robotics for beginners: using robot kits to build embodied agents

tutorial
AAMAS-03, Melbourne, Australia

presenters:

Professor Elizabeth Sklar, Columbia University, New York, USA
email: *sklar@cs.columbia.edu*
web: *http://www.cs.columbia.edu/˜sklar*

Professor Simon Parsons, City University of New York, USA
email: *s.d.parsons@csc.liv.ac.uk*
web: *http://www.csc.liv.ac.uk/˜sp*

this material is based in part on lecture notes by Andreas Birk, Maja Mataric and Michael Wooldridge. we gratefully acknowledge access to this material.

schedule.

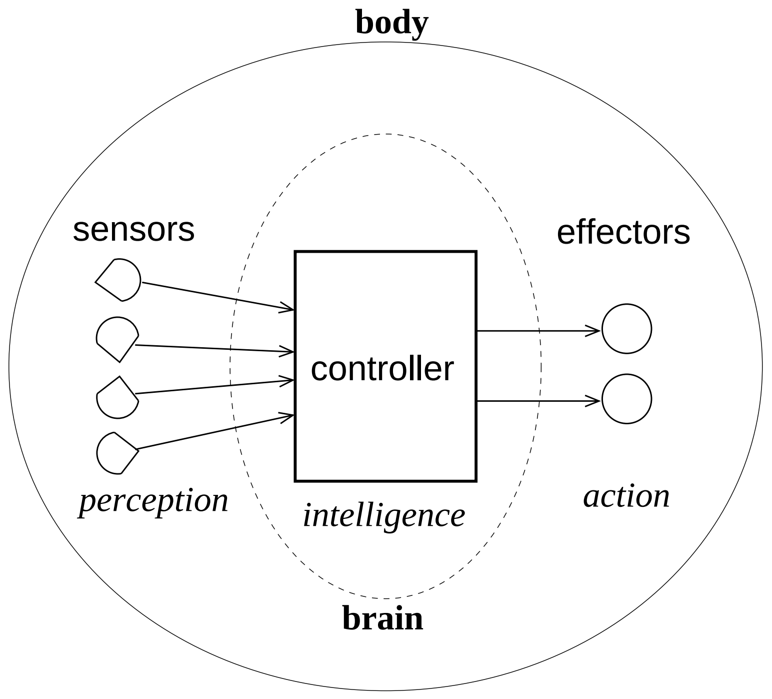
(1)	9.00am	-	10.00am:	autonomous agents and autonomous robotics.
	10.00am	-	10.30am:	break.
(2)	10.30am	-	12.00pm:	LEGO Mindstorms and Not-Quite C.
	12.00pm	-	1.00pm:	lunch.
(3)	1.00pm	-	2.00pm:	behaviour-based robotics.
(4)	2.00pm	-	3.00pm:	BDI architectures.
	3.00pm	-	3.30pm:	break.
(5)	3.30pm	-	4.30pm:	using robotics kits in undergraduate education.
	4.30pm	-	5.00pm:	discussion.

(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&Norvig, 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



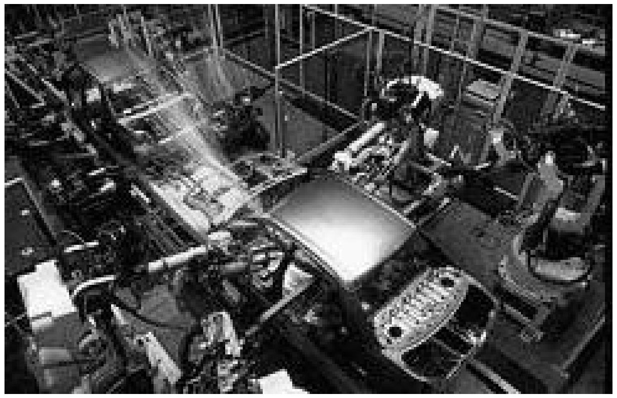
our canonical agent

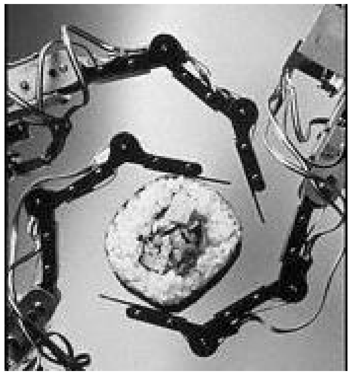
(1) a bit of robot history.

- the word *robot* came from the Czech word *robota*, which means *slave*
- used first by playwrite 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*

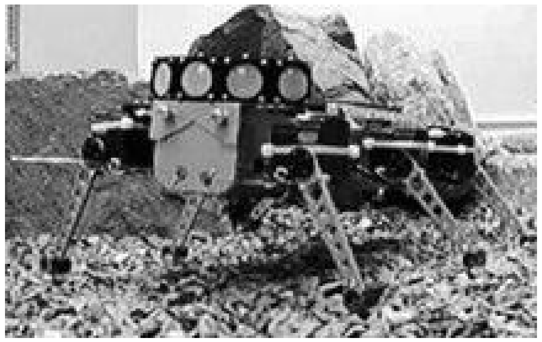
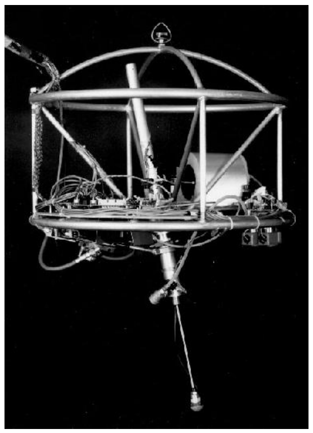




some manipulator robots

(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 $(roll, pitch, yaw)$
    - 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*

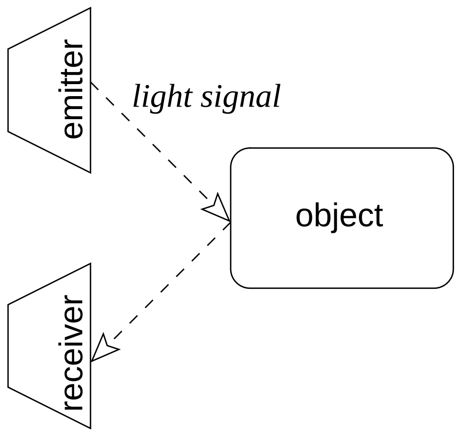
(1) sensors.

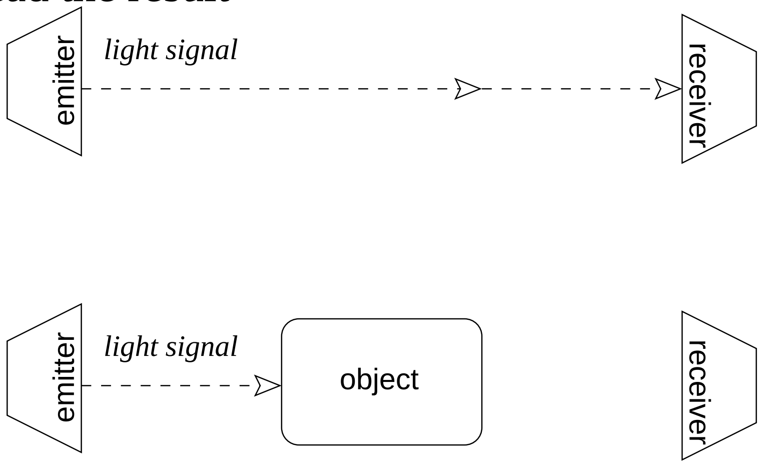
- ⇒ perception
  - *proprioceptive:* know where your joints/sensors are
  - *odometry:* know where you are
- function: to convert a physical property into an electronic signal which can be interpreted by the robot in a useful way

property being sensed	type of sensor
contact	bump, switch
distance	ultrasound, radar, infra red (IR)
light level	photo cell, camera
sound level	microphone
smell	chemical
temperature	thermal
inclination	gyroscope
rotation	encoder
pressure	pressure gauge
altitude	altimeter

(1) more on sensors.

