


# Lecture 18

## An Introduction to Robotics: Part II

Rob Gaizauskas

# Lecture Outline

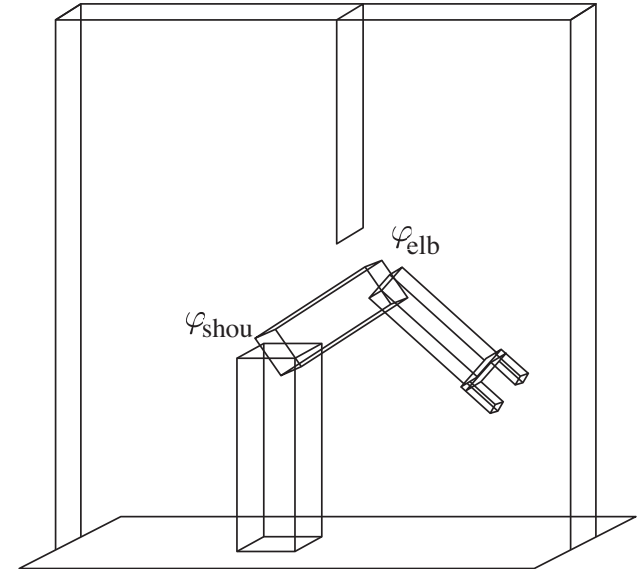
- Definition and Types of Robots
- Sensors and Effectors
- Robotic Perception
- Planning to Move 
- Moving
- Robot Software Architectures
- Applications
- Reading: (Readings that begin with \* are **mandatory**)
  - \*Russell and Norvig (2010), Chapter 25 “Robotics”

# Planning to Move

- Robot choices all ultimately come down to deciding how to move effectors
- The **point-to-point motion** problem is to move robot or end effector to a designated target location
- The **compliant motion** problem is to move robot when in contact with another object (e.g. pushing a box or screwing in a light bulb)
- First need to find a suitable representation in which to describe motion-planning problem
  - **Configuration space** – space of robot states defined by location, orientation and joint angles – turns out to be better than 3D space
- The **path planning** problem is to find a path from one configuration to another in configuration space

# Configuration Space

- Consider the arm in the figure
- Has two joints – shoulder and elbow that move independently
  - Can alter  $(x,y)$  co-ordinates of gripper
  - Cannot move in  $z$  direction
- So, configuration could be fully described by a 4D co-ordinate:
  - $(x_e, y_e)$  for location of elbow
  - $(x_g, y_g)$  for location of gripper
- These co-ordinates constitute a **workspace representation**
  - Co-ordinates of robot specified in same co-ordinate system as object to be manipulated/avoided

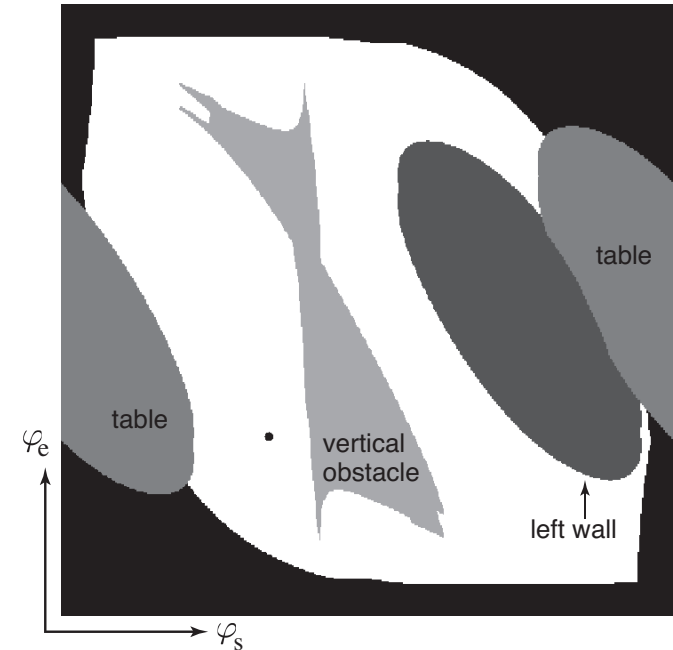


# Configuration Space

- Problem with workspace representation: not all workspace co-ordinates are reachable
  - Linkage constraints of robot rule out some workspace co-ordinates
  - E.g. elbow and gripper are always a fixed distance apart
- Thus a motion planner working over workspace co-ordinates faces problem of generating paths that satisfy these constraints
- Instead plan in **configuration space** – represent state of robot by configuration of joints rather than Cartesian co-ordinates of its elements

# Configuration Space

- In example can represent state of robot with two angles  $\phi_s$  and  $\phi_e$  for the shoulder and elbow joints
- If no obstacles, robot can take on any value in configuration space
- To plan motion
  - connect present configuration and target configuration by a straight line
  - Move joints at constant velocity along this path to target location

