today’s topics:
• introduction to robotics

(1) autonomous agents and autonomous robotics.
• we will be discussing autonomous mobile robots
• what is a robot?
  – “a programmable, multifunction manipulator designed to move material, parts, tools or specific devices through variable programmed motions for the performance of various tasks.” [Robot Institute of America]
  – “an active, artificial agent whose environment is the physical world” [Russell&Norvig, p773]
• what is an agent?
  – “anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.” [Russell&Norrig, p32]
• what is autonomy?
  – no remote control!
  – an agent makes decisions on its own, guided by feedback from its sensors; but you write the program that tells the agent how to make its decisions environment.

(1) our definition of a robot.
• robot = autonomous embodied agent
• has a body and a brain
• exists in the physical world (rather than the virtual or simulated world)
• is a mechanical device
• contains sensors to perceive its own state
• contains sensors to perceive its surrounding environment
• possesses effectors which perform actions
• has a controller which takes input from the sensors, makes intelligent decisions about actions to take, and effects those actions by sending commands to motors

(1) a bit of robot history.
• the word robot came from the Czech word robota, which means slave
• used first by playwright Karel Capek, “Rossum’s Universal Robots” (1923)
• human-like automated devices date as far back as ancient Greece
• modern view of a robot stems from science fiction literature
• foremost author: Isaac Asimov, “I, Robot” (1950)
• the Three Laws of Robotics
  1. A robot may not injure a human being, or, through inaction, allow a human being to come to harm.
  2. A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.
  3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.
• Hollywood broke these rules: e.g., “The Terminator” (1984)
(1) effectors.

- comprises all the mechanisms through which a robot can effect changes on itself or its environment
- actuator = the actual mechanism that enables the effector to execute an action; converts software commands into physical motion
- types:
  - arm
  - leg
  - wheel
  - gripper
- categories:
  - manipulator
  - mobile

(1) mobile robots.

- classified by manner of locomotion:
  - wheeled
  - legged
- stability is important
  - static stability
  - dynamic stability

(1) degrees of freedom.

- number of directions in which robot motion can be controlled
- free body in space has 6 degrees of freedom:
  - three for position \((x, y, z)\)
  - three for orientation \((\text{roll, pitch, yaw})\)
    - yaw refers to the direction in which the body is facing
      i.e., its orientation within the \(xy\) plane
    - roll refers to whether the body is upside-down or not
      i.e., its orientation within the \(yz\) plane
    - pitch refers to whether the body is tilted
      i.e., its orientation within the \(xz\) plane
- if there is an actuator for every degree of freedom, then all degrees of freedom are controllable \(\Rightarrow\) holonomic
- most robots are non-holonomic

(1) sensors.

- \(\Rightarrow\) perception
  - proprioceptive: know where your joints/sensors are
  - odometry: know where you are
- function: to convert a physical property into an electronic signal which can be interpreted by the robot in a useful way

<table>
<thead>
<tr>
<th>property being sensed</th>
<th>type of sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>contact</td>
<td>bump, switch</td>
</tr>
<tr>
<td>distance</td>
<td>ultrasound, radar, infra red (IR)</td>
</tr>
<tr>
<td>light level</td>
<td>photo cell, camera</td>
</tr>
<tr>
<td>sound level</td>
<td>microphone</td>
</tr>
<tr>
<td>smell</td>
<td>chemical</td>
</tr>
<tr>
<td>temperature</td>
<td>thermal</td>
</tr>
<tr>
<td>inclination</td>
<td>gyroscope</td>
</tr>
<tr>
<td>rotation</td>
<td>encoder</td>
</tr>
<tr>
<td>pressure</td>
<td>pressure gauge</td>
</tr>
<tr>
<td>altitude</td>
<td>altimeter</td>
</tr>
</tbody>
</table>
more on sensors.

- operation
  - passive: read a property of the environment
  - active: act on the environment and read the result

- noise
  - internal: from inside the robot
  - external: from the robot’s environment
  - calibration: can help eliminate/reduce noise

environment.

