Soar Documentation

Release 1.5.2

Andrew Antonitis

Feb 08, 2019

Contents:

Soar (Snakes on a Robot) is a Python framework for simulating and interacting with robots.

The software is designed to be the following:

- **painless**: Using Soar for its intended purpose should be *trivial*. A student using Soar as part of an intro robotics course should, in the ideal case, have to look at *Getting Started* and nothing else.
- **extensible**: Soar can support nearly any type of robot and any type of connection, so long as the user provides a suitable interface. Connect to robot's over a serial port, WiFi, Bluetooth, etc–Soar is interface-agnostic. Though Soar provides basic physics for 2D collision detection and movement, the physics of simulated worlds and objects can be completely overidden.
- **simulation-driven**: The most typical use case of Soar will be to run some stepwise simulation on a certain robot type, with some *brain* controlling it. It is not primarily designed for persistent robots that are always on or for situations where stepwise interaction is not suitable.
- **multiplatform**: Soar uses no platform specific features, and uses Python's standard GUI package, Tkinter, for its GUI. Soar should thus work on any platform with a standard Python interpreter of version 3.5 or later. Soar has been tested on Fedora 25 GNU/Linux, and Windows 8. If an issue arises running Soar on your platform, open an issue on GitHub.
- **open source**: Soar is licensed under the LGPLv3, and may be used as a library by projects with other licenses.

To start using Soar, read the *Getting Started* guide, or look at the documentation for *brains* and *worlds*.

CHAPTER 1

Getting Started

This guide will outline how to install and use Soar. For greater detail see the documentation on *brains* and *worlds*, or perhaps the *Module Documentation*.

1.1 Installation

Installing Soar is (hopefully) painless and primarily done 3 ways, ordered by decreasing ease:

Note:

- Most Python installations will already have `setuptools`, necessary to install Soar, but if not, see this documentation to install it.
- Installing Soar will also install pyserial version 3.0 or later, as well as matplotlib version 2.0 or later.
- Soar was developed exclusively with Python 3.5 or later in mind. Your mileage may vary or be nonexistent if using an earlier version.

1.1.1 From PyPI

Soar can be installed from the Python Package Index (PyPI) by running `pip install soar`.

This will install the latest stable (not development) release.

1.1.2 From Releases

An arbitrary stable (not development) Soar release can be installed from the github releases, by downloading the `zip` archive and running `pip install <path-to-zip>`.

1.1.3 From latest source

Clone or download the git repo, navigate to the directory, then run:

```
python3 setup.py sdist
cd dist
pip install Soar-<version>.tar.gz
```