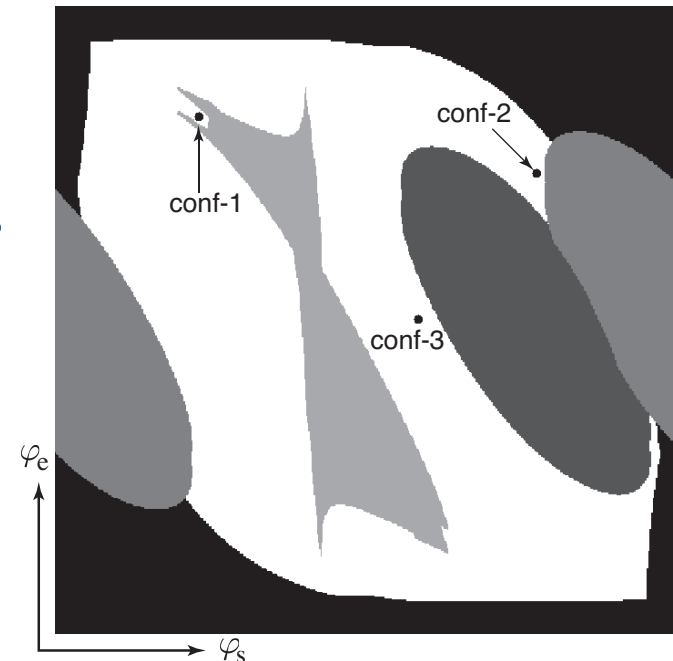
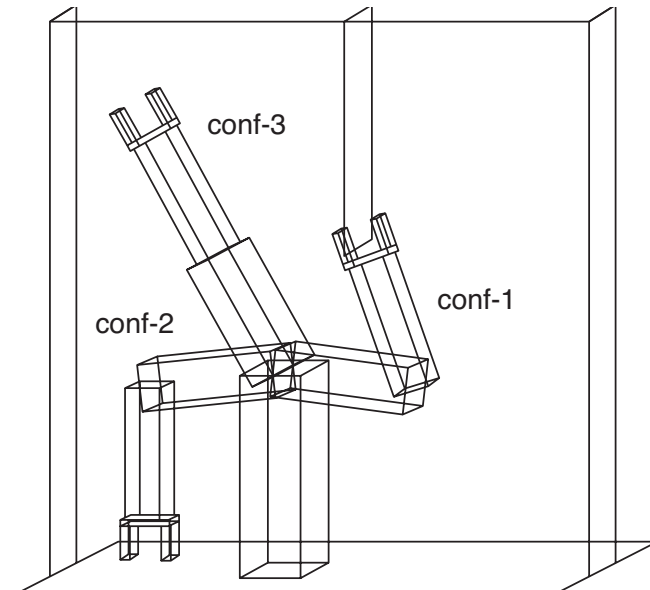


# Configuration Space (cont)

- Problem with using configuration space is that task is usually expressed in workspace co-ordinates
  - So need to map between workspace co-ordinates and configuration space co-ordinates
- Configuration space co-ords  $\rightarrow$  workspace co-ords
  - Simple co-ordinate transformation – linear for prismatic joints, trigonometric for revolute joints
  - These transformations known as **kinematics**
- Workspace co-ords  $\rightarrow$  configuration space co-ords
  - Known as **inverse kinematics**
  - Difficult, especially for robots with many DOFs
  - Solution in general is not unique
  - In our 2D example there are 0 to 2 inverse kinematic solutions for any set of workspace co-ordinates
  - Many robots have sufficient DOFs to lead infinitely many solutions

# Configuration Space

- Obstacles in workspace may have simple forms in workspace co-ords but complex shapes in configuration space
- Example shows
  - Obstacle hanging from ceiling
  - Space of configurations the robot can reach in white (**free space**)
  - Space of unreachable configurations (**occupied space**)
    - Black = where robot collides with self
    - 2 ovals on side = table on which robot sits
    - 3<sup>rd</sup> oval = left wall
    - Irregular shape in middle = ceiling obstacle
- Since shape of free space can be very complicated is typically probed rather than constructed explicitly



