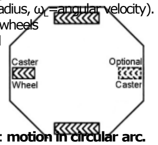
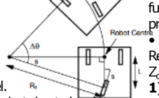
1) Intro
Robot: A physically-embodied, artificially intelligent device with sensing and actuation. It can sense, and act, and thinks to connect sensing with action.
A Washing Machine can be argued to be a robot rather than an **appliance** as it has sensing, actuation and processing. Can be argued as an appliance as the workspace is inside the device, and robots typically need be smarter (react to changing things rather than set environment).
Examples: Robot Arms (factories), rescue, delivery robots, roombas, farming. Home environments can be equally challenging to the outdoors.

2) Robot Movement
Consider: movement control, obstacle avoidance, localisation, mapping, path mapping. Ideally our robots are independent, untethered and self-contained
Levels of Autonomy:
1. Teleoperation (Remotely-Operated Vehicles)
2. Semi-autonomous/Supervised (e.g. Mars rovers, driver assist systems).
3. Fully autonomous (roombas, autonomous cars (?))

We focus on wheeled robots on flat surfaces:
World Frame: 3D coordinate space anchored on env.
Robot Frame: 2D coordinate space relative to robot.
1) Degrees of Motion Freedom (DOF):
• A rigid body which translates and rotates along 1D path has 1 translational DOF. e.g: a train.
• A rigid body which translates and rotates on a 2D plane has 3 DOF: 2 translational (x, y for loc), 1 rotational (rotate clock/anticlockwise) e.g: our robot.
• A rigid body which translates and rotates in a 3D volume has 6 DOF: 3 translational (we need an x, y, z param to represent loc), 3 rotational (we can rotate up in z - ie plane lifting nose, rotate around in 2d ie changing direction of plane, rotate orientation ie: turn to fly upside down).
Example: a flying robot.
• Holonomic robots move instantaneously in any direction in the space of its DOF - ie the train example. • Otherwise, a robot is called non-holonomic. Most are non-holonomic - we need to orient / turn ourselves to move in a direction / we can't just translate that way. • Example: a car only has two types of input -> speed and angle. Obviously not holonomic because we have 2 inputs only vs 3 degrees of freedom.
2) The Movement of Differential Drive Robots:
Define wheel velocities V_L & V_R (linear velocities over ground). $V_L = r_L\omega_L$ (r = wheel radius, ~~ω = angular velocity~~). W = distance between the two wheels



• Two driving wheels on left and right with individual motors.
• Steering is done by setting different wheel speeds
• Straight line motion: $V_L = V_R$
• Turn on the spot: $V_L = -V_R$
• Other combinations of speeds: **motion in circular arc.**
Equations the curved path of the robot:
$R = W(V_R+V_L) / 2(V_R - V_L)$ $\Delta\theta = (V_R - V_L)\Delta t / W$
AKA: supply the velocities, the measured W, and time, and we get the radius of the circle / angle change.
3) Circular Path of Car-Like Tricycle Robots:
Two front free running wheels. Has a single steerable and drivable wheel at the back. With no wheel slip:
$R = L / \tan s$, $\Delta\theta = v\Delta t \sin s / L$.
Speed: $v = r_w\omega$



Where: L = distance between the central point between the front wheels, and the back wheel;
s = angle of turn of the back wheel.
We could measure all these values, but due to hard to model factors (surface slip, tyre softness) calibrating these things via experiments is better (to work out the constant scaling between the motor reference angle & distance).

2.1) Robot Motion in Practice
Gears and Encoders turn high angular velocity/low torque motors into low rotation/high torque forces to the wheel.
Pulse Width Modulation: set a power level to send to the motor. We'll pass voltage with a fixed amp but with the amount of "fill-in" set by PWMer. The actual velocity will depend on a number of factors.
Feedback / Servo Control:
1) Measure what the robot is doing
2) compare it to what we want to do
3) record and try to minimize the difference (error) by adjusting the power supply

The motors we have record rotational motor position in degrees. **Modes**: 1) position control (demand is a constant) 2) velocity control (where demand increases linearly with time). Pulse modulation just supplies energy on and off so that the total proportion is the % we need.
PID (Proportional or Differential) Control:
• Error: e(t) = demand - actual position
• PID Expression: sets power as a function of error:
$$P(t) = k_p e(t) + k_i \int_{t_n}^{t} e(\tau)d\tau + k_d \frac{de(t)}{dt}$$
•k_p, k_i, k_d are gain constants which we can tune.
• k_p is the main term: high values give rapid response but we possibly oscillate. • k_i is integral term – reduces steady state error. • k_d – differential term: reduces settling time

Robot moving on 2D plane has a location by a state vec of 3 Parameters: x = y = θ = 0. $-\pi < \theta <= \pi$.
During a straight line period of motion of distance D:
$(x^{new}, y^{new}, \theta^{new})^T = (x + D\cos\theta, y + D\sin\theta, \theta)^T$
During a pure rotation of angle α:
$(x^{new}, y^{new}, \theta^{new})^T = (x, y, \theta+\alpha)^T$
In the general case of movement on an angle:
$$(x^{new}, y^{new}, \theta^{new}) = \begin{bmatrix} x+R(\sin(\Delta\theta+\theta)-\sin\theta)\\ y-R(\cos(\Delta\theta+\theta)-\cos\theta)\\ \theta+\Delta\theta\end{bmatrix}$$

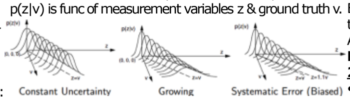
2.1.1) Position Based Planning:
Allows movement through a series of predefined waypoints
• Robot movements are composed of straight line moves and turns on the spot. Minimises total distance moved.
• Assume that the robot's current pose is (x, y, θ) and the next waypoint is to (W_x, W_y).
1) First rotate to point to the waypoint: direction vector to point in is: $(d_x\ d_y) = [W_x - x]$ $[W_y - y]$
Thus the absolute orientation α is: $\alpha = \tan^{-1} d_y / d_x$
We must take care to ensure α is in the correct quadrant of $-\pi < \alpha \le \pi$, when $\tan^{-1}$ is in the range $-\pi/2 < \alpha \le \pi/2$. This can be achieved with atan2(dy, dx).
2) Thus the angle the robot must rotate through is therefore $\beta = \alpha - \theta$. For efficiency – pick the lesser angle by add/subbing 2π to have $-\pi < \beta \le \pi$.
3) $d = \sqrt{d_x^2 + d_y^2}$