- accessible vs inaccessible
  - robot has access to all necessary information required to make an informed decision about to do next

- deterministic vs nondeterministic
  - any action that a robot undertakes has only one possible outcome.

- episodic vs non-episodic
  - the world proceeds as a series of repeated episodes.

- static vs dynamic
  - the world changes by itself, not only due to actions effected by the robot

- discrete vs continuous
  - sensor readings and actions have a discrete set of values.

state.

- knowledge about oneself and one’s environment
  - kinematics = study of correspondence between actuator mechanisms and resulting motion
    - motion:
      - rotary
      - linear
  - combines sensing and acting
    - did i go as far as i think i went?

- but one’s environment is full of information
- for an agent, what is relevant?

control.

- autonomy
- problem solving
- modeling
  - knowledge
  - representation
- control architectures
- deliberative control
- reactive control
- hybrid control
to be truly autonomous, it is not enough for a system simply to establish direct numerical relations between sensor inputs and effector outputs

- a system must be able to accomplish goals
- a system must be able to solve problems

⇒ need to represent problem space
  - which contains goals
  - and intermediate states

there is always a trade-off between generality and efficiency
  - more specialized ⇒ more efficient
  - more generalized ⇒ less efficient

(1) autonomy.

(1) modeling the robot’s environment.

- modeling
  - the way in which domain knowledge is embedded into a control system
  - information about the environment stored internally: internal representation
  - e.g., maze: robot stores a map of the maze “in its head”

- knowledge
  - information in a context
  - organized so it can be readily applied
  - understanding, awareness or familiarity acquired through learning or experience
  - physical structures which have correlations with aspects of the environment and thus have a predictive power for the system

(1) problem solving: example.

- GPS = General Problem Solver [Newell and Simon 1963]
- Means-Ends analysis

<table>
<thead>
<tr>
<th>operator</th>
<th>preconditions</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH(obj, loc)</td>
<td>at(robot, obj) ∧ large(obj)∧ clear(obj) ∧ armempty()</td>
<td>at(obj, loc)∧</td>
</tr>
<tr>
<td>CARRY(obj, loc)</td>
<td>at(robot, obj) ∧ small(obj)</td>
<td>at(obj, loc)∧ at(robot, loc)</td>
</tr>
<tr>
<td>WALK(loc)</td>
<td>none</td>
<td>at(robot, loc)</td>
</tr>
<tr>
<td>PICKUP(obj)</td>
<td>at(robot, obj)</td>
<td>holding(obj)</td>
</tr>
<tr>
<td>PUTDOWN(obj)</td>
<td>holding(obj)</td>
<td>¬holding(obj)</td>
</tr>
<tr>
<td>PLACE(obj1, obj2)</td>
<td>at(robot, obj2) ∧ holding(obj1)</td>
<td>on(obj1, obj2)</td>
</tr>
</tbody>
</table>

(1) memory.

- divided into 2 categories according to duration
- short term memory (STM)
  - transitory
  - used as a buffer to store only recent sensory data
  - data used by only one behaviour
  - examples:
    - avoid-past: avoid recently visited places to encourage exploration of novel areas
    - wall-memory: store past sensor readings to increase correctness of wall detection
- long term memory (LTM)
  - persistent
  - metric maps: use absolute measurements and coordinate systems
  - qualitative maps: use landmarks and their relationships
  - examples:
    - Markov models: graph representation which can be augmented with probabilities for each action associated with each sensed state
(1) knowledge representation.

- must have a relationship to the environment (temporal, spatial)
- must enable predictive power (look-ahead), but if inaccurate, it can deceive the system
- explicit: symbolic, discrete, manipulable
- implicit: embedded within the system
- symbolic: connecting the meaning (semantics) of an arbitrary symbol to the real world
- difficult because:
  - sensors provide signals, not symbols
  - symbols are often defined with other symbols (circular, recursive)
  - requires interaction with the world, which is noisy
- other factors
  - speed of sensors
  - response time of effectors

(1) components of knowledge representation.