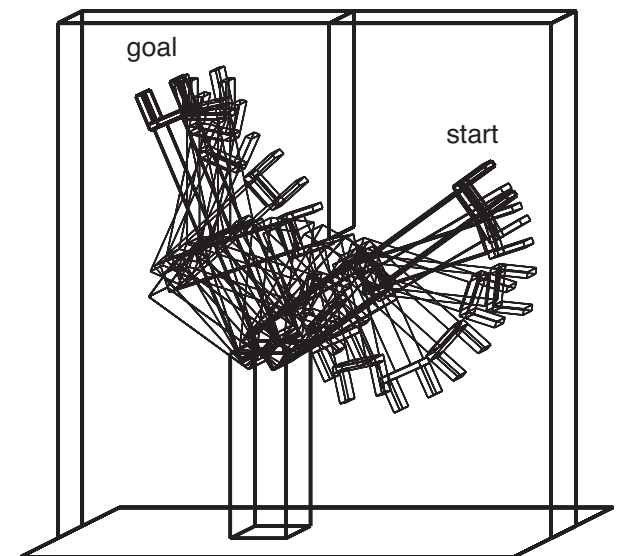
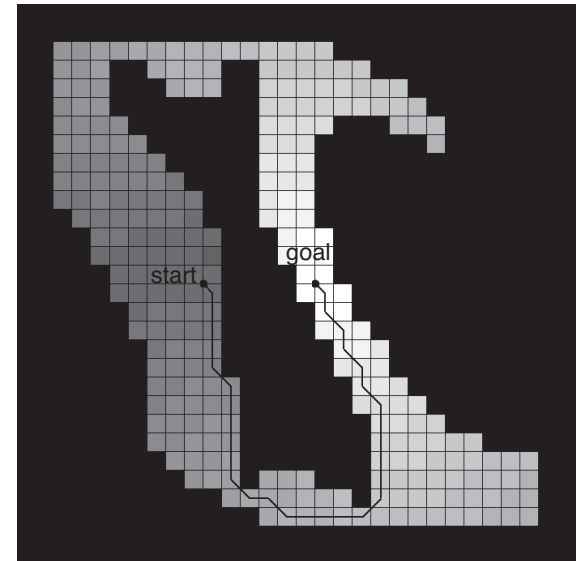


# Cell Decomposition Methods for Planning

- One approach to path planning uses cell decomposition
  - Free space is decomposed into a finite number of contiguous regions called cells
- Each cell has property that path planning within cell has a simple solution – e.g. moving along straight line
- Path planning then becomes a discrete graph search problem for which many solutions exist

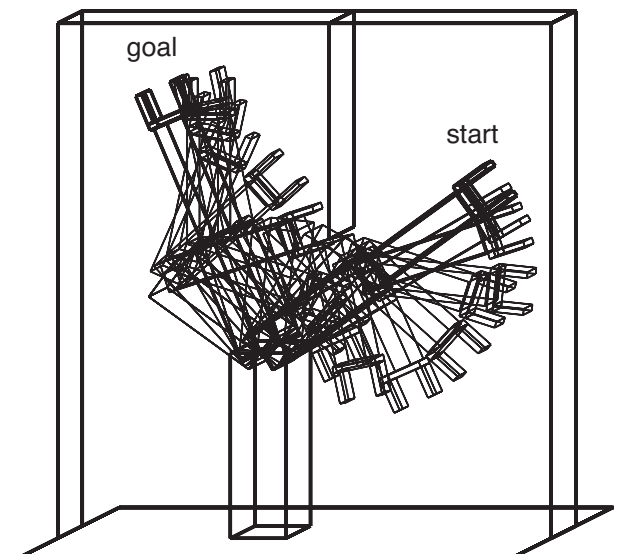
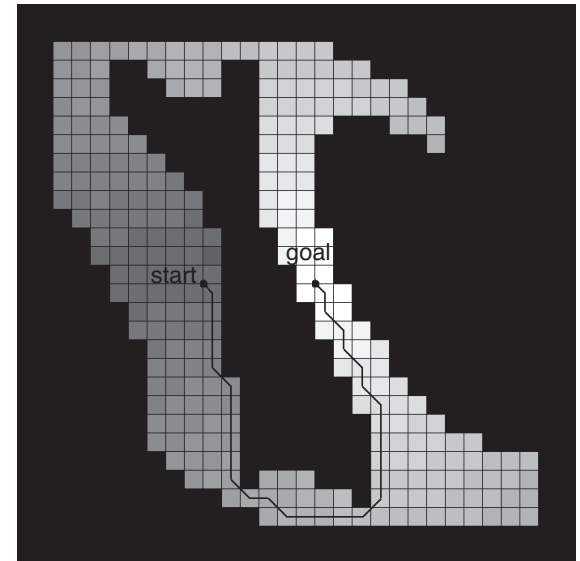
# Cell Decomposition Methods for Planning

- Simplest solution is a regularly spaced grid
- Example shows solution path that is optimal for grid size
  - Shading shows **value** of each cell = cost of shortest path from that cell to the goal
- Approach is simple to implement but has 3 limitations
  1. Workable only for low dimensional configuration spaces – number of cells grows exponentially with number of dimensions



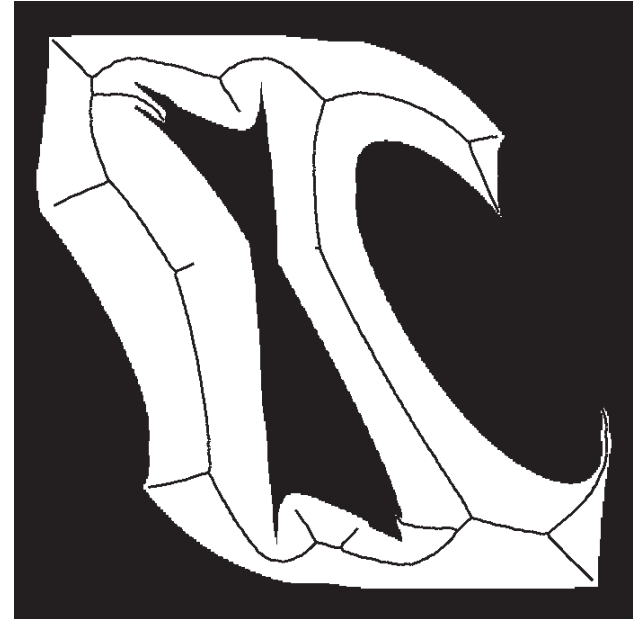
# Cell Decomposition Methods for Planning

