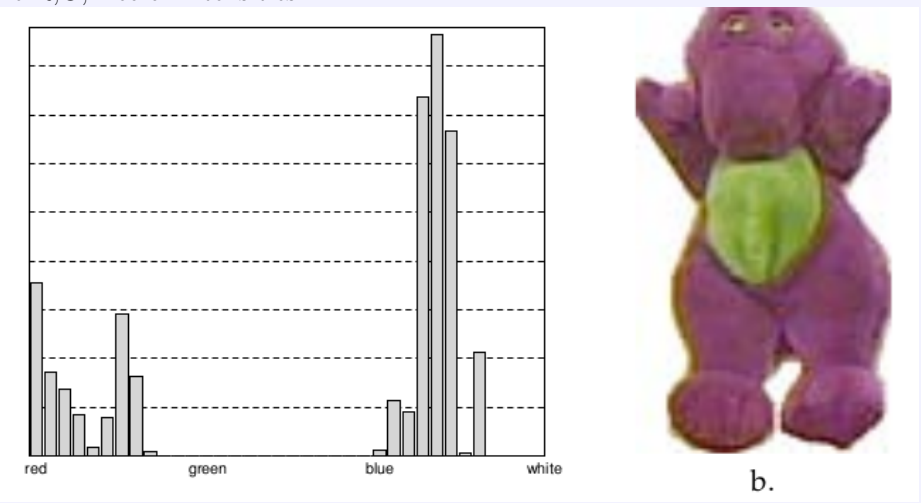
RGB is a function of the sensitivity of the sensor to reflected light of each color. The sum of those intensities may vary wildly from frame to frame depending on the distance of the object due to intensity of the reflected light. The object appears to "errode" with changes in lighting. CCD Cameras are also notoriously insensitive to red, meaning that one of the three color planes is not as helpful in distinguishing colors. HSI and SCT colour spaces aim to reduce visual erosion since the Hue - the main wavelength measured (**perceptually meaningful dimensions**) will not change with the object's relative position, only its saturation and intensity will! Equipment to capture HSI images is expensive, and conversions between colour spaces sometimes fail.

Region Segmentation

Finding groups of pixels related to each other via color, within a certain threshold and identifying the centroids of those groups. Requires high contrast between the **foreground** (object of interest) and the **background** to work well.

Color histogramming

a type of histogram (bar chart basically), the user specifies range of values for each bar, (bucket) the size of the bar is the number of data points falling within the bar's "range". These ranges could be set to capture different values of either the R,G,B color intensities.



Such histograms can be **subtracted bucket-wise** from each other as a form of distance measure to compare image stimuli.

Stereopsis

The method of triangulating depth data from 2 POV's

Stereo camera pairs

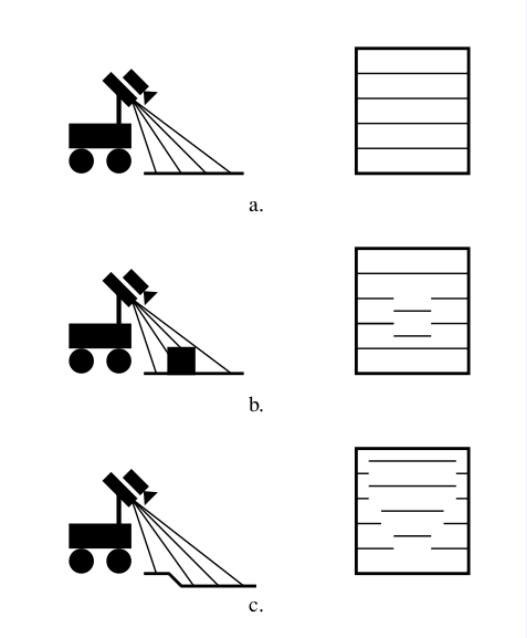
Usage of two cameras to extract range data by finding the same point on the images received from two (most likely parallel) cameras, and then finding the depth information using the geometry of the cameras. It can be hard to find the same point on two pictures ()**correspondence problem**), the method of picking a spot of interest is called an **interest operator**. Cameras can be mounted in parallel to produce **rectified images** (the distance between the two cameras is then known as the **disparity**). This can save computation time since the point of interest will appear in the same line of the image on both cameras (**epipolar lines**)

Optic flow

Information to do with: Shadow cues, texture, expected size of objects

Light stripping

Method of projecting a pattern of light onto a surface of interest and observing the distortion to the pattern to visualise the surface and/or distance information. Does not work that well in natural conditions due to noise.



Laser ranging

like radar but using light (**lidar**), scanning components are expensive, a planar laser range finder is a cheaper alternative. Produces an intensity and range map.

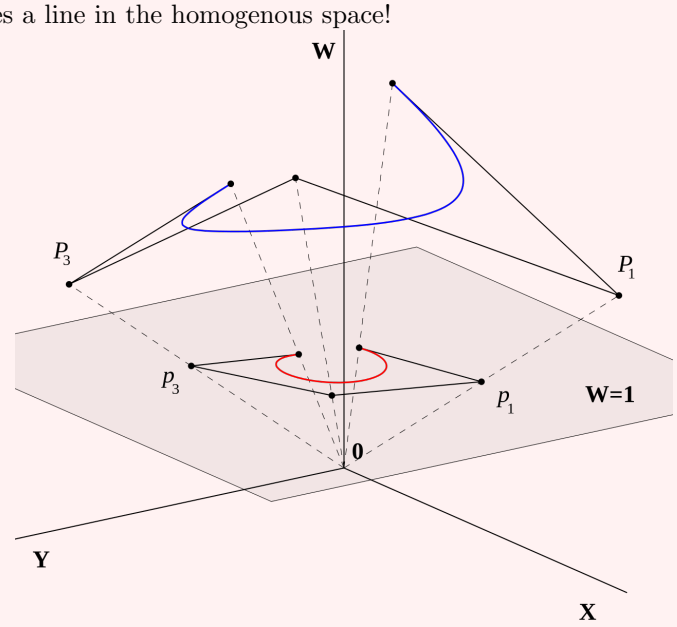
Range segmentation

Segmenting the image based on range data, can be used to determine the geometry of surfaces

Image Basics

Homogenous coordinates

Homogenous (aka similar) coordinates are coordinates in space with one more dimension than in the corresponding cartesian space, in this space we can express linear translations as linear matrix transformations! Every point in the cartesian space becomes a line in the homogenous space!



Conversion to homogenous coordinates:

$$\begin{bmatrix} x \\ y \\ \vdots \end{bmatrix} = \begin{bmatrix} x \\ y \\ \vdots \\ 1 \end{bmatrix} \tag{1}$$

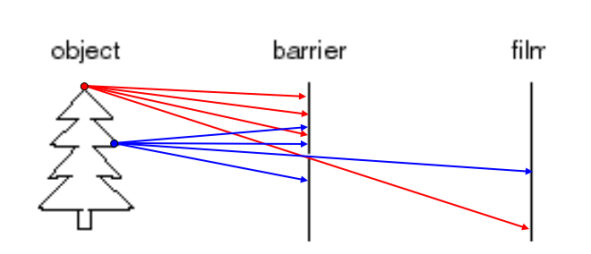
Conversion from homogenous coordinates

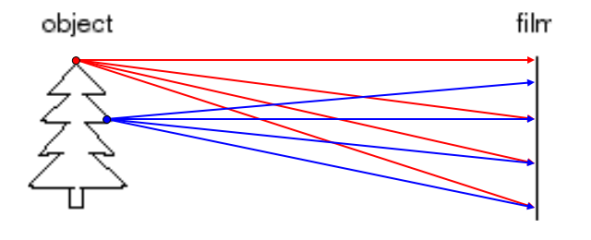
$$\begin{bmatrix} x \\ y \\ \vdots \\ w \end{bmatrix} = \begin{bmatrix} x/w \\ y/w \\ \vdots \end{bmatrix} \quad w \neq 0 \tag{2}$$

Notice how a point in homogenous space can be multiplied by any constant, and yet when it is converted back to normal space, it becomes the same point. **The ratio** between the components defines the line in homogenous space.

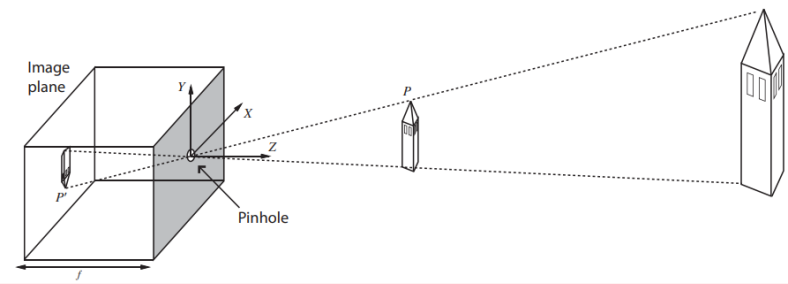
Pinhole camera

Capturing on a simple plane does not work because multiple rays from the same point in the scene travel to multiple parts of the film. We want the film to capture a single "ray" per point of interest





A camera setup using a tiny hole to filter and hence focus the light onto a single clear image.



Using similar triangles, the point P:(X, Y, Z) maps to point P' on the 2d surface of the image plane, at a distance f (**focal length**) from the pinhole as follows:

$$x = \frac{-fX}{Z}, y = \frac{-fY}{Z}, z = f \tag{3}$$

This projection of scene point to camera point can be expressed as a linear matrix transformation in homogenous space:

$$P_h = \begin{bmatrix} X \\ Y \\ -Z/f \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1/f & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{4}$$

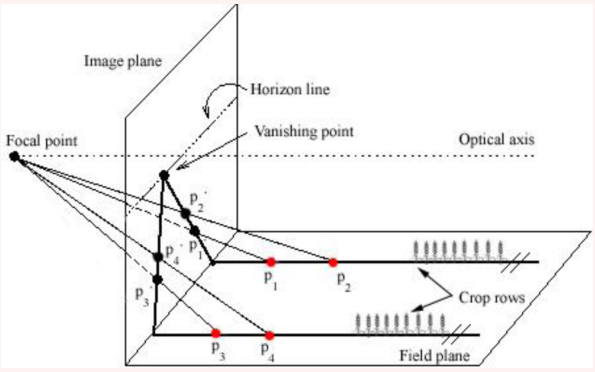
To retrieve the projected point in cartesian space we simply divide by the third coordinate and discard it.

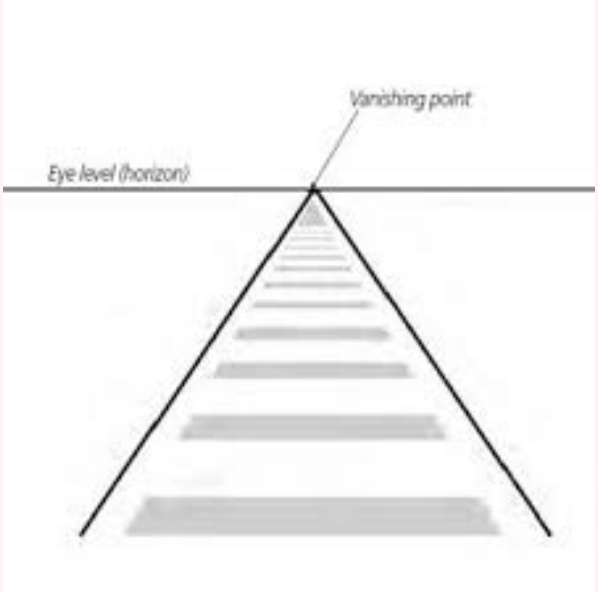
$$P_c = \begin{bmatrix} X/(-Z/f) \\ Y/(-Z/f) \end{bmatrix} = \begin{bmatrix} -fX/Z \\ -fY/Z \end{bmatrix} \tag{5}$$

Which is identical to the projection above.
This projection, preserves straight lines (**colinearity**) and their intersections, but looses information about angles and lengths (due to multiple points in 3D possibly mapping to the same point in 2D)
Lines directly passing through the focal point are projected as points.
Planes are preserved but those passing through the focal point are projected as lines.