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





reflectance *break-beam*

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

(1) 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.

(1) state.

- knowledge about oneself and one's environment
  - *kinematics* = study of correspondance 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?

(1) control.

- autonomy
- problem solving
- modeling
  - knowledge
  - representation
- control architectures
- deliberative control
- reactive control
- hybrid control

(1) autonomy.

- 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) problem solving: example.

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

operator	*preconditions*	*results*
$PUSH(obj, loc)$	$at(robot, obj) \wedge large(obj) \wedge$ $clear(obj) \wedge armempty()$	$at(obj, loc) \wedge$ $at(robot, loc)$
$CARRY(obj, loc)$	$at(robot, obj) \wedge small(obj)$	$at(obj, loc) \wedge$ $at(robot, loc)$
$WALK(loc)$	$none$	$at(robot, loc)$
$PICKUP(obj)$	$at(robot, obj)$	$holding(obj)$
$PUTDOWN(obj)$	$holding(obj)$	$\neg holding(obj)$
$PLACE(obj1, obj2)$	$at(robot, obj2) \wedge holding(obj1)$	$on(obj1, obj2)$

(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) 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 layers:
    1. reactive layer
    2. planner (deliberative layer)
    3. integration layer to combine them
    4. (but this is hard to do!)

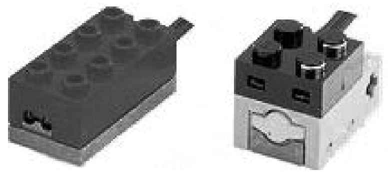
schedule.

(1)	9.00am	-	10.00am:	autonomous agents and autonomous robotics.
	10.00am	-	10.30am:	$\rightarrow$ **Break** $\leftarrow$
(2)	10.30am	-	12.00pm:	Not-Quite C and LEGO Mindstorms.
	12.00pm	-	1.00pm:	lunch.
(3)	1.00pm	-	2.00pm:	behaviour-based robotics.
(4)	2.00pm	-	3.00pm:	BDI architectures.
	3.00pm	-	3.30pm:	break.
(5)	3.30pm	-	4.30pm:	using robotics kits in undergraduate education.
	4.30pm	-	5.00pm:	discussion.

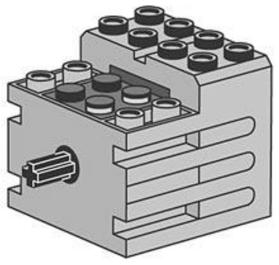
(2) LEGO Mindstorms.