2.1.2) Local Planning: (Dynamic Window Approach for Differential Drive Robots)
1) Consider all possible movements within time dt (we can set V_L, V_R between 0 and max velocity). 9 possible actions: each of V_L, or V_R can go up, down or stay same. Max acceleration * dt (time step) is the max change we can make.
2) For each of them, look ahead some extra time τ and predict where we end up using the motion eqns frm before
3) Calculate the **benefit** (B) and **cost** (C) of these motions: We define W_C & W_B ourselves (how much we value being close to the target / an obstacle). Target is at pos (T_x, T_y)
B = $W_B * D_f$, D_f is the amount we got closer to the target
$$D_f = \sqrt{(T_x - x)^2 + (T_y - y)^2} - \sqrt{(T_x - x_{new})^2 + (T_y - y_{new})^2}$$
C = $W_C * C_f$, C_f is distance to closest obstacle at (O_x, O_y)
$$C_f = D_{safe} - \left(\sqrt{(O_x - x_{new})^2 + (O_y - y_{new})^2} - r_{robot} - r_{obstacle}\right)$$
r_{robot} and $r_{obstacle}$ are the robot and obstacle radii. (O_x, O_y) are found by searching through the obstacles. (if we know where they are - a sensor should detect them)
We choose the path with maximum B – C and follow that path for dt time.

2.1.3) Global Planning (Wavefront Method)
• Brute force 'flood fill' breadth first search of whole environment. Finds the shortest route, but slow.
• **Rapidly Exploring Randomised Trees (RRT) Method**: Algorithm grows a tree of connected nodes by randomly sampling points and extending the tree a short step from the closest node. Expands rapidly into new areas, but without the same guarantees.

3) Sensors
Sensors are either proprioceptive or exteroceptive.
• **Proprioceptive:** self-sensing; e.g: motor encoders and internal force sensors – improve a robot's sense of internal state and thus can improve motion. The reading is simply a function of the state of the robot: $Z_p = Z_p(x)$. Can rely on previous states or rates of change of state too.
• **Exteroceptive:** (monitor the outward environment). Readings depend on robot state and the world around us: $Z_o = Z_o(x, y)$. We parameterise world state: e.g: coord grid.
1) Touch Sensors: Binary on/off. No processing – switch open/closed means current flows/doesn't. Single valued.
2) Light Sensors: Detect intensity of light from a single forward direction, with some angular sensitivity. Multiple sensors in diff directions can be used to guide steering.
Lego Sensors can also emit light for it to reflect off close targets, for short-range obstacle detection and avoidance.
3) Sonar: Measures depth with ultrasonic time pulse and measuring time to return. Usual angular width: 10-20deg. Accurate (to the cm) but can be noisy in presence of complex shapes. Max range: few meters. Ring of sonars can do obstacle detection & avoidance.
4) Laser Range Finders: measures depth. Lidar sensors return an array of depth measurements from a scanning beam. Submillimetre accuracy, works on most surfaces. Scans in 2D plane, but can get 3D ones. Bulky, expensive.
5) External Sensors (vision/cameras): Generalises light sensors. Returns a large rectangular array of measurements. It only measures light intensity, from just one image we can't tell if objects are "small/close" or "large/far away". We can use cameras as planar sensors if we have extra scene knowledge / we know we're observing the ground plane. More on this in section 5.
3.1) Reacting on Sensor Results:
1) Collision Handling: on collision, reverse and turn a fixed angle to find a new direction to navigate in. We can also randomise the angle.
2) Servoing: Technique where control params are tied to sensor readings – negative feedback loop used to update them. High update freq.

Examples of Servoing in action:
1) Proportional Control using an External Sensor:
Set demand proportional to neg error (diff between desired sensor and true sensor value): e.g: velocity:
$v = -k_p(z_{desired} - z_{actual})$
Where k_p is a proportional gain constant found via tuning.
A more general case of PID Control
2) Visual Servoing to Control Steering:
For a robot with tricycle / car-wheel configuration.
$s = k_p\alpha$ (where s is the angle of turn of the back wheel, and alpha is the angle bw front midpoint & obstacle centre) will guide the robot to collide with the obstacle.
$s = k_p(\alpha - \sin^{-1}R/D)$ will avoid the obstacle who has radius R, with a distance D from its center.
3) Wall following with Sonar:
• Use a sideways looking sonar to find distance z to wall
• Use velocity control and loop at e.g: 20Hz
• To maintain distance d from the wall: $V_R - V_L = K_p(z - d)$.
Symmetricity can be done use a constant offset V_C:
$V_R = V_C + ½ K_p(z - d)$ $V_L = V_C - ½ K_p(z - d)$
Problem: If our angle to the wall gets too large, the sonar doesn't accurately measure distance anymore. **Solutions**: 1) ring of sonars & combine measurements. OR 2) Mount the sonar in front of wheels; couples wall rotation & dist.
Probabilistic Sensor Modelling:
• Sensing is uncertain. Readings are perturbed.
• Having callibrated a sensor (find reading uncertainty) we can build a probabilistic model. This is a probability dist (**likelihood function**) of form: $p(z_o|x, y)$. Gaussian.
• This likelihood function fully describes sensor performance p(z|v) is func of measurement variables z & ground truth v.



For simple algorithms like servoing, we need to preprocess raw sensor readings to make them useful:
1. **Temporal Filtering**: Smooth/median of last few sensor readings of single reading sensors (sonar). Elims outliers.
2. **Geometric Filtering / Feature Detection:** On sensors that report array of measurements; fit geometric shapes (lines/corners) to the data, and output the params of those shapes rather than the measurements (LSM?)

Combining: Sensing and Action Loops:
• We need to combine the data from multiple sensors, and process them to decide how to respond
• World model approach: capture data (and manipulate it), plan actions to achieve goal, exec plan, if world changes during exec, stop and replan. Powerful, but expensive.

