Computer Science Education
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/ncse20

Robotics for computer scientists: what's the big idea?

David S. Touretzky[a]
[a] Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA.
Published online: 28 Oct 2013.

To cite this article: David S. Touretzky , Computer Science Education (2013): Robotics for computer scientists: what's the big idea?, Computer Science Education, DOI: 10.1080/08993408.2013.847226

To link to this article: http://dx.doi.org/10.1080/08993408.2013.847226

Computer Science Education, 2013
http://dx.doi.org/10.1080/08993408.2013.847226

Routledge
Taylor & Francis Group

Robotics for computer scientists: what's the big idea?

David S. Touretzky*

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA

(*Received 28 May 2013; accepted 3 September 2013*)

Modern robots, like today's smartphones, are complex devices with intricate software systems. Introductory robot programming courses must evolve to reflect this reality, by teaching students to make use of the sophisticated tools their robots provide rather than reimplementing basic algorithms. This paper focuses on teaching with Tekkotsu, an open source robot application development framework designed specifically for education. But, the curriculum described here can also be taught using ROS, the Robot Operating System that is now widely used for robotics research.

Keywords: robotics; Tekkotsu; ROS (Robot Operating System); curriculum design

1. Introduction

Outside of our most elite universities, robotics courses offered to computer science majors today use hardware and software that were designed for middle school children. The undergraduates in these courses are limited to programming simple reactive behaviors rather than exploring the big ideas that underlie intelligent perception and action (Touretzky, 2010).

This paper presents a vision of what an introductory robotics course for computer science undergraduates should look like. I have been offering a course based on these principles at Carnegie Mellon for the past eight years, and have worked with colleagues at several Historically Black Colleges and Universities (HBCUs) in a consortium called the ARTSI Alliance (Williams et al., 2008) to develop similar courses at their schools (Boonthum-Denecke, Touretzky, Jones, Humphries, & Caldwell, 2011). The software framework, Tekkotsu, and the curriculum materials used in these courses are freely available at wiki.Tekkotsu.org. Some other schools offering serious robotics courses are using ROS, the widely used Robot Operating System from Willow Garage (Quigley et al., 2009). Tekkotsu and ROS are both free,

*Email: dst@cs.cmu.edu

open source projects, and have many similarities. But they also differ in some important respects, which are discussed below.

Unlike most other areas of computer science where students need only a laptop to participate, robotics courses are *laboratory* courses requiring access to specialized equipment for substantial periods of time. The main impediment to the maturation of robotics education for CS undergraduates has been the scarcity of appropriate equipment.

A decade ago, the Sony AIBO robot dog was an ideal platform for teaching classic robotics topics such as computer vision, path planning, landmark-based navigation, and forward and inverse kinematics (Veloso, Rybski, Lenser, Chernova, & Vail, 2006). It has taken years to recover from Sony's surprise 2006 exit from the robotics market. But, reasonably priced and capable mobile robots – albeit without the charm of the AIBO – are again available today. The advent of a new robotics competition specifically for computer science students (Carnegie Mellon University, 2013) will help publicize the existence of these robots. And the development of improved software frameworks means that algorithms and representations once considered graduate-level material are now accessible to undergraduates. Computer science educators who take advantage of these tools can lead their students far beyond the reactive line tracking and wall following exercises they undertook in middle or high school.

2. Essential questions and big ideas

A first course in robotics should introduce students to the "essential questions" and "big ideas" of the field (Wiggins & McTighe, 2005). Table 1 gives a list of 10 such questions, the big ideas they introduce, and their underlying technologies. An earlier version of this list was presented in Touretzky (2012). A robotics survey course can discuss these big ideas, but real mastery requires hands-on experimentation. Section 5 will discuss current options for hardware platforms suitable for an undergraduate robotics lab.

An introductory robotics elective should not assume prior experience with robots, but it is essential that students be competent programmers and debuggers. Thus, it is important to distinguish between an introductory robotics course and an introductory *programming* course that happens to use robots. The ACM/IEEE Joint Task Force on Computing Curricula refers to the latter as a "thematically-focused introductory course" and cites two other popular themes: computational biology and digital media manipulation (Joint Task Force on Computing Curricula, 2013). Any of these may be effective ways to kindle student interest in computing, but a course that uses robots to introduce such elementary concepts as variables, conditionals, and loops can cover very little of the content from Table 1.

Table 1. Essential questions and big ideas that capture the essence of robotics.

Essential question	Big idea	Underlying technologies
1. How do robots know what to do?	Autonomous robot behaviors are **mechanisms** constructed from carefully designed algorithms and representations	State machines; event-based architectures
2. How do robots see the world?	Robots use sophisticated but imperfect **computer vision algorithms** to deduce real world object representations from arrays of pixels	Hough transforms; AprilTags; object recognition algorithms such as SIFT and SURF; face detection algorithms; etc.
3. How do robots know where they are?	Robots estimate their position in the world using a combination of **odometry**, **visual landmarks**, and other types of sensor information	Particle filters; SLAM (Simultaneous Localization and Mapping) algorithms
4. How do robots know where to go?	Robots navigate through the world using a **path planner** to search for routes around obstacles	Path planning algorithms such as RRTs (Rapidly-exploring Random Trees)
5. How do robots control their bodies?	Robots describe their bodies as **kinematic trees** and use **kinematics solvers** to translate between joint angles and body coordinates	Kinematic description files; Denavit-Hartenberg conventions; forward and inverse kinematics solvers
6. What can we do when a robot becomes too complex for one person to fully understand it?	Robots are complex software systems that employ standard **abstraction** and **software engineering** techniques to **manage complexity**	Modular design; coding standards; class libraries; documentation generators
7. How do we calculate the quantities needed to make a robot function?	Geometry, trigonometry, and linear algebra are the **mathematical underpinnings** of much of robotics	Software libraries for linear algebra, angular arithmetic, quaternions, etc
8. How can robots solve complex problems?	Robots use **task planning** to search a space of **world states** to find a path to a goal state	Task planners; domain description languages; plan execution and monitoring architectures
9. How should robots behave around people?	Successful **human–robot interaction** requires awareness of humans in the environment and adherence to social conventions such as not following too closely	Human-tracking tools such as the Microsoft Kinect; face and gesture recognition software; speech recognition systems; natural language dialog systems
10. How can robots work together?	**Inter-robot communication** and **multi-robot coordination** algorithms allow robots to collaborate	Communication primitives; multi-robot planners