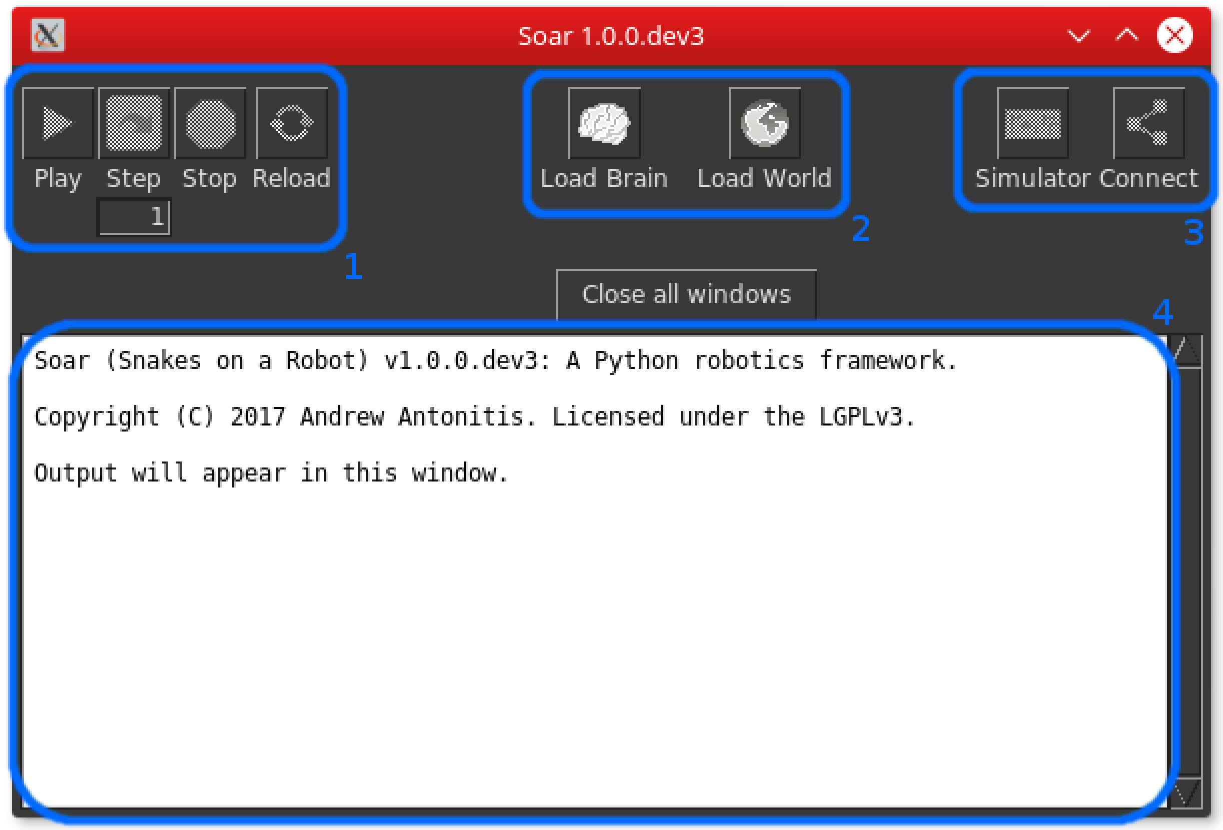
1.2 Usage

There are two major 'modes' of operation that Soar offers: simulation of a robot (and possibly other objects), or connecting via a robot-defined interface. The latter is only usable when running from the GUI; when running headless or from another project, only simulation may be used.

Soar's functionality can be accessed through multiple interfaces, documented below.

Also see the documentation for *Brains* and *Worlds*.

1.2.1 GUI



```
Soar (Snakes on a Robot) v1.0.0.dev3: A Python robotics framework.
Copyright (C) 2017 Andrew Antonitis. Licensed under the LGPLv3.
Output will appear in this window.
```

Fig. 1: Soar v1.0.0dev3 in KDE

Soar can be started in GUI mode simply by running `soar` from the command line. The main interface is fully resizable.

From the main interface, there are multiple panels to interact with:

1. The playback panel. When a robot controller has been loaded, either simulated or through some robot-defined interface, this is available and can be used to control playback. Pressing the `Play` button for the first time will start the controller and repeatedly step the robot. Pressing it while a simulation is running will `Pause` it. The `Step` button, and its associated entry, will step the controller that many times, although this functionality is only available for simulations. The `Stop` button stops the controller, and the `Reload` button reloads the currently loaded brain and world.
2. The brain/world loading panel. These buttons are used to load Soar brains and worlds. The default directory for brains is the user's home directory. The default directory for worlds is Soar's own soar/worlds/ directory, installed with the project. In order to simulate a robot, both a brain and world must be loaded. If connecting through a robot-defined interface, only a brain must be loaded and any loaded world will be ignored.
3. The simulation/connect panel. These buttons are used to prepare for simulation, or connecting to a robot-defined interface. If a controller has already been loaded, clicking either of these will kill it and reload the brain and/or world and prepare a controller of the desired type.
4. The output panel. All controller-related output will appear here. Any text the brain prints to `stdout` (via `print()`) will appear prefixed by `'>>>'`. If an exception occurs when running the controller, its full traceback will also be printed here. The panel is cleared whenever a reload occurs.

The simulation window opens whenever a simulation controller is loaded. Soar widgets like `soar.gui.plot_window.PlotWindow` are linked to this window, and will be closed whenever it is closed. User-created windows may also be linked to it via use of the *`soar.hooks.tkinter_hook()`*.

The simulation window opens with a default maximum dimension (width or height) of 500 pixels, but may be resized to any size that matches the aspect ratio of the corresponding world.

1.2.2 Command Line/Headless

See the *Command Line Reference* and the documentation for *Logging*.

When running in headless mode, both a brain and world file are required. The simulation will be immediately started, and may never complete if the brain does not raise an exception or call *`soar.hooks.sim_completed()`*. Typical usage might be to capture the `stdout` and `stderr` of the process, terminate it after a set period or time, or ensure that the brain will end the simulation at some point.

1.2.3 In another project

To use Soar from within another Python project, import *`soar.client.main()`* and pass arguments accordingly. Unless you desire to build Soar's GUI interface when invoking this function, you will have to pass `headless=True`.

Note: When using Soar's entrypoint from another Python project, you have the advantage of being able to use file-like objects such as `StringIO` as log outputs instead of actual files.