4) Probabilistic Robotics:
• **Systematic Error** - constant error term after all experiments. Can be calibrated for.
• **Zero Mean Errors** - spread around point when removing systematic error. Can't be calibrated - due to random noise. Error Distribution in the world frame will grow as the robot moves further around the square.
• We can model the zero mean errors probabilistically: in many cases a Gaussian is suitable.
1) During a **straight-line period of motion** of distance D:
$(x_{new}, y_{new}, \theta_{new})^T = (x + (D+e)\cos\theta, y+(D+e)\sin\theta, \theta+f)^T$
2) During a pure rotation of angle α:
$(x_{new}, y_{new}, \theta_{new})^T = (x, y, \theta+\alpha+g)^T$
e, f and g are uncertainty terms, with zero mean and a gaussian distribution. Models the motion; stddev is found experimentally via calibration. Remember, variance sums.

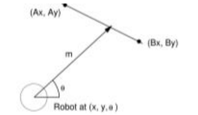
4.1) Probabilistic Inference
Sensing -> Action procedures are locally effective but limited. Complicated problems require longer-term representations and consistent scene models.
Solution: Probabilistic approaches acknowledge uncertainty and uses models to abstract useful information from data. Builds a best estimate of the situation. Goal: incrementally updated probabilistic estimate robot position on the map.
Uncertainty: Every robot action & reading is uncertain. If we combine actions & readings to make state estimates it will be uncertain.
For probabilistic inference:
• We want to find our state, and the world state.
• A weighted combination of prior knowledge with new measurements, as a Bayesian Network.
• **Sensor Fusion**: combine data from many sources.
• This composite state estimate decides next robot action.
Bayesian Probabilistic Inference:
P(XZ)=P(Z|X)P(X)=P(X|Z)P(Z)➔P(X|Z) = P(Z|X)P(X)/P(Z)
P(X): prior, P(X|Z): posterior, P(Z): marginal likelihood.
This is used to take in new info about the robot state given new info, in order to construct the new state (next prior).
Ways of representing the Probability Dist P(X)
1) PDFs can be discretized to representthem in the system, with n bins and a value for each bin Higher res, expensive, and n->inf tends towards the continuous function.
Area under PDF p(x) from a to b: $\int_a^b p(x)dx$
2) Using a Gaussian Distribution: $p(x) = \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
A wide prior multiplied by a likelihood curve produces a tighter posterior. Product of two Gaussians is Gaussian.

We can represent probability distributions in terms of weighted particles!Weight sum = 1. Simple, represents many dists well (including multi-modal). Poor performance with low num particles, but high num is costly.

4.2) Probabilistic Localisation:
• We have a map of the environment in advance
• We want to become certain about the robot position
• We store and update a probability (particle) distribution representing our position estimate.
We care about the 2D case.
A particle p_i is a point estimate = $(x_i\ y_i, \theta_i, w_i)$.
Normalised: weight sum of the N particles (e.g 100) = 1

4.2.1) Monte Carlo Localisation:
• Can be thought off as a Bayesian Probabilistic Filter or a Genetic Algorithm where bad particles are penalised.
• Cloud of particles represents uncertain robot state.
• Robot Pos = Individual sum of vec elems * weightSteps:
  • This can be used for **waypoint navigation**
• MCL can solve both tracking/kidnapped robot problems.
  • For tracking (Continous Localisation) – we assume the init the particles at a known position. Updates: given a good estimate of where we were, using reading est new pos
  • For kidnapped robot (Global Localisation) - init particles randomly. Needs many sonars to be done accurately.

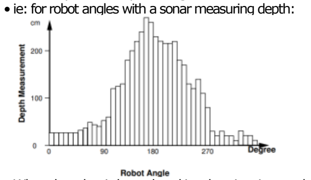
MCL Steps:
1) Motion Prediction based on Proprioceptive Sensors:
For each particle p_i:
• Use the uncertain movement update formulae from 2.1
• Using a gaussian with stddevs e, f, g perturb particles.
Variance is proportional to amount moved as it is additive. Better to overestimate than underestimate, but don't go too high or we'll oscillate!
Angles can wrap around past the $-\pi$ to π range. This is OK! **Don't clamp them, or mean will be inaccurate!**
2) State Update based on Outward-Looking Sensors
• Each particle represents a hypothesis to check
• We use Bayes' Rule for updates, given a reading z, and the particle's state X.   P(X|Z) = P(Z|X)P(X)/P(Z)
• With a measurement z, we update the weight like so:
$W_{i(new)} = P(z \mid x_i) * w_i$
The denominator is a constant factor. Doesn't need to be calculated as we'll later remove it via normalisation.



If robot is at pose (x, y, θ) then its forward distance to an infinite wall passing through (A_x, A_y) and (B_x, B_y) is:
$$m = \frac{(B_y - A_y)(A_x - x) - (B_x - A_x)(A_y - y)}{(B_y - A_y)\cos\theta - (B_x - A_x)\sin\theta}$$
But we need to work out which wall we're looking at:
• The world coordinates at which the forward vector from the robot will meet the wall are:
$$\begin{pmatrix} x + m\cos\theta \\ y + m\sin\theta\end{pmatrix}.$$
Using this you can check if the sonar should hit between the endpoint limits of the wall.
• If the sonar would independently hit several of these walls, obviously the closest is the one it will actually respond to.
• Now that we calculate m, we can carry out the measurement update, which should depend on the **difference** z-m (ie: that's what we feed into the Bayesian formula).
• We also require a sensor standard deviation, depending on how uncertain we are on its values.
• We use **robust** likelihood functions to model the fact that sensors can report garbage values by adding small const K.
• $p(z|m) \propto K + e^t$, where $t = -(z-m)^2/(2\sigma_s^2)$
• K stops us killing off particles that are far off the reading as aggressively. As our sensor reading may have just been wrong, and we don't want to wipe out all our good particles after a simple sensor mistake! A ring of sensors would help.
• If the angle β between the sonar and wall normal is wrong (exceeds some known angle c after which the sonar performs poorly), we discard the reading. Don't update.
$$\beta = \cos^{-1}\left(\frac{\cos\theta(A_y - B_y) + \sin\theta(B_x - A_x)}{\sqrt{(A_y - B_y)^2 + (B_x - A_x)^2}}\right)$$
3) Normalisation: Calculate the sum of all weights, and then divide each weight by it.
4) Resampling - Biased Roulette Wheel Selection:
• Generate a cumulative prob distribution array.
• Generate N particles (N was the prev num parts we had):
  • Generate a random float between 0 & 1, and select the particle whose cum prob it intersects. Copy this particle into the new set. Can skip normalisation for efficiency.