3. Software frameworks: Tekkotsu and ROS

Tekkotsu (a Japanese word for *framework*) is an open source robotics application development framework based on C++, with an extensive collection of GUI tools written in Java for portability (Tira-Thompson & Touretzky, 2011). Tekkotsu was originally developed to facilitate programming the Sony AIBO robot dog (Tira-Thompson, 2004), but was later made platform-independent, with support added for a variety of robot types including a hand-eye system (Nickens, Tira-Thompson, Humphries, & Touretzky, 2009), the Chiara hexapod robot (Atwood & Berry, 2008), and the Calliope family of mobile manipulators (Touretzky, Watson, Allen, & Russell, 2010). The code base currently consists of roughly 250,000 lines of C++ and Java, with some utility programs written in Perl and Ruby.

Tekkotsu makes extensive use of advanced C++ features such as templates, multiple inheritance, polymorphism, namespaces, and functors. As a consequence, students who learn Tekkotsu programming also frequently report that they have developed a better understanding of C++. Because it is a large software system, students also gain experience with basic software engineering concepts such as class libraries, and they learn to navigate online documentation that is generated automatically using doxygen. Baltes and Anderson, describing a mixed reality infrastructure for robotics instruction, have likewise observed that working on robotics assignments gave students useful software engineering experience (Baltes & Anderson, 2010).

Tekkotsu's main competitor is ROS from Willow Garage (Quigley et al., 2009). ROS was developed as a research platform, but it is also used for teaching. Tekkotsu was designed with education as the primary goal, although it is also used for research. The different emphases led to different design choices, which are discussed in more detail in Tira-Thompson & Touretzky (2011). To briefly summarize: Tekkotsu utilizes a single language and single address space model with tightly integrated software components, whereas ROS emphasizes orthogonality of components, which may be written in multiple languages (primarily C++ and Python), with each running in a separate process. The ROS approach provides for greater modularity and scalability, while Tekkotsu is able to make greater use of C++ abstraction facilities and offers a shallower learning curve and easier debugging.

The curriculum described in this paper could be taught using either Tekkotsu or ROS. Some features that are presently unique to Tekkotsu could be ported to ROS in the future.

4. Supporting infrastructure

Mobile robots are complex software systems. To make these systems comprehensible to and usable by undergraduates, it is important to provide good supporting infrastructure at multiple levels, from fundamental math-

ematical operations to high level mechanisms for communication among software components.

4.1. Mathematical support

Vectors and matrices appear extensively in robotics and graphics programming (ref. Big Idea #7); yet aside from Matlab, none of the popular languages used in these areas (e.g. C, C++, Java, or Python) includes linear algebra support in the language specification. Instead, there are multiple, incompatible matrix packages from which users must choose.

Tekkotsu includes a package called *fmat* that provides efficient support for the small, fixed-sized vectors and matrices commonly used in robotics. It includes column and row vectors, general $m \times n$ matrices, and specialized rotation matrices and 3D transformation matrices utilizing homogeneous coordinates. *Fmat* also includes support for quaternions and axis-angle representations which are used to express 3D rotations. The Eigen package used by ROS provides similar facilities.

Many undergraduate CS programs include a linear algebra requirement, but these courses are usually taught by Mathematics departments and involve no programming. This illustrates an unfortunate disconnect in the CS curriculum: students take a substantial amount of math – typically three semesters of calculus, plus linear algebra, and some other mathematical topic such as differential equations or probability, yet they have few opportunities to apply any of this material in their computer science work.

Schools that do not require a linear algebra course may find it advantageous to teach linear algebra in the context of a robotics or graphics elective. For example, Full Sail University's BS in Game Development program includes a linear algebra course that combines the traditional mathematical material with coding of graphical display and collision detection routines that are common in gaming (Full Sail University, 2013). Since putting concepts to use promotes both understanding and retention, computer science programs should seriously consider replacing their linear algebra math requirement with a graphics or robotics application-themed version.

Besides linear algebra, another area where Tekkotsu provides mathematical support is angular arithmetic. Tekkotsu provides four classes for representing ranges of angular values. The AngTwoPi class represents angles from 0 to 2π and is used to encode robot headings, while the AngSignPi class represents angles from $-\pi$ to π and is used to encode differences between two headings or bearings. The other two classes are AngPi, used to encode orientations of symmetric objects such as lines, and AngSignTwoPi, used to encode turns with an explicit turn direction. Each class includes the usual overloaded arithmetic operators and assures that value are always within

the specified range, wrapping around as necessary. By using these classes instead of the generic *float* type, programmers can make their assumptions clear and avoid some common programming errors.

4.2. Frames of reference

Tekkotsu provides explicit support for three fundamental frames of reference: the camera image (*cam-centric*), the robot itself (*egocentric*), and the world (*allocentric*). An innovative feature of Tekkotsu's Point class is that it includes a reference frame in addition to the x, y, and z coordinate values. Arithmetic operators on Point objects are therefore able to check for reference frame compatibility, which helps detect coding errors in which programmers inadvertently mix reference frames. Points are displayed with a reference frame prefix, e.g. the notation `e:[500,0,0]` indicates a point in egocentric coordinates 500 mm ahead of the robot.