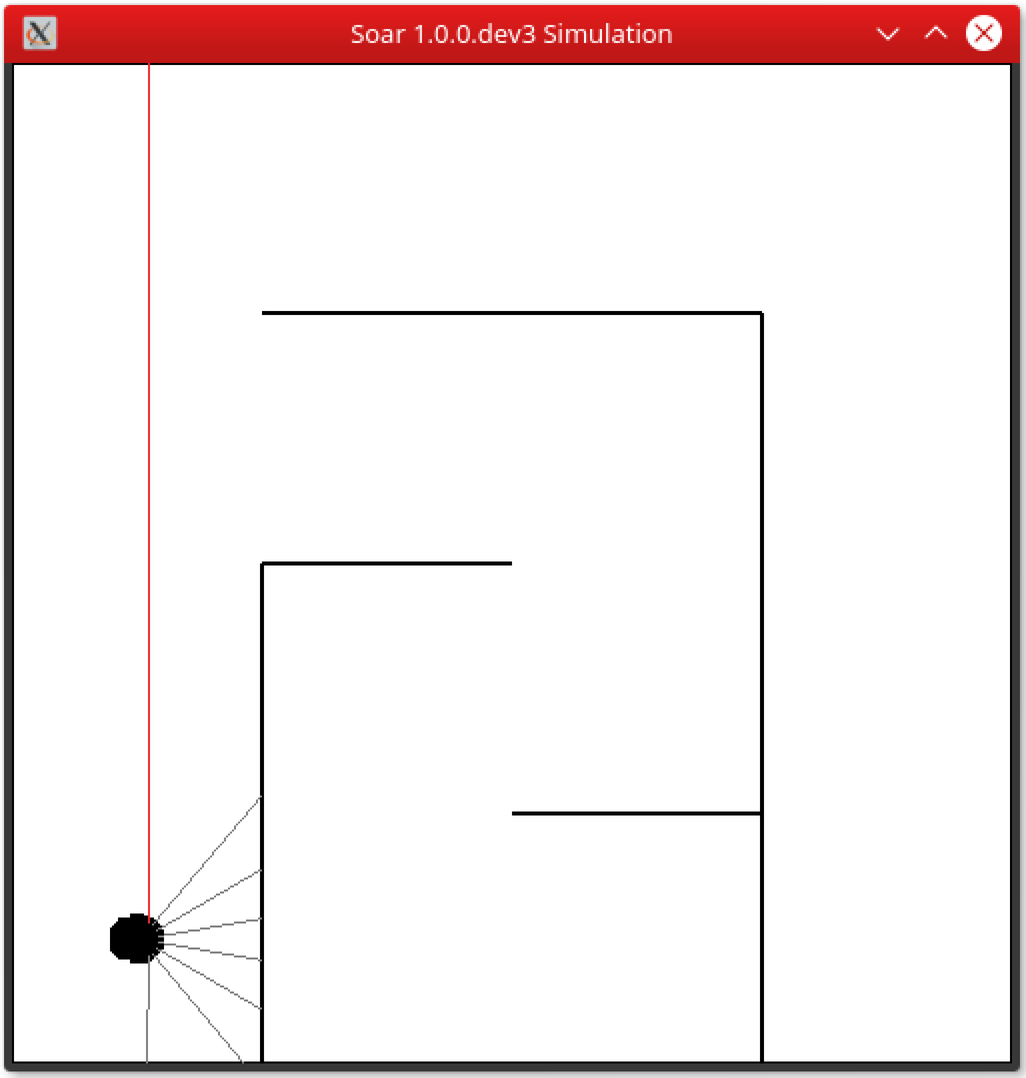


Fig. 2: Soar v1.0.0dev3 Simulation in KDE

CHAPTER 2

Brains

A Soar brain is a Python module used to control a robot. Brains can be used to control the movement of a simulated robot, connect to a real interface to control a real one, or multiplex between the two. Whenever a brain is loaded by Soar, the content of the module is compiled and executed in an isolated namespace, meaning that variable names will not conflict with those defined by Soar, except where documented below. All Python builtins are available to brains–they are normal Python modules with certain reserved words used by Soar.

2.1 Properties

A brain file must have the following attributes to be usable in Soar:

- `robot`: This should be an instance of *`soar.robot.base.BaseRobot`*, an instance of a subclass (like *`soar.robot.pioneer.PioneerRobot`*, or some object that supports identical methods. `robot` is what defines the interface, if any, to connect to something outside of Soar, as well as what can be done with that robot type–i.e movement, sensor readings, etc.
- `on_load()`, `on_start()`, `on_step(duration)`, `on_stop()`, `on_shutdown()` functions. While not strictly necessary (if any of these are not defined, they will be silently replaced by empty functions by Soar, they are the main ways in which the brain actually interacts with the controller.
  - `on_load()` is called exactly once, when the controller is loaded, and always after the robot's corresponding `on_load()` method.
  - `on_start()` is also called exactly once, when the controller is started for the first time, just before the first actual step. The robot's `on_start()` method is always called just before the brain's.
  - `on_step(duration)` is called whenever the controller steps, *before* the robot's `on_step(duration)` method is called. `duration` specifies how long the step lasts, which may not be constant from step to step. Note that if this function takes longer to execute than `duration`, the `duration` argument that the robot receives will be lengthened accordingly.
  - `on_stop()` is called exactly once when the controller stops, *before* the robot's `on_stop()` method is called.