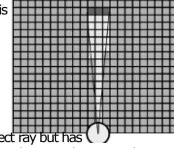
Compass Sensors:
• Enables the robot to estimate position without drift.
• Compass measures bearing β relative to north
• The likelihood $P(\beta|x_i)$ only depends on the θ part of x_i.
• If the compass is correct: $\beta = \gamma - \theta$, where γ is the magnetic bearing of the x coord axis of frame W.
• So we should assess the uncertainty in the compass, and set a likelihood depending on the diff between β & γ, e.g: e^t, where $t = -(\beta-(\gamma-\theta))^2/ 2\sigma_c^2$
• Using a compass and a sonar reduces ambiguity from measurements. Or we can use multiple sonars.

5) Advanced Sensing
5.1) Global Localisation via Place Recognition
• Alternative relocalisation technique: involves making many measurements at chosen locations and learning their characteristics. Doesn't need a prior map but needs training.
• The robot can only recognise the locations it has learned.
• For instance: place the robot at the target location, spin the robot and take a regularly spaced set of sonar readings (e.g one per degree)
• The raw measurements are stored to describe the location: a place descriptor or signature.
• ie: for robot angles with a sonar measuring depth:



• When the robot is later placed in a location, it can take a set of measurements (producing a histogram) and select the stored histogram signature it best matches. Least SM.
Estimating Orientation:
• If the test histogram and one of the saved locations can be brought into close agreement by only a shift, the robot is in the same place but rotated.
• The amount of shift to get the best agreement is a measurement of the rotation.
• To save the computational cost of always trying every shift, we can build a signature which is invariant to robot rotation, such as a histogram of occurrences of certain depth measurements.
• Matching tests can then be carried out on this directly.
• Once the correct location has been found, the shifting procedure to find the robot's orientation need only be carried out for that one location

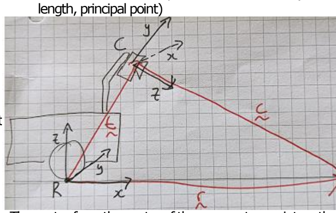
5.2) Probabilistic Occupancy Based Grid Mapping
• We know the robot's location, but we need to infer what parts around the robot are navigable free space, and what parts are obstacles, using a grid representation of world.
• Occ. Grids accumulate uncertain sensor info to solidify towards precise maps
Steps: 1) Define an area on the ground we'd like to map and choose a square grid cell size.
2) For each cell i, we store and update a probability of occupancy $P(O_i)$ that it is occupied by an obstacle.
3) $P(E_i)$ is the corresponding probability that the cell is empty, where $P(O_i) + P(E_i) = 1$.
4) We init the occupancy probabilities for unexplored space to a constant prior value; e.g: 0.5. If we think there's not many obstacles, init lower.
5) Occupancy maps are often visualised with a greyscale value for each cell: from black for $P(O_i) = 1$ to white for $P(O_i) = 0$; intermediate values are shades of grey.
Update:
• We report depth Z = d. This provides evidence that cells around distance d in front of the robot are more likely to be occupied. But also, that cells in front of the robot at depths less than d are more likely to be empty.



• A sonar beam is not a perfect ray but has a width (e.g. 10–15◦ as it spreads out and we can take account of this as shown.
• For each cell, we must test if it lies within the beam given the robot's position. We do not learn anything about cells beyond the beam width or beyond the measured depth.
• The test is done by noting the two vectors:
C: the grid cell that we want to test, and Z, the beam stoppage point. Calculate the dot product between the two, and if its less than beam width/2 we know it's a cell we're interested in. Z =(zcosx, zsinx), x is orientation angle.
C = (gx - x, gy - y) where gx, gy is the vector from the origin to C, x, y is the origin vec.
• For each cell we apply bayes rule:
$$P(O_i|Z) = \frac{P(Z|O_i)P(O_i)}{P(Z)}$$
As in MCL, we could avoid calculating $P(Z)$ by also calculating $P(E_i|Z)$ and normalising since we know $P(O_i|Z) + P(E_i|Z) = 1$.
$$P(E_i|Z) = \frac{P(Z|E_i)P(E_i)}{P(Z)}$$
• if we take the ratio
$$\left(\frac{P(O_i|Z)}{P(E_i|Z)}\right) = \left(\frac{P(Z|O_i)}{P(Z|E_i)}\right) * \left(\frac{P(O_i)}{P(E_i)}\right)$$
We can use the odds notation: $o(A) = \frac{P(A)}{P(\bar{A})}$.
$$o(O_i|Z) = \left(\frac{P(Z|O_i)}{P(Z|E_i)}\right) * o(O_i)$$
Taking logs:
$$\ln o(O_i|Z) = \ln\left(\frac{P(Z|O_i)}{P(Z|E_i)}\right) + \ln o(O_i)$$
So in this form, for each cell we store $\ln o(O_i)$ and update it additively. Cells with probability 0.5 of occupancy will have log odds 0; positive log odds means probability > 0.5; negative log odds means probability < 0.5 Also we normally cap log odds at certain positive and negative limits.

Likelihood Model of a Sensor - Summary
• We need models for the likelihood func for sensor performance. We can consider the ratio of likelihoods we need to update log odds
• Log odds update $U = \ln P(Z|O_i) / P(Z|E_i)$, for each cell
$P(Z|O_i)$: probability of obtaining the sensor value we did given that the cell is occupied; $P(Z|E_i)$ is the probability of obtaining that value given that the cell is empty.
• For cells within the sonar beam but closer than the measured depth Z = d: $P(Z|O_i)/P(Z|E_i)$ is less than 1: we can choose a constant negative value for U.
• For cells within the sonar at around the measured depth Z = d: $P(Z|O_i)\ P(Z|E_i)$ is greater than 1; we can choose a constant positive value for U.
• Note that we are somewhat oversimplifying in occupancy grids in assuming the the probabilities of occupancy for each cell are independent.
• Occupancy maps soon converge. Probability vals must be thresholded so we can make decisions about where we can drive. Large occ maps are mem intensive though.