4.3. The Tekkotsu "crew"

The Tekkotsu "crew" is a collection of interacting software modules that provide high level functionality for perception, navigation, and manipulation (Touretzky & Tira-Thompson, 2010). Users can invoke these components directly by filling out a request template. Requests can also be sent from higher level crew members to lower level ones. From lowest to highest, the members are:

(1) **Lookout:** manages the robot's sensor package: pointing the head; taking camera images or rangefinder readings; simple scanning and tracking operations.
(2) **MapBuilder:** visual perception: object detection, mapping from camera coordinates to local and world coordinates.
(3) **Pilot:** path planning, navigation, localization, and collision avoidance.
(4) **Grasper:** object manipulation using the robot's arm or gripper.

The interactions between crew members include the MapBuilder invoking the Lookout to obtain camera images; the Pilot calling on the MapBuilder to detect visual landmarks; and the Grasper utilizing the Pilot to position the body so that the arm can reach an object, and then calling on the MapBuilder to obtain updated object position estimates.

The algorithms used by the crew are tuned to work well on the specific robots Tekkotsu supports. For example, on the Calliope2SP robot, which has a simple two degree-of-freedom arm with gripper, the Grasper acquires an object by pre-positioning the gripper and then using the Pilot to drive it into the object. On the Calliope5KP, which has a more capable five degree-of-freedom arm, the Grasper uses an entirely different strategy. First

it uses the Pilot to position the body at an optimum distance from the object, then it uses an arm path planner to calculate a trajectory that will bring the gripper to the correct pose for grasping while avoiding collisions with the body or other objects.

Abstraction is a primary design goal for the crew: it is important that the user be able to write a generic request to pick up an object and let the Grasper worry about the best way to achieve that intent. But, finer control is available if desired, e.g. on the Calliope5KP robot the user has the option of choosing a specific *grasp strategy*: side grasp, overhead grasp, or unconstrained, rather than allowing the Grasper to decide.

The ROS equivalent of the Tekkotsu crew is a collection of "services" for functions such as localization and navigation. ROS decouples its services so that different implementations of a service can be substituted without affecting other services.

4.4. Shape spaces

A powerful unifying feature in Tekkotsu is the "shape space": a collection of objects describing 2D and 3D shapes in a specific coordinate system (Touretzky, Halelamien, Tira-Thompson, Wales, & Usui, 2007). There is a separate shape space for each of the three fundamental frames of reference (camera, local, and world). These spaces serve multiple purposes and are used by the entire crew, and by user applications.

Information can enter a shape space in three ways. The first is perception: a vision algorithm can recognize a geometric figure (e.g. a line, an ellipse, a polygon) or an object (e.g. a cylinder, or an AprilTag) and create a shape in camera space that describes it. The MapBuilder then uses the camera pose and other information to perform a coordinate transformation and infer the location of the shape in 3D egocentric (local) space. Finally, the robot's position and heading are used to transform the shape to allocentric (world) coordinates and add it to the robot's world map.

A graphical tool called the SketchGUI allows the user to examine any of the shape spaces in real time. Thus, as the robot looks around and constructs a map, the user can watch the map develop.

The second way that information can enter a shape space is by the user creating shapes directly. For example, if the robot is to use a pre-defined map, that map can be created in the world shape space by constructing shapes that represent arena boundaries or maze walls, visual landmarks, and the initial positions of objects. Another important use of user-constructed shapes is communicating a request to the crew. If the user wants the robot to go to an arbitrary location on the world map, they can construct a point shape with those coordinates and drop that shape into a Pilot request. Because the world shape space is visible in the SketchGUI, the user will

see a graphical representation of their request, and then see the robot's pose change on the map as the Pilot guides the robot to the destination.

The third way that information enters a shape space is that it can be put there for illustrative or diagnostic purposes by the crew. For example, the Pilot uses a particle filter to estimate the robot's position and heading. These particles are displayed on the world map (i.e. they are represented in the world shape space) and updated automatically as the robot moves. As position uncertainty accumulates, users can watch the particle cloud disperse. When the robot relocalizes, the cloud collapses again.

The Pilot uses an RRT-based path planner (Kuffner & LaValle, 2000) to generate a navigation plan that avoids obstacles and ensures that the robot arrives at its destination with the proper orientation. When the Pilot is executing a navigation request, it draws the planned path in the world shape space. If the user requests the Pilot to display the search tree, a graphical representation of the tree is also added to the shape space, as in Figure 1. This can be helpful when path planning fails and the user needs to diagnose the problem (Pockels, Iyengar, & Touretzky, 2012).

One reason a Pilot or Grasper operation might fail is that the requested goal state is in collision with some obstacle. In this situation, the Pilot or Grasper automatically displays the bounding boxes of the robot's components and those of the obstacles, so that the collision is easily visualized. The display is accomplished by creating additional shapes in the world shape space.

ROS uses the rviz visualization tool to display graphical objects in a manner similar to Tekkotsu's SketchGUI. But in keeping with Tekkotsu's emphasis on integrated functionality, the SketchGUI does more than display shapes. It also conveys semantic information, such as whether an object has been marked as a landmark to be used for localization, or marked as an obstacle to be avoided by the path planner.

4.5. State machine language

State machines are very commonly used to specify robot behaviors. Tekkotsu's state machine formalism is built on top of an event-based, message passing architecture that was Tekkotsu's original programming metaphor. Today, users code almost exclusively in the state machine language. They construct behaviors in two stages. First, they define their own classes of state nodes that inherit from a variety of built-in node classes and contain additional task-specific functionality in C++. They then specify how these state node classes and various transition classes are instantiated to construct a state machine instance.