- 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://www.baumfamily.org/nqc/index.html`
- a smattering of NQC follows
- basic command-line operation:

```
bash# nqc -d <rcx-program-file>
bash# nqc -firmward <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

type	meaning	description
`int`	pass by value	value can change inside function, but changes won’t be seen by caller
`int &`	pass by reference	value can change inside function, and changes will be seen by caller; only variables may be passed
`const int`	pass by value	value cannot be changed inside function; only constants may be passed
`const int &`	pass by reference	value cannot be changed inside function; value is read each time it is used

(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

operator	action
$abs()$ $sign()$	absolute value sign of operand
$++$, $--$	increment, decrement
$-$, ~, !	unary minus, bitwise negation, logical negation
$*$, $/$, $\%$, $+$, $-$	multiply, divide, modulo, addition, subtraction
$<<$, $>>$	left and right shift
$>$, $<$, $>=$, $<=$, $==$, $!=$	relational and equivalence operators
&, $\wedge$, $\|$	bitwise AND, XOR, OR
&&, $\|\|$	logical AND, OR
$=$ $+=$, $-=$, $*=$, $/=$, $\&=$, $\|=$	assignment operator shortcut assignment operators
$\|\|=$ $+-=$	set variable to absolute value of expression set variable to sign (-1,+1,0) of expression

(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`)[1]
- 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>`

[1] macro redefinition not allowed

(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>
  catch ( <catch-events> ) <handler>
  .
  .
  catch <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: counters.

- 3 counters, numbered 0..2[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] note that these overlap with global storage locations so these should be reserved if they are going to be used; see `#pragma reserve` description

(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[3]: `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

[3]a subset is shown

(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

exercise 1: build a simple line-following robot.

- build and program a simple line-following robot
- to be distributed at the tutorial:
  - CDROM with NQC
  - sample code
  - building plans

schedule.

(1)	9.00am	-	10.00am:	autonomous agents and autonomous robotics.
	10.00am	-	10.30am:	break
(2)	10.30am	-	12.00pm:	Not-Quite C and LEGO Mindstorms.
	12.00pm	-	1.00pm:	$\rightarrow$ **lunch.** $\leftarrow$
(3)	1.00pm	-	2.00pm:	behaviour-based robotics.
(4)	2.00pm	-	3.00pm:	BDI architectures.
	3.00pm	-	3.30pm:	break.
(5)	3.30pm	-	4.30pm:	using robotics kits in undergraduate education.
	4.30pm	-	5.00pm:	discussion.

(3) behaviour-based robotics.

- control
- behaviour-based systems
- expressing behaviours
- behavioural encoding
- representations
- behaviour coordination
- emergent behaviour

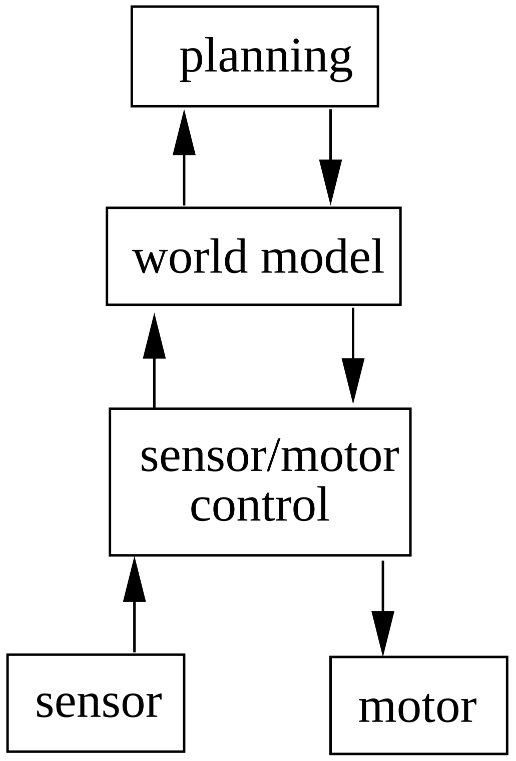
(3) control: models

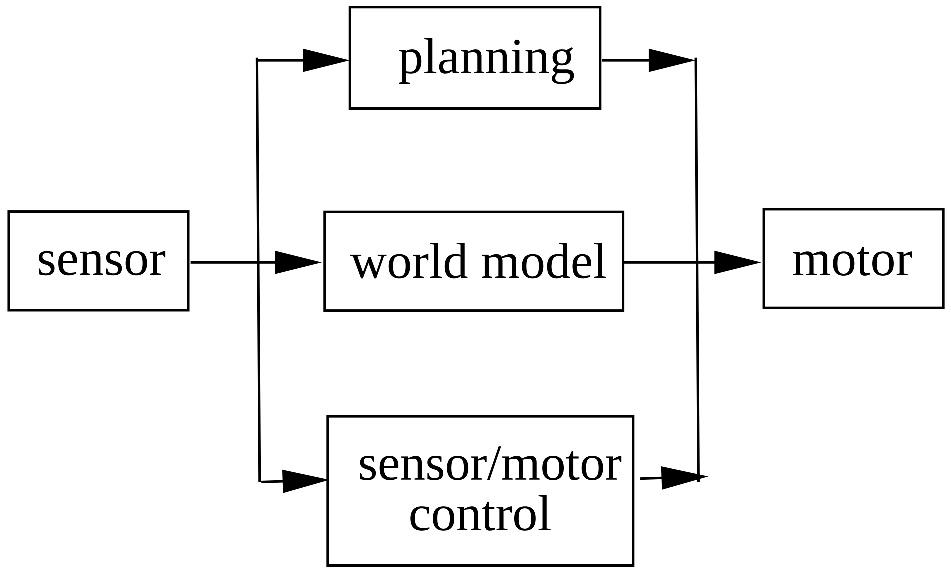
- we like to make a distinction between:
  - classic “model-based” AI
    - symbolic representations
  - neo “behaviour-based” AI
    - numeric representations
- classic models are “good old-fashioned AI”, in the tradition of McCarthy.
- behaviour-based models are “nouvelle AI” in the tradition of Brooks.
- (There are also hybrid models that combine aspects of both model-based and behaviour-based, but we have no time to cover them in detail here.)

(3) control: classic models.

- deliberative... sense, plan, act
  - functional decomposition
  - systems consist of sequential modules achieving independent functions
    - sense world
    - generate plan
    - translate plan into actions
- reactive architectures
  - task-oriented decomposition
  - systems consist of concurrently executed modules achieving specific tasks
    - avoid obstacle
    - follow wall

(3) control: two orthogonal flows.





(3) control: behaviour based systems.

- behaviours are the underlying module of the system
- behavioural decomposition
  - rather than a functional or a task-oriented decomposition
- systems consist of sequential modules achieving independent functions
- natural fit to robotic behaviour
  - generate a motor response from a given perceptual stimulus
  - basis in biological studies
  - biology is an inspiration for design
- abstract representation is avoided

(3) behaviour based systems: behaviour vs action.

behaviour is:

- based on dynamic processes
  - operating in parallel
  - lack of central control
  - fast couplings between sensors and motors
- exploiting emergence
  - side-effects from combined processes
  - using properties of the environment
- reactive

(3) behaviour-based systems: behaviour vs action.

action is:

- discrete in time
  - well defined start and end points
  - allows pre- and post-conditions
- avoidance of side-effects
  - only one (or a few) actions at a time
  - conflicts are undesired and avoided
- deliberative

actions are building blocks for behaviours.

(3) behaviour-based systems: properties.

- achieve specific tasks/goals
  - avoid others, find friend, go home
- typically execute concurrently
- can store state and be used to construct world models/representations
- can directly connect sensors to effectors
- can take inputs from other behaviours and send outputs to other behaviours
  - connection in networks
- typically higher-level than actions (go home, not turn left 45 degrees)
- typically closed loop, but extended in time
- when assembled into distributed representations, behaviours can be used to look ahead but at a time-scale comparable with the rest of the system

(3) behaviour-based systems: key properties.

- ability to act in real time
- ability to use representations to generate efficient (not only reactive) behaviour
- ability to use a uniform structure and representation throughout the system (so no intermediate layer)

(3) behaviour-based systems: challenges.

- how can representation be effectively distributed over the behaviour structure?
  - time scale must be similar to that of real-time components of the system
  - representation must use same underlying behaviour structure for all components of the system
- some components may be reactive
- not every component is involved with representational computation
- some systems use a simple representation
- as long as the basis is in behaviours and not rules, the system is a BBS

(3) behaviour-based systems: what are behaviours?

- behaviour: anything observable that the system/robot does
  - how do we distinguish internal behaviours (components of a BBS) and externally observable behaviours?
  - should we distinguish?
- reactive robots display desired external behaviours
  - avoiding
  - collecting cans
  - walking
- but controller consists of a collection of rules, possibly in layers
- BBS actually consist and are programmed in the behaviours, which are higher granularity, extended in time, capable of representation

(3) behaviour-based systems: expressing behaviours.

- behaviours can be expressed with various representations
- when a control system is being designed, the task is broken down into desired external behaviours
- those can be expressed with
  - functional notation
  - stimulus response (SR) diagrams
  - finite state machines/automata (FSA)
  - schema

(3) expressing behaviours: functional notation.

- mathematical model:
  - represented as triples (S, R, β)

    S = stimulus

    R = range of response

    β = behavioural mapping between S and R
- easily convert to functional languages like LISP