- `on_shutdown()` is called exactly once when the controller is shut down, *before* the robot's `on_shutdown()` method. Typically, this function performs any cleanup desired.

These names should be considered reserved by Soar when used in a brain module.

Note: Typically, using `print()` within a Soar brain will cause the ouput to be prepended by three carets `'>>>'`. To prevent this, pass the argument `raw=True` to the `print` function.

2.2 Hooks

Hooks are optional functions that brains can import to interact with Soar on a more flexible level than the controller provides. Hooks include everything defined in *`soar.hooks`*, as well as GUI widgets like `soar.gui.plot_window.PlotWindow`.

The names of hooks, as well as widget names like `PlotWindow`, should be considered reserved names and not used otherwise, as the client detects their presence by name.

Note: The usage of hooks outside of the controller methods (`on_load()`, `on_start()`, etc) is allowed but discouraged. Users may create `PlotWindow` plots in the main body of the brain module, hook Tkinter widgets into the *`tkinter_hook`*, etc, with no issues, but should be aware that hooks like *`sim_completed`* and *`elapsed`* may not function as expected outside of a controller (they do nothing and return `0.0`, respectively).

CHAPTER 3

Worlds

A Soar world is a Python file containing the definition of a world used for simulation. It may contain any number of simulated objects, and subclasses of `soar.sim.world.World` may change its behavior further.

3.1 Properties

To be usable in Soar, each world file must have the following attributes:

- `world`: An instance of `soar.sim.world.World`, or an instance of a subclass defined by the user.

`WorldObjects` may be added to the world in the initial constructor, or after the object has been created, as long as this is done in the world file at some point.

CHAPTER 4

Logging

Soar logging is done via JSON objects separated by newlines. All logged objects will contain a `type` key. There are various types of logged objects:

- `meta`: A typical `meta` object might look like:

  ```
  {"version": "1.0.0dev3", "brain": "path/to/brain.py", "type": "meta", "world":
  ↪"path/to/world.py", "simulated": false}
  ```

  containing information about the controller when it was loaded. When a simulation completes, the following `meta` object is logged:

  ```
  {"type": "meta", "completed": <time elapsed>}
  ```

- `step`: Step objects are logged before every step, and one is logged after the simulation is stopped as well:

  ```
  {"type": "meta", "step": <step_number>, "elapsed": <time elapsed>, "brain_print":
  ↪<brain stdout>, "robot": <serialized robot data>}
  ```

  containing information about the step.

- `plot`: Plot objects are logged whenever the user closes a simulation with a `PlotWindow` open:

  ```
  {"type": "plot", "data": <hex data of png image of plot>}
  ```

CHAPTER 5

soar

Soar (Snakes on a Robot) v1.5.2

An extensible Python framework for both simulating and interacting with robots.

5.1 soar.client

Soar client entrypoint.

Classes and functions for interacting with Soar in general. This module serves as the main entrypoint to the package. Projects desiring to invoke a Soar instance should import this module and call *soar.client.main()*.

Examples

This will invoke a GUI instance, which will terminate when the main window is closed, and always return 0:

```
from soar.client import main
return_value = main()
```

If invoking a headless instance, paths to brain and world files should be specified:

```
from soar.client import main
return_value = main(brain_path='path/to/brain.py', world_path='path/to/world.py',␣
↪headless=True)
```

In this case, the return value will be 1 if an exception occurred and 0 otherwise.

Logging is handled via passing a path:

```
from soar.client import main
return_value = main(logfile='path/to/logfile')
```

or, using a file-like object:

```
from soar.client import main
return_value = main(logfile=open('path/to/logfile', 'r+'))
```

`soar.client.`**`future`**(*name*, **args*, ***kwargs*)

Adds a function with data for the client to execute in the future, by putting it on the client's internal queue.

Note that if an exception or controller failure occurs, the future may never be executed as the client's queue will be purged.

Certain futures accept an optional callback parameter.

Parameters

- **`name`** – The future to execute, defined in *`soar.common`*. and contained in *`future_map`*.
- **`*args`** – The variable length arguments to pass to the function.
- **`**kwargs`** – The keyword arguments to pass to the function.

`soar.client.`**`future_map = {0:  <function make_gui at 0x7ff3591b20d0>, 1:  <function load_bra`**

A mapping from future names to their actual functions.

`soar.client.`**`main`**(*brain_path=None*, *world_path=None*, *headless=False*, *logfile=None*, *step_duration=0.1*, *realtime=True*, *options=None*)

Main entrypoint, for use from the command line or other packages. Starts a Soar instance.