The formalism includes several useful extensions: state machines can nest hierarchically; multiple states can be active simultaneously; and states can pass messages to their successors as part of a transition's firing.

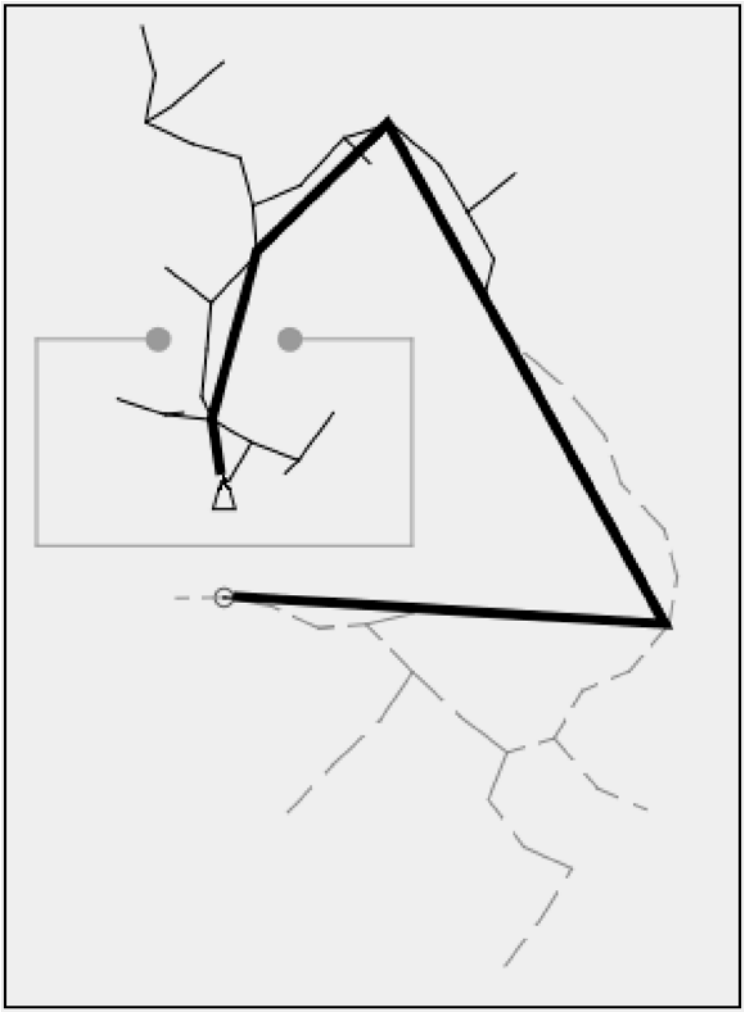

Figure 1. Using the RRT-Connect algorithm to escape from a box. The robot's current pose is shown by the small triangle, and a goal location below the box is shown as a hollow circle. The search tree rooted at the robot's location is shown as a thin black line, while the search tree rooted at the goal is shown as a dashed gray line. When the trees meet, a solution has been found. After applying a smoothing algorithm, the resulting solution path is shown as a thick line.

Building state machines on top of the C++ class system ensures that users have the full power of C++ available in every state node. The principal drawback is verbosity: there is a lot of boilerplate involved in defining new classes and then instantiating the classes and linking the nodes and transitions together. Generating all this code is tedious and is an opportunity to introduce errors such as forgetting to register a node with its parent. To alleviate these problems, we developed a state machine shorthand notation. Users code in a mixture of this shorthand and C++, and

a preprocessor converts this to pure C++ just prior to compilation. Code written in the shorthand form is 25 to 50% shorter than the pure C++ version.

The state machine language includes direct support for the crew. There are built in node types for constructing MapBuilder, Pilot, and Grasper requests, and built in transition types that detect specific Pilot or Grasper events, e.g. collision detection or path planning failure.

Figure 2 shows a complete Tekkotsu program written in the shorthand notation. The program has the robot look around for red cylinders, and if one is found, pick it up. The program defines a custom node class FindCylinder, a subclass of MapBuilderNode, to search for the cylinders. The node composes a MapBuilder request that will be sent to the MapBuilder upon completion of the node's doStart method on lines 7 and 8. A second custom node class, GrabIt, is a subclass of GrasperNode. It composes a grasper request by specifying the object that is to be grasped. This object is a cylinder shape in the world shape space. If no cylinder was found, the `find_if` function will return an *invalid* shape, similar to a null pointer. Lines 13 and 14 check for this case and cancel the grasp request, so it is never sent to the Grasper.

The `$setupmachine` section instantiates node and transition classes to construct a state machine instance. This state machine consists of five nodes, two of which are explicitly named: *startnode* is an instance of FindCylinder, and *grab* is an instance of GrabIt. The other three nodes are speech nodes that do not require explicit names.

Transitions between nodes are written as labeled arrows. The `=C=>` arrow is a *completion* transition that fires upon normal completion of a node action, such as when the MapBuilder completes the request sent to it by a MapBuilderNode, or when a SpeechNode finishes speaking its assigned text. The `=F=>` is a *failure transition*; it fires when a node signals that it has encountered some type of failure. In the GrabIt node, the `cancelThisRequest()` call signals such a failure. The two `=GRASP=>` transitions check for normal and abnormal completion of the grasp request.

4.6. Kinematics

Kinematics can be introduced using just a bit of linear algebra, but if students are to become proficient at applying these ideas to real robots, a substantial amount of infrastructure is required that is not covered in standard textbooks.