6) Camera Measurements
• Cameras are projective sensors. Each pixel captures the intensity and colour of light arriving at camera centre from one particular vector direction.
• We don't know geometric info directly from the camera, as depth is unknown (is an object big or just close?)
• If we know the camera is observing the ground plane, then every pixel in image captured corresponds to a specific point on the ground plane.
• This correspondence between points in the ground plane **specified in the robot camera plane,** and pixels in the image is fixed. It depends on:
  • Position and orientation of camera relative to robot.
  • The intrinsic calibration parameters of camera (focal length, principal point)



The vector from the centre of the camera to a point on the ground, expressed in the coordinate frame C of the camera is: $c^C = R^{CR}(r^R - t^R)$.
R^{CR}: is the rot matrix turning robot frame R to cam frame C
t^R: vector from the robot centre to the camera centre
r^R: vector from the robot centre to the point (both expressed in the robot frame).
This makes complete sense! It's just the vector from the camera to robot origin -> vector from robot origin to point; and then just transform that into cam space coordinates!
A perspective camera projects a 3D point at vector c^C to image coordinates (u, v) via the camera calibration matrix $(u\ v\ 1)^T \propto Kc^C$
Homogenous Coordinates: remember Graphics?
(u v 1) -> Homogenous coords, we care about u & v, but the 3D element is for scaling and so we can do 3D vec ops.

6.1) Ground Plane Homography
The projected image coords (u, v) of a point at position r^R is given by: $(u\ v\ 1)^T \propto KR^{CR}(r^R - t^R)$
Same matrix as before, but with K – a scaling camera calibration matrix.
If we know a point is on the ground plane: $r^R = (x^R\ y^R\ 0)^T$ (as obviously, z = 0)
Thus:
$$\mathbf{r}^R - \mathbf{t}^R = \begin{pmatrix} x^R - t_x^R \\ y^R - t_y^R \\ -t_z^R \end{pmatrix} = \mathrm{T}\begin{pmatrix} x^R \\ y^R \\ 1\end{pmatrix}$$
$$\mathrm{T} = \begin{bmatrix} 1 & 0 & -t_x^R \\ 0 & 1 & -t_y^R \\ 0 & 0 & -t_z^R \end{bmatrix}$$
So, we can write $(u\ v\ 1)^T \propto KR^{CR}T(x^R\ \ t^R\ 1)^T$
The three 3×3 matrices K (the camera calibration matrix), R^{CR} (rotation matrix of the orientation of the camera relative to the robot frame), and T (encodes the translation of the camera relative to the robot frame) all have unknown elements. Rather than estimating all of these elements separately, we can roll all of them up into a single matrix, and then just empirically obtain that. Easy! $H = KR^{CR}T$
H is also a 3 × 3 matrix known as the **ground plane homography**. Therefore: $(u\ v\ 1) \propto H(x^R\ \ y^R\ \ 1)$

6.2) Direct Calibration of the Ground Plane Homography
Instead of measuring individual params, we can estimate the homography directly from correspondences.
Mark 4+ points on the ground plane in carefully measured positions. Capture an image, and then record the image coordinates of the points. Obviously,
$$\mathrm{H} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix}$$
From one correspondence (u, v) → (x, y), (ground plane to image plane) we get three equations:
$$\begin{pmatrix} ku \\ kv \\ k \end{pmatrix} = \begin{pmatrix} ax + by + c \\ dx + ey + f \\ gx + hy + 1 \end{pmatrix}$$

From this we can deduce:
(gx + hy + 1)u = ax + by + c
⇒ ax + by + c – uxg – uyh = u
and (gx + hy + 1)v = dx + ey + f
⇒ dx + ey + f – vxg – vyh = v

With 4 corresponding points we can use each one to produce 2 lines of the following system which can be solved for a..h easily using np.linalg.lstsq().
$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -u_1x_1 & -u_1y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -v_1x_1 & -v_1y_1 \\ x_2 & y_2 & 2 & 0 & 0 & 0 & -u_2x_2 & -u_2y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 2 & -v_2x_2 & -v_2y_2 \\ x_3 & y_3 & 3 & 0 & 0 & 0 & -u_3x_3 & -u_3y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 3 & -v_3x_3 & -v_3y_3 \\ x_4 & y_4 & 4 & 0 & 0 & 0 & -u_4x_4 & -u_4y_4 \\ 0 & 0 & 0 & x_4 & y_4 & 4 & -v_4x_4 & -v_4y_4 \end{bmatrix}\begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} = \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \end{bmatrix}$$
Now that we have the homography H we can use its inverse to find ground plane coordinates for an image space point.

However it's only valid for **ground plane points.** So in practice, we can use it for point-like obstacles or marker detection on the ground.
We could experiment with markers in known positions to find the uncertainty in ground plane measurements.
We could also use it for the case a camera observes the ground in front of the robot and the intersection of the ground walls or obstacles. Using ideas from computer vision, we can segment the image into ground plane and obstacle regions. Colour-based segmentation or U-Net NN classifiers.

The region at the bottom of the image closest to the robot, will usually lie in the ground plane. We are interested in the boundary between the ground plane and obstacles. These boundary points lie in the ground plane, so our homography is valid.
By finding the ground plane coordinates of a number of points on the boundary, we turn our camera into a kind of simple laser scanner which can simultaneously measure the distance to a set of points on a wall.

7) Simultaneous Location and Mapping

- Fundamental problem in mobile robotics:

*A body with quantitative sensors moves through a previously unknown, **static environment**, mapping it and calculating its egomotion (localisation).*

- SLAM is needed for truly autonomous robots, when little is known about the environment beforehand (no prior map), when we don't want to use GPS or beacons, and when the robot needs to know where it actually is.
- We build the map incrementally, and localise it with respect to the map that grows and is gradually refined.
- Most SLAM algos make maps of *natural scene features*.
  - Laser/sonar: wall segments, planes, corners, etc.
  - Vision: salient point features, lines, textured surfaces.
- Features should be distinctive and easily recognisable from different viewpoints to enable reliable matching (also called correspondence or data association)
- Thanks to the assumption that the world is static, we can just extend probabilistic estimation to the map features as well (e.g: we store some random inited state, and then we update it as we get measurements until we converge to the truth). In SLAM we store and update a joint distribution over the states of both the robot and the mapped world… and if the data is good enough it just works.
- New features are gradually discovered as the robot explores so the dimension of this joint estimation problem will grow