Parameters

- **`brain_path`** (*`optional`*) – The path to the initial brain to load. Required if headless, otherwise not required.
- **`world_path`** (*`optional`*) – The path to the initial world to load. Required if headless, otherwise not required.
- **`headless`** (*`bool`*) – If `True`, run Soar in headless mode, immediately running a simulation.
- **`logfile`** (*`optional`*) – The path to the log file, or a file-like object to log data to.
- **`step_duration`** (*`float`*) – The duration of a controller step, in seconds. By default, 0.1 seconds.
- **`realtime`** (*`bool`*) – If `True`, the controller will never sleep to make a step last the proper length. Instead, it will run as fast as possible.
- **`options`** (*`dict`*) – The keyword arguments to pass to the robot whenever it is loaded.

Returns 0 if Soar's execution successfully completed, 1 otherwise.

5.2 soar.common

Soar common constants.

Contains named constants for sending messages to the client and determining the type of a controller.

Most use cases of soar will never need to use these constants or order client futures.

The values of the constants themselves are arbitrary.

`soar.common.`**`CONTROLLER_COMPLETE = 9`**

Signal that the controller has completed and the simulation can end.

`soar.common.`**`CONTROLLER_FAILURE = 12`**
Signals a controller failure.

`soar.common.`**`CONTROLLER_IO_ERROR = 11`**
Signals an IO error, which usually occurs when connecting with a real robot.

`soar.common.`**`DRAW = 19`**
Draw an object on the GUI's canvas.

`soar.common.`**`EXCEPTION = 22`**
Returned by wrapped functions to signal that an exception occurred during their execution.

`soar.common.`**`GUI_ERROR = 21`**
Signals an exception occurring somewhere in Tkinter callback

`soar.common.`**`GUI_LOAD_BRAIN = 15`**
Forces loading a brain as if it were done through the GUI.

`soar.common.`**`GUI_LOAD_WORLD = 16`**
Forces loading a world as if it were done through the GUI.

`soar.common.`**`LOAD_BRAIN = 1`**
Loads a brain, optionally calling the `callback` argument, and prints if `silent` is `False`.

`soar.common.`**`LOAD_WORLD = 2`**
Loads a world, optionally calling the `callback` argument, and prints if `silent` is `False`.

`soar.common.`**`LOGGING_ERROR = 13`**
Signals a logging error. These are typically ignored entirely.

`soar.common.`**`MAKE_CONTROLLER = 3`**
Makes and loads the controller (type is based on `simulated`), and calls `callback`.

`soar.common.`**`MAKE_GUI = 0`**
No arguments. Build the main Tkinter-based GUI and enter its event loop

`soar.common.`**`MAKE_WORLD_CANVAS = 20`**
Tell the GUI to make the simulator canvas.

`soar.common.`**`NOP = 18`**
Does nothing besides calling an optional callback

`soar.common.`**`PAUSE_CONTROLLER = 5`**
Pauses the controller, calling an optional callback.

`soar.common.`**`SET_HOOKS = 17`**
Initializes the hooks

`soar.common.`**`SHUTDOWN_CONTROLLER = 8`**
Shuts down the controller.

`soar.common.`**`START_CONTROLLER = 4`**
Starts the controller, calling an optional callback.

`soar.common.`**`STEP_CONTROLLER = 6`**
Steps the controller, where the first argument is the number of steps, infinite if `None`.

`soar.common.`**`STEP_FINISHED = 14`**
Signals that the step thread has finished.

`soar.common.`**`STOP_CONTROLLER = 7`**
Stops the controller and calls an optional callback.

5.3 soar.controller

Controller classes and functions for controlling robots, simulated or real.

class soar.controller.**Controller**(*client_future*, *robot*, *brain*, *simulated*, *gui*, *step_duration=0.1*, *realtime=True*, *world=None*, *log=None*)

Bases: `object`

A class for interacting with simulated or real robots.

Initialized by the client and used to call the user defined methods of the robot and brain.

running
bool – Indicates whether the controller is currently running–that is, repeatedly stepping.

started
bool – Indicates whether the controller has been started.

stopped
bool – Indicates whether the controller has been stopped.

step_count
int – The number of steps that have elapsed so far.

elapsed
float – The number of seconds spent actually running. Unless a step takes longer than `step_duration`, this will typically be the *step_count* multiplied by `step_duration`. If any steps take longer, the additional time will also be counted here.