The first issue is how to describe the structure of a robot as a tree of kinematic reference frames, one per joint or end-effector. Denavit-Hartenberg conventions specify how to link these reference frames together via a series of translations and rotations, which can be expressed using transformation matrices and homogeneous coordinates. A textbook will typically describe

```
#include "Behaviors/StateMachine.h"

$nodeclass Demo1 {

  $nodeclass FindCylinder :
       MapBuilderNode(MapBuilderRequest::worldMap) : doStart {
    mapreq.addObjectColor(cylinderDataType, "red");
    mapreq.searchArea = Lookout::groundSearchPoints();
  }

  $nodeclass GrabIt : GrasperNode(GrasperRequest::grasp) : doStart {
    graspreq.object = find_if<CylinderData>(worldShS);
    if ( ! graspreq.object.isValid() )
      cancelThisRequest();
  }

  $setupmachine{
    startnode: FindCylinder =C=> grab

    grab: GrabIt
    grab =GRASP(noError)=> SpeechNode("Got it")
    grab =GRASP=> SpeechNode("Error during grasp")
    grab =F=> SpeechNode("Couldn't find a cylinder") =C=> startnode
  }
}

REGISTER_BEHAVIOR(Demo1);
```

Figure 2. Tekkotsu state machine program to find a red cylinder and pick it up.

these conventions and then work out the DH parameters for some example robot arms. This is as far as one can go without committing to a software implementation, but that is precisely what students need in order to experiment.

Tekkotsu provides the infrastructure for students to experiment with kinematic calculations. This includes a data structure for representing kinematic trees and conventions for describing such trees as human-readable XML files. Real kinematic descriptions contain more than just the four DH parameters (d, θ, r, α). They must include limits on joint travel, since most joints in physical robots cannot rotate a full 360°, and joint offset values, since the zero position on a servo does not necessarily align with the origin of a joint's reference frame.

Tekkotsu provides a graphical tool called the DH Wizard that allows students to browse a kinematic tree, alter any of its parameters, and immediately observe the effect on the modeled robot in the Mirage simulator. Figure 3 shows the tool being run on the kinematic description of the Calliope5KP robot.

The second issue when teaching kinematics is how to perform forward kinematics calculations. Mathematically, this is just a chain of matrix-vector multiplies, but if we want students to experiment, we must provide in-

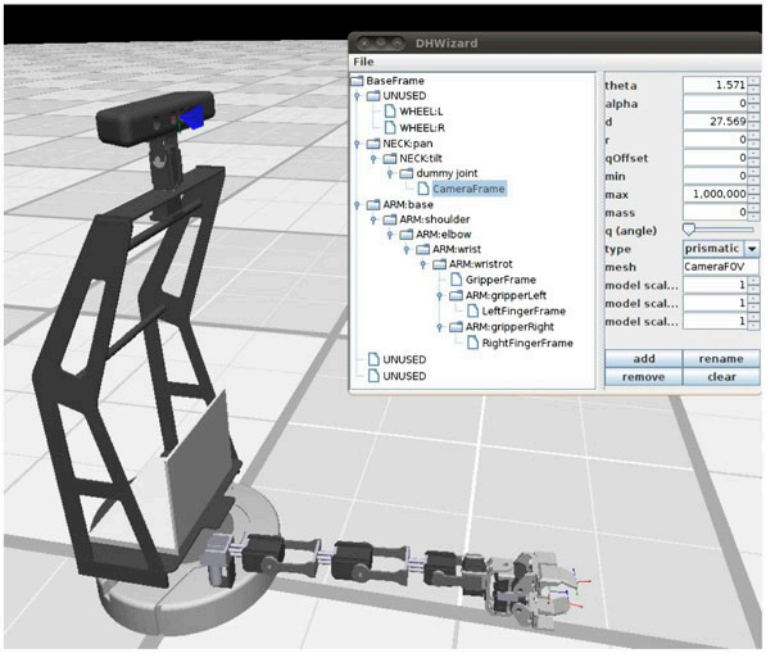

Figure 3. Using the DH Wizard to edit the kinematic description of the Calliope5KP robot.

frastructure to facilitate this. Tekkotsu provides methods for calculating the transformation matrix between any two kinematic joints or reference frames, and it includes a special instance of the kinematics engine that is tied to the robot's state representation so current joint angles can be used in the calculation. With these tools, for example, the current position in 3D space of the gripper relative to the robot's origin (the "base frame") can be computed in a single line of code. We do this by extracting the translation portion of the gripper-to-base transformation matrix:

```
fmat::Column<3> gripperPosition
  = kine->jointToBase(GripperFrame).translation();
```

The third big issue in kinematics instruction is inverse kinematics. Tekkotsu includes analytic solvers for common cases such as pan/tilts, 2 and 3 degrees-of-freedom planar arms, and simple 3-dof legs. It also includes a gradient descent solver that can handle more complex cases, such as solutions for the 5-dof arm on the Calliope5KP that involve orientation constraints.

Using these tools, we can construct compelling kinematics demos, such as the CameraTrackGripper and GripperTrackCamera demos built in to Tekkotsu. In CameraTrackGripper, the robot relaxes its arm so that it can be freely manipulated by the student. The demo uses forward kinematics to calculate the gripper position from the arm joint angles, and then inverse kinematics for the pan/tilt to keep the camera pointed at the gripper. By

monitoring the camera image while moving the arm around, the student can verify that the calculations are accurate. GripperTrackCamera on the Calliope5KP is even more impressive. The student drives the pan/tilt using the standard teleoperation tool, and the demo performs an inverse kinematics calculation for the arm to keep the left finger centered in the camera image and, as far as possible, a fixed distance from the lens. Students who do not have access to a Calliope5KP can run the demo in simulation.

Forward kinematics calculations in ROS are done using the *tf* package, while inverse kinematics uses the *MoveIt!* motion planning package.