```
coordinate-behaviours [
   move-to-classroom ( detect-classroom-location ),
   avoid-objects ( detect-objects ),
   dodge-students ( detect-students ),
   stay-to-right-on-path ( detect-path ),
   defer-to-elders ( detect-elders )
] = motor-response
```

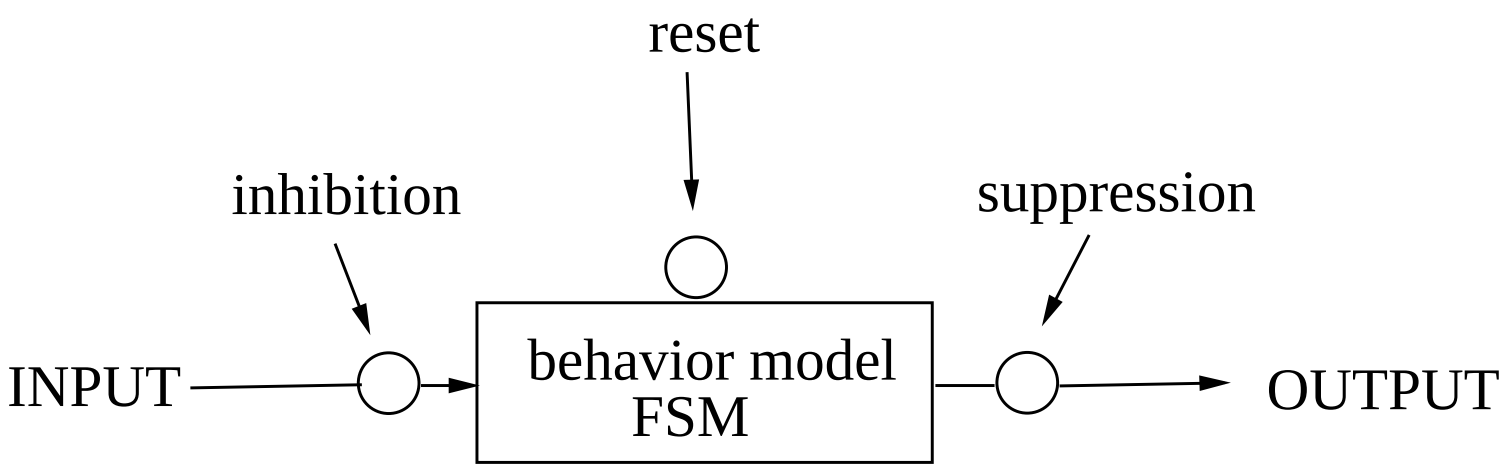
(3) expressing behaviours: FSA diagrams.

- states of the diagram can also be called behaviours, diagrams show sequences of behaviour transitions
- situated automata.
  - formalism for specifying FSAs that are situated [Kaelbling & Rosenschein, 1991]
  - task described in high-level logic expressions, as a set of goals and a set of operators that achieve (ach) and maintain (maint) the goals
  - once defined, tasks can be compiled into circuits (using special purpose languages), which are reactive

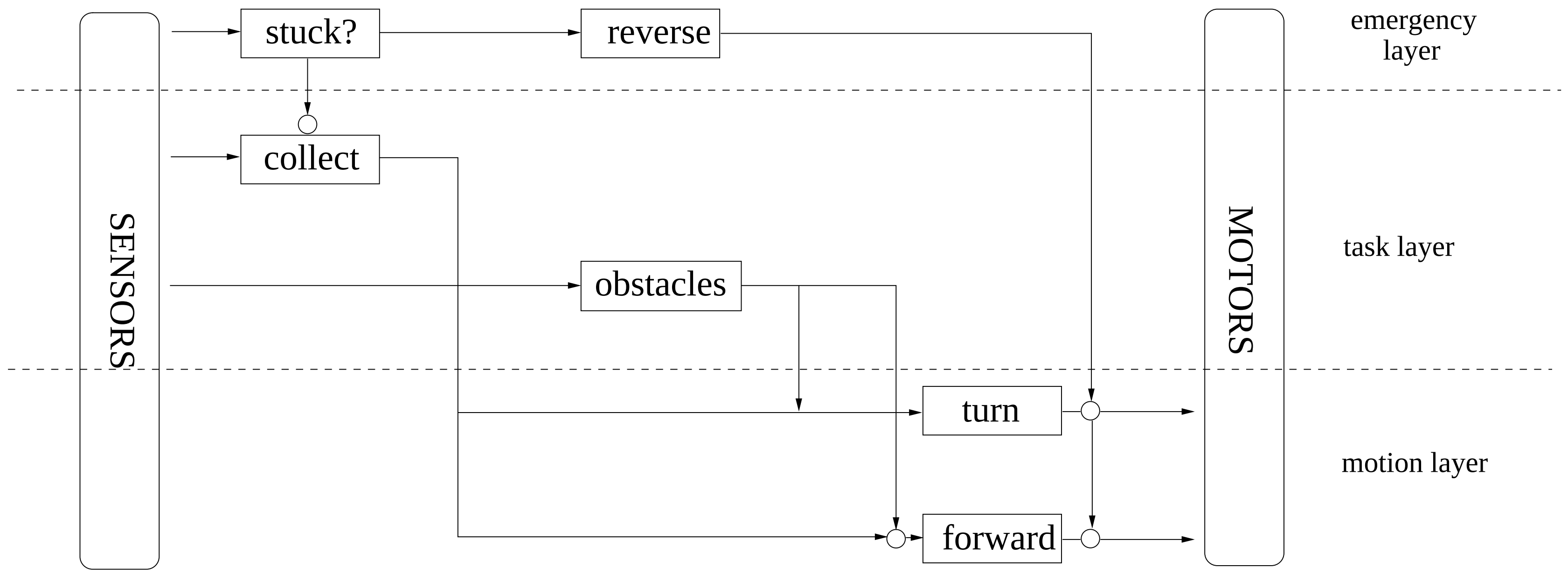
```
(defgoalr (ach in-classroom)
(if (not start-up)
    (maint (and (maint move-to-classroom)
                (maint avoid-objects)
                (maint dodge-students)
                (maint stay-to-right-on-path)
                (maint defer-to-elders)))))
```

(3) expressing behaviours: subsumption architecture.

- Rodney Brooks, 1986, MIT AI lab
- reactive elements
- behaviour-based elements
- layered approach based on levels of competence
- augmented finite state machine:



(3) expressing behaviours: subsumption architecture.



(3) expressing behaviours: formally.

- behavioural response in physical space has a strength and an orientation
- expressed as (S, R, β)
- S = stimulus, necessary but not sufficient condition to evoke a response (R); internal state can also be used
- β = behavioural mapping categories
  - null
  - discrete
  - continuous

(3) expressing behaviours: behavioural mapping.

- discrete encoding
  - expressed as a finite set of situation-response pairs/mappings
  - mappings often include rule-based form IF-THEN
  - examples:
    - ∗ Gapps [Kaelbling & Rosenschein]
    - ∗ subsumption language [Brooks]
- continuous encoding
  - instead of discretizing the input and output, a continuous mathematical function describes the input-output mapping
  - can be simple, time-varying, harmonic
  - examples:
    - ∗ potential field
    - ∗ schema
  - problems with local minima, maxima, oscillatory behaviour

(3) expressing behaviours: motor schemas.

- type of behaviour encoding
- based on schema theory
- provide large grain modularity
- distributed, concurrent schemas used
- based on neuroscience and cognitive science
- represented as vector fields
- decomposed into *assemblages* by fusion, not competition
- assemblages
  - recursively defined aggregations of behaviours or other assemblages
  - important abstractions for constructing behaviour-based robots

(3) expressing behaviours: schemas.

- representation
  - responses represented in uniform vector format
  - combination through cooperative coordination via vector summation
  - no predefined schema hierarchy
  - arbitration not used — gain values control behavioural strengths
- designing with schemas
  - characterize motor behaviours needed
  - decompose to most primitive level, use biological guidelines where appropriate
  - develop formulas to express reactions
  - conduct simple simulations
  - determine perceptual needs to satisfy motor schema inputs
  - design specific perpetual algorithms
  - integrate/test/evaluate/iterate

(3) behavioural encoding: strengths and weaknesses.

- strengths
  - support for parallelism
  - run-time flexibility
  - timeliness for development
  - support for modularity
- weaknesses
  - niche targetability
  - hardware retargetability
  - combination pitfalls (local minima, oscillations)

(3) behaviour coordination.

- BBS consist of collection of behaviours
- execution must be coordinated in a consistent fashion
- coordination can be
  - competitive
  - cooperative
  - combination of the two
- deciding what to do next.
  - action-selection problem
  - behaviour-arbitration problem

(3) behaviour coordination.

- competitive coordination.
  - perform arbitration (selecting one behaviour among a set of candidates)
    - priority-based: subsumption
    - state-based: discrete event systems
    - function-based: spreading of activation action selection
- cooperative coordination.
  - perform command fusion (combine outputs of multiple behaviours)
  - voting
  - fuzzy (formalized voting)
  - superposition (linear combinations)
    - potential fields
    - motor schemas
    - dynamical systems

(3) emergent behaviour: what is it?

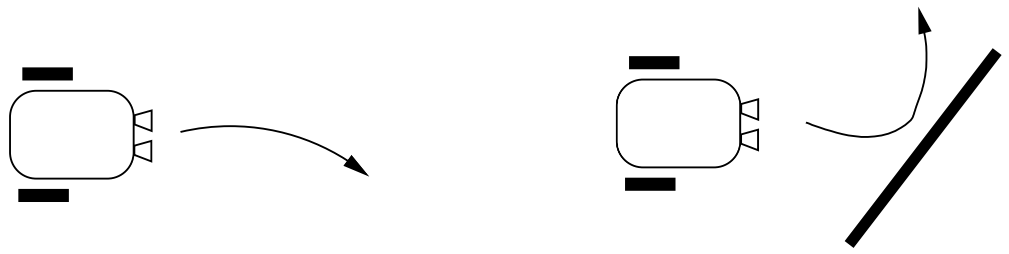
- important but not well-understood phenomenon
- often found in behaviour-based robotics
- robot behaviours “emerge” from
  - interactions of rules
  - interactions of behaviours
  - interactions of either with environment
- coded behaviour
  - in the programming scheme
- observed behaviour
  - in the eyes of the observer
  - emergence
- there is no one-to-one mapping between the two!

(3) emergent behaviour: how does it arise?

- is it magic?
  - sum is greater than the parts
  - emergent behaviour is more than the controller that produces it
- interaction and emergence.
  - interactions between rules, behaviours and environment
  - source of expressive power for a designer
  - systems can be designed to take advantage of emergent behaviour
- emergent flocking.
  - program multiple robots:
    - dont run into any other robot
    - dont get too far from other robots
    - keep moving if you can
  - when run in parallel on many robots, the result is flocking

(3) emergent behaviour: wall following.

coded behavior

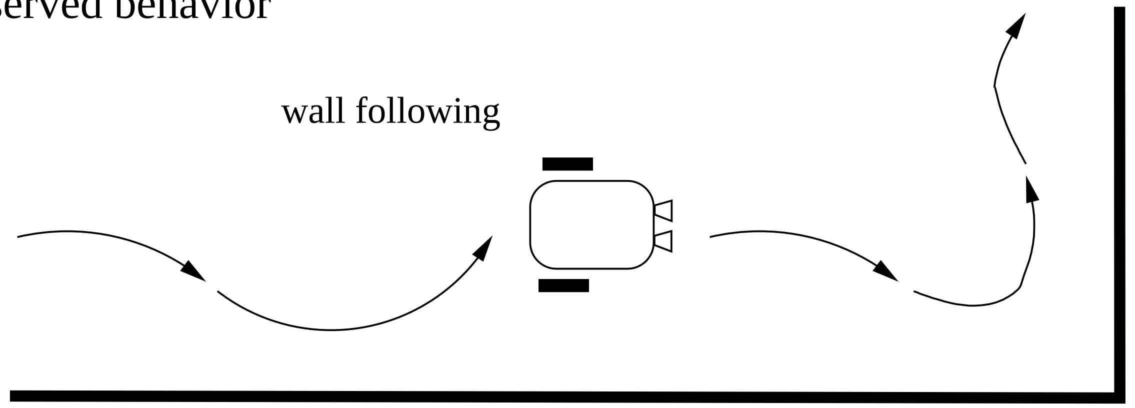


forward motion,
with slight right turn

obstacle avoidance

observed behavior

wall following



(3) emergent behaviour: wall following.

- can also be implemented with these rules:
  - if too far, move closer
  - if too close, move away
  - otherwise, keep on
- over time, in an environment with walls, this will result in wall-following
- is this emergent behaviour?
- it is argued yes because
  - robot itself is not aware of a wall, it only reacts to distance readings
  - concepts of “wall” and “following” are not stored in the robot’s controller
  - the system is just a collection of rules

(3) emergent behaviour: conditions on emergence.

- notion of emergence depends on two aspects:
  - existence of an external observer, to observe and describe the behaviour of the system
  - access to the internals of the controller itself, to verify that the behaviour is not explicitly specified anywhere in the system
- unexpected vs emergent.
  - some researchers say the above is not enough for behaviour to be emergent, because above is programmed into the system and the “emergence” is a matter of semantics
  - so emergence must imply something unexpected, something “surreptitiously discovered” by observing the system.
  - “unexpected” is highly subjective, because it depends on what the observer was expecting
  - naïve observers are often surprised!
  - informed observers are rarely surprised
- once a behaviour is observed, it is no longer unexpected
- is new behaviour then “predictable”?

exercise 2: build a behaviour-based robot.

- build and program a behaviour-based robot.

(4) BDI architecture.

- in this part of the tutorial, we take a detailed look at the BDI model of agency — an architecture for practical reasoning.
- this architecture was motivated to a great extent by Bratman's theory of plans, intentions, and practical reasoning.
- the BDI model has been implemented many times:
  - the Intelligent Resource-bounded Machine Architecture (IRMA);
  - the Procedural Reasoning System (PRS).
- the BDI model combines three components:
  - a *philosophical* component;
  - a *software architecture* component;
  - a *logical* component.
- we begin by introducing the Philosophy of BDI: Bratman's theory of human practical reasoning and the notion of an agent as an *intentional system*. . .

(4) BDI philosophy: the intentional stance.

- when explaining human activity, it is often useful to make statements such as:

  Betsy took her umbrella because she *believed* it was going to rain.
  Simon worked hard because he *wanted* to get tenure.

- these statements make use of a *folk psychology*, by which human behaviour is predicted and explained through the attribution of *attitudes*, such as believing and wanting (as in the above examples), hoping, fearing, and so on.
- the attitudes employed in such folk psychological descriptions are called the *intentional* notions.
- the philosopher Daniel Dennett coined the term *intentional system* to describe entities 'whose behaviour can be predicted by the method of attributing belief, desires and rational acumen'.

  'A *first-order* intentional system has beliefs and desires (etc.) but no beliefs and desires *about* beliefs and desires. . . . A *second-order* intentional system is more sophisticated; it has beliefs and desires (and no doubt other intentional states) about beliefs and desires (and other intentional states) — both those of others and its own'.

(4) BDI philosophy: the intentional stance.

- is it legitimate or useful to attribute beliefs, desires, and so on, to computer systems?
- McCarthy argued that there are occasions when the *intentional stance* is appropriate:

> ‘To ascribe *beliefs*, *free will*, *intentions*, *consciousness*, *abilities*, or *wants* to a machine is <u>legitimate</u> when such an ascription expresses the same information about the machine that it expresses about a person. It is <u>useful</u> when the ascription helps us understand the structure of the machine, its past or future behaviour, or how to repair or improve it. It is perhaps never <u>logically required</u> even for humans, but expressing reasonably briefly what is actually known about the state of the machine in a particular situation may require mental qualities or qualities isomorphic to them. Theories of belief, knowledge and wanting can be constructed for machines in a simpler setting than for humans, and later applied to humans. Ascription of mental qualities is <u>most straightforward</u> for machines of known structure such as thermostats and computer operating systems, but is <u>most useful</u> when applied to entities whose structure is incompletely known’.

(4) BDI philosophy: the intentional stance.

- what objects can be described by the intentional stance?
- as it turns out, more or less anything can… consider a light switch:

  > 'It is perfectly coherent to treat a light switch as a (very cooperative) agent with the capability of transmitting current at will, who invariably transmits current when it believes that we want it transmitted and not otherwise; flicking the switch is simply our way of communicating our desires'. (Yoav Shoham)

- but most adults would find such a description absurd!
- why is this?
- the answer seems to be that while the intentional stance description is consistent,

  > '… it does not *buy us anything*, since we essentially understand the mechanism sufficiently to have a simpler, mechanistic description of its behaviour'. (Yoav Shoham)

(4) BDI philosophy: the intentional stance.

- put crudely, the more we know about a system, the less we need to rely on animistic, intentional explanations of its behaviour.
- but with very complex systems, a mechanistic, explanation of its behaviour may not be practicable.
- *as computer systems become ever more complex, we need more powerful abstractions and metaphors to explain their operation — low level explanations become impractical.*

  the intentional stance is such an abstraction.

(4) BDI philosophy: practical reasoning

- practical reasoning is reasoning directed towards actions — the process of figuring out what to do:

  practical reasoning is a matter of weighing conflicting considerations for and against competing options, where the relevant considerations are provided by what the agent desires/values/cares about and what the agent believes.

- distinguish practical reasoning from *theoretical reasoning*. Theoretical reasoning is directed towards beliefs.
- human practical reasoning consists of two activities:
  - *deliberation*

    deciding *what* state of affairs we want to achieve;
  - *means-ends reasoning*

    deciding *how* to achieve these states of affairs.
- the outputs of deliberation are *intentions*.

(4) practical reasoning: intentions.

1. intentions pose problems for agents, who need to determine ways of achieving them.

   If I have an intention to ϕ, you would expect me to devote resources to deciding how to bring about ϕ.

2. intentions provide a “filter” for adopting other intentions, which must not conflict.

   If I have an intention to ϕ, you would expect me to adopt an intention ψ such that ϕ and ψ are mutually exclusive.

3. agents track the success of their intentions, and are inclined to try again if their attempts fail.

   if an agent’s first attempt to achieve ϕ fails, then all other things being equal, it will try an alternative plan to achieve ϕ.

4. agents believe their intentions are possible.

   that is, they believe there is at least some way that the intentions could be brought about.

(4) practical reasoning: intentions.

5. agents do not believe they will not bring about their intentions.

   It would not be rational of me to adopt an intention to ϕ if I believed ϕ was not possible.

6. under certain circumstances, agents believe they will bring about their intentions.

   It would not normally be rational of me to believe that I would *bring my intentions about; intentions can* fail. *Moreover, it does not make sense that if I believe ϕ is inevitable I would adopt it as an intention.*

7. agents need not intend all the expected side effects of their intentions.

   If I believe $\phi \Rightarrow \psi$ and I intend that ϕ, I do not necessarily intend ψ also. (Intentions are not closed under implication.)

   This last problem is known as the *side effect* or *package deal* problem. I may believe that going to the dentist involves pain, and I may also intend to go to the dentist — but this does not imply that I intend to suffer pain!

(4) practical reasoning: intentions.

- notice that intentions are much stronger than mere desires:

  > my desire to play basketball this afternoon is merely a potential influencer of my conduct this afternoon. It must vie with my other relevant desires [. . .] before it is settled what I will do. In contrast, once I intend to play basketball this afternoon, the matter is settled: I normally need not continue to weigh the pros and cons. When the afternoon arrives, I will normally just proceed to execute my intentions.

- the added strength comes from some kind of *commitment* to the subject of the intention.
- there are computational advantages to this.

(4) BDI software architecture: implementing BDI Agents.

- a first pass at an implementation of BDI agent:

```
Agent Control Loop Version 1
1.  while true
2.      observe the world;
3.      update internal world model;
4.      deliberate about what intention
          to achieve next;
5.      use means-ends reasoning to get
          a plan for the intention;
6.      execute the plan
7.  end while
```

- (we will not be concerned with stages (2) or (3).)

(4) BDI software architecture: rationality.

- problem: deliberation and means-ends reasoning processes are not instantaneous. They have a *time cost*.
- suppose the agent starts deliberating t_0, begins means-ends reasoning at t_1, and begins executing the plan at time t_2.

  time to deliberate is

  $$t_{deliberate} = t_1 - t_0$$

  and time for means-ends reasoning is

  $$t_{me} = t_2 - t_1$$
- further suppose that deliberation is *optimal* in that if it selects some intention to achieve, then this is the best thing for the agent. (Maximises expected utility.)
- so at time t_1, the agent has selected an intention to achieve that would have been optimal *if it had been achieved at* t_0.

  but unless $t_{deliberate}$ is vanishingly small, then the agent runs the risk that the intention selected is no longer optimal by the time the agent has fixed upon it.

  this is *calculative rationality*.

(4) BDI software architecture: rationality.

- deliberation is only half of the problem: the agent still has to determine *how* to achieve the intention.
- so, this agent will have overall optimal behaviour in the following circumstances:
  1. when deliberation and means-ends reasoning take a vanishingly small amount of time; or
  2. when the world is guaranteed to remain static while the agent is deliberating and performing means-ends reasoning, so that the assumptions upon which the choice of intention to achieve and plan to achieve the intention remain valid until the agent has completed deliberation and means-ends reasoning; or
  3. when an intention that is optimal when achieved at time t_0 (the time at which the world is observed) is guaranteed to remain optimal until time t_2 (the time at which the agent has found a course of action to achieve the intention).

(4) BDI software architecture: rationality.

- let's make the algorithm more formal.