Parameters

- **client_future** – The function to call to schedule a future for the client to execute.
- **gui** – The currently *soar.gui.soar_ui.SoarUI* instance, if any, or `None` if in headless mode.
- **simulated** (*bool*) – If `True`, the controller will simulate the robot. Otherwise it will treat the robot as real.
- **robot** – An instance of *soar.robot.base.BaseRobot* or a subclass.
- **brain** – The currently loaded brain module, supporting the `on_load()`, `on_start()`, `on_step()`, `on_stop()`, and `on_shutdown()` methods.
- **realtime** (*bool*) – If `True`, stepping takes real time–that is, the controller will sleep for whatever time is not used running the step, until the step has taken at least `step_duration` seconds. Otherwise, no sleeping will occur; however the elapsed time will behave as if each step was at least `step_duration` long.
- **world** – An instance of *soar.sim.world.World* or one of its subclasses if one is loaded, or `None`.
- **step_duration** (*float*) – The duration of a single step, in seconds.
- **log** – A callable that accepts a `dict`-like object as an argument to log to a file, or `None`, if no logging
- **to take place.** (*is*) –

failure()
Called when the controller fails.

load()
Called when the controller is loaded.

log_step_info()
Log information about the current step.

run(*n=None*)
Runs the controller, starting it if necessary, for one or many steps, or without stopping.

Parameters n – If `None`, run forever, at least until stopped. If 0, start the controller, if it has not yet been started. Otherwise, for `n > 0`, run for that many steps.

shutdown()
Called when the controller is shut down.

stop()
Called when the controller is stopped.

5.4 soar.errors

Soar error classes.

exception `soar.errors.`**GUIError**
Bases: *`soar.errors.SoarError`*

Raised when an error occurs while drawing the world or one of its objects.

exception `soar.errors.`**LoggingError**
Bases: *`soar.errors.SoarError`*

Raised when an error occurs writing to a log file. Typically, these are ignored and merely printed.

exception `soar.errors.`**SoarError**
Bases: `Exception`

An umbrella class for Soar exceptions.

This is raised by the client if it encounters an error it cannot handle, such as an exception that occurs in headless mode.

exception `soar.errors.`**SoarIOError**
Bases: *`soar.errors.SoarError`*

Raised whenever an error occurs communicating with a real robot.

Typically, this is raised within the robot class when a connection could not be established, timed out, or invalid data that cannot be dealt with was received.

`soar.errors.`**printerr**(**args*, ***kwargs*)
A wrapper around `print` to print to `sys.stderr`.

5.5 soar.gui

Soar GUI Modules.

Contains the modules for building and displaying Soar's GUI, as well as all GIFs used for icons or animations, and widgets for plot generation, etc.

5.5.1 soar.gui.canvas

Soar canvas classes and methods.

Defines `SoarCanvas` and `SoarCanvasFrame` classes, as well as a method of creating them from an instance of `soar.sim.world.World`.

class `soar.gui.canvas.`**`SoarCanvas`**(*parent*, ***options*)

Bases: `tkinter.Canvas`

A metered, resizable Tkinter canvas. All drawing operations take metered arguments, where x values increase rightwards and y values increase upwards.

Parameters

- **`parent`** – The parent window or widget in which the canvas is placed.
- **`**options`** – Tk options.

`create_arc`(**args*, ***kw*)
Create arc shaped region with coordinates x1,y1,x2,y2.

`create_bitmap`(**args*, ***kw*)
Create bitmap with coordinates x1,y1.

`create_image`(**args*, ***kw*)
Create image item with coordinates x1,y1.

`create_line`(**args*, ***kw*)
Create line with coordinates x1,y1,. . . ,xn,yn.

`create_oval`(**args*, ***kw*)
Create oval with coordinates x1,y1,x2,y2.

`create_polygon`(**args*, ***kw*)
Create polygon with coordinates x1,y1,. . . ,xn,yn.

`create_rectangle`(**args*, ***kw*)
Create rectangle with coordinates x1,y1,x2,y2.

`create_text`(**args*, ***kw*)
Create text with coordinates x1,y1.

`on_resize`(*event*)

`remap_coords`(*coords*)

class `soar.gui.canvas.`**`SoarCanvasFrame`**(*parent*, ***options*)

Bases: `tkinter.Frame`

A resizable frame that holds a `SoarCanvas`.

`on_resize`(*event*)

`soar.gui.canvas.`**`canvas_from_world`**(*world*, *toplevel=<class 'tkinter.Toplevel'>*, *close_cmd=None*)

Return a `soar.gui.canvas.SoarCanvas` in a new window from a World. Optionally, call a different `toplevel()` method to create the window, and set its behavior on close.

Additionally, if an object in the world defines `on_press`, `on_release`, and `on_motion` methods, bind those to the associated mouse events for that tag, making it draggable.

Parameters

- **`world`** – An instance of `soar.sim.world.World` or a subclass.

- **`toplevel`** (*`optional`*) – The function or method to call to create the window itself.
- **`close_cmd`** (*`optional`*) – The function or method to call when the window is destroyed.