Vanishing point

Any two parallel lines will converge to a certain point on the image as long as their directions are the same





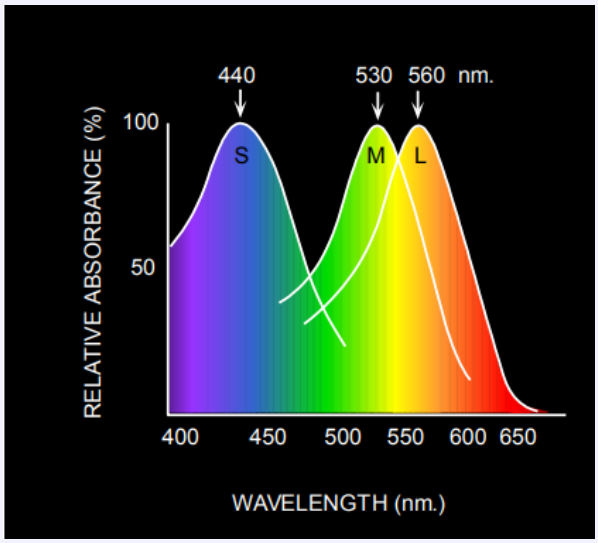
Detector response curve

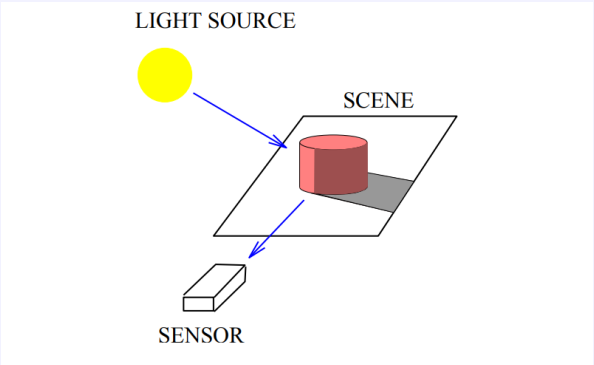
The curve showing which frequencies of light a detector perceives the most and which will dominate the actual "perceived" or "central" wavelength of light, i.e. the curve showing which wavelength of light a detector is most sensitive to. Each sensor type acts as a filter to the incomming light, and can produce an output signal proportional to the amount of its central wavelength absorbed.

The wavelength signal perceived is a function of many things:

- type of source light
- the reflective properties of the objects in the scene
- the sensor detector curve

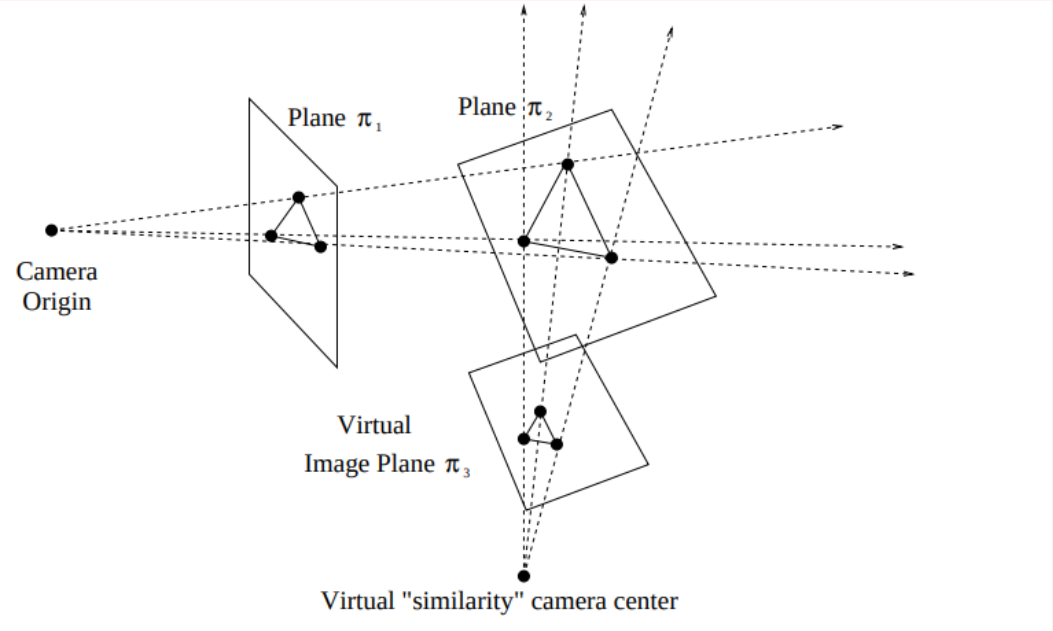
As such knowing the "real" wavelength of the light is very difficult.





Homography

An invertible linear transformation $\mathbf{P}$ that maps points from one plane to another (think of it as a change of POV)



Given at least 4 corresponding points on each plane defining a POV, we can perform a least-square estimation of $\mathbf{P}$:

$$\mathbf{P} = \begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{bmatrix} \tag{6}$$

let $\mathbf{p} = (p_{11}, p_{12}, p_{13}p_{21}, p_{22}, p_{23}, p_{31}, p_{32}, p_{33})$

let $\mathbf{A}_i = \begin{bmatrix} 0 & 0 & 0 & -u_i & -v_i & -1 & y_i u_i & y_i v_i & y_i \\ u_i & v_i & 1 & 0 & 0 & 0 & -x_i u_i & -x_i v_i & -x_i \end{bmatrix}$

construct $\mathbf{A} = \begin{bmatrix} A_1 \\ A_2 \\ \dots \\ A_N \end{bmatrix}$

Compute SVD($\mathbf{A}$) = $\mathbf{UDV}'$

$\mathbf{p}$ is last column of $\mathbf{V}$ (eigenvector of smallest eigenvalue of $\mathbf{A}$)

Then once we know the homography $\mathbf{P}$, then we can map (u,v) onto (x,y) using:

$$\begin{pmatrix} \lambda x \\ \lambda y \\ \lambda \end{pmatrix} = \mathbf{P} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \tag{7}$$

(λ representing the fact that this coordinate is in homogenous space)

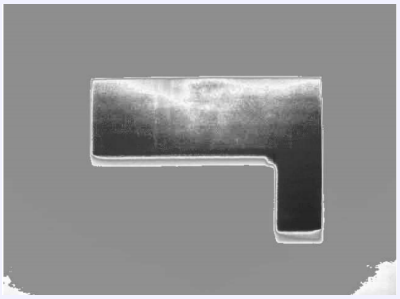
Focus problems

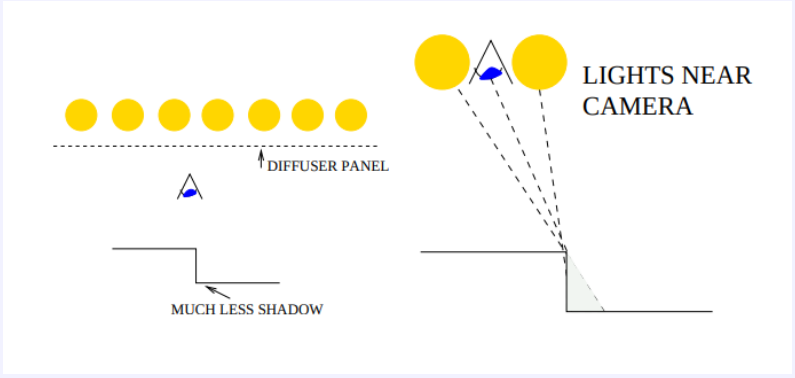
Focus set to one distance, and other nearby distances in focus (depth of focus). Further or closer not so well focused.

Solutions: Use smaller aperture and brighter light

Shadow problems

False colours due to different intensity of light (shadows) make it difficult to separate shapes of interest from shadows. (is the white part under this part a shadow or the edge ?)
Main cause of the problem: point of light sources, the perceived brightness at a surface is proportional to the **square** of the distance between the surface and the light source.

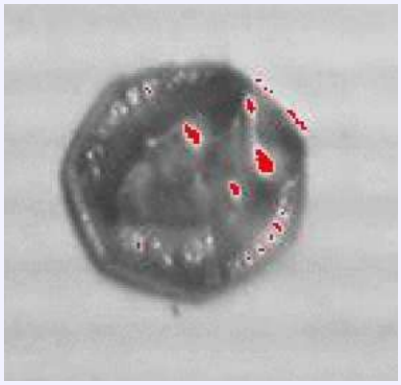




Solutions: increase ambient lighting by using diffusing panels or lots of point lights

Specularities/highlights

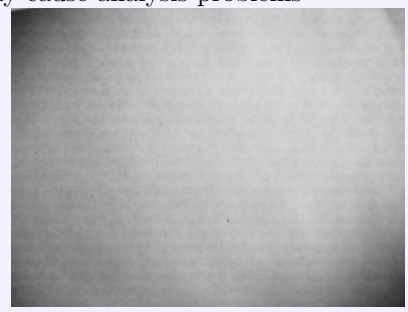
(Saturated pixels set to red)



Solutions: increase ambient lighting by using diffusing panels or lots of point lights, or use smaller aperture, reduce gain and adjust gamma

Non-uniform ilumination

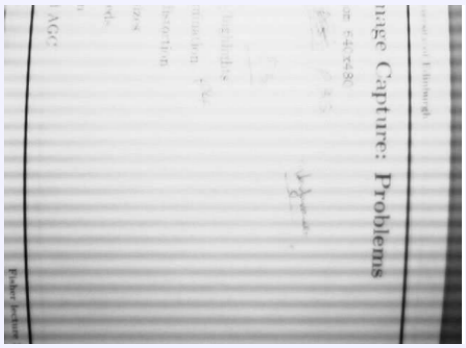
Contrast on background enhanced: may cause analysis problems

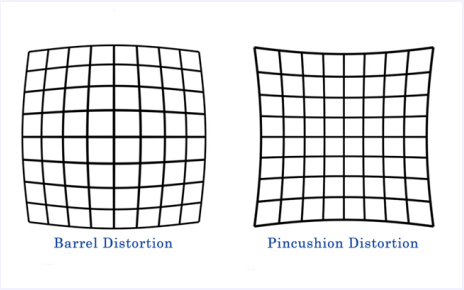


Solutions: increase ambient lighting by using diffusing panels or lots of point lights

Radial lens distortion

Lenses sometimes slightly distort the image "radially" making accurate measurements hard





Solutions: more expensive lenses, view from further away

Image Segmentation

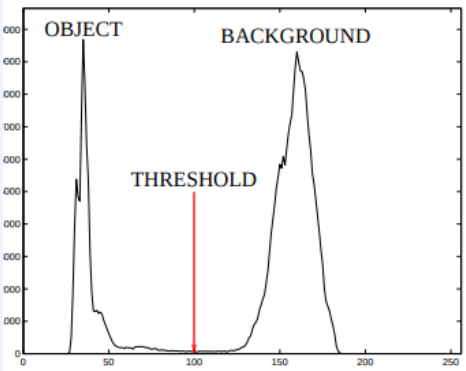
Approaches

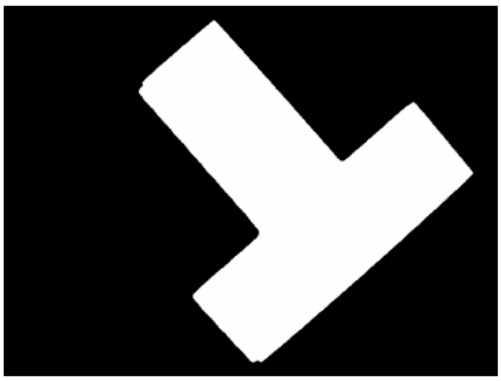
Image segmentation is the process of grouping pixels which belong together semantically, i.e. perhaps because they belong to the same object.
We can segment based on many facts:

- Contrast - objects have different lightness : use thresholding
- Change - objects different from background : background models
- Similarity - objects have consistent colours : colour clustering

Thresholding

This method assumes that pixels are separable based on their color values. We can pick threshold boundaries for each color value and select regions based on regions of pixels which fall in those boundaries.