```
Agent Control Loop Version 2
1.  $B := B_0$; /* initial beliefs */
2.  while true do
3.      get next percept $\rho$;
4.      $B := brf(B, \rho)$;
5.      $I := deliberate(B)$;
6.      $\pi := plan(B, I)$;
7.      $execute(\pi)$
8.  end while
```

- we can now develop this in a number of ways.

(4) BDI software architecture: deliberation.

- how does an agent deliberate?
  - begin by trying to understand what the *options* available to it are;
  - *choose between them*, and *commit* to some.

  Chosen options are then intentions.
- The $deliberate$ function can be decomposed into two distinct functional components:
  - *option generation*
    in which the agent generates a set of possible alternatives; and
  - *filtering*
    in which the agent chooses between competing alternatives, and commits to achieving them.
- represent option generation via a function, $options$, which takes the agent's current beliefs and current intentions, and from them determines a set of options (= *desires*).

(4) BDI software architecture: deliberation.

- in order to select between competing options, an agent uses a *filter* function.

```
Agent Control Loop Version 3
1.
2.   $B := B_0$;
3.   $I := I_0$;
4.   while true do
5.       get next percept $\rho$;
6.       $B := brf(B, \rho)$;
7.       $D := options(B, I)$;
8.       $I := filter(B, D, I)$;
9.       $\pi := plan(B, I)$;
10.      $execute(\pi)$
11.  end while
```

(4) BDI software architecture: commitment strategies.

- how long should an agent maintain an intention? How *committed* should it be?

  Some time in the not-so-distant future, you are having trouble with your new household robot. You say “Willie, bring me a beer.” The robot replies “OK boss.” Twenty minutes later, you screech “Willie, why didn’t you bring me that beer?” It answers “Well, I intended to get you the beer, but I decided to do something else.” Miffed, you send the wise guy back to the manufacturer, complaining about a lack of commitment. After retrofitting, Willie is returned, marked “Model C: The Committed Assistant.” Again, you ask Willie to bring you a beer. Again, it accedes, replying “Sure thing.” Then you ask: “What kind of beer did you buy?” It answers: “Genessee.” You say “Never mind.” One minute later, Willie trundles over with a Genessee in its gripper. This time, you angrily return Willie for overcommitment.

(4) BDI software architecture: commitment strategies.

After still more tinkering, the manufacturer sends Willie back, promising no more problems with its commitments. So, being a somewhat trusting customer, you accept the rascal back into your household, but as a test, you ask it to bring you your last beer. Willie again accedes, saying “Yes, Sir.” (Its attitude problem seems to have been fixed.) The robot gets the beer and starts towards you. As it approaches, it lifts its arm, wheels around, deliberately smashes the bottle, and trundles off. Back at the plant, when interrogated by customer service as to why it had abandoned its commitments, the robot replies that according to its specifications, it kept its commitments as long as required — commitments must be dropped when fulfilled or impossible to achieve. By smashing the bottle, the commitment became unachievable (Philip Cohen).

(4) BDI software architecture: commitment strategies.

- the mechanism an agent uses to determine when and how to drop intentions is a *commitment strategy*.
- the following three commitment strategies are commonly discussed in the literature of rational agents:
  - *blind commitment*
    a blindly committed agent will continue to maintain an intention until it believes the intention has actually been achieved. Blind commitment is also sometimes referred to as *fanatical* commitment.
  - *single-minded commitment*
    a single-minded agent will continue to maintain an intention until it believes that either the intention has been achieved, or else that it is no longer possible to achieve the intention.
  - *open-minded commitment*
    an open-minded agent will maintain an intention as long as it is still believed possible.