- state
  - totally vs partially vs un- observable
  - discrete vs continuous
  - static vs dynamic
- spatial: navigable surroundings and their structure; metric or topological maps
- objects: categories and/or instances of detectable things in the world
- actions: outcomes of specific actions on the self and the environment
- self/ego: stored proprioception (sensing internal state), self-limitations, capabilities
  - perceptive: how to sense
  - behaviour: how to act
- intentional: goals, intended actions, plans
- symbolic: abstract encoding of state/information

(1) types of representations.

- maps
  - euclidean map
    - represents each point in space according to its metric distance to all other points in the space
  - topological map
    - represents locations and their connections, i.e., how/if they can be reached from one another, but does not contain exact metrics
  - cognitive map
    - represents behaviours; can store both previous experience and use for action
    - used by animals that forage and home (animal navigation)
    - may be simple collections of vectors
- graphs
  - nodes and links
- Markov models
  - associates probabilities with states and actions

(1) control architecture.

- a control architecture provides a set of principles for organizing a control system
- provides structure
- provides constraints
- refers to software control level, not hardware!
- implemented in a programming language
- don’t confuse “programming language” with “robot architecture”
- architecture guides how programs are structured
(1) classes of robot control architectures.

- **deliberative**
  - look-ahead; think, plan, then act
- **reactive**
  - don’t think, don’t look ahead, just react!
- **hybrid**
  - think but still act quickly
- **behaviour-based**
  - distribute thinking over acting

(1) deliberative control.

- classical control architecture (first to be tried)
- first used in AI to reason about actions in non-physical domains (like chess)
- natural to use this in robotics at first
- example: Shakey (1960’s, SRI)
  - state-of-the-art machine vision used to process visual information
  - used classical planner (STRIPS)
- planner-based architecture
  1. sensing (S)
  2. planning (P)
  3. acting (A)
- requirements
  - lots of time to think
  - lots of memory
  - (but the environment changes while the controller thinks)

(1) reactive control.

- operate on a short time scale
- does not look ahead
- based on a tight loop connecting the robot’s sensors with its effectors
- purely reactive controllers do not use any internal representation; they merely react to the current sensory information
- collection of rules that map situations to actions
  - simplest form: divide the perceptual world into a set of mutually exclusive situations recognize which situation we are in and react to it
  - (but this is hard to do!)
- example: subsumption architecture (Brooks, 1986)
  - hierarchical, layered model

(1) hybrid control.

- use the best of both worlds (deliberative and reactive)
- combine open-loop and closed-loop execution
- combine different time scales and representations
- typically consists of three scales:
  1. reactive layer
  2. planner (deliberative layer)
  3. integration layer to combine them
  4. (but this is hard to do!)
(2) LEGO Mindstorms.

- Hitachi h8300 microprocessor called RCX
- with an IR transceiver

- and 3 input ports, for:
  - light sensor
  - touch sensor

- and 3 output ports, for:
  - motors
  - light bulbs

(2) programming the LEGO Mindstorms.

- you write programs on your computer and download them to the RCX using an IR transmitter ("communication tower")

- Mindstorms comes with RoboLab — graphical programming environment
- but people have built other interfaces, e.g.:
  - Not-Quite C (NQC)
  - Brickos
  - lejos

(2) Not-Quite C.

- programming language based on C which runs on the RCX
- first you need to download firmware onto the RCX so that it will understand the NQC code which you write
- then you can write programs
- NQC is mostly like C, with some exceptions...
- for download and full documentation:
  http://bricxcc.sourceforge.net/nqc/
- a smattering of NQC follows
- basic command-line operation:
  bash# nqc -d <rcx-program-file>
  bash# nqc -firmware <firmware-file>
  bash# nqc -help
- note that the NQC subset presented is for RCX 2.0

(2) NQC: program structure.

- comprised of global variables and code blocks
  - variables are all 16-bit integers
  - code blocks:
    - tasks
    - inline functions
    - subroutines
- features include:
  - event handling
  - resource allocation mechanism
  - IR communication
(2) NQC: tasks.