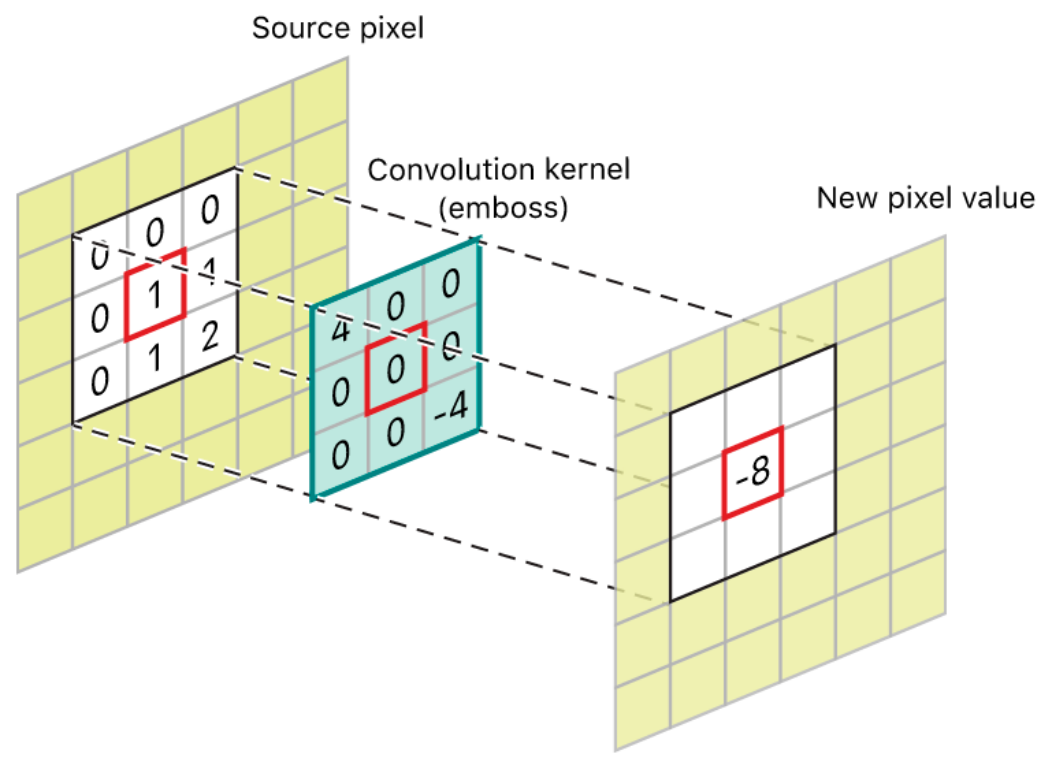
Histogram Thresholded Image

problems:

- Distributions may be broad and have some overlap leading to misclassified pixels
- variations in lighting might cause parts of the object to be missing, or shadows to be classified as objects
- color distributions might have more than 2 peaks

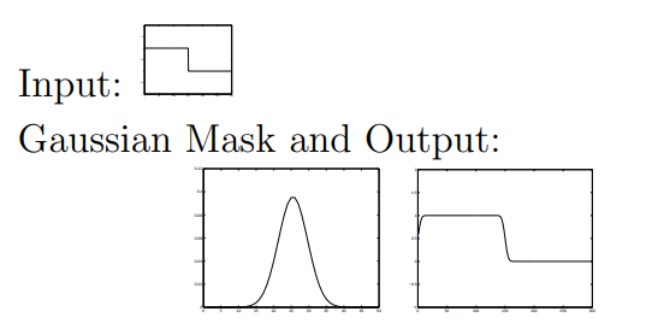
Convolutions

General-purpose image (and signal) processing function.
can be used to remove noise, smooth data, or detect features!
In the case of thresholding, we can use convolutions to smooth the histogram. Imagine convolutions as a sliding window, where each point in the original image is replaced with the weighted average of the window at that position with the pixels.



Convolution in 1D, with kernel of size (odd) N (even kernels require padding with zeros):

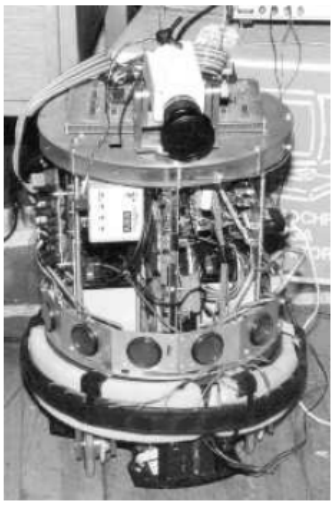
$$Output(x) = \sum_{i=-\lfloor N/2 \rfloor}^{\lfloor N/2 \rfloor} weight(i) * input(x-i) \tag{8}$$



Convolution in 2D, with kernel of size (odd) N:

$$Output(x) = \sum_{i=-\lfloor N/2 \rfloor}^{\lfloor N/2 \rfloor} \sum_{j=-\lfloor N/2 \rfloor}^{\lfloor N/2 \rfloor} weight(i,j) * input(x-i, y-j) \tag{9}$$

Smoothing kernel (2d gaussian)

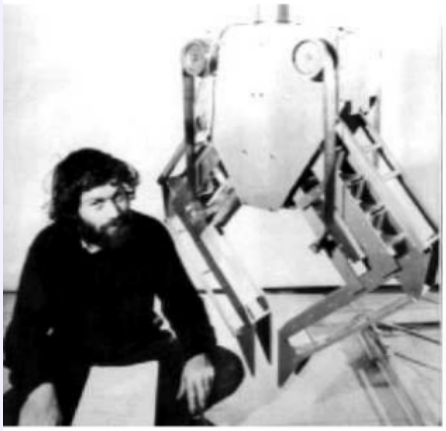


$*$

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

$=$

Edge Detection kernel



$*$

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$=$

Edge detection

$*$

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$=$

Background removal

If we have 2 images, one with just the background (**B**) and one with background and foreground (the image **I**), we can

$$N = I - B \tag{10}$$

This difference will zero-out pixels with identical values to the background, and only leave those values which are different (either positive or negative depending on if the foreground is brighter or darker than the background at each point)
We can do this for each channel of the image, and perform thresholding on the logical or between all the resulting differential pictures.

$$thr(\mid I_r - B_r \mid) \parallel thr(\mid I_g - B_g \mid) \parallel thr(\mid I_b - B_b \mid)$$

BACKGROUND FOREGROUND DIFFERENCE

we can also use division instead of substraction to achieve a similar effect:

$$N = I/B \tag{11}$$

This in effect removes the effects of illumination since:

$$background(i,j) = illumination(i,j) \cdot bg\_reflectance(i,j) \tag{12}$$

$$object(i,j) = illumination(i,j) \cdot obj\_reflectance(i,j) \tag{13}$$

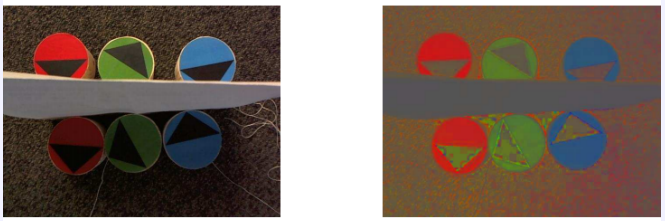
The pixels with a value of 1 are going to be the background, pixels with value > 1 are lighter objects and pixels with values < 1 are darker objects (than the background)
In both of these techniques, we might need to use an operator such as the **open** operator to remove noise artifacts (with values which are just around the values which signify background pixels but not quite)
Neither will work well when the background in I and B varies wildly.

RGB Normalisation

differences in lighting can be dealt with by normalising the RGB values of the image:

$$(r', g', b') = (\frac{r}{r+g+b}, \frac{g}{r+g+b}, \frac{b}{r+g+b}) \tag{14}$$

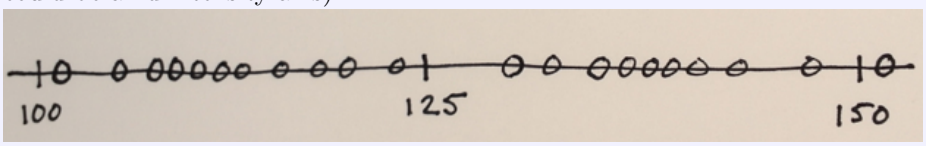
since multiplying all values r,g,b in the original space by a constant, changes the brightness of the color, we remove this effect thanks to the equation above, mapping all different brightness values of the same colour to one value.



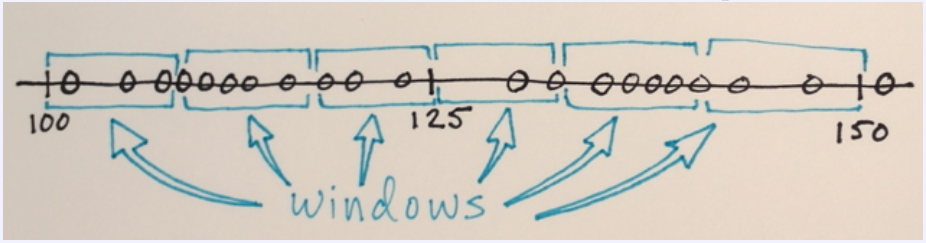
Mean Shift Segmentation

We can segment the image by performing clustering on the pixels by their color values (or any attributes for that reason)!
The algorithm works as follows:

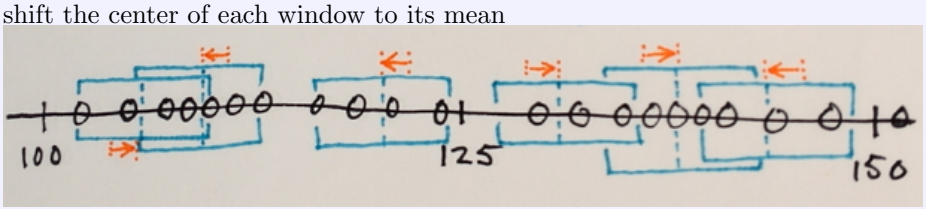
1. create a feature space over the attributes chosen to represent each pixel (for example for a grayscale this could be a 1d intensity axis)



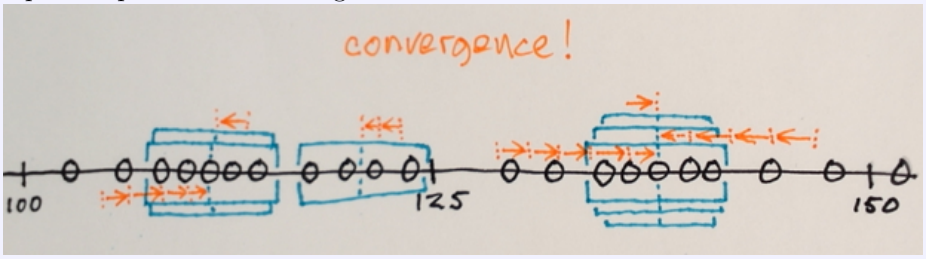
2. distribute a number of "search windows" or kernels over the space



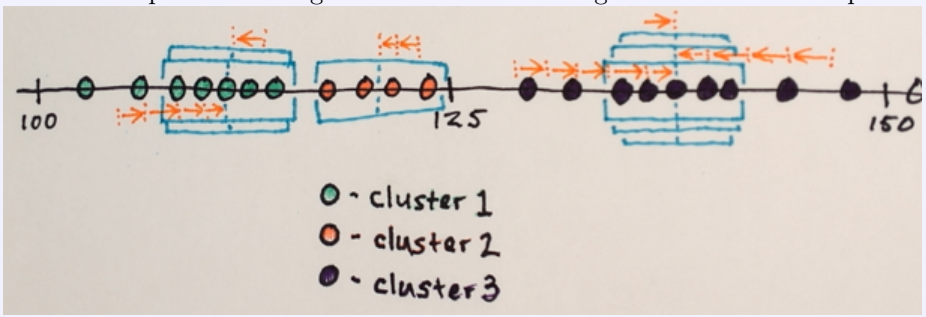
3. calculate each window's mean
4. shift the center of each window to its mean



5. repeat steps 3-4 until convergence



6. merge windows ending up in close-enough locations, and call these the clusters
7. cluster each pixel according to which cluster its original window ended up at

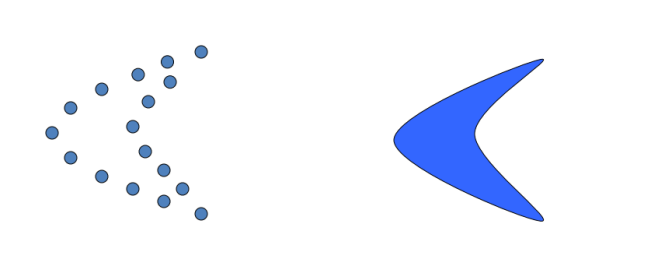


the feature space can contain any number of dimensions, and so we could include spatial, color, texture-data, and so on. This is a very versatile algorithm. It is application-independent, model-free (does not assume any shape of clusters), only requires a single parameter (window size h) which affects the scale of the clustering It is robust to outliers and finds a variable number of modes given the same h.
The output is heavily dependent on the window size h, however. And the selection of h is not trivial. The whole algorithm is rather expensive and does not scale well with the dimension of the feature space.