5. Hardware platforms

The ability to interact with the physical world is a key part of robotics' appeal. While simulated robots can be used to some extent and are even preferable for some types of debugging, keeping students motivated requires giving them access to real robots. Another consideration is the limited fidelity of present day simulation environments. For efficiency reasons, they do not accurately model natural lighting, camera idiosyncrasies, backlash, friction, and so on. As a result, applications that work in simulation may not function reliably in the messier environment of the real world. The only way to be sure is to test on a physical robot.

A robot platform suitable for the type of course described here should provide five things: a camera, an accurate pan/tilt mechanism so the robot can track objects and landmarks, reasonable odometry to support navigation, some sort of manipulator, and a computer with sufficient processing power to implement the algorithms discussed in the course. The inexpensive robots used in robot-themed CS1 courses do not meet these criteria.

The best low-cost solutions presently available mount a netbook or laptop on a small mobile base such as the iRobot Create. One such robot is the Calliope2SP, jointly developed by Carnegie Mellon and RoPro Design, Inc. Beginning in 2014, it will serve as the platform for a national robotics competition sponsored by the Institute for African American Mentoring in Computing Sciences (iAAMCS) and held in conjunction with the annual Tapia Conference. A more advanced model, the Calliope5KP, is presently in use at Carnegie Mellon. The Turtlebot 2, developed by Willow Garage for use with ROS, is also in this class, although it lacks a pan/tilt and manipulator. Table 2 gives the specifications for these robots, which are shown in Figure 4.

Besides Create-based platforms, the robot with the most significant penetration in the university market is the Nao humanoid from Aldebaran Robotics, which replaced the Sony Aibo as the standard platform in the RoboCup robot soccer competition. The Nao retails for $12–16,000; there is also a legless version for roughly $6800. The Darwin-OP from Robotis is a similar humanoid that retails for $10–12,000. Unfortunately, the relatively high prices of these robots has impeded their wider adoption.

Table 2. Educational robots for undergraduate robotics courses.

	Calliope2SP	Calliope5KP	Turtlebot 2
Source	RoPro Design, Inc.	RoPro Design, Inc.	Willow Garage
Base	iRobot Create	iRobot Create	Yujin Kobuki
Camera	Sony PlayStation Eye	Microsoft Kinect	Microsoft Kinect or ASUS Xtion
Pan/tilt	Yes	Yes	No
Arm	2 dof + gripper	5 dof + 2 fingers	None
Servos	Robotis AX-12 (5)	Robotis AX series (3) and RX series (6)	None
Computing	Netbook with dual-core 1.6 GHz Intel Atom processor, 2 GB RAM, and 320 GB hard drive		
Price (assembled)	$2500	Approx. $4000	$1600

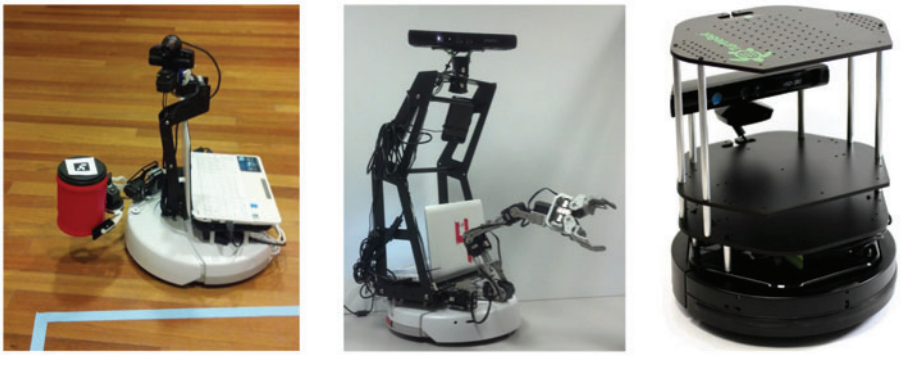

Figure 4. The Calliope2SP, Calliope5KP, and Turtlebot 2 robots.

6. Teaching the big ideas

The concepts in Table 1 can be introduced to beginning students by demonstration without going into implementation details. Tekkotsu's emphasis on making robot operations transparent means that some of these demonstrations happen automatically. For example, Figure 5 shows a code fragment that requests the Pilot to take the robot to a point one meter north and half a meter west of its starting location, and to arrive there on a southerly heading. When this demo is run, students will see the planned path displayed in the world shape space. They may not yet know how a path planner works, but they can see its effects. As the robot executes the path, both the robot shape and its localization particles move through the shape space, and the particle cloud begins to disperse due to accumulated error.

```
1  $nodeclass GoThere : PilotNode(PilotTypes:goToShape) : doStart {
2    NEW_SHAPE(destination, PointData,
3              new PointData(worldShS, Point(1000, 500, 0, allocentric)));
4    pilotreq.targetShape = destination;
5    pilotreq.targetHeading = M_PI;
6  }
```

Figure 5. Pilot request that invokes the navigation path planner.

We can make the situation more interesting by introducing objects into the environment that act as both obstacles and landmarks. Now the path planner must calculate a trajectory around the obstacles. (It finds the obstacles by looking in the world shape space). As the robot moves, the Pilot can correct its motion error by visually acquiring the landmarks and relocalizing. It does this automatically if it finds suitable objects in the world map.

The next step for teaching path planning would be to introduce the RRT-Connect algorithm (Kuffner & LaValle, 2000). This algorithm is typically described using an oversimplified domain where the robot is symmetric and holonomic, so orientation is not a factor. Students can read pseudo-code for the algorithm and examine pictures of the search trees it constructs. They can also learn how collision detection is accomplished using the separating axis theorem.