- Multi-tasking program structure
  ```
  task <task-name> {
    // task code goes in here
  }
  ```
- Up to 10 tasks
- Invoked using `start <task-name>`
- Stopped using `stop <task-name>`

(2) NQC: inline functions.

- Functions can take arguments but always `void`
  ```
  void <function-name> ( <arguments> ) {
    // function code goes in here
  }
  ```
- Return statement, just like C
- Arguments:
<table>
<thead>
<tr>
<th>type</th>
<th>meaning</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>pass by value</td>
<td>value can change inside function, changes won’t be seen by caller</td>
</tr>
<tr>
<td>int &amp;</td>
<td>pass by reference</td>
<td>value can change inside function, changes will be seen by caller, only variables may be passed</td>
</tr>
<tr>
<td>const int</td>
<td>pass by value</td>
<td>value cannot be changed inside function, only constants may be passed</td>
</tr>
<tr>
<td>const int &amp;</td>
<td>pass by reference</td>
<td>value cannot be changed inside function, value is read each time it is used</td>
</tr>
</tbody>
</table>

(2) NQC: subroutines.

- Subroutines cannot take any arguments
  ```
  sub <subroutine-name> {
    // subroutine code goes in here
  }
  ```
- Allow a single copy of code to be shared by multiple callers
- So more efficient than inline functions
- Cannot be nested

(2) NQC: variables.

- All are 16-bit signed integers
- Scope is either global or local (just like C)
- Use as many local variables as possible (for efficiency)
- Arrays
  - Declaration just like C
  - Cannot pass whole arrays to functions (but can pass individual array elements)
  - Cannot use shortcut operators on array elements (`++`, `--`, `+=`, `-=`, etc)
  - Cannot do pointer arithmetic
- Hexadecimal notation, e.g.: `0x12f`
- Special values: `true` (non-zero) and `false` (zero)
(2) NQC: operators.

- operators listed in order of precedence

<table>
<thead>
<tr>
<th>operator</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>absolute value</td>
</tr>
<tr>
<td>sign()</td>
<td>sign of operand</td>
</tr>
<tr>
<td>++, −−</td>
<td>increment, decrement</td>
</tr>
<tr>
<td>−, ′, ′</td>
<td>unary minus, bitwise negation, logical negation</td>
</tr>
<tr>
<td>∗, /, %, +, −</td>
<td>multiply, divide, modulo, addition, subtraction</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
<td>left and right shift</td>
</tr>
<tr>
<td>&gt;, &lt;, &gt;=, &lt;=, ==</td>
<td>relational and equivalence operators</td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td>bitwise AND, XOR, OR</td>
</tr>
<tr>
<td></td>
<td>logical AND, OR</td>
</tr>
<tr>
<td>=</td>
<td>assignment operator</td>
</tr>
<tr>
<td></td>
<td>shortcut assignment operators</td>
</tr>
<tr>
<td></td>
<td>set variable to absolute value of expression</td>
</tr>
<tr>
<td></td>
<td>set variable to sign (-1,+1.0) of expression</td>
</tr>
</tbody>
</table>

(2) NQC: preprocessor.

- following directives included:
  - #include "<filename>"
    - file name must be listed in double quotes (not angle brackets)
  - macro definition (#define, #ifdef, #ifndef, #undef)
  - conditional compilation (#if, #elif, #else, #endif)
  - program initialization
    - special initialization function (_init) called automatically (sets all 3 outputs to full power forward, but not turned on)
    - suppress it using: #pragma noinit
    - redirect it using: #pragma init <function-name>
  - reserving global storage locations (there are 32): #pragma reserve <value>

(2) NQC: branching statements.

- if / else — just like C
  
  if ( <condition> ) <consequence>
  if ( <condition> ) <consequence> else <alternative>

- switch — just like C
  
  switch ( <expression> ) {
    case <constant-expression1> : <body>
      .
      .
    case <constant-expressionN> : <body>
    default : <body>
  }

(2) NQC: looping statements.