Description of Segments

Shape

a set of points in the plane, or a continuous outline (silhouette)

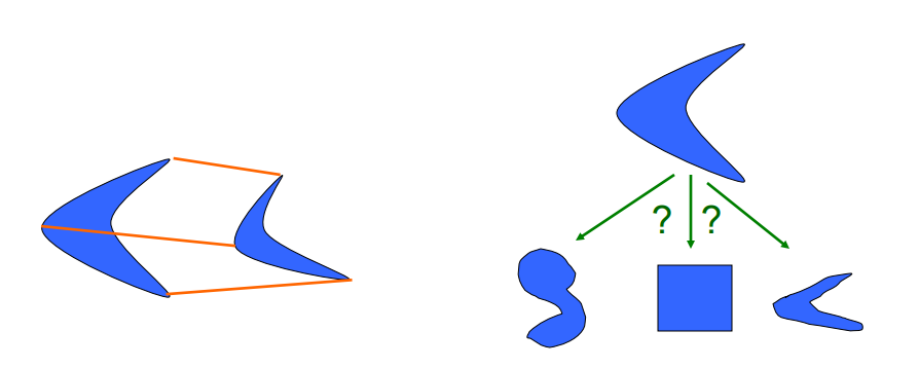


Cues

shapes can give us cues (**interior** and **boundary** cues)about the objects they outline.
Some classes are defined purely by the boundary of the shape, some are defined purely by the **contents/interior** of the shape (i.e. texture,color), and some are defined by a mixture of both

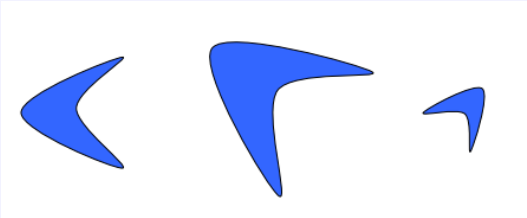
Correspondence and recognition

We can draw conclusions about similarities between shapes using **point-to-point** correspondences or **shape characteristics** to help us recognize objects belonging to certain classes.

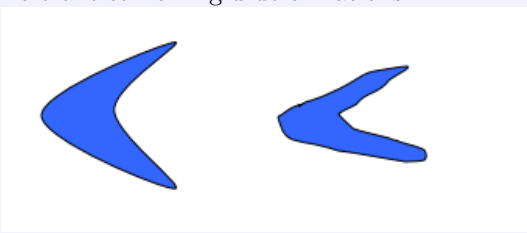


Good methods of finding similarities will be :

- Invariant to rigid transformations like: translation, rotation and scale



- Tolerant to non-rigid deformations



Global shape descriptors

Shape descriptors which put a number of a certain characteristic of a shape based on its **entirety** - hence "global".

Convexity

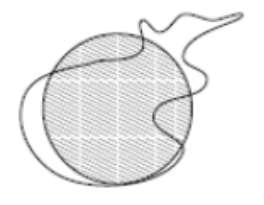
Convexity describes the ratio of a shape's convex hull to its perimeter, values of 1 mean that the shape is entirely convex, and values < 1 mean the shape is less convex.

$$conv = \frac{P_{hull}}{P_{shape}} \tag{15}$$

Compactness

Compactness describes how close the perimeter of the shape is to the perimeter of the circle with the same area.

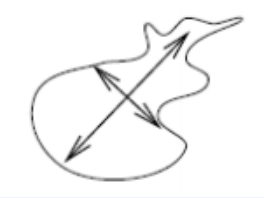
- If the circle of equal area has a smaller perimeter, this value will be smaller than 1, meaning that the shape's "mass" is distributed in a less compact manner.
- If the circle of equal area has equal parimeter, this value will be equal to 1, meaning the shape's "mass" is distributed as compactly as possible.
- This value cannot be greater than 1, as the circle is the most compact distribution of mass



$$comp = \frac{2\sqrt{A\pi}}{P_{shape}} \tag{16}$$

Elongation

The elongation is simply the ratio of the principal axes, i.e. the aspect ratio of a shape, this value can be anywhere between 0 (flat line) and ∞ (also flat line) This can be computed by taking the cross product of the principal axes with their length being set to the eigen values of the covariance matrix (if you treat each pixel as a data point)



$$elong = \frac{c_{yy} + c_{xx} - \sqrt{(c_{yy} + c_{xx})^2 - 4(c_{xx}c_{yy} - c_{xy}^2)}}{c_{yy} + c_{xx} + \sqrt{(c_{yy} + c_{xx})^2 - 4(c_{xx}c_{yy} - c_{xy}^2)}} \tag{17}$$

Properties of these global descriptors

+ Invariant to translation/rotation/scale (rigid)
+ Robust to shape deformations (non-rigid)
+ Simple
+ Fast to compute
- These do not find any point correspondences,
- Little power to discriminate between shapes (Can you discriminate between the shape of a horse and a plane with these ?)

Moments

Moments in mathematics are measures which put a number on the function of interest's graph. A shape can be thought of like the graph of some function defined on the 2D space (**f(x,y)**)
Family of stable **binary** (and grey level) shape descriptions which can be made invariant to translation, rotation and scaling
Let p_{yx} be the pixel value $\in 0, 1$ at row y and column x
Area $A = \sum_y \sum_x p_{yx}$
Center of mass $(\hat{y}, \hat{x}) = (\frac{1}{A}\sum_y \sum_x y \cdot p_{yx}, \frac{1}{A}\sum_y \sum_x x \cdot p_{yx})$ i.e. average of x and y values weighted by "mass"

Translation invariant

let $u, v \in \mathbb{Z}$
then a family of 'central' (translation invariant) moments can be defined as:

$$m_{uv} = \sum_y \sum_x (y - \hat{y})^u (x - \hat{x})^v p_{yx} \tag{18}$$

notice how with $u, v = 2$ this is somewhat similar to variance and a little close to the moment of inertia ($\sum_p mr^2$).

This moment encapsulates the distribution of points around the center of mass, thanks to this it does not matter where the shape is positioned.

Scale invariant

We can make this family of moments invariant by noticing the fact that if we double the dimensions uniformly, then the moment m_{uv} increases by a factor of $2^u 2^v$ w.r.t weightings $(y - \hat{y}, x - \hat{x})$ and its area increases by 4. Hence $A^{\frac{u+v}{2}+1}$ grows by a factor of $4 \cdot 2^u 2^v$, Therefore the ratio:

$$\mu_{uv} = \frac{m_{uv}}{A^{\frac{u+v}{2}+1}} = \frac{m_{uv}}{m_{00}^{\frac{u+v}{2}+1}} \tag{19}$$

is invariant to scale (it cancels out the effects of increasing area, i.e. area = 1)

Rotation invariant

We can generate a similar moment using complex numbers and multiple scale-invariant moments which is invariant to rotation:
let $c_{uv} = \sum_y \sum_x ((y - \hat{y}) + i(x - \hat{x}))^u ((y - \hat{y}) - i(x - \hat{x}))^v p_{yx}$
then let:

$$\begin{aligned} s_{11} &= c_{11}/A^2 \\ s_{20} &= c_{20}/A^2 \\ s_{21} &= c_{21}/A^{2.5} \\ s_{12} &= c_{12}/A^{2.5} \\ s_{30} &= c_{30}/A^{2.5} \end{aligned} \tag{20}$$

we can combine these to get rotation invariant descriptors in similar magnitudes like so:

$$\begin{aligned} ci_1 &= real(s_{11}) \\ ci_2 &= real(10^3 \cdot s_{21} \cdot s_{12}) \\ ci_3 &= 10^4 \cdot real(s_{20} \cdot s_{12}^2) \\ ci_4 &= 10^4 \cdot imag(s_{20} \cdot s_{12}^2) \\ ci_5 &= 10^6 \cdot real(s_{30} \cdot s_{12}^3) \\ ci_6 &= 10^6 \cdot imag(s_{30} \cdot s_{12}^3) \end{aligned} \tag{21}$$

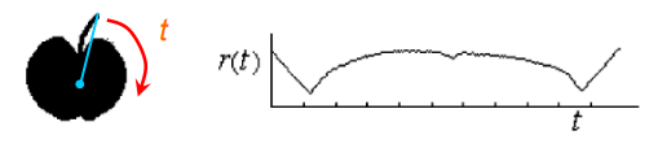
Shape signatures

We can represent the shape using a 1D function (**f(t)**) defined via the points on the boundary of the shape. Once we have such descriptors, we can establish similarity between two shapes using: $\int f(t) - f(t')$ i.e. the difference between the shape's descriptors integrated over t

Centroid distance

for angle t, and point on boundary at that angle $p(t)$

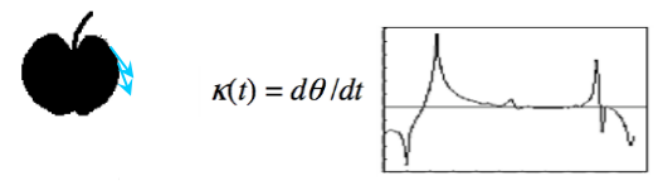
$$r(t) = d(p(t), centroid) \tag{22}$$



Curvature

for angle t, and angle θ representing the angle between points $p(t)$ and $p(t + \Delta t)$ on the boundary at the angles t and $t + \Delta t$

$$k(t) = d\theta/dt \tag{23}$$

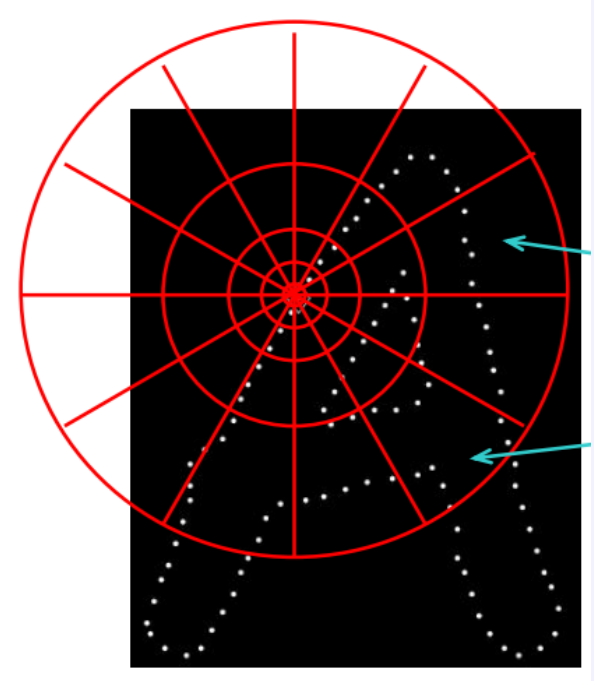


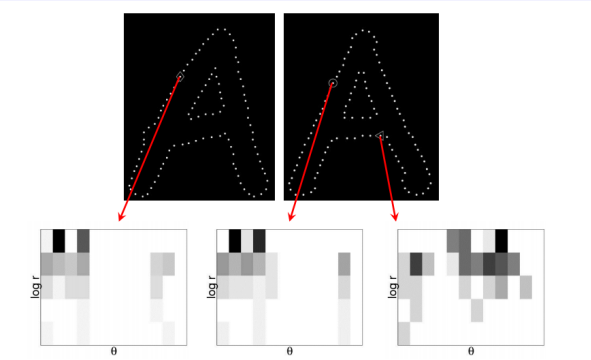
Properties of shape signatures

+ invariant to translation,scale(if shape is normalized), rotation (if orientation is normalized)
+ point correspondences (if both descriptors are aligned)
+ informative
+ deformations affect signature locally and not globally (i.e. at a single point of the signature)
~ manages to handle shape deformation to some degree
- where to start t ? high computational cost of alignment of two signature functions
- sensitive to noise (especially with derivatives)

Shape Context

Shape context is a shape descriptor utilizing the local properties of points on the boundary of each shape to establish **point-to-point** correspondences
We do this by counting the number of other points around the points on the boundary of each shape in each bin of a polar-coordinate "kernel" (this forms a histogram)





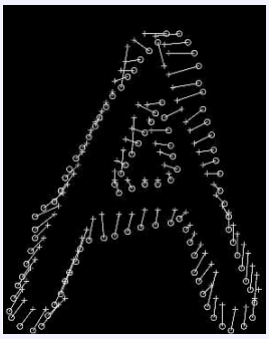
We can compare the K-bin histograms $h_i(k), h_j(k)$ of two points i, j on different shapes respectively, using the chi-squared test:

$$C(i,j) = \frac{1}{2}\sum_{k=1}^{K}\frac{(h_i(k) - h_j(k))^2}{h_i(k) + h_j(k)} \tag{24}$$

This establishes a cost function over which we can pair-up the corresponding points on each shape, by finding the least-cost matching $\pi(p)$ of points on one shape to the other (perhaps using the hungarian or blossom algorithms) which minimizes the total cost:

$$H(\pi) = \sum_{p \in all\_points} C(p, \pi(p)) \tag{25}$$

thus establishing a point-to-point correspondence between two shapes:



Propertis of shape signatures

+ invariant to translation
+ invariant to scaling (if we normalize the radial distances between points in each shape by their mean)
+ informative - describes points in the context of the overall shape
+ handles non-rigid deformations quite well - more sensitive for deformations closest to the point of interest due to shape of kernel
- not invariant to scale (but could be added by measuring angles in terms of tangents at each point instead of global coordinates)
- many parameters (# and size of bins, # of iterations, # number of points, etc..)
- very expensive computationally

Object recognition

Assumptions & Approaches

Approaches

Several approaches to classification/recognition. Choose the same class as objects with:

- **Shape** - similar shape descriptors
- **Appearence** - similar pixel values
- **Geometric** - similar structures in similar places with similar parameters
- **Graph** - similar part relationships
- **Bag of words** - similar local feature descriptors (frankenstein objects made up of smaller objects)

Assumptions

Assumptions made in this course:

- Flat objects, viewed orthographically
- Always looked at from same distance
- Good contrast everywhere
- No specularities
- shape-based recognition only

Shape-based recognition

1. Extract object from image via segmentation
2. Compute its properties
3. Use those properties to compute the class it belongs to
4. Learn/improve the model properties for the classes

Probabilistic object recognition

The process of classifying the shape into a class by calculating the probability of it belonging to each class.