7.1) Carrying out SLAM

- The robot starts with zero uncertainty. It then measures a feature A with a little uncertainty.
- It then drives forwards, and its uncertainty grows (as to where it thinks it is)
- It then sees and initialises features B and C, they inherit the robot uncertainty + a little more
- The robot then drives back to its starting position, and sees A so it now knows where it is! We also know slightly better where A was. As a result, thanks to comparison of a previous measurement: uncertainty of the robot position shrinks, and the uncertainty of the positions of B and C is updated too to shrink. T
- We remeasure B, and note its location. We also get a better idea of where we are too, so C's uncertainty shrinks too.
- SLAM with Joint Gaussian Uncertainty:
  - The most common and efficient way to represent the high-dimensional probability distributions we need to propagate in SLAM is as a joint Gaussian distribution. Updates can be made via the Extended Kalman Filter.
  - PDF represented with state vector and covariance matrix.

$$\hat{\mathbf{x}} = \begin{pmatrix} \hat{\mathbf{x}}_v \\ \hat{\mathbf{y}}_1 \\ \hat{\mathbf{y}}_2 \\ \vdots \end{pmatrix} , \quad \mathbf{P} = \begin{bmatrix} P_{xx} & P_{xy_1} & P_{xy_2} & \dots \\ P_{y_1x} & P_{y_1y_1} & P_{y_1y_2} & \dots \\ P_{y_2x} & P_{y_2y_1} & P_{y_2y_2} & \dots \\ \vdots & \vdots & \vdots & \end{bmatrix}$$

  - The state vector contains the robot state and all the feature states. $\mathbf{x}_v$ is robot state, e.g. (x, y, θ) in 2D; $\mathbf{y}_i$ is feature state, e.g. (X, Y) in 2D.
- Ie: with have a vector of feature states and the robot state, and then just the covariance matrix (ie: the standard deviations of a feature with respect to another).
- SLAM can be done with a single camera, every time a feature point is detected in an image, it provides a measurement of the angular direction of the feature relative to the camera.
- SLAM enables good roombas, oculus, drones, ARKit.

Positioning and sparse/semi dense reconstruction is now rather mature… and enabling real products.

Probabilistic SLAM is limited to small domains due to:

- Poor computational scaling of probabilistic filters.
- Growth in uncertainty at large distances from map origin makes representation of uncertainty inaccurate.
- Data Association (matching features) gets hard at high uncertainty

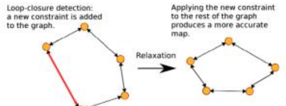
Local Metric Place Recognition Global Optimisation

Practical modern solutions to large scale mapping follow a *metric/topological* approach which approximates full metric SLAM. They need the following elements:

- Local metric mapping to estimate trajectory and make local maps.
- Place recognition, to perform 'loop closure' or relocalise the robot when lost.
- Map optimisation/relaxation to optimise a map when loops are closed.

- One very effective way to detect when an 'old' place is revisited is to save images at regular intervals and use an image retrieval approach (where each image is represented using a Visual Bag of Words which has very much the same character as our invariant sonar descriptors).
- In fact we can make an interesting SLAM system using only place recognition. Topological SLAM with a graph-based representation.
- We simply keep a record of places we have visited and how they connect together, without any explicit geometry information.
- Adapted to symbolic planning and navigation.

- The edges between linked nodes are annotated with relative motion information; could be from local mapping or purely incremental information like odometry or visual odometry.
- Apply **pose graph optimisation (relaxation)** algorithm, which computes the set of node positions which is maximally probable given both the metric and topological constraints.
- Pose graph optimisation only has an effect when there are loops in the graph.



A factor graph is a probabilistic graphical model which represents the factorisation structure of problems like SLAM. Each factor (dot) is the likelihood of one measurement, which depends on a subset of the variables (circles) in the graph.

The general definition of a Gaussian factor is:

$$f_s(\mathbf{x}) = Ke^{-\frac{1}{2}[(\mathbf{z}_s - \mathbf{h}_s(\mathbf{x}_s))^\top \Lambda_s (\mathbf{z}_s - \mathbf{h}_s(\mathbf{x}_s))]}$$

Here $\mathbf{z}_s$ is the measurement represented by this factor, and $\mathbf{h}_s$ is a model of how the measurement depends on a subset of variables $\mathbf{x}_s$. Matrix Λ_s is the precision (inverse covariance) of the measurement. K is a constant. The total likelihood of all measurements is the product of all the factors:

$$p(\mathbf{x}) = \prod_s f_s(\mathbf{x}_s)$$

To 'solve' a factor graph, we want to find the most probable values of the variables $\mathbf{x}$ given all of the measurements; or more general the marginal probability distributions over those variables.

$$p(\mathbf{x}) = \prod_s f_s(\mathbf{x}_s)$$

There are various techniques for factor graph inference, which usually take advantage of the graph sparsity structure, and have different advantages, e.g.:

- Global batch optimisation (e.g. bundle adjustment in computer vision)
- Incremental filtering and marginalisation (e.g. Extended Kalman Filter in MonoSLAM and many early SLAM methods).
- Incremental piece-wise optimisation (iSAM, etc.)
- Distributed inference via Gaussian Belief Propagation.

Elastic Fusion:

Maps a scene **with** millions of surfels and handles small loop closures using deformation.

Relies on a depth camera and GPU processing.

8) Robotics Practicals

Practical 1: Accurate Robot Motion

Investigating and Understanding Motor Control:

- Motor_DegreesPerSecond and Motor_Power perform similar functions. One motor is dictates the velocity of the other - ie: if we power one motor, or turn it as if it is a dial, we change the velocity of the other motor.
- **Difference:** For Power, when we resist the spinning motor on the power script then let go - the velocity of that motor returns to what it was previously as it has a **fixed** power output set to it.
- For DPS: when we resist and let go of the motor during the DPS script, the spinning motor jumps up in velocity momentarily, as because it needs to make up for the lack of degrees spun per second while being resisted so spinning faster for a split second will satisfy the setting put on it by the independent motor. This makes sense, as Power is a constant setting, while the DPS script operates per unit time.