- ... approach is simple to implement but has 3 limitations
  2. Problem of **mixed** cells – cells that lie in partly free, partly occupied space
    - If path is planned through such cells it may not work (unsound)
    - But if they are avoided perhaps only solution is missed (incomplete)
  3. Any path through a discretized state space will not be smooth
- Refinement to the approach exist that address these issues



# Skeletonization Methods

- Another approach to path-planning is **skeletonization**
  - Key idea: reduce robot's free space to a 1D representation for which planning is easier
  - This 1D representation is called a **skeleton** of the configuration space
- Figure shows **Voronoi graph** of free space – set of all points that are equidistant to two or more obstacles
- To plan robot
  - First moves to point on Voronoi graph – can be done by straight line movement in configuration space
  - Then follows Voronoi graph to closest point to target
  - Then leaves Voronoi graph and moves in straight line to target

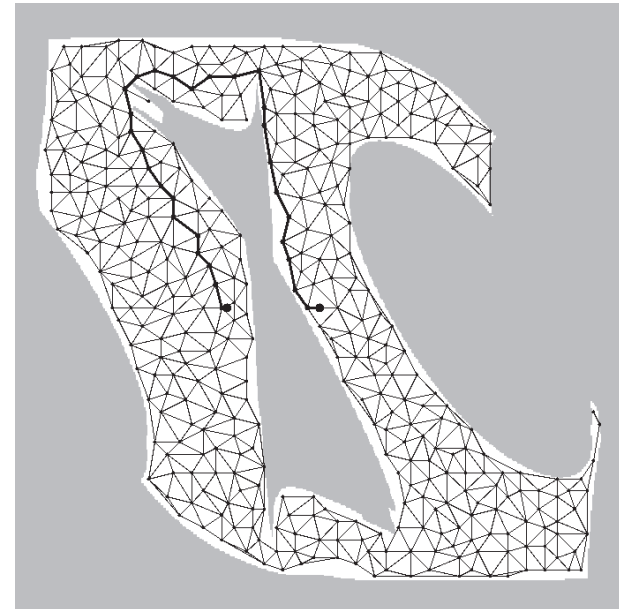


# Skeletonization Methods (cont)

- This approach reduces the problem to finding a path on the Voronoi graph (VG) which is
  - Generally 1D
  - Has finitely many points where three or more 1D curves intersect
- Finding the shortest path on the VG is a discrete graph search problem for which many standard algorithms exist
- Advantages:
  - Searching for shortest path on the VG is easy
  - Paths tend to maximise clearance from obstacles
- Disadvantages:
  - Difficult to apply to higher-dimensional configuration spaces
  - Can result in unnecessarily large detours when configuration space is wide open
  - Computing VG can be hard in configuration space, especially where the shapes of obstacles may be complex

# Skeletonization Methods

- An alternative to the Voronoi graph is the **probabilistic roadmap**
- Starts by creating a graph whose nodes are randomly generated in configuration space and retained only if there are in free space
- Arcs are then drawn between pairs of nodes only if it is “easy” to reach one node from the other, e.g. by a straight line in free space
- Result is a randomized graph in free space
- Add robot’s start and goal configurations and path planning then reduces to discrete graph search again
- Theoretically, approach is incomplete as bad choice of random points may result in no path. But:
  - Can bound probability of failure in terms of number of point generated and geometric properties of configuration space
- With refinements this approach can scale to higher dimensional configuration spaces better than most alternatives



# Moving – Dynamics and Control

- So far have assumed robot can follow any path that our path planning algorithm produces
  - Not true: robots have inertia and cannot execute arbitrary paths except at arbitrarily slow speeds
  - Robots get to specify forces (e.g. torque applied to rotational joint) NOT to specify locations
- Solution might seem to be to extend our model from a **kinematic state** to **dynamic state** representation
  - i.e. record not just positions of effectors (e.g. angles of joints) but **rate of change** of these positions
  - Can capture this information **in differential equations**
- But, doubles dimensionality of representation and makes many motion planning algorithms intractable