Bayes rule

we can calculate the probability of feature vector $\mathbf{x}$ (which may be a collection of shape descriptor values, or any other properties) being drawn from the probability distribution which best describes the class c as:

$$p(c|\mathbf{x}) = \frac{p(\mathbf{x}|c)p(c)}{p(\mathbf{x})} = \frac{p(\mathbf{x}|c)p(c)}{\sum_k p(\mathbf{x}|k)p(k)} \tag{26}$$

where:

- $p(\mathbf{x}|c)$ is the probability of observing the feature vector $\mathbf{x}$ if it belongs to class c (using the distribution of feature vectors from class c)
- $p(c)$ is the *a priori* probability of observing a feature vector from class c (before making any observations)
- $p(\mathbf{x})$ is the total probability of seeing the feature vector $\mathbf{x}$ amongst all the classes

Multivariate Gaussian distribution

how do we model the probability $p(\mathbf{x}|c)$ of observing each feature class, knowing some feature vectors belonging to each class ?
We can perform Maximum likelihood estimation (MLE) on the observed $k > n$ (n being the dimensionality of $\mathbf{x}$)"training" instances of data and build a multivariate gaussian distribution for each class. MLE yields the following values for each class:

- mean vector of each feature $\mathbf{m}_c$ of dimension n - average value of each feature in class c:

$$\mathbf{m}_c = \frac{1}{k}\sum_{i=1}^{k}\mathbf{x}_i \tag{27}$$

- covariance matrix $\mathcal{A}_c$ - the nxn matrix of co-variances betwen each pair of features/properties:

$$\mathcal{A}_c = \frac{1}{k-1}\sum_{i=1}^{k}(\mathbf{x}_i - \mathbf{m}_c)(\mathbf{x}_i - \mathbf{m}_c)^T \tag{28}$$

With those properties the multivariate gaussian is formed as follows:

$$p(\mathbf{x}|c) = \frac{1}{(2\pi)^{\frac{n}{2}}}\frac{1}{|\mathcal{A}_c|^{\frac{1}{2}}}\exp^{-\frac{1}{2}[(\mathbf{x}-\mathbf{m}_c)^T\mathcal{A}_c^{-1}(\mathbf{x}-\mathbf{m}_c)]} \tag{29}$$

Recognition Algorithmics

We split the data into:

- a **training** set to estimate the model's parameters (e.g. the gaussian distributions)
- a **validation** set to pick the ideal "hyper" parameters which affect the performance of the algorithm without necessarily affecting the underlying model
- a **test** set to evaluate the performance of the algorithm

note: *we must have more training samples than the dimensions of the feature vectors used!*

Chamfer-based shape matching

We can employ an entirely different method of object recognition. In a way similar to 2D convolutions:

- Extract edges/contours of image we want to identify object in (perhaps using convolutions)
- Create a chamfer or "template" which forms the shape of the object we want to identify in the image
- Slide it over the image, at each point find the "chamfer distance" by calculating the average distance of points on the chamfer to closest edges in the image:

$$D_{chamfer}(T, I) = \frac{1}{|T|} \sum_{t \in T} d_I(t) \tag{30}$$

where:

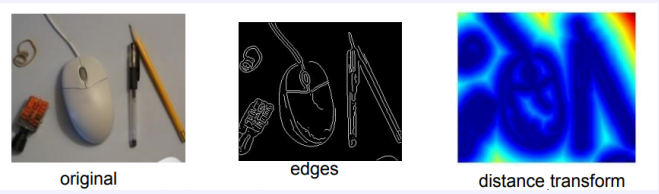
T, I are the sets of template and image points respectively

$d_I(t)$ is the minimum distance for any template point t to any point in the image I

$|T|$ is the number of points in the template

Optimisations

the naive implementation is very expensive as we re-compute the distances between each time. Instead we can do this only once by producing a look-up image of distances encoding the distance between each pixel in the image to the nearest edge inside it:



Robotics

Introduction to perception and action

Robot

A robot is a:

> *"reprogrammable, multifunctional manipulator designed to move material, parts, tools, or specialized devices through variable programmed motions for the performance of a variety of tasks"*
> - Robot institute of America

Robots are needed to perform tasks which are:

- **Dangerous**: exploration, chemical spill cleanup, disarming bombs, disaster cleanup
- **Boring and/or repetitive**: welding car frames, manufacturing parts
- **High precision or high speed**: electronics testing, surgery, precision machining.

Most robots exhibit at least some of the following:

1. **Sense** their environment as well as their own state
2. Exhibit **intelligence** in behaviour, especially planned behaviour which mimics humans or other animals
3. **Act** upon their environment, move around, opearte a mechanical limb, sense actively, communicate ...

Perception

The sensory experience of the world around us.

Types of sensory information

There are two main categories of sensory information:

- **Semantic** information - what is out there ?
- **Metric** information - where is it exactly ?

Some examples of such information include:

- Distance
  - vision
  - hearing
  - smell
- Contact
  - taste
  - pressure
  - temperature
- Internal
  - balance
  - actuator position and movement
  - pain or damage

Types of perception

There are three main categories of perception:

- **Exteroception** - the perception of external stimuli or objects
- **Proprioception** - the perception of self-movement and internal state
- **Exproprioception** - the perception of relations and changes of relations between the body and the environment (a mix of both of the above)

Actuation

An **effector** is a tool used by a robot to perform some task. An **actuator** is used to move the robot either indirectly via joint movement or directly (propulsion?).
There are two main types of joints:

- Rotary (revolute)
- Prismatic (linear)

Degrees of freedom

There are two different definitions for DoF's

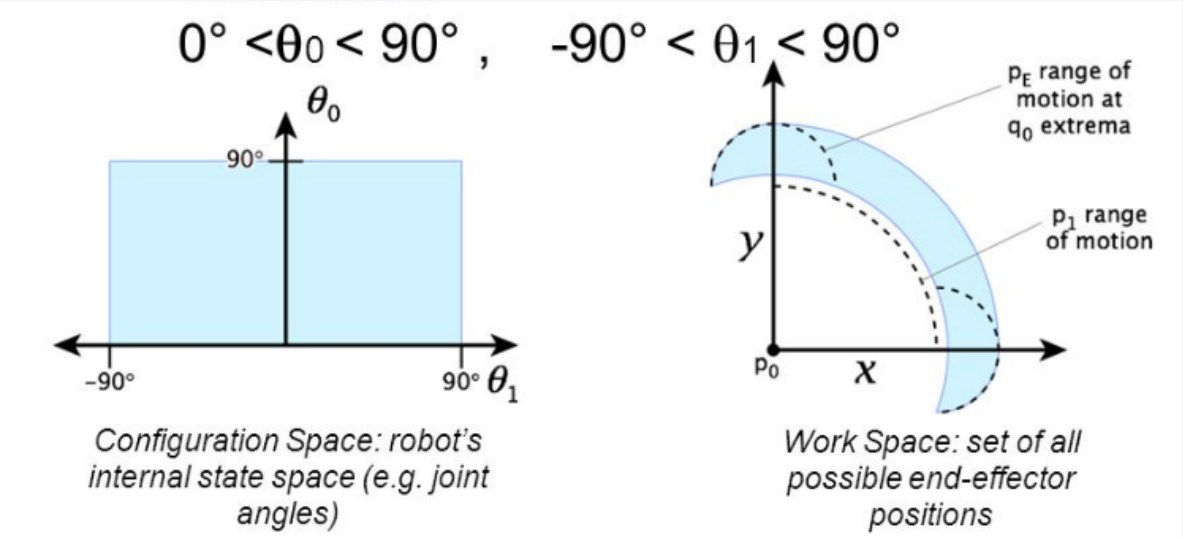
- DoF's for a task (**n**) - the number of linearly independent axes of motion required to perform a certain task (6 DoF's required to move freely in 3D space)
- DoF's of a robot (**m**) - the number of actuators the robot can move (not necessarily independent)

Notice how a robot that consists of 6 linear actuators pointing in the same direction has 3 DoF's but cannot move at all in 3D space, even though it has the required number of DoF's. So to count the "real" DoF's of a robot we might only count the independent axes of motion. Even though all the parameters of each actuator define the state of the robot.
If $m > n$, the robot is **redundant**. If $m < n$, the robot is **underactuated**. (Regardless of it's "real" DoF's)

Rigid-body motion

Configuration and Work spaces

We usually refer to the vector of all actuator parameters of a robot with the label **q**, and the position of the **end-effector** position resulting from that configuration with the label: **x**
The set of all possible configurations of the joint parameters is called the **Configuration** space.
The set of all "reachable" via end-effector, positions is called the **work** space.

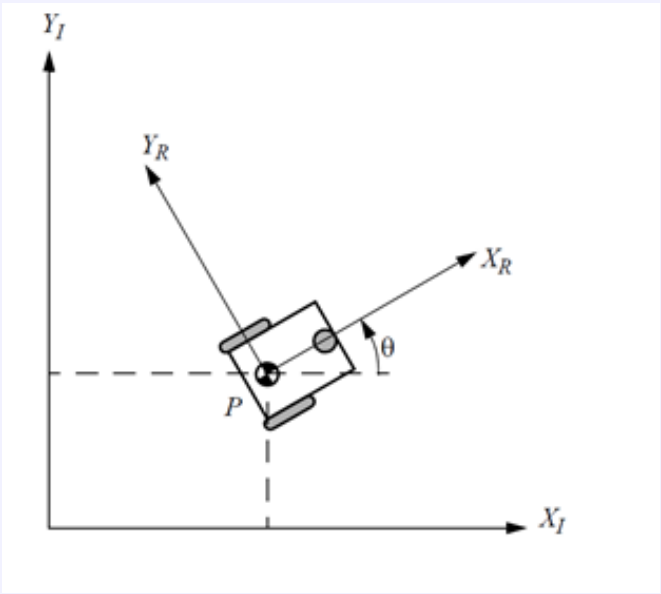


Reference frames

This is a very important concept, you cannot talk about motion, without first specifying the coordinate system you're working in. From the point of view of an observer at a train station, the trains are moving, from the point of view of an observer inside the train, the train is stationary and the world is moving.

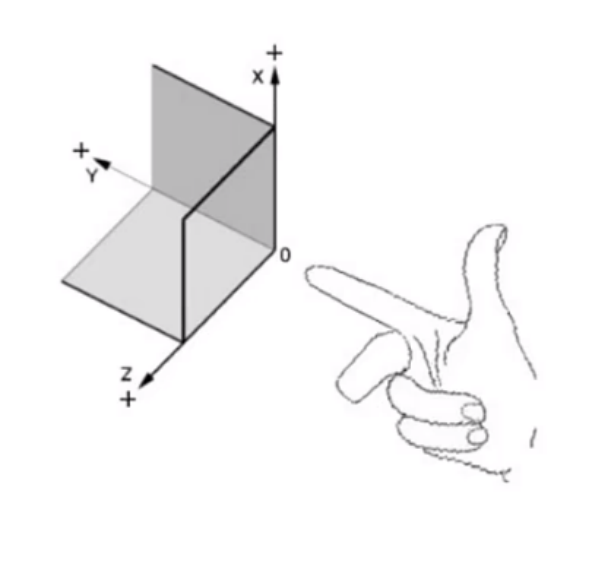
We can think of reference frames as coordinate systems, we are mostly interested in coordinate systems attached to either stationary objects, or objects moving at a constant velocity (**inertial frames**). Think trains again, if you drop something in a train which is either stationary or moving at a constant speed, the object will drop straight down as expected since it inherits the train's velocity. If the train is **accelerating** however, the object you drop still inherits the train's velocity, but **not** its acceleration, and therefore the object seems to fall towards the back of the train while falling.