Robot API:

- `BP.set_motor_power(BP.PORT A, power, deg)`: set power level range: -100 to 100, and optionally degrees. Makes the motor move without PID control.
- `BP.set_motor_power(BP.PORT A, BP.MOTOR_FLOAT)`: set the motor to 'float' without power, such that it can be turned by hand (no resistance). Encoder can still be read
- `BP.get motor encoder(BP.PORT A)`: returns the current encoder position in degrees.
- `BP.offset_motor_encoder(BP.PORT_A, BP.get_motor encoder(BP.PORT_A))` zeroes the encoder count.
- `BP.set_motor_position(BP.PORT A, degrees)`: set a position demand for the motor in degrees, and start PID control to reach it.
- `BP.set_motor_dps(BP.PORT A, dps)`: set a velocity demand for the motor in degrees per second, and start PID control to achieve it.
- `BP.get motor status(BP.PORT A)`: return a tuple of four values: current status flag, power in percent, encoder position in degrees and current velocity (DPS).
- `BP.set_motor_limits(BP.PORT A, power, dps)`: set limits on the power and degrees per second that will be used in PID control. Useful to protect your BrickPi and motors from overloading (you should stay below 70%).
- `BP.set_motor position_kp(BP.PORT A, kp)`: set PID proportional gain constant; default is 25.
- `BP.set motor position kd(BP.PORT A, kd)`: set PID differential gain constant; default is 70.
- `BP.reset all()`: disable all motors and sensors.

PORT A,B,C,D are for motors. Ports 1-4 are for sensors.

Practical 2: Sensors and Feedback Control

1) *"Write a program to drive your robot forward until it hits an obstacle. React in a sensible way: if the bump is left (triggers the left sensor), reverse and turn right. If right, reverse and turn left. If the bump is straight ahead it triggers both touch sensors. Turn left or right."*

- Mount a touch sensor on left front and right front of robot, with a bumper mounted in front.
- Initialise sensor: `BP.set_sensor_type(BP.PORT_2, BP.SENSOR_TYPE.TOUCH)` `BP.get_sensor(BP.PORT_2)`
- offset motor encoder; set power limits.
- In a while loop:
- Move forward some small distance (set_motor_position, and straight line movement). Check the left and right sensors (Boolean == 1 = collision).
- If left == 1 == right == 1 reverse, and then turn right (turn right -> left wheel mov fwd, right bckwd, equally)
- if left == 1; turn right. if right == 1, turn left.
- Between every mov, offset motor encoders.

2) **Forward and backward proportional servoing with a Sonar Sensor**

- We want to move closer and closer to a wall, but not actually touch it (maintain some set distance), by accelerating until we get closer, and then we slow and stop. Will need tuning to not be jerky and be accurate.
- Mount a sonar on the front of the robot facing straight forwards, parallel with the ground
- Loop and monitor sonar readings. `BP.set_sensor_type(BP.PORT_1, BP.SENSOR_TYPE.NXT_ULTRASONIC)` `value = BP.get_sensor(BP.PORT_1)`
- Use velocity control: set the velocity demands of both wheels to be proportional to the error between the desired distance and the currently measured distance using proportional control with a single proportional gain value. The speed of the wheels, both equal, should be set to be proportional to the negative of this error.

ie: $v_L = v_R = -K_p(d - z)$

- Remember the sonar sometimes fucks up, so discard outliers / take a median of a few.

3) Wall Following:

- Mount a sonar to the left (or right) of the robot. Slightly adjust the program from before.
- When the sensor is our desired distance away (30cm), we need to drive forwards
- Otherwise, if for instance our sonar is right mounted, if the distance > 30cm, speed the left wheel up and slow the right down. If < 30cm, speed right, slow left down.
- For proportional control, we should set the difference between the speeds of the wheels to be proportional to the negative 'error' between the measured distance z and the desired distance of d = 30cm. The average speed of the two wheels should stay at a pre-chosen constant value (we always increase left wheel speed by same amount we dec right wheel speed & viceversa).

$v_R = v_C + \frac{1}{2}K_p(z - d)$ $\quad$ $v_L = v_C - \frac{1}{2}K_p(z - d)$

With the right gain value, things will be smooth.