```
Agent Control Loop Version 4

    $B := B_0$;
    $I := I_0$;
    while true do
        get next percept $\rho$;
        $B := brf(B, \rho)$;
        $D := options(B, I)$;
        $I := filter(B, D, I)$;
        $\pi := plan(B, I)$;
        while not $empty(\pi)$ do
            $\alpha := hd(\pi)$;
            $execute(\alpha)$;
            $\pi := tail(\pi)$;
            get next percept $\rho$;
            $B := brf(B, \rho)$;
            if not $sound(\pi, I, B)$ then
                $\pi := plan(B, I)$
            end-if
        end-while
    end-while
```

- an agent has commitment both to *ends* (i.e., the state of affairs it wishes to bring about), and *means* (i.e., the mechanism via which the agent wishes to achieve the state of affairs).
- currently, our agent control loop is overcommitted, both to means and ends.
- modification: *replan* if ever a plan goes wrong.

(4) BDI software architecture: intention reconsideration.

```
Agent Control Loop Version 5
⋮
10.     while not $empty(\pi)$
                or $succeeded(I, B)$
                or $impossible(I, B)$) do
11.         $\alpha := hd(\pi)$;
12.         $execute(\alpha)$;
13.         $\pi := tail(\pi)$;
14.         get next percept $\rho$;
15.         $B := brf(B, \rho)$;
16.         if not $sound(\pi, I, B)$ then
17.             $\pi := plan(B, I)$
18.         end-if
19.     end-while
20. end-while
```

- still overcommitted to intentions: never stops to consider whether or not its intentions are appropriate.
- modification: stop to determine whether intentions have succeeded or whether they are impossible.
- *single-minded commitment.*

(4) BDI software architecture: intention reconsideration.

- our agent gets to reconsider its intentions once every time around the outer control loop, i.e., when:
  - it has completely executed a plan to achieve its current intentions; or
  - it believes it has achieved its current intentions; or
  - it believes its current intentions are no longer possible.
- this is limited in the way that it permits an agent to *reconsider* its intentions.
- modification: reconsider intentions after executing every action.

(4) BDI software architecture: intention reconsideration.

```
Agent Control Loop Version 6
⋮
10.     while not ($empty(\pi)$
                or $succeeded(I, B)$
                or $impossible(I, B)$) do
11.         $\alpha := hd(\pi)$;
12.         $execute(\alpha)$;
13.         $\pi := tail(\pi)$;
14.         get next percept $\rho$;
15.         $B := brf(B, \rho)$;
16.         $D := options(B, I)$;
17.         $I := filter(B, D, I)$;
18.         if not $sound(\pi, I, B)$ then
19.             $\pi := plan(B, I)$
20.         end-if
21.     end-while
22. end-while
```

(4) BDI software architecture: intention reconsideration.

- but intention reconsideration is *costly*!
- a dilemma:
  - an agent that does not stop to reconsider its intentions sufficiently often will continue attempting to achieve its intentions even after it is clear that they cannot be achieved, or that there is no longer any reason for achieving them;
  - an agent that *constantly* reconsiders its attentions may spend insufficient time actually working to achieve them, and hence runs the risk of never actually achieving them.
- solution: incorporate an explicit *meta-level control* component, that decides whether or not to reconsider.

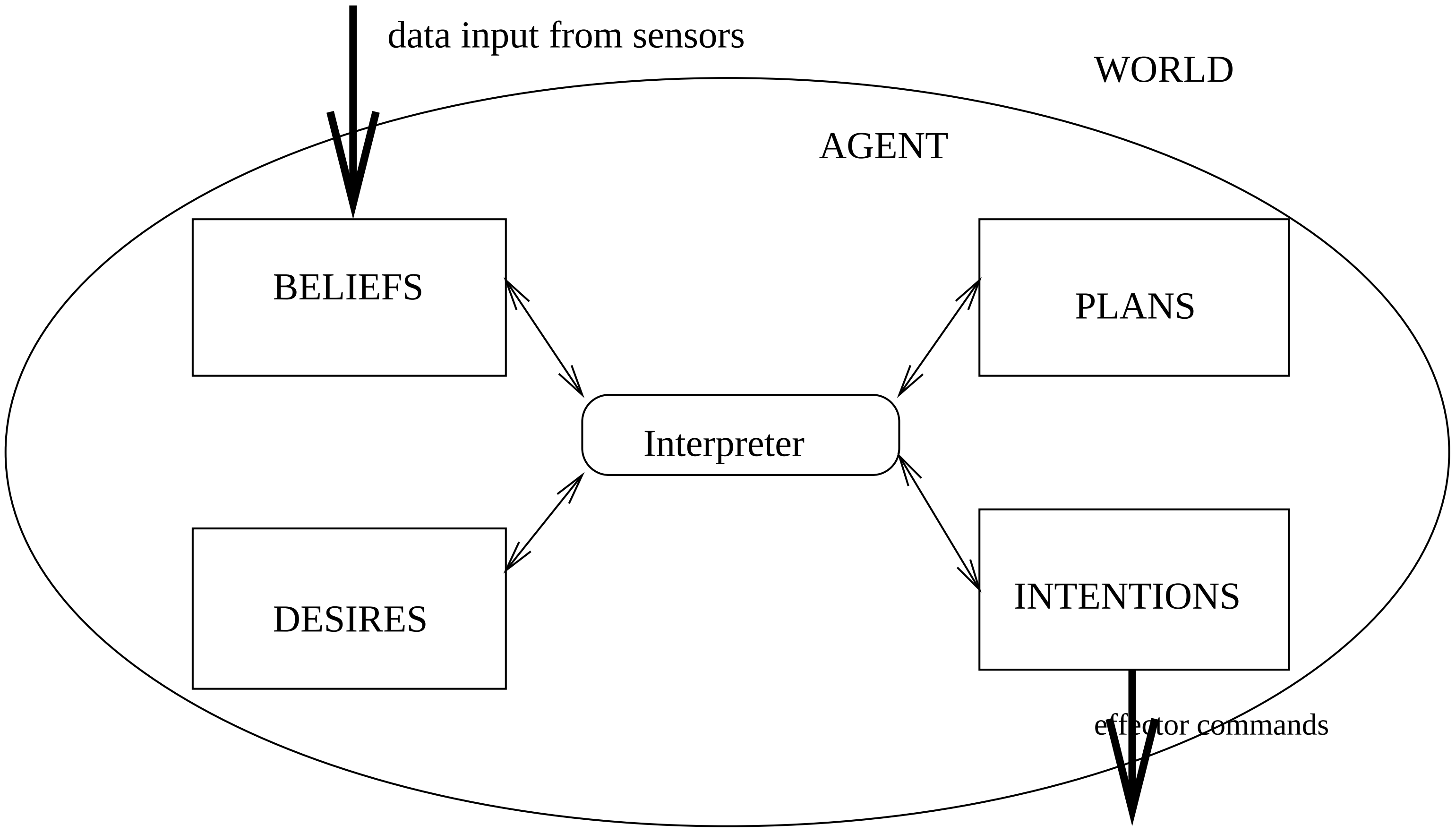
(4) BDI software architecture: meta-level control.