If we express the movements of our robot in terms of a frame which accelerates, we will see similar artifacts.



Right handed coordinates

The most common way of assigning the axes names is using the right handed rule:

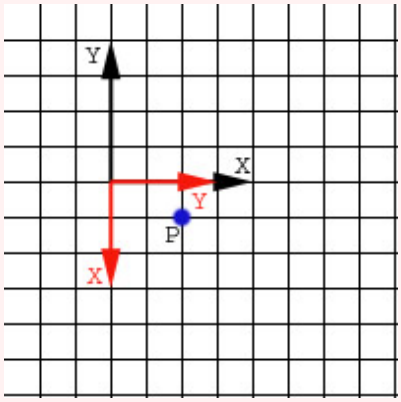


Changing between frames

We can express the coordinates of in one frame, within another frame's coordinates. This can be done using matrix transformations.
The matrix transformation which transforms points in frame A to points in frame B - will convert points from frame B to points in frame A

Transforming between points in different frames

Consider the black and red reference frames A,B respectively below:



The point P can be expressed as either $P_A = \begin{pmatrix} 1/2 \\ -1/4 \end{pmatrix}$ or $P_B = \begin{pmatrix} 1/3 \\ 2/3 \end{pmatrix}$, relative to either frame.
What does this mean precisely ? Here both these frames use different basis vectors, and these coordinates underneath represent this:

$$\begin{aligned} &(1) \quad P_A = 1/2 \cdot i_A - 1/4 \cdot j_A \\ &(2) \quad P_B = 1/3 \cdot i_B + 2/3 \cdot j_B \end{aligned} \tag{31}$$

Notice how the definitions i_A, j_A and i_B, j_B are ambigious, since we can express these in any coordinate system. How can we convert between points P_A and P_B ?

- $P_A \to P_B$: simply use equation (1) but express i_A, j_A in the coordinate system of frame B:

$$i_A = 0 \cdot i_B + \frac{4}{3} \cdot j_B, \quad j_A = -\frac{4}{3} \cdot i_B + 0 \cdot j_B \tag{32}$$

$$P_{A\ in\ B} = 1/2 \cdot \begin{pmatrix} 0 \\ 4/3 \end{pmatrix} - 1/4 \cdot \begin{pmatrix} -4/3 \\ 0 \end{pmatrix} = \begin{pmatrix} 1/3 \\ 2/3 \end{pmatrix} \tag{33}$$

- $P_B \to P_A$: simply use equation (2) but express i_B, j_B in the coordinate system of frame A:

$$i_B = 0 \cdot i_A + -\frac{3}{4} j_A, \quad j_B = \frac{3}{4} i_A + 0 \cdot j_A \tag{34}$$

$$P_{B\ in\ A} = 1/3 \cdot \begin{pmatrix} 0 \\ -3/4 \end{pmatrix} + 2/3 \cdot \begin{pmatrix} 3/4 \\ 0 \end{pmatrix} = \begin{pmatrix} 1/2 \\ -1/4 \end{pmatrix} \tag{35}$$

Expressing points from one frame in another frame

We can express the transition between frames as a matrix transformation, a matrix whose columns are unit vectors of frame B expressed in frame A's coordinate system:

- Maps points in frame B's coordinate system to points in frame A's coordinate system
- Aligns the frame A with the frame B
- Expresses coordinates originally with respect to B's coordinate system, as coordinates with respect to A's coordinate system

$$P_A(P_B) = \begin{bmatrix} i_{B\ in\ A} & j_{B\ in\ A} \end{bmatrix} P_B \tag{36}$$

So in our concrete example, to express points in B with respect to A's coordinate system:
$\begin{bmatrix} 0 & -3/4 \\ -3/4 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -3/4 \end{bmatrix}$

Rotation

Rotations between frames are nothing more than changes of basis, to rotate from frame A to frame B, we form a matrix with columns formed by frame B's basis vectors in A's space (using geometric relationships)
Using this method we can form elementary rotation matrices around each axis:

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{37}$$

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{pmatrix} \tag{38}$$

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \tag{39}$$

Properties

All rotation matrices:

- Are **orthogonal** $R^T = R^{-1}$
- Have determinants of 1 (do not change vector lengths)
- Can be fully defined with 3 variables

Translation as a matrix transformation

Translation is a non-linear transformation and as such cannot be represented as a matrix transformation. It can however if we use Homogenous coordinates!
We can combine a rotation R and translation by O (in this order) into a single homogenous matrix transformation:

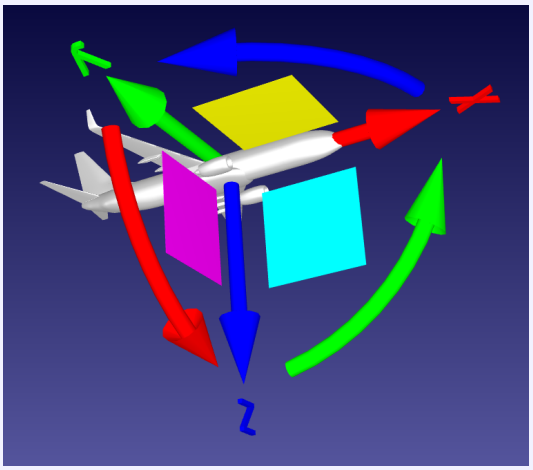
$$A = \begin{bmatrix} R_{3\times3} & O_{3\times1} \\ 0_{1\times3} & 1 \end{bmatrix}_{4\times4} \tag{40}$$

Properties

$$A^{-1} = \begin{bmatrix} R^T & -R^T O \\ 0 & 1 \end{bmatrix} \tag{41}$$

Orientation

We know how to describe a position in space, but how do we define the orientation of something in 3D space ? There are many conventions, all based on the idea of picking a specific series of rotations, around different axes in a specific order.
In general we can describe any orientation with 3 variables



Euler angles

To describe an orientation with the Euler angle convention, we perform rotations as follows:

$$R = R_z(\alpha)R_{y'}(\beta)R_{z''}(\gamma) = \begin{pmatrix} c_\alpha c_\beta c_\gamma - s_\alpha s_\gamma & -c_\alpha c_\beta s_\gamma - s_\alpha c_\gamma & c_\gamma s_\beta \\ s_\alpha c_\beta c_\gamma + c_\alpha s_\gamma & -s_\alpha c_\beta s_\gamma + c_\alpha c_\gamma & s_\gamma s_\beta \\ -s_\beta c_\gamma & s_\beta s_\gamma & c_\beta \end{pmatrix} \tag{42}$$

Notice how each rotation rotates the frame result of the previous rotation.
To calculate the parameters from any euler rotation matrix we use:

$$\alpha = atan_2(r_{23}, r_{13}), \beta = atan_2(\sqrt{r_{13}^2 + r_{23}^2}, r_{33}), \gamma = atan_2(r_{32}, -r_{31}) \tag{43}$$

Roll,Pitch,Yaw angles

This convention is mostly used in aerospace engineering:

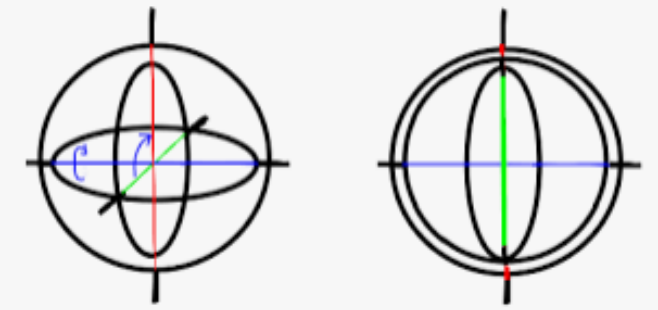
$$R = R_z(\alpha)R_{y'}(\beta)R_{x''}(\gamma) = \begin{pmatrix} c_\alpha c_\beta & c_\alpha s_\beta s_\gamma - s_\alpha c_\gamma & c_\alpha s_\beta c_\gamma + s_\alpha s_\gamma \\ s_\alpha c_\beta & s_\alpha s_\beta s_\gamma + c_\alpha c_\gamma & s_\alpha s_\beta c_\gamma - c_\alpha c_\gamma \\ -s_\beta & c_\beta s_\gamma & c_\beta c_\gamma \end{pmatrix} \tag{44}$$

To calculate the parameters from any RPY rotation matrix we use:

$$\alpha = atan_2(r_{21}, r_{11}), \beta = atan_2(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2}), \gamma = atan_2(r_{32}, -r_{33}) \tag{45}$$

Gimbal lock

If we create a Euler rotation matrix with $\beta = 0$ (or any multiple of π), the rotation order devolves from: zyz, to zz, since the y rotation matrix becomes the identity matrix. While this does not matter when we're just trying to rotate a frame, It does matter if we want to retrieve the parameters from this rotation matrix we will fail. The parameters could be any number of infinite combinations. Gimbal lock is more serious when we're dealing with real devices such as gyroscopes, in which case the rotation axes can physically become coupled and unable to be separated.

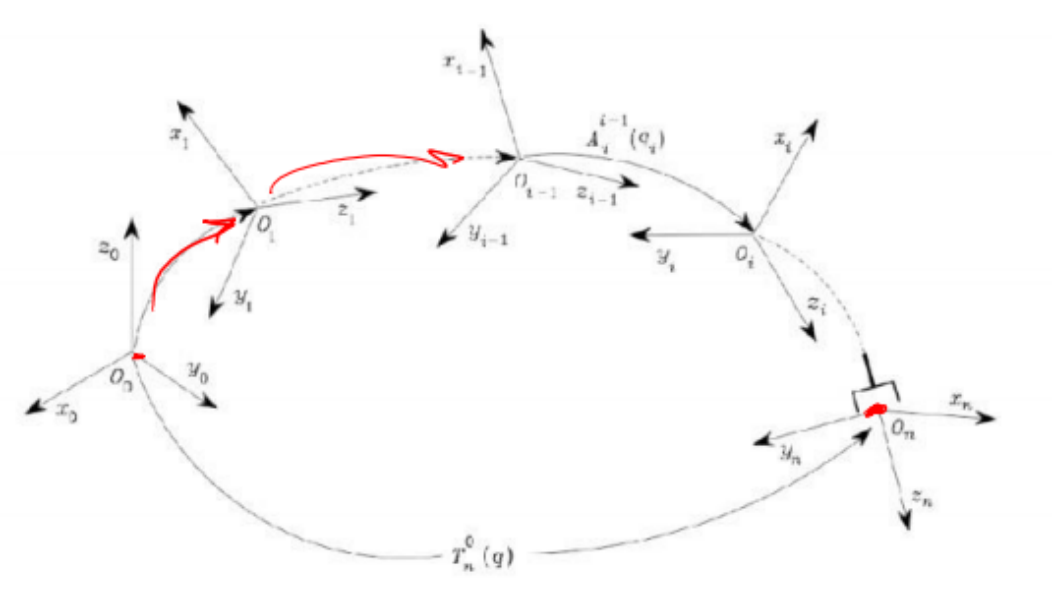


Forward kinematics

Forward Kinematics function

Forward kinematics is a way of computing the end-effector position from the parameters of the robot's actuators. It is as simple as chaining the transformations between each frame in the robot's joints, and expressing the position of the end effector in the ground frame:

$$\mathbf{x} = A_{g \to l_0}(\mathbf{q}) A_{l_1 \to l_2}(\mathbf{q}) \dots A_{l_n \to e}(\mathbf{q}) = k(\mathbf{q}) \tag{46}$$



Denavit-Hartenberg convention (D-H)

The amount of ways in which we can pick our frames at each joint is way too big, and using this many parameters is wasteful. D-H proposes a way of deriving frames and transformations between them with only 4 parameters:

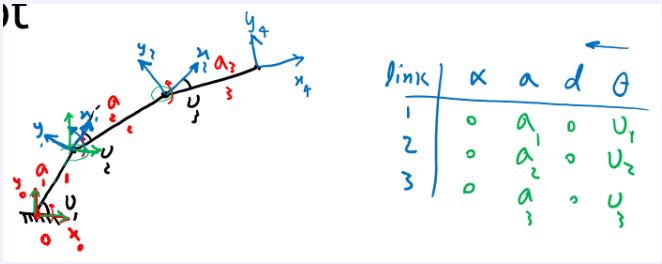
- Place the z-axis in the direction of motion for linear joints and perpendicular to the rotation direction for revolute joints
- Select the x-axis so that it is perpendicular to both the current and previous z-axis, and such that it intersects the previous z-axis (the distance along this axis between the z-axis is called the common normal)
- Select the origin of each joint's frame to be at the intersection of its x and z axes
- Complete the right-handed coordinate frame with y

Once we select our frames in this way we can create transformations between each frame pair using the following operation order:

$$R_{z,\theta_i}T_{z,d_i}T_{x,a_i}R_{x,\alpha_i} = \begin{pmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{47}$$

the parameters at each frame i (apart from the ground frame) represent:

- θ_i: the angle (including rotation of the joint) about z_{i-1} between the previous x_{i-1} and x_i
- d_i: the distance along the previous z_{i-1} to the common normal (think up/down) i.e. the next x_i axis
- α_i: the angle about the common normal/ new x_i axis from the old z_{i-1} to new z_i axis
- a_i: the length of the common normal



Velocity of end-effector

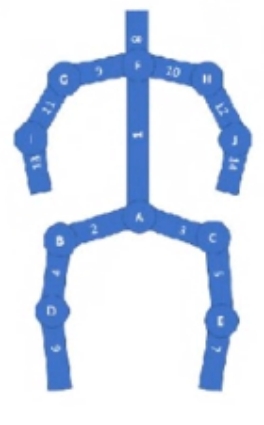
Given the change in $\mathbf{q}$ can we find out the resulting velocity of the end-effector $\dot{\mathbf{x}}$? Yes, using the jacobian:

$$\dot{\mathbf{x}}_e = \frac{\partial k(\mathbf{q})}{\partial \mathbf{q}}\dot{\mathbf{q}} = J(\mathbf{q})\dot{\mathbf{q}} \tag{48}$$

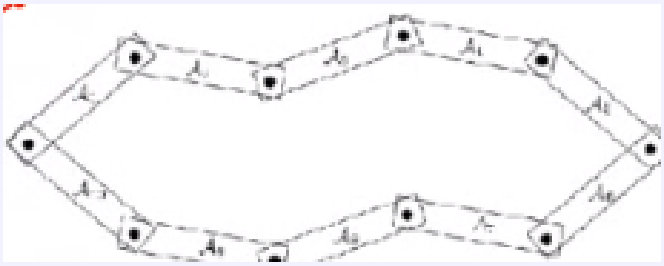
$$J(\mathbf{q}) = \frac{\partial k(\mathbf{q})}{\partial \mathbf{q}} = \begin{pmatrix} \frac{\partial k_1(\mathbf{q})}{\partial q_1} & \frac{\partial k_1(\mathbf{q})}{\partial q_2} & \cdots & \frac{\partial k_1(\mathbf{q})}{\partial q_m} \\ \frac{\partial k_2(\mathbf{q})}{\partial q_1} & \frac{\partial k_2(\mathbf{q})}{\partial q_2} & \cdots & \frac{\partial k_2(\mathbf{q})}{\partial q_m} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial k_n(\mathbf{q})}{\partial q_1} & \frac{\partial k_n(\mathbf{q})}{\partial q_2} & \cdots & \frac{\partial k_n(\mathbf{q})}{\partial q_m} \end{pmatrix} \tag{49}$$

Open vs Closed chains

in an open kinematics chain, each joint may rotate or translate indepdently:



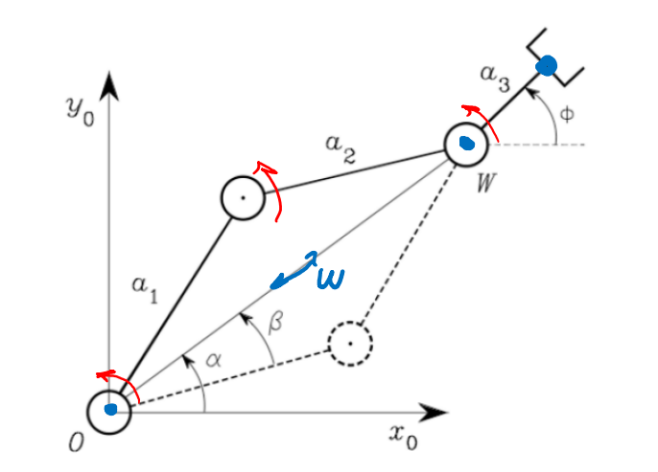
In a closed kinematics chain, joint motion may be coupled. Rotating or translating one joint may force another to to rotate or translate.



Motion representation

Many ways to skin a cat

We have a way of solving for the end-effector position using the geometry of a robot, but how do we do the reverse and find the angles required to put the end-effector in any position we desire ?

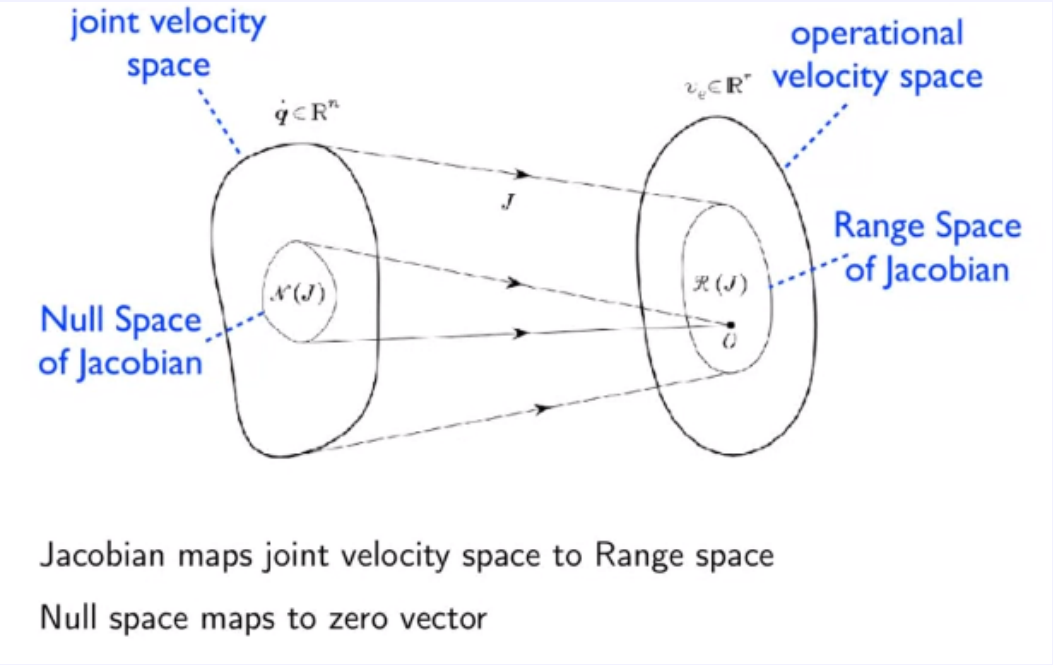


For the robot above:

- No configuration reaching $\mathbf{x}$ with $|\mathbf{x}| > a_1 + a_2 + a_3$
- 1 configuration reaching $\mathbf{x}$ with $|\mathbf{x}| = a_1 + a_2 + a_3$
- 2 configurations reaching $\mathbf{x}$ with $|\mathbf{x}| < a_1 + a_2 + a_3$

Jacobian mapping

The jacobian can be thought of as a mapping of joint velocity to end-effector velocity. The null-space of the jacobian are all the joint velocities which **do not move the end-effector**



Numerical solution

Given $\mathbf{x_d}$ the desired position of end-effector, and $\mathbf{x}$ the current position of end effector, we can use the inverse of our Jacobian to find the velocity (or change in) the parameters $\mathbf{\Delta q}$ which always brings us closer to the desired configuration!

$$\begin{aligned} \dot{\mathbf{x}} &= J\dot{\mathbf{q}} \\ \mathbf{\Delta q} &= \int_{\mathbf{x}}^{\mathbf{x}_d} J^{-1}\dot{\mathbf{x}}\ \mathrm{dt} \\ \mathbf{\Delta q} &= J^{-1}\mathbf{x}\big|_{\mathbf{x}}^{\mathbf{x_d}} \\ \mathbf{\Delta q} &= J^{-1}(\mathbf{x_d} - \mathbf{x}) \end{aligned} \tag{50}$$

Notice how this means the resultant velocity at each joint is just the sum of all joint velocities at each intermediate position! When calculating this in real scenarios, we can simply perform differentiation/integration numerically (as opposed to analytically), and take the resulting velocity of joints at each step each frame as the small change to q which brings us closer to the goal:

$$\mathbf{q} = \mathbf{\Delta q} + \mathbf{q}_0 = J^{-1}\left(\mathbf{x_d} - \mathbf{x}\right) + \mathbf{q_0} \approx kJ^{-1}\left(\mathbf{x_d} - \mathbf{x}\right) + \mathbf{q_0} \tag{51}$$

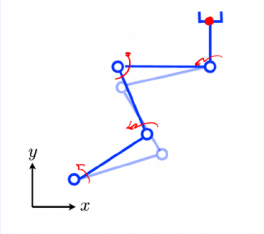
where k is the size of the step we want to take at each iteration.
Note: this does not really work well in practice because the measurements of q can be very noisy

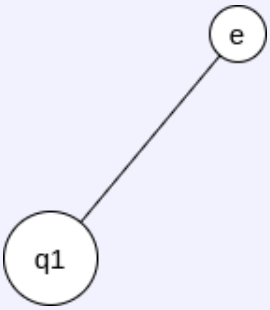
Non-invertible jacobians

There are many reasons for which we might not be able to find the inverse of the Jacobian J^{-1}

Under-actuated or Redundant robot

Notice that as soon as the robot has less or more parameters than dimensions in its work-space, the jacobian is no longer square $(x \times q)$





Singularitiies

Whenever two of the jacobian's columns or rows become linearly dependent, it loses rank and therefore its determinant drops to zero, preventing us from finding the inverse.
consider the following jacobian:

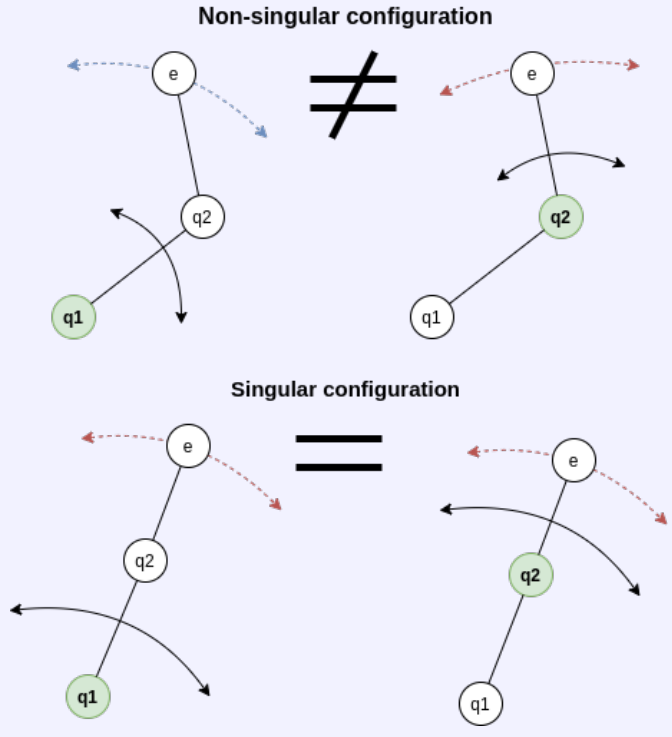
$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix} \tag{52}$$

What does this configuration mean ? Have a look at what happens when we try to apply velocities to all our joints:

$$J \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad J \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \tag{53}$$

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 2 \\ 2 \end{bmatrix} \tag{54}$$

both joints result in velocities in the same **direction**, but with different magnitudes. Assuming the joints can act in different directions, this configuration causes us to lose a degree of freedom! Such positions occur whenever joints align, or extend fully. A similar situation is represented in the figure below:



While we technically still can move the joints in such a way as to reach any position in the work space, the Jacobian tells us a different story! That's because at that instant of time, any movement of our joints results in the same direction of movement, should we somehow leave this configuration the next iteration, the jacobian will be back to normal!