Once students are familiar with the RRT-Connect search algorithm, they can deepen their understanding by constructing test cases and seeing where the algorithm succeeds and where it fails (Pockels et al., 2012). This can be done in Tekkotsu by setting up simulated worlds with various obstacle configurations, and varying the path planning parameters in the Pilot request. The results can be visualized by displaying the search trees in the world shape space along with the robot's start and goal locations and any obstacles the user defined. An example is shown in Figure 1.

For still deeper understanding, students can learn to call the path planner directly instead of invoking it implicitly via the Pilot. Tekkotsu provides a simple x–y planner for a symmetric, holonomic robot that can be used for such exercises. Alternatively, students can use the more complex non-holonomic x–y–θ planner that is actually invoked by the Pilot. Both planners pull obstacles from the world shape space and can display their results in that space.

The most advanced students can read the C++ code for the path planners to appreciate how they are implemented, including how path smoothing is accomplished and how collision detection is performed efficiently on complex shapes.

A summary of our approach to teaching the big ideas is:

(1) Begin by demonstrating how the idea is actually realized on the robot.

(2) Explain the algorithm, possibly in simplified form.
(3) Have students experiment with the algorithm to gain an appreciation of its parameters, performance characteristics, and limitations.
(4) Have students put the new functionality to use as part of a larger behavior, possibly in a final project at the end of the course.

Besides path planning, this approach fits several of the other big ideas well: computer vision, localization with a particle filter, forward and inverse kinematics, and (potentially) human-robot interaction technologies such as face detection and speech understanding.

Examples of student projects over the years include getting two AIBO robots to cooperate to push an object, getting a Chiara to calculate depth from stereo by shifting its body left or right while pointing its single camera at a scene, and getting a Calliope5KP robot to locate and pick up a packet of crackers, carry it to another robot, and toss it into a basket mounted on top.

Many of the students in the Carnegie Mellon course choose to develop novel extensions to the framework as their final project. The DH Wizard tool, Tekkotsu's inter-robot communication primitives, and the WorldBuilder tool that eases creation of virtual environments for the Mirage simulator all started out as course projects. Non-undergraduates have also produced some striking projects. A graduate student who had taken the course as an undergraduate went on to program the Chiara to play chess on a real chessboard for his masters thesis (Coens, 2010). And a high school student, who learned Tekkotsu programming in a summer enrichment program, programmed the Chiara to walk up to a keyboard and play Ode to Joy with its right front leg. His video explaining how he used Tekkotsu's vision and kinematics primitives to accomplish this (Iyengar, 2011) won a prize in the 2011 Association for the Advancement of Artificial Intelligence video competition.

The course materials are still evolving, and not all the big ideas from Table 1 are covered yet. For example, Tekkotsu presently provides little in the way of human–robot interaction facilities (Big Idea #9) besides sound effects and text-to-speech, although some students have experimented with gesture recognition, face detection, and speech recognition modules. Likewise, there is not yet any support for multi-robot coordination (Big Idea #10) beyond basic inter-robot messaging, although a student project has explored how to share world maps between robots. A task planner (Big Idea #8) is currently in development. Covering all the big ideas using the hands-on, experimental approach advocated here is probably not feasible in a one-semester course, but should be easily doable in a two-course sequence.

7. Robotics in the CS2013 curriculum

Consider the complex software that goes into a modern web browser: an HTML interpreter, renderers for various image types, a font manager, layout

engines for text and tables, a JavaScript interpreter, client code for multiple networking protocols, and more. We introduce students to the web by first having them use a browser where all these technologies are seamlessly integrated. Then we teach them how to create simple web pages of their own. And then we introduce advanced features such as frames, forms, style sheets, and scripts. Web designers need to understand some details of the underlying technologies, such as the effects of JPEG or GIF encoding on an image, but they are never asked to write their own JPEG or GIF renderer. When a system depends on so many different technologies, it becomes impractical to master all their implementation details. And in an introductory course, these would be a distraction. A similar argument could be made concerning application development for smartphones.

The software behind modern robots is already more complex than a web browser or smartphone, and is continuing to advance. As a result, I believe introductory robot programming courses need to evolve in the direction of web design courses, where the focus is on understanding and using the technologies rather than reimplementing simplified versions of them. Contrast this view with the description of a robotics elective in the CS2013 Ironman draft (Joint Task Force on Computing Curricula, 2013), which reflects the current state of many robotics courses. Of the eight desired learning outcomes for the robotics elective, three are skill-based. The first of these is:

> **2.** *Integrate sensors, actuators, and software into a robot designed to undertake some task.*

If the sensor is a photodiode or ultrasonic rangefinder, the actuators are a couple of servos, and the software is a few pages of C code running on a microcontroller; we have the typical embedded systems view of robotics found in an electrical engineering department. But the kind of robot computer scientists should be using today is, like a smartphone, built from much more sophisticated components that are already carefully integrated.

The second skill-based learning objective, which involves robot control architectures, is more appropriate for computer scientists:

> **3.** *Program a robot to accomplish simple tasks using deliberative, reactive, and/or hybrid control architectures.*

The final skill-based learning objective in the CS2013 Ironman draft involves reimplementing fundamental algorithms:

> **4.** *Implement fundamental motion planning algorithms within a robot configuration space.*

When robots were less capable and there were no comprehensive software frameworks available, reimplementing basic algorithms was a sensible thing to do in an introductory course. And reimplementation of some key algorithms might still be appropriate in an advanced robotics course, just as reimplementation of a JPEG decoder or ray tracer might be assigned in an advanced graphics course. But with today's sophisticated hardware and powerful software frameworks, it seems more important that students first learn how to use their many components effectively.