Practical 3 & 4:

```
def updateParticleForwardMov(self,
particle, D)->(float, float, float,
float):
error_e = gauss(0, self.std_dev_e)
f = gauss(0, self.std_dev_f)
theta = particle[2]
weight = particle[3]
x_term = particle[0] + (D + error_e) *
cos(theta * pi / 180)
y_term = particle[1] + (D + error_e) *
sin(theta * pi / 180)
angle_term = theta + f
seed()
return (x_term, y_term, angle_term,
weight)

def updateParticleRot(self, particle,
rot) -> (float, float, float, float):
g = gauss(0, self.std_dev_g)
angle_term = rot + g
seed()
return (particle[0], particle[1],
angle_term, particle[3])

def generateCurrLocation(self) ->
(float, float, float):
 xPos = 0
 yPos = 0
 angle = 0
 for particle in self.particles.data:
   xPos += particle[0] * particle[3]
   yPos += particle[1] * particle[3]
   angle += particle[2] * particle[3]
return (xPos, yPos, angle)
```

```
def model_particle_distribution(self, movementType, mov):
partics = []
if (movementType is util.MovementType.TRANSLATION):
for i in range(0, self.NUM_PARTICLES):
  partics.append(self.updateParticleForwardMov(self.particles.data[i], mov))
elif (movementType is util.MovementType.ROTATION):
  for i in range(0,self.NUM_PARTICLES):
     partics.append(self.updateParticleRot(self.particles.data[i], mov))
self.particles.set_particles(partics)

def rot_efficient(self, Wx, Wy, rotateCrude) -> float:
    rotateCrude %= 360
    if rotateCrude > 180: rotateCrude = -(360 - rotateCrude)
    elif rotateCrude < -180: rotateCrude = 360 + rotateCrude
    return rotateCrude  * util.ANGLE_SF

def navigateToWaypoint(self, Wx, Wy):
  (x, y, theta) = self.generateCurrLocation()
  (dx, dy) = (Wx - x, Wy - y)
   alpha = -(atan2(dy, dx))
   l2Dist = (dx*dx + dy*dy)**0.5
   rotateCrude = -((alpha * (180/pi) + theta))
   rotateBy = self.rot_efficient(Wx, Wy, rotateCrude)
   util.reset_encoders(self.BP)
   self.BP.set_motor_position(self.left, -rotateBy)
   self.BP.set_motor_position(self.right, rotateBy)
   time.sleep(abs(rotateBy/util.ANGLE_SF)/15+1)
   self.model_particle_distribution(util.MovementType.ROTATION, theta +
                                    rotateBy/util.ANGLE_SF)
   self.update(self.BP.get_sensor(self.sensor)+21)
   util.reset_encoders(self.BP)
   tMov = 20 * util.TRANSLATION_SF
   for i in range(0, floor(l2Dist / 20)):
       self.BP.set_motor_position(self.right, tMov)
       self.BP.set_motor_position(self.left, tMov)
       time.sleep(4)
       self.model_particle_distribution(util.MovementType.TRANSLATION, 20)
       self.update(self.BP.get_sensor(self.sensor)+21)
       util.reset_encoders(self.BP)
   endMov = (l2Dist % 20) * util.TRANSLATION_SF
   self.BP.set_motor_position(self.right, endMov)
   self.BP.set_motor_position(self.left, endMov)
   time.sleep(4)
   self.model_particle_distribution(util.MovementType.TRANSLATION,l2Dist%20)
        self.update(self.BP.get_sensor(self.sensor)+21)
        util.reset_encoders(self.BP)
        (x, y, theta) = self.generateCurrLocation()

def normalise(self, ps):
    weights = 0
    normalised_particles = []
    for p in ps: weights += p[3]
        for p in ps:
            normalised_particles.append((p[0], p[1], p[2], p[3] / weights))
    return normalised_particles

def resample(self, ps, cdfs):
    resampled = []
    for i in range(0, self.NUM_PARTICLES):
      prob_pt = uniform(0, 1)
      for j in range(0, self.NUM_PARTICLES-1):
        if (cdfs[j] <= prob_pt and cdfs[j+1] >= prob_pt):
         resampled.append((ps[i][0],ps[i][1],ps[i][2],1/self.NUM_PARTICLES))
    endVal = ps[len(ps)-1]
    resampled.append((endVal[0],endVal[1],endVal[2],1/self.NUM_PARTICLES))
    return resampled

def update(self, z):
    if (z >= 255): return
    particles = []
    for p in self.particles.data:
      particles.append(self.calculate_likelihood(p[0], p[1], p[2], p[3], z))
      normalised_particles = self.normalise(particles)
      cdf_arr = []
      cumsum = 0
      for p in self.particles.data:
         cumsum += p[3]
         cdf_arr.append(cumsum)
      resampled_particles = self.resample(normalised_particles, cdf_arr)
      self.particles.set_particles(resampled_particles)

def expected_depth(self, wall, cLoc) -> int:
     (Ax, Ay, Bx, By) = wall
     (x, y, theta) = cLoc
      numerator = (By - Ay) * (Ax - x) - (Bx - Ax) * (Ay - y)
      denom = (By - Ay) * cos(theta) - (Bx - Ax) * sin(theta)
      if (denom < 2): return -1
      return numerator / denom


def weight_update(self, m, z) -> int:
      numerator = - (z - m) ** 2
      denom = 2 * (2.5) ** 2
      exponent = numerator / denom
      K = 0.2
      return e ** exponent + K
```

```
def calculate_likelihood(self, x, y, theta, w, z) -> (float, float, float,
float):
      angle = 0
      candidate_walls = self.circuit.walls
      for (Ax, Ay, Bx, By) in self.circuit.walls:
           numerator = cos(theta) * (Ay - By) + sin(theta) * (Bx - Ax)
           denominator = sqrt((Ay - By)**2 + (Bx - Ax)**2)
           angle = acos(numerator/denominator) * 180/pi
           print(f"My angle: {angle}")
           if angle < 36 and angle > -36:
               candidate_walls.append((Ax, Ay, Bx, By))
      cLoc = self.generateCurrLocation()
      m = self.determine_closest_wall_dist(cLoc, candidate_walls)
      w = self.weight_update(m, z)
      return (x, y, theta, w)

  def determine_closest_wall_dist(self, cLoc, candidate_walls) -> int:
      (x, y, theta) = cLoc
      minDist = 200
      for wall in candidate_walls:
          m = self.expected_depth(wall, cLoc)
          if m < 0:
              continue
          (projx, projy) = (x + m*cos(theta), y + m*sin(theta))
          l2Dist = sqrt(projx ** 2 + projy ** 2)
          if (l2Dist < minDist):
              minDist = l2Dist
      return minDist
```

Practical 5: Planar Camera Calibration

- The only purpose of this practical is to discover your robot's ground plane homography.
- Done with segmenation to find the location of red blobs in the image space, and then compare this to the known locations on the ground plane so we can least squares solve for the H homography matrix. Discover correspondences: don't need to worry about this – this is more of a computer vision problem.
- Solving the linear system:

```
# Form and solve linear system
A = np.array([[x1, y1, 1, 0, 0, 0, -u1 * x1, -u1 * y1],
              [0, 0, 0, x1, y1, 1, -v1 * x1, -v1 * y1],
              [x2, y2, 1, 0, 0, 0, -u2 * x2, -u2 * y2],
              [0, 0, 0, x2, y2, 1, -v2 * x2, -v2 * y2],
              [x3, y3, 1, 0, 0, 0, -u3 * x3, -u3 * y3],
              [0, 0, 0, x3, y3, 1, -v3 * x3, -v3 * y3],
              [x4, y4, 1, 0, 0, 0, -u4 * x4, -u4 * y4],
              [0, 0, 0, x4, y4, 1, -v4 * x4, -v4 * y4]])
b = np.array([u1, v1, u2, v2, u3, v3, u4, v4])
R, residuals, RANK, sing = np.linalg.lstsq(A, b,
rcond=None)
# Build homography matrix
H = np.array([[R[0], R[1], R[2]],
              [R[3], R[4], R[5]],
              [R[6], R[7], 1]])
print (H)
HInv = np.linalg.inv(H)

def HtransformXYtoUV(H, xin, yin):
    xvec = np.array([xin, yin, 1])
    uvec = H.dot(xvec)
    uout = uvec[0]/uvec[2]
    vout = uvec[1]/uvec[2]
    return(uout, vout)

def HtransformUVtoXY(HInv, uin, vin):
    uvec = np.array([uin, vin, 1])
    xvec = HInv.dot(uvec)
    xout = xvec[0]/xvec[2]
    yout = xvec[1]/xvec[2]
    return(xout, yout)
```