Inverse Kinematics

Pseudo-inverse

How can we then in general find the angles which position the end-effector wherever we desire?
A general solution to the non-invertibility problem (in the case of redundancy) is the use of a **pseudo-inverse**. pseudo-inverses deal with tall-matrices, i.e. systems of equations which have an infinite number of solutions. For example the Moore-Penrose pseudo-inverse, finds a solution which minimizes the distance $|Ax - b|$ for the system $Ax = b$
There are two possible inverses which yield two different solutions in our case:

$$J^+ = (J^T J)^{-1} J^T = J^T (J J^T)^{-1} \tag{55}$$

Properties

$$\begin{aligned} JJ^+J &= J \\ J^+JJ^+ &= J^J \\ (JJ^+)^T &= JJ^+ \\ (J^+J)^T &= J^+J \end{aligned} \tag{56}$$

- As the determinant of J gets closer to zero (as J grows closer to a **singularity**), the pseudo-inverse J^+'s elements grow towards infinity!

Redundancy-proof IK methods

There are many methods to perform Inverse Kinematics (*None of these deal with singularities! only the redundancy!*):

Pseudo-inverse method

$$\dot{\mathbf{q}} = J^+ \dot{\mathbf{x}} \tag{57}$$

Weighted Inverse Kinematics

Result of minimising the expression (solving IK the constraints of our robot while minimising $|\dot{\mathbf{x}} - J\dot{\mathbf{q}}|$):

$$g(\dot{\mathbf{q}}) = \frac{1}{2}\dot{\mathbf{q}}^T W \dot{\mathbf{q}} + \lambda^T (\dot{\mathbf{x}} - J\dot{\mathbf{q}}) \tag{58}$$

Where W is a positive definite matrix (i.e. the mass distribution of the arm, in which case this minimizes the kinetic energy of the system)
Using lagrange-multipliers, results in:

$$\dot{\mathbf{q}} = W^{-1} J^T (J W^{-1} J^T)^{-1} \dot{\mathbf{x}} \tag{59}$$

Dealing with singularities

In order to avoid singularities we need to employ additional goals and methods!

Pseudo inverse + secondary task

We introduce a secondary task, we can describe the "general solution" as follows: "maximise $\dot{\mathbf{q}}_\mathbf{0}$, while trying to hit the desired position"

$$\dot{\mathbf{q}} = J^+\dot{\mathbf{x}} + (\mathbf{I} - J^+J)\dot{\mathbf{q}}_\mathbf{0} \tag{60}$$

We define $\dot{\mathbf{q}}_\mathbf{0}$ as follows:

$$\dot{\mathbf{q}}_\mathbf{0} = k_0\left(\frac{\partial w(\mathbf{q})}{\partial \mathbf{q}}\right)^T = k_0\begin{bmatrix}\frac{\partial w}{\partial q_1}\\ \vdots \\ \frac{\partial w}{\partial q_n}\end{bmatrix} \tag{61}$$

i.e. the derivative of some function w of the joint values with respect to the joint values. Each element of the derivative may be numerically estimated as: $\frac{\Delta w}{\Delta q}$

To avoid singularities we may define w as :

$$w(\mathbf{q}) = \sqrt{det(J(\mathbf{q})J^T(\mathbf{q}))} \tag{62}$$

i.e. we try to maximise the determinant of J!

Damped pseudo inverse

Instead of J^+ we can use the damped pseudo inverse:

$$J^\star = J^T(JJ^T + k^2\mathbf{I})^{-1} \tag{63}$$

with k being the damping factor. This pseudo-inverse will limit the size of entries (mostly near-singularities, while farther from singularities, this has little effect) in the pseudo-matrix in effect "damping" the rise to infinity and normalizing the behaviour of our robot.

"Spring" method

We can attach a fictitious spring at the end-effector with the other end connected to the target position. More correctly we can move in the direction of the force a spring would exert if it were there.

$$\Delta q = J^T K(\mathbf{x}_d - \mathbf{x}_e) \tag{64}$$

This works because the transpose of the Jacobian actually maps forces at the end effector to equivalent forces in the joints required for equivalent work done (explained in the dynamics section). We simply take those torques, and use their directions (we can take the magnitudes in proportion as we like)
Note:This method requires that the target is stationary!

Dynamics & Statics

Statics

In statics we consider systems at rest, where all the forces are balanced - under **static equilibrium**. We use the idea of **virtual displacement** and **virtual forces** to examine the system and work out equivalences between forces! Under static equilibrium, the **work done** by all forces is zero (because there is no motion at all). Since no work is done we can use the idea of infinitesimal virtual displacements to quantify the net zero work done by each force. This is analogous to hanging a mass off a scale, waiting till the system is at rest and taking a reading for the weight of the mass, we can also artificially pull down on the mass to see what happens to the weight - except in this case no real forces are changed, and the whole system is in a freeze-frame, i.e. no time passes!

Work

The work done is basically the energy change in a system caused by an application of a force. A change of energy happens only if something moves!
The work done is defined as:

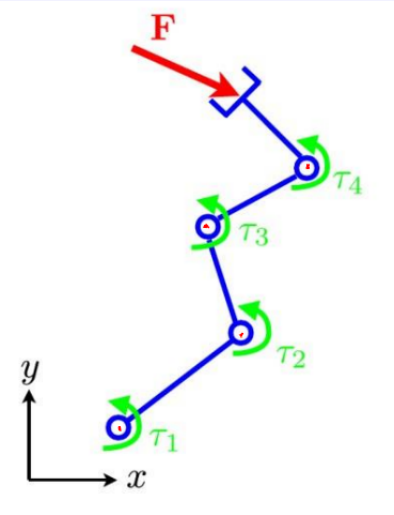
$$w = Fs \tag{65}$$

where w is the work done (Joules or Newton-meters), F is the force (Newtons), and s is the displacement (meters)
If a force only affects the kinetic energy of an object (i.e. no heat loss and no increase in potential energy) the work done is related to the change in kinetic energy $\left(\frac{1}{2}mv^2\right)$:

$$w = \Delta E_k = \frac{1}{2}mv_1^2 - \frac{1}{2}mv_0^2 \tag{66}$$

Torque at joints required to balance a force

Have a look at the situation below:



Imagine a force is to be applied to the end-effector of a robot, some kid is pushing a toy robot-arm at the end-effector and we want to counter-act the force how do we do that with statics ? Well, first of all we need a static equilibrium, assume that all the forces are balanced. Which forces are involved ?
We have the force F acting at the end-effector, and all the torques r_1 through r_4 acting on the motors.
Essentially what we need all the torques combined to equal the force at the end effector but just in the other direction - let's ignore the direction for a moment.
We introduce a virtual displacement due to the force, which causes an equivalent virtual displacement of the joint angles, these two are related geometrically, and this relationship is:

$$\delta\mathbf{x} = J\delta\mathbf{q} \tag{67}$$

This arbitrary displacement is a theoretical tool we use to work out constraints and equivalences. Since everything is at equilibrium then no work is done, as well as this, no **virtual work** is done either:

$$\delta w = 0 = \mathbf{r}^T\delta\mathbf{q} - \mathbf{F}^T\delta\mathbf{x} = \mathbf{r}^T\delta\mathbf{q} - \mathbf{F}^T J\delta\mathbf{q} = \delta\mathbf{q}^T(\mathbf{r} - J^T\mathbf{F}) \tag{68}$$

From this, since the virtual displacement δq is non-zero (but infinitesimal), we must have that:

$$\begin{aligned} \mathbf{r} - J^T\mathbf{F} &= 0 \\ \mathbf{r} &= J^T\mathbf{F} \end{aligned} \tag{69}$$

i.e. to counter-act the force we would need to apply a torque of $J^T\mathbf{F}$ in the opposite direction! This equivalence works only in the absence of gravity and friction, but is still quite neat.

Forward Dynamics

Forward dynamics is the function relating the state of the robot as well as the forces acting on it, to the reactive accelerations generated on the robot!
The general equation of motion for all rigid-body robots(manipulators specifically) is:

$$\ddot{\mathbf{q}} = I^{-1}(\mathbf{q})(-B(\mathbf{q})[\dot{\mathbf{q}}\dot{\mathbf{q}}] - C(\mathbf{q})[\dot{\mathbf{q}}^2] - G(\mathbf{q}) + \mathbf{r}) \tag{70}$$

Where:

- I: Inertia matrix
- B: Coriolis matrix
- C: Centrifugal matrix
- G: Gravity matrix
- $[\dot{\mathbf{q}}\dot{\mathbf{q}}] = (\dot{q_1}\dot{q_2}, \dot{q_1}\dot{q_3} \ \dots \ \dot{q}_{n-1}\dot{q_n})^T$
- $[\dot{\mathbf{q}}^2] = \left(\dot{q_1}^2, \dot{q_2}^2, \dots, \dot{q_n}^2\right)^T$

Inverse Dynamics

The inverse dynamics, specifies the forces we need to apply (e.g. torques) to generate desired state and motion. Unlike with kinematics, the inverse dynamics equation is very easy to find (because $I(\mathbf{q}$ is always positive semi-definite!)):

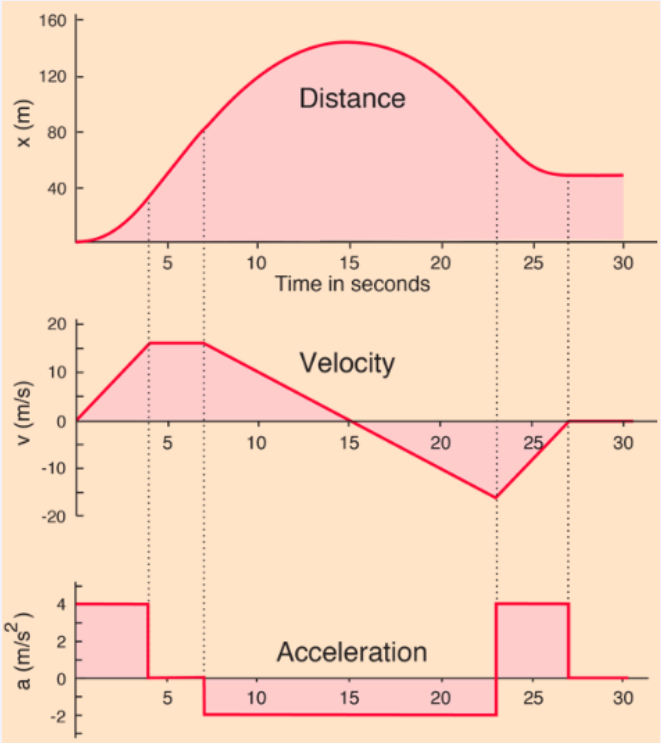
$$I(\mathbf{q})\ddot{\mathbf{q}} + B(\mathbf{q})[\dot{\mathbf{q}}\dot{\mathbf{q}}] + C(\mathbf{q})[\dot{\mathbf{q}}^2] + G(\mathbf{q}) = \mathbf{r} \tag{71}$$

Simulating motion

Now these equations above get extremely complicated, they are differential equations of second order, how do we put them to practice ?
If we are interested in simulating how a manipulator will behave under certain forces ,we can use the forward dynamics equation, which gives us an expression for: $\ddot{\mathbf{q}}(t)$ We can integrate this (using Euler's method) to get: (**the classic equations of motion**)

$$\begin{aligned}\dot{\mathbf{q}}(t) &= \dot{\mathbf{q}}(t-\Delta t) + \ddot{\mathbf{q}}(t-\Delta t)\Delta t \\ \mathbf{q}(t) &= \mathbf{q}(t-\Delta t) + \dot{\mathbf{q}}(t-\Delta t)\Delta t + \frac{1}{2}\ddot{\mathbf{q}}(t-\Delta t)^2\end{aligned} \tag{72}$$

Think about it, if we divide time into discrete time steps, and at step t we know the previous velocity we were traveling at, as well as the way we accelerated last time, we can calculate the velocity at the new time step by retaining the velocity and adding ontop of that the result of the acceleration! A similar case goes for our position, only this time we also include the effect of translation due to the previous velocity as well!



Also try to internalize that:

- The velocity of an object is the derivative of its displacement with respect to time: $v = \frac{ds}{dt}s$, i.e. change in distance over time
- The acceleration of an object is the derivative of its velocity with respect to time: $a = \frac{dv}{dt}v$, i.e. change in speed over time
- The opposite relationships hold, i.e. the displacement, is the integral of velocity over time, and velocity is the integral of acceleration over time

Note: this is exactly how physics in game engine's/simulations works(maybe not the robot manipulator part, but the integration of acceleration part)