Another important difference between the robotics curriculum described here and the proposed CS2013 Robotics elective is the omission of vision and kinematics. CS2013 does contain a separate Perception and Computer Vision elective that includes both vision and speech recognition, and rightly focuses on algorithms for feature extraction and pattern classification. But vision is an integral part of modern robotics, essential for several of the topics that *are* included in the elective, such as localization, mapping, and navigation (the robot must be able to see the landmarks), and also motion planning (so it can see the obstacles to be avoided or objects to be manipulated). Likewise, although forward and inverse kinematics are covered in a CS2013 Computer Animation elective, these should not be entirely omitted from Robotics.

The present time is a transition period in robotics education. More capable hardware platforms are available but have not yet been widely adopted. At the same time, students are being introduced to complex software frameworks in contexts such as web design and smartphone application design, but not yet in robot application design. As our robots become even more complex and capable, a shift in instructional strategy appears inevitable.

Acknowledgements

The author thanks the anonymous referees for several helpful suggestions.

Funding

This work was supported in part by National Science Foundation award [grant number CNS-1042322].

References

Atwood, T., & Berry, K. (2008, November/December). Chiara: Designed for computer science R&D. *Robot Magazine*, pp. 12–13.

Baltes, J., & Anderson, J. E. (2010). Leveraging mixed reality infrastruture for robotics and applied AI instruction. In *Proceedings of EAAI-10: The First Symposium on Educational Advances in Artificial Intelligence*. Menlo Park, CA: AAAI Press.

Boonthum-Denecke, C., Touretzky, D. S., Jones, E. J., Humphries, T., & Caldwell, R. (2011). The ARTSI Alliance: Using robotics and AI to recruit African–Americans to computer science research. In *Proceedings of FLAIRS-24*. Palm Beach, FL: AAAI Press.

Carnegie Mellon University. (2013). *Carnegie Mellon joins launch of NSF-sponsored alliance to Mentor African–American computer scientists*. Retrieved from https://news.cs.cmu.edu/article.php?a=3737

Coens, J. (2010). *Taking Tekkotsu out of the plane* (Master's thesis), Carnegie Mellon University, Computer Science Department. Retrieved from http://reports-archive.adm.cs.cmu.edu/anon/2010/CMU-CS-10-139.pdf. For videos, see http://Chiara-Robot.org/Chess

Full Sail University. (2013). *Linear algebra course*. Retrieved from http://www.fullsail.edu/degrees/campus/game-development-bachelors/courses/linear-algebra-GEN-242

Iyengar, A. (2011). *Chiara robot plays the piano*. Retrieved from aivideo.org or http://www.youtube.com/watch?v=-e8zmGypBDg

Joint Task Force on Computing Curricula. (2013). *Ironman Draft version 1.0*. Retrieved from http://ai.stanford.edu/users/sahami/CS2013/ironman-draft/cs2013-ironman-v1.0

Kuffner, J. J., & LaValle, S. M. (2000). RRT-connect: An efficient approach to single-query path planning. In *ICRA'2000 IEEE*. San Francisco, CA.

Nickens, G. V., Tira-Thompson, E. J., Humphries, T., & Touretzky, D. S. (2009). An inexpensive hand-eye system for undergraduate robotics instruction. In *SIGCSE 2009* (pp. 423–427). Chattanooga, TN: Association for Computing Machinery.

Pockels, J. A., Iyengar, A., & Touretzky, D. S. (2012). Graphical display of search trees for transparent robot programming. In *Proceedings of the Twenty-Fifth International Florida Artificial Intelligence Research Society Conference (FLAIRS-25)*. Marco Island, FL: Association for Computing Machinery.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J. ... Ng, A. (2009). ROS: An open-source robot operating system. In *Proceedings of ICRA-2009 IEEE*. Kobe, Japan.

Tira-Thompson, E. J. (2004). *Tekkotsu: A rapid development framework for robotics* (Masters thesis). Carnegie Mellon University.

Tira-Thompson, E. J., & Touretzky, D. S. (2011). The Tekkots robotics development environment. In *Proceedings of ICRA-2011*. Shanghai, China: IEEE.

Touretzky, D. S. (2012). Seven big ideas in robotics and how to teach them. In *Proceedings of SIGCSE'12*. Raleigh, NC: Association for Computing Machinery.

Touretzky, D. S., Watson, O., Allen, C. S., & Russell, R. J. (2010). Calliope: Mobile manipulation from commodity components. In *Papers from the 2010 AAAI Robot Workshop Technical report WS-10-09*. Atlanta, GA: AAAI Press.

Touretzky, D. S. (2010). Preparing computer science students for the robotics revolution. *Communications of the ACM, 53*, 27–29.

Touretzky, D. S., Halelamien, N. S., Tira-Thompson, E. J., Wales, J. J., & Usui, K. (2007). Dual-coding representations for robot vision in Tekkotsu. *Autonomous Robots, 22*, 425–435.

Touretzky, D. S., & Tira-Thompson, E. J. (2010). The Tekkotsu "crew": Teaching robot programming at a higher level. In *Proceedings of EAAI-10: The First Symposium on Educational Advances in Artificial Intelligence*. Menlo Park, CA: AAAI Press.

Veloso, M. M., Rybski, P. E., Lenser, S., Chernova, S., & Vail, D. (2006). CMRoboBits: Creating an intelligent AIBO robot. *AI Magazine, 27*, 67–82.

Wiggins, G., & McTighe, J. (2005). *Understanding by design* (expanded 2nd ed.). Upper Saddle River, NJ: Pearson Education.

Williams, A. B., Touretzky, D. S., Manning, L., Walker, J. J., Boonthum, C., Forbes, J., & Doswell, J. T. (2008). The ARTSI Alliance: Recruiting underrepresented students to computer science and robotics to improve society. In *Papers from the 2008 AAAI Spring Symposium: Using AI to motivate greater participation in computer science* (pp. 110–111). AAAI.