- while, do...while, for — just like C
  
  while ( <condition> ) <body>
  do <body> while ( <condition> )
  for ( <statement0> ; <condition> ; <statement1> ) <body>

- also use of break and continue statements just like C

- repeat loop (not like C):
  
  repeat ( <expression> ) <body>
  - <expression> is evaluated once, indicating the number of times to perform the body statements

- until loop (not like C):
  
  until ( <condition> );
  - effectively a while loop with an empty body; program waits until condition is true before proceeding
(2) NQC: resource acquisition.

- acquire ( <resources> ) <body>
- acquire ( <resources> ) <body> catch <handler>
- resource access control given to task that makes the call
- execution jumps to catch handler if access is denied
- note that access can be lost in mid-execution of a task with a higher priority requests the resource; to set task’s priority, use SetPriority( <p> ) where <p> is between 0..255; note that lower numbers are higher priority
- resource returned to the system when <body> is done
- example:
  acquire( ACQUIRE_OUT_A ) {
    Wait( 1000 );
  }
  catch {
    PlaySound( SOUND_UP );
  }

(2) NQC: event handling.

- monitor ( <events> ) <body>
- monitor ( <events> ) <body> catch ( <catch-events> ) <handler>
- you can configure 16 events, numbered 0..15 and use EVENT_MASK() macro to identify
  monitor( EVENT_MASK(2) | EVENT_MASK(3) | EVENT_MASK(4) ) {
    Wait( 1000 );
  }
  catch ( EVENT_MASK(4) ) {
    PlaySound( SOUND_DOWN ); // event 4 happened
  }
  catch {
    PlaySound( SOUND_UP ); // event 2 or 3 happened
  }

(2) NQC: sensors.

- identifiers: SENSOR_1, SENSOR_2, SENSOR_3
- SetSensorType( <sensor>,<type> )
  - sets sensor type
    - <type> is one of: SENSOR_TYPE_NONE, SENSOR_TYPE_TOUCH, SENSOR_TYPE_TEMPERATURE, SENSOR_TYPE_LIGHT or SENSOR_TYPE_ROTATION
- SetSensorMode( <sensor>,<mode> )
  - sets sensor mode
    - <mode> is one of: SENSOR_MODE_RAW, SENSOR_MODE_BOOL, SENSOR_MODE_PERCENT, SENSOR_TYPE_LIGHT or SENSOR_TYPE_ROTATION
- SensorValue( <sensor> )
  - reads sensor value

(2) NQC: outputs.

- identifiers: OUT_A, OUT_B, OUT_C
- SetOutput( <outputs>,<mode> )
  - sets output mode
    - <mode> is one of: OUT_OFF, OUT_ON or OUT_FLOAT
- SetDirection( <outputs>,<direction> )
  - sets output direction
    - <direction> is one of: OUT_FWD, OUT_REV or OUT_TOGGLE
- SetPower( <outputs>,<power> )
  - sets output power (speed)
    - <power> is one of: OUT_LOW, OUT_HALF, OUT_FULL or 0.7 (lowest..highest)
- multiple <output> identifiers can be added together
- also: On( <outputs> ), Off( <outputs> ), Fwd( <outputs> ), Rev( <outputs> ), OnFwd( <outputs> ), OnRev( <outputs> ), OnFor( <outputs>,<time> ) (where <time> is in 100ths of a second)
(2) NQC: sound.

- **PlaySound( <sound> )**
  - Plays a sound
  - **<sound>** is one of: SOUND_CLICK, SOUND_DOUBLE_BEEP, SOUND_DOWN, SOUND_UP, SOUND_LOW_BEEP or SOUND_FAST_UP

- **PlayTone( <frequency>, <time> )**
  - Plays "music"
  - **<frequency>** is in Hz
  - **<time>** is in 100ths of a second
  - For example:
    PlayTone( 440, 100 )

(2) NQC: LCD display.

- **SelectDisplay( <mode> )**
  - Displays sensor values
  - **<mode>** is one of: DISPLAY_WATCH, DISPLAY_SENSOR_1, DISPLAY_SENSOR_2, DISPLAY_SENSOR_3, DISPLAY_OUT_A, DISPLAY_OUT_B or DISPLAY_OUT_C

- **SetUserDisplay( <value>, <precision> )**
  - Displays user values
  - **<value>** is the value to display
  - **<precision>** is the number of places to the right of the decimal point (?)