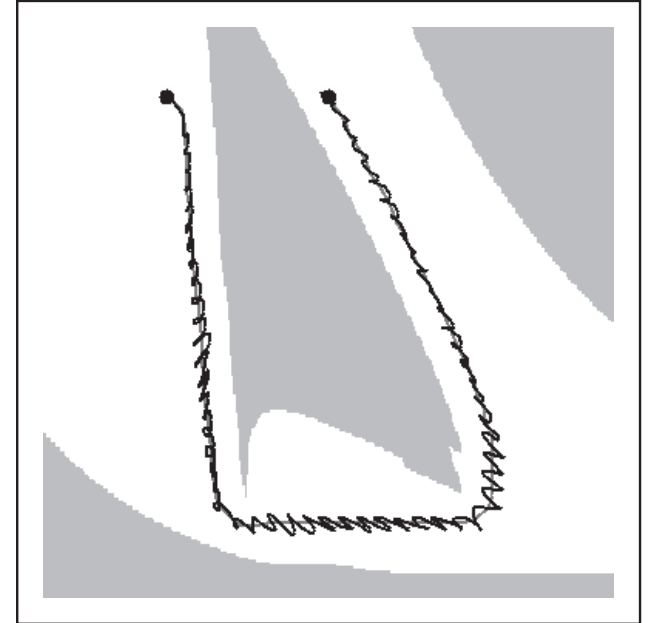
# Moving – Dynamics and Control (cont)

- Common solution is to stick with kinematic state representations for motion planning but use a separate **controller** for keeping the robot on the path
- Controllers keep robot on track by using real-time feedback from the environment to achieve a control objective
  - If objective is to keep robot on preplanned path controller is called a **reference controller** and the path **a reference path**
- Keeping a robot on a pre-specified path might seem to be a simple problem ...



# Moving – Dynamics and Control (cont)

- Simple solution is for robot to generate an opposing force whenever it deviates from path (due to noise or limits on the forces it can apply)
  - Force magnitude proportional to deviation
- However, as figure shows, this leads robot to vibrate excessively around the path
  - Result from inertia – when driven back to reference path robot overshoots, leading to a further correction in reverse direction and so on



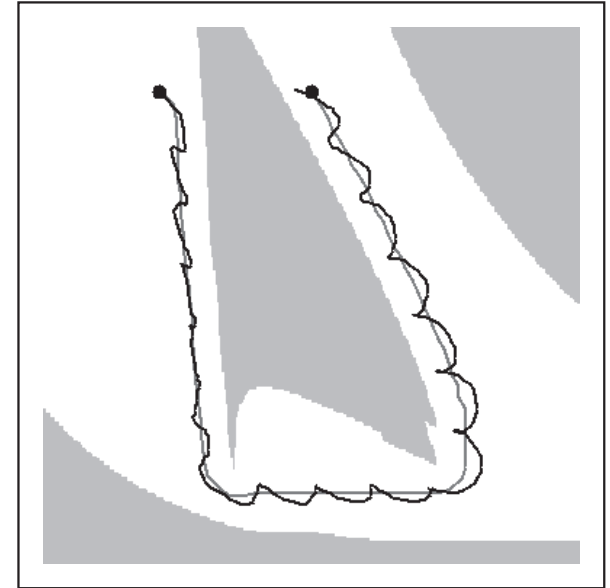
# Moving – Dynamics and Control (cont)

- Controllers that provide force in negative proportion to observed error are called **P controllers** (“P” for proportional)
- If we let
  - $y(t)$  be the reference path, parametrised by time index  $t$
  - $x_t$  be the robot state at time  $t$
  - $K_p$  be a constant called the **gain parameter**Then the control  $a_t$  generated by a P controller has the form

$$a_t = K_p (y(t) - x_t)$$

# Moving – Dynamics and Control (cont)

- $K_p$  regulates how strongly the control corrects for deviations between the actual state and the reference state
- Reducing  $K_p$  might appear to offer a solution
  - But all it really does is slow down the oscillation
  - Figure shows effect of reducing gain  $K_p$  from 1 to .1
- Problems like this are addressed in **control theory** and considerably better solutions are available
  - A reference controller is said to be **stable** if small perturbations lead to bounded error between robot and reference signal
  - It is **strictly stable** if given small perturbations it can return to and then stay on its reference path
- The simplest controller that achieves strict stability in this domain is a **PD controller** (proportional derivative)

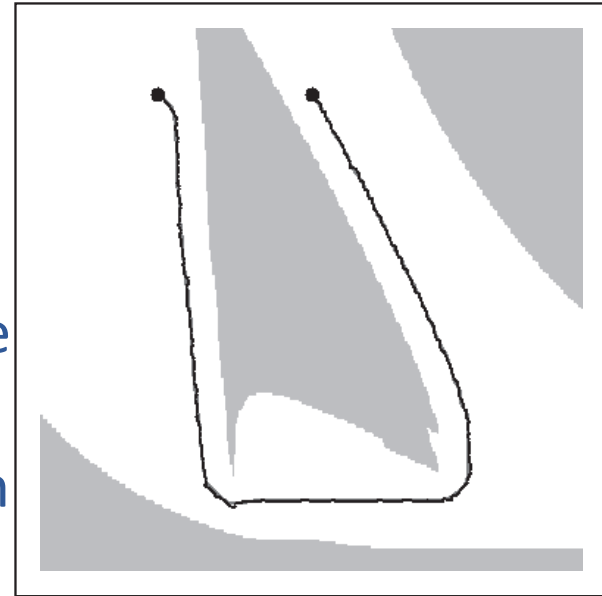


# Moving – Dynamics and Control (cont)

- PD controllers are described by:

$$a_t = K_P(y(t) - x_t) + K_D \frac{\delta(y(t) - x_t)}{\delta t}$$

- PD controllers extend P controllers by adding differential term proportional to rate of change of error over time
- Has effect of damping the controlled system
  - If error is changing rapidly differential term counterbalances proportional term
  - If error persists over time, differential term vanishes and proportional term dominates
- Figure shows PD controller with  $K_p = .3$  and  $K_D = .8$ 
  - Notice path is much smoother and no sign of obvious oscillations
- Further refinements (**PID controllers**) regulate in the case of systematic external forces that may not be part of the model
  - E.g. car driving on banked surface; wear and tear in robot arms



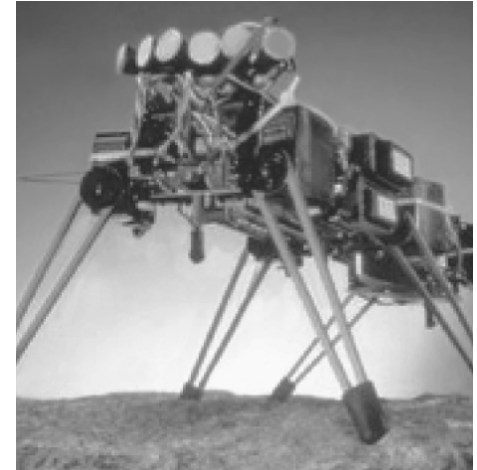
# Reactive Control

- Path planning requires the robot to have some model of the external environment in order for it to construct a reference path
- The approach has several difficulties:
  - Models may be hard to obtain – esp. for complex or remote environments or if robot has limited sensors
  - Even if we do have a sufficiently accurate model, computational difficulty or localisation error might make the approach infeasible
- Alternative is a reflex agent architecture using **reactive control**

# Reactive Control (cont)

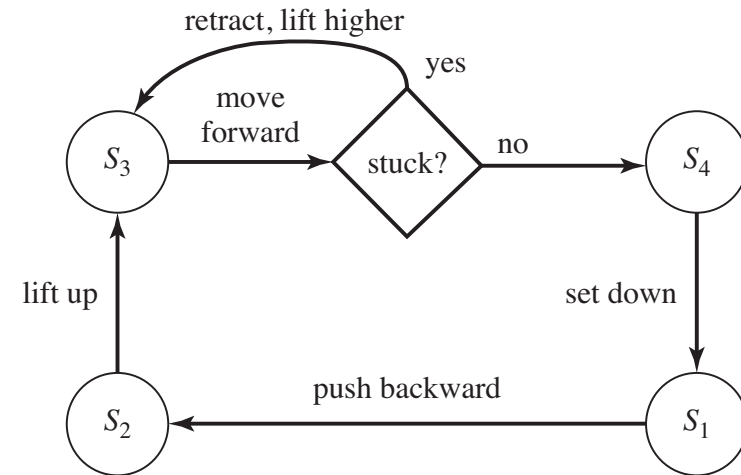
Example: multi-legged robot trying to lift leg over obstacle

- Could give robot rule:
  - if leg hits obstacle, move leg back, lift leg  $h$  cm higher and try again
- Could think of  $h$  as modelling an aspect of the world
- Or, could think of it as a variable of the robot controller, with no direct physical meaning
- Pictured robot (Genghis) is designed for walking on rough terrain
  - Has inadequate sensors to obtain model of terrain for path planning
  - And with 12 DOF (2 for each leg) planning would be computationally intractable



# Reactive Control (cont)

- Instead specify controller without an environmental model
- First chose a **gait** – a pattern of movement of legs
- A statically stable gait is:
  - First move right front, right rear, left centre legs
  - Then move other three
- Works well on flat terrain but an obstacle may prevent leg from moving forwards
- In this case can use a finite state controller as shown
- Such controllers can generate very robust walking patterns over rugged terrain



# Reactive Control (cont)

- Reactive controllers such as the one described are model-free and do not plan or use search to generate controls
  - Yet feedback from environment plays key role in what happens
- Behaviour like this that emerges from interaction between a simple controller and a complex environment is called **emergent behaviour**
- While strictly all robot behaviour is emergent – since no model is perfect – the term is normally used just for techniques that do not use explicit models of the environment
  - Emergent behaviour is also characteristic of biological organisms



# Robotic Software Architectures

- A methodology for structuring algorithms is called a **software architecture**
- Modern software architectures for robotics address how to integrate reactive control and model-based planning
  - Strengths and weaknesses of the two are complementary
  - Reactive control is sensor-driven
    - Good for low-level decision making in real-time
    - Not good for global control decisions that require information that cannot be sensed at the time of decision making
  - Deliberation-based planning better for such global decisions
- Hence most robot architectures combine reactive techniques at low level with deliberative techniques at higher levels
  - Called **hybrid architectures**