Returns The new *`SoarCanvas`*.

5.5.2 soar.gui.output

Soar output classes.

Tk widgets allowing the capture and display of text output in the GUI.

class `soar.gui.output.`**`OutputFrame`**(*master*, *initial_text=''*)

Bases: `tkinter.Frame`

A read-only Tk output frame that can display normal and error output.

Parameters

- **`master`** – The parent widget or window.
- **`initial_text`** (*`str, optional`*) – The text to have initially in the output frame.

`clear`()

Clear the entire text field.

`error`(*text*)

Insert error output text at the end of the text field.

Parameters **`text`** (*`str`*) – The text to insert.

`insert`(*text*, **tags*)

Insert text at the end of the text field.

Parameters

- **`text`** (*`str`*) – The text to insert.
- **`*tags`** – Variable length `str` list of tags to attach to the text. The `'output'` tag signifies normal output, and the `'error'` tag signifies that the text will be red.

`link`(*text*)

Insert a clickable link at the end of the text field.

Parameters **`text`** (*`str`*) – The link text to insert.

`output`(*text*)

Insert normal output text at the end of the text field.

Parameters **`text`** (*`str`*) – The text to insert.

class `soar.gui.output.`**`SoarIO`**(*write_func*)

Bases: `_io.StringIO`

`write`(*s*)

Write string to file.

Returns the number of characters written, which is always equal to the length of the string.

5.5.3 soar.gui.plot_window

Tkinter wrapper for plotting using matplotlib

Note: Unlike use of the `soar.hooks.tkinter_hook()`, use of this module will not force brain methods to run on the main thread alongside Soar's GUI event loop.

However, methods of `PlotWindow` *will* run on the main thread, regardless of what thread the other brain methods run in.

`PlotWindow` is wrapped by the client to ensure that the proper mode (GUI or headless) is enforced regardless of the arguments passed to the constructor by the brain methods.

The client will also ensure that, if logging is occurring, any `PlotWindow` objects will have their image data included in the log whenever the controller is shut down.

Based on code written by Adam Hartz, August 1st 2012.

5.5.4 soar.gui.soar_ui

Soar Main GUI classes.

Classes for building the main GUI, which allows the user to load brains & worlds, start simulations, etc.

class `soar.gui.soar_ui.`**`ButtonFrame`**(*master*, *image=None*, *text=None*, *command=None*, *state=None*)

Bases: `tkinter.Frame`

A Tk frame containing an image Button and a Label immediately beneath it, arranged via the grid geometry manager.

`button`

The button inside the frame.

`label`

The label inside the frame.

Parameters

- **`master`** – The parent widget or window in which to place the frame.
- **`image`** (*`optional`*) – The image to place inside the button.
- **`text`** (*`optional`*) – The text to place inside the label.
- **`command`** (*`optional`*) – The function to call when the button is clicked.
- **`state`** (*`optional`*) – The state of the button, either NORMAL or DISABLED.

`config`(*image=None*, *text=None*, *command=None*, *state=None*)

Sets the parameters of the button/label.

Parameters

- **`image`** (*`optional`*) – The image to place inside the button.
- **`text`** (*`optional`*) – The text to place inside the label.
- **`command`** (*`optional`*) – The function to call when the button is clicked.
- **`state`** (*`optional`*) – The state of the button, either NORMAL or DISABLED.

class `soar.gui.soar_ui.`**`IntegerEntry`**(*parent*, *value=”*, ***kwargs*)
Bases: `tkinter.Entry`

A Tk entry that only allows integers.

Parameters

- **`parent`** – The parent Tk widget or window.
- **`value`** (*`str`*) – The initial value. Must be able to be cast to `int`.
- **`**kwargs`** – Arbitrary Tk keyword arguments.

class `soar.gui.soar_ui.`**`LoadingIcon`**(*parent*, *file*, *frames*)
Bases: `tkinter.Label`

An animated loading icon that can be shown or hidden.

Parameters

- **`parent`** – The parent Tk widget or window.
- **`file`** (*`str`*) – The path to an animated `.gif`. The last frame should be empty/transparent.
- **`frames`** (*`int`*) – The number of frames in the `.gif`.

class `soar.gui.soar_ui.`**`SoarUI`**(*client_future*, *client_mainloop*, *parent=None*, *title='Soar 1.5.2'*)
Bases: `tkinter.Tk`

The main GUI window.

Parameters

- **`client_future`** – The function to call to schedule a future with the client.
- **`client_mainloop`** – The client's mainloop function, restarted after the main thread switches to Tk execution.
- **`parent`** (*`optional`*) – The parent window. It is almost always unnecessary to change this from the default.
- **`title`** (*`str, optional`*) – The main window title.