```
Agent Control Loop Version 7
⋮
10.     while not ($empty(\pi)$ or $succeeded(I, B)$
                or $impossible(I, B)$) do
11.         $\alpha := hd(\pi)$;
12.         $execute(\alpha)$;
13.         $\pi := tail(\pi)$;
14.         get next percept $\rho$;
15.         $B := brf(B, \rho)$;
16.         if $reconsider(I, B)$ then
17.             $D := options(B, I)$;
18.             $I := filter(B, D, I)$;
19.         end-if
20.         if not $sound(\pi, I, B)$ then
21.             $\pi := plan(B, I)$
22.         end-if
23.     end-while
24. end-while
```

(4) BDI software architecture: implemented systems.

- we now make the discussion even more concrete by introducing an actual agent architecture: the PRS.
- in the PRS, each agent is equipped with a *plan library*, representing that agent's *procedural knowledge*: knowledge about the mechanisms that can be used by the agent in order to realise its intentions.
- the options available to an agent are directly determined by the plans an agent has: an agent with no plans has no options.
- in addition, PRS agents have explicit representations of beliefs, desires, and intentions, as above.

(4) BDI software architecture: the PRS.



(4) BDI software architecture: plans in PRS.

- each plan in the PRS has:
  - a *body*, which represents the ‘program’ part of the plan;
  - an *invocation condition*, which specifies when the plan becomes ‘active’ — active plans in the PRS correspond to *desires*;
  - a *pre-condition*, which specifies what must be true in order for the plan to be successfully executed.
- let:
  - Π be a *plan library* for the agent;
  - $inv(\pi)$ be invocation condition for plan $\pi \in \Pi$;
  - $pre(\pi)$ be pre-condition for plan $\pi \in \Pi$.

(4) BDI software architecture: option generation in PRS.

- the function *options*:

```
function $options(b, d, i)$
begin
    $options := \emptyset$
    for $\pi \in \Pi$ do
        if $unifies(inv(\pi), b)$ then
            if $unifies(pre(\pi), b)$ then
                if $unifies(post(\pi), I)$ then
                    $options := options \cup \{\pi\}$
                end-if
            end-if
        end-if
    end-for
    return $options$
end
```

(4) BDI software architecture: deliberation in PRS.

- the function $filter$:

```
function $deliberate(b, d, i)$
begin
    if $\#d \leq 1$ then
        return $d$
    else
        return $random(d)$
    end-if
end
```

- we can add make more sophisticated deliberation to our agent by adding *meta-level plans*.

exercise 3: build a BDI-controlled robot.

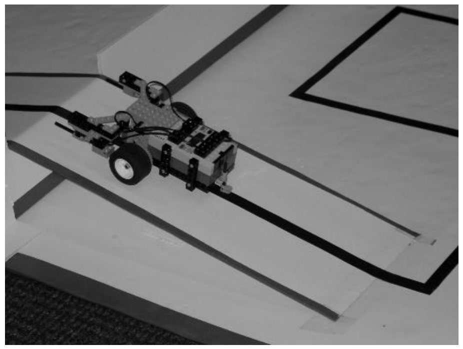
- build and program a BDI-controlled robot.

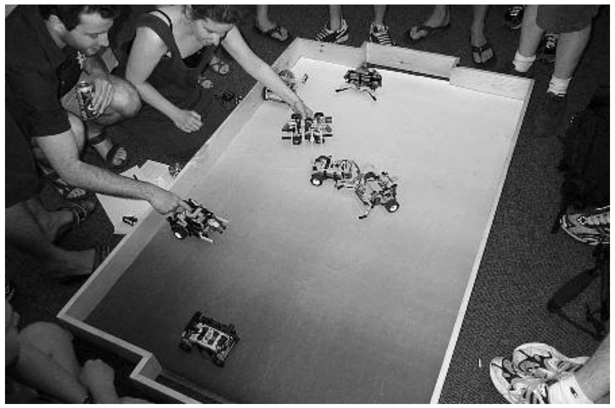
schedule.

(1)	9.00am	-	10.00am:	autonomous agents and autonomous robotics.
	10.00am	-	10.30am:	break
(2)	10.30am	-	12.00pm:	Not-Quite C and LEGO Mindstorms.
	12.00pm	-	1.00pm:	lunch.
(3)	1.00pm	-	2.00pm:	behaviour-based robotics.
(4)	2.00pm	-	3.00pm:	BDI architectures.
	3.00pm	-	3.30pm:	$\rightarrow$ **break.** $\leftarrow$
(5)	3.30pm	-	4.30pm:	using robotics kits in undergraduate education.
	4.30pm	-	5.00pm:	discussion.

(5) using robotics kits in undergraduate education.

- *educational robotics* — the use of robotics as a hands-on medium for learning, not just about robotics, but about other topics as well
- courses
  - introductory robotics
  - artificial intelligence
  - autonomous agents and multi agent systems
  - introductory programming
- contests/projects
  - line-following — maze, obstacle course
  - soccer
  - dance
  - *ala* RoboCup





(5) course structure.

- lectures and labs
- robotics component is project-based
- students work in groups (of 2-4)
- groups leave a deposit for the kits :-)
- this is a *science*
  - conduct experiments
  - record results
  - write up lab reports
- robotics are introduced as a means to demonstrate technical components of the course
- give them something to start with and they will run with it
- contests are a great motivator
- dance project is optional (e.g., “extra credit”)

(5) cautions.

- group work is *never* balanced
  - make them submit *individual* project reports
  - ask each group member to tell you what they did and what the others did; how everybody contributed (as part of their report)
- contests can motivate but can get in the way of learning the science
  - emphasize configuration management
  - emphasize thoughtful and structured experimentation
  - deadlines lead to hacking
- everybody will want to take your course
  - set limits if you can
- equipment will get lost/broken
  - get a deposit for every piece of equipment you lend out
  - give them a parts list with the kit you distribute (they don't need all 700 parts!)
- batteries cost money

(5) recommendations.

- learning to communicate
  - make them WRITE
  - make them TALK
- plan curricular topics carefully
  - depth
  - breadth
- hold contests in public
  - let them invite their friends
  - invite your colleagues
  - take pictures!

(5) resources.

- NQC download and documentation:
  `http://www.baumfamily.org/nqc/index.html`
- introduction to robotics web page:
  `http://www.cs.columbia.edu/˜sklar/teaching/spring2001/mc375/`
- artificial intelligence web pages:
  `http://www.cs.columbia.edu/˜sp/4701-2.html`
  `http://www.sci.brooklyn.cuny.edu/˜parsons/courses/cis32-fall2002/`
- educational robotics resource web page:
  `http://agents.cs.columbia.edu/˜er`
  (contains MANY links to course web pages and Mindstorms resource sites)

schedule.

(1)	9.00am	-	10.00am:	autonomous agents and autonomous robotics.
	10.00am	-	10.30am:	break
(2)	10.30am	-	12.00pm:	Not-Quite C and LEGO Mindstorms.
	12.00pm	-	1.00pm:	lunch.
(3)	1.00pm	-	2.00pm:	behaviour-based robotics.
(4)	2.00pm	-	3.00pm:	BDI architectures.
	3.00pm	-	3.30pm:	break.
(5)	3.30pm	-	4.30pm:	using robotics kits in undergraduate education.
	4.30pm	-	5.00pm:	$\rightarrow$ **discussion.** $\leftarrow$