# Subsumption Architecture

- Proposed by Brooks (1986), it's a framework for assembling reactive controllers out of **finite state machines** (FSMs)
- Nodes in FSMs may contain tests for sensor variables and action of machine can be conditioned on outcome of such tests
- Arcs between nodes generate message when traversed that get sent to robot effectors or to other FSMs
- FSMs also have internal clocks that control time takes to traverse an arc (hence called **augmented** FSMs, or AFSMs)
- Example shown for hexapod robot above
  - 4 states + sensor test between states  $s_3$  and  $s_4$
  - Allows robot to react to events arising from its interaction with the environment

# Subsumption Architecture (cont)

- Subsumption architecture has additional primitives for
  - Synchronising AFSMs
  - Combining output values of multiple, possibly conflicting AFSMs
- Allow programmer to build increasingly complex controllers in bottom-up fashion. E.g. could
  - Begin with AFSMs for individual legs
  - Then add AFSM for coordinating multiple legs
  - Then add higher level behaviours for, e.g., collision avoidance, involving backing up and turning

# Subsumption Architecture (cont)

- By contrast building such a behaviour using a configuration-space path-planning algorithm would be prohibitively difficult
  - Would need model of terrain
  - Configuration space of robot with 6 legs each with two independent motors is 18 dimensional
    - $6 \times 2 = 12$  for legs
    - 6 for location and orientation of robot with respect to environment
  - Need to consider unforeseen events – e.g. robot sliding down slope – P-type controllers might not be able to help

# Subsumption Architecture (cont)

But, subsumption architecture has its own problems:

- AFSMs driven by raw sensor input
  - OK if data is reliable and sufficient for decision making
  - Fails if sensor data needs to be integrated over time
  - Thus subsumption-style controllers mostly applied to simple tasks like following a wall or moving towards a light
- Lack of deliberative capability means it is difficult to modify subsumption-style robot for a new task
  - Hence tend to be single task only
- Subsumption-style controllers difficult to understand and program
  - Interplay between dozens of interacting AFSMs and the environment beyond what most human programmers can comprehend
- Thus subsumption architecture now rarely used, despite great historical importance

# Three-layer architecture

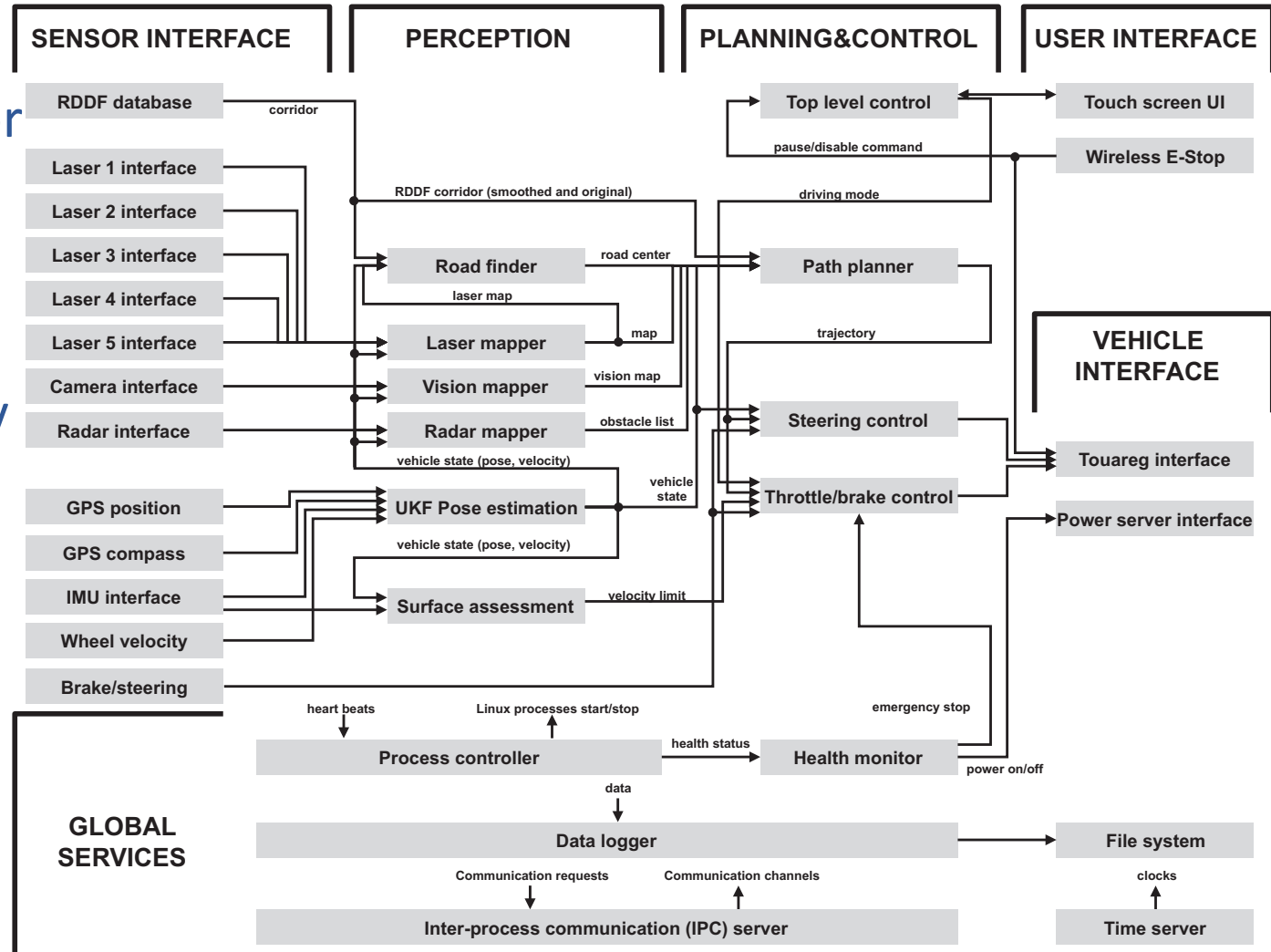
- Hybrid architectures combine reaction and deliberation
- Most popular is three-layer architecture:
  - **Reactive layer** (milliseconds):
    - Provides low level control using tight sensor-action loop
  - **Executive layer** (seconds):
    - Accepts directives from deliberative layer and sequences them for reactive layer (e.g. take set of via-points from path planner and make decisions about which reactive component to invoke)
    - Integrates sensor information into an internal state representation using, e.g. localisation and mapping routines)
  - **Deliberative layer** (minutes):
    - Generates global solutions to complex tasks using planning + models (either supplied or learned from data and using info from executive Layer)