(2) NQC: IR communication.

- Simple communication can send single (one-byte) messages with values between 0..255

- **x = Message()**
  - Reads and returns the most recently received message

- **ClearMessage()**
  - Clears the message buffer

- **SendMessage( <message> )**
  - Sends a message
  - **<message>** is a value between 0..255

(2) NQC: serial IR communication.

- Serial communication allows up to 16-byte messages

- For example:
  - **SetSerialComm( SERIAL_COMM_DEFAULT );**
  - **SetSerialPacket( SERIAL_PACKET_DEFAULT );**
  - **SetSerialData( 0, 10 );**
  - **SetSerialData( 1, 25 );**
  - **SendSerial( 0, 2 );**

- **SetSerialData( <byte-number>, <value> )**
  - Puts data in one byte of the 16-byte transmit buffer
  - **<byte-number>** is between 0..15

- **SendSerial( <start-byte>, <number-of-bytes> )**
  - Sends all or part of the transmit buffer
  - **<start-byte>** is between 0..15
  - **<number-of-bytes>** is between 1..16
(2) NQC: timers.

- allows setting/getting of timers with 100th of a second resolution (fast mode) or 10th of a second resolution (default)
- 4 timers, numbered 0..3
- `ClearTimer(<n>)`
  - clears specified timer
  - `<n>` is between 0..3
- `x = Timer(<n>)`
  - returns the value of specified timer (for default resolution)
- `x = FastTimer(<n>)`
  - returns the value of specified timer (for 100th of a second resolution)
- `SetTimer(<n>,<value>)`
  - sets specified timer
  - `<value>` can be any constant or expression

(2) NQC: event handling.

- allows up to 16 events
- `SetEvent(<event>,<source>,<type>)`
  - configures an event
  - `<event>` is between 0..15
  - `<source>` is the source of the event (e.g., SENSOR_1)
  - `<type>` is one of: EVENT_TYPE_PRESSED, EVENT_TYPE_RELEASED, EVENT_TYPE_PULSE (indicates a toggle), EVENT_TYPE_EDGE, EVENT_TYPE_LOW (use `SetLowerLimit()` to set threshold), EVENT_TYPE_HIGH (use `SetUpperLimit()` to set threshold), EVENT_TYPE_NORMAL, EVENT_TYPE_MESSAGE
- `ClearEvent(<event>)`
  - clears configuration

(2) NQC: counters.

- 3 counters, numbered 0..2
- `ClearCounter(<n>)`
  - clears specified counter
- `IncCounter(<n>)`
  - increments specified counter
- `DecCounter(<n>)`
  - decrements specified counter
- `x = Counter(<n>)`
  - gets the value of specified counter

(2) NQC: data logging.

- `CreateDatalog(<size>)`
  - creates a data log for recording sensor readings, variable values and the system watch
  - `<size>` is the number of points to record; 0 clears the data log
- `AddToDatalog(<value>)`
  - adds a value to the data log
- `UploadDatalog(<start>,<count>)`
  - uploads the contents of the data log
  - to upload and print the content of the data log to the computer from the command-line:
    bash# nqc -datalog
    bash# nqc -datalog_full
(2) NQC: miscellaneous functions

- **Wait( <time> )**
  - to sleep
  - <time> is a value in 100ths of a second

- **SetRandomSeed( <n> )**
  - x = Random( <n> )
  - sets random number seed and generates/returns random number between 0..<n>

- **SelectProgram( <n> )**
  - sets the current program
  - <n> is between 0..4

- **x = Program()**
  - gets currently selected program

- **x = BatteryLevel()**
  - monitors the battery and returns the battery level in millivolts

- **SetWatch( <hours>,<minutes> )**
  - sets system clock
  - <hours> is between 0..23
  - <minutes> is between 0..59

- **x = Watch()**
  - gets value of system clock in minutes