# Pipeline Architecture

- Like the subsumption architecture, **pipeline architecture** executes multiple processes in parallel
- However, modules are like those in three-layer architecture
- Pipeline consists of sequence of layers, e.g. for control of autonomous car
  - **Sensor interface layer**: accepts raw sensor input
  - **Perception layer**: updates internal models of environment based on sensor data
  - **Planning and control layer**: adjusts robot's plans and turn them into controls
  - **Vehicle interface layer**: passes controls back to the vehicle
- Key is that all layers function asynchronously in parallel. E.g.
  - While perception layer processes most recent sensor data, control layer bases decisions on slightly older data
  - Analogous to human brain – we perceive, plan and act at same time

# Pipeline Architecture

- Software architecture for a robot car
- All modules process data simultaneously





# Applications

- **Industry and Agriculture**
  - Assembly lines; paint stripping from ships; mining
- **Transportation**
  - Autonomous helicopters; automatic wheelchairs; container movers; indoor load movers
- **Robotic Cars**
  - Google cars; Darpa Urban Challenge entrants
- **Health Care**
  - Surgical assistants (instrument placement); robotic walkers; personal care assistants
- **Hazardous Environments**
  - Nuclear waste cleanup (Fukushima, Chernobyl); World Trade Center (9/11); minefield clean up and bomb defusal

# Applications (cont)

- **Exploration**
  - Mars explorers; international space station; mapping sunken ships and abandoned mines; aerial reconnaissance
- **Personal Services**
  - Autonomous vacuum cleaners, lawn mowers, golf caddies; information kiosks; Siri
- **Entertainment**
  - Robotic software
- **Human Augmentation**
  - Legged walking machines for carrying disabled people; robotic hands; robotic teleoperation (master-slave configuration – robot mimics human movements) – e.g. for underwater exploration

# Summary

- Robots are **mechanical or virtual artificial agents**, usually an electro-mechanical machine that is **guided by a computer program** or electronic circuitry
- Robots have **sensors** for perceiving the environment and **effectors** for acting on it
- **Robotic perception** is about deriving decision-relevant information from sensor data.
  - Requires an internal representation and method for updating it over time.
  - Examples of perception are: **localization**, **mapping** and **object recognition**
  - Probabilistic filtering algorithms, like **Monte Carlo localization**, that maintain the belief state (a posterior distribution over state variables) are useful for robot perception

# Summary (cont)

- Robot motion **planning** is usually done in **configuration space**, which specifies the location and orientation of the robot and its joint angles
  - Configuration space search algorithms include **cell decomposition** and **skeletonization**
- Given a reference path, the path can be executed by a **P-type controller**, which accommodates small perturbations introduced by noise or inertial constraints
- An alternative to deriving a path from a model of the environment is to use **reactive controllers**, which directly specify how a robot is to respond to the environment
- Robotic **architectures** offer a framework for combining the many modules comprising a robot. Architectures include
  - **Subsumption architecture**
  - **Three layer architecture**
  - **Pipeline architecture**

# References

Russell, Stuart and Norvig, Peter (2010) Artificial Intelligence: A Modern Introduction (3<sup>rd</sup> ed). Pearson. Chapter 25.

Wikipedia: Robotics. <https://en.wikipedia.org/wiki/Robotics> (visited 05/12/15).

Wikipedia: Robots. <https://en.wikipedia.org/wiki/Robot> (visited 05/12/15).