`attach_window`(*w*, *linked=True*)
Attach an existing window to the SoarUI.

Parameters **`linked`** (*`bool`*) – If `True`, the window will be destroyed whenever the simulator window is destroyed.

Returns The window that was attached.

`brain_cmd`()
Called when the brain button is pushed.

`brain_ready`()
Configure buttons and paths when a brain is loaded.

`cancel_all_loading`()
Hide the loading icon and delete all button state frames.

`close_windows`(*close_unlinked=False*)
Close windows, optionally unlinked ones, clear the draw queue, and set the simulator canvas to `None`.

Parameters **`close_unlinked`** (*`bool`*) – If `True`, closes all windows. Otherwise, closes only the linked ones.

connect_cmd()
Called when the connect to robot button is pushed.

connect_load()
Called when the real robot's requested reload has finished.

connect_ready()
Called when the real robot is ready.

controller_failure()
Called by the client when the controller fails.

controller_io_error()
Called when an error occurs connecting to the real robot.

done_loading()
Re-enable user interaction, and hide the loading icon.

future(*func*, **args*, *after_idle=False*, ***kwargs*)
Executes a function (asynchronously) in the GUI event loop ASAP in the future.

gui_error()
Called when a GUI error occurs, such as an error while drawing a world or window.

initialize()
Initialize the grid geometry.

loading()
Disable user interaction and animate the loading icon.

mainloop(*n=0*)
Enter the Tk event loop, and restart the client as a new thread.

Redirect stdout and stderr to the GUI's output frame.

on_close()
Called when the main window is closed.

pause_cmd()
Called when the pause button is pushed.

pause_ready()
Called when the controller has finished pausing.

play_cmd()
Called when the play button is pushed.

play_ready()
Called after the controller has started playing.

reload_cmd(*reload_controller=True*, *clear_output=True*, *silent=False*, *close_unlinked=False*, *callback=None*)
Kill the controller, close windows, and reload the brain and world.

Parameters

- **reload_controller** (*bool*) – If `True`, immediately reload whatever controller was in effect previously.
- **close_unlinked** (*bool*) – If `True`, close all windows, not just the linked ones.
- **clear_output** (*bool*) – If `True`, clears the output of the output frame.
- **silent** (*bool*) – If `True`, stops the client from printing `'LOAD BRAIN'`-like messages.

- **`callback`** – The function to call after the reload has finished, or `None`.

`reload_finished`(*reload_controller*, *sim_canvas*, *connected*, *callback*)
Called after the client has finished reloading.

Parameters

- **`reload_controller`** (*`bool`*) – If `True`, reload the previous controller.
- **`sim_canvas`** – If not `None`, the controller to be reloaded is the simulator.
- **`connected`** – If `True`, the controller to be reloaded is the real robot controller.
- **`callback`** – A function to call once the reload has finished, or `None`.

`report_callback_exception`(*exc*, *val*, *traceback*)
Report callback exception to sys.stderr, as well as notifying the Soar client.

In addition, signal any events that may need to be flagged for *`synchronous_future`*.

`reset`(*clear_output=True*)
Reset all of the button states to what they are at initialization, before any files are loaded.

Parameters **`clear_output`** (*`bool, optional`*) – If `True`, clear the contents of the output frame.

`sim_cmd`()
Called when the simulator button is pushed.

`sim_load`()
Called when the simulator's reload has finished.

`sim_ready`()
Called when the simulator is ready.

`step_cmd`()
Called when the step button is pushed.

`step_finished`()
Called when the controller finishes multi-stepping.

`stop_cmd`()
Called when the stop button is pushed.

`stop_ready`()
Called when the controller has stopped.

`synchronous_future`(*func*, **args*, *after_idle=False*, ***kwargs*)
Executes a function in the GUI event loop, waiting either for its return, or for a Tk exception.

`toplevel`(*linked=True*)
Add a new window to the UI's internal list, and return a new Toplevel window, optionally linked.

Parameters **`linked`** (*`bool`*) – If `True`, the window will be destroyed whenever the simulator window is destroyed.

Returns The new Toplevel window.

`world_cmd`()
Called when the world button is pushed.

`world_ready`(*auto_sim_load=False*)
Configure buttons and paths when a world is ready.

5.6 soar.hooks

Functions for brains to hook into various elements of Soar.

To make use of a given hook, a brain should import it from here. It will be redefined by the client or controller accordingly.

Treat all hook names as reserved words, and do not use them as arbitrary variable names.

`soar.hooks.`**`elapsed`**`()`

Get the time that has elapsed running the controller.

Set by the controller before the brain's `on_load()` function is called.

Returns The elapsed time in seconds, as defined in *`soar.controller.Controller.elapsed`*.

Return type float

`soar.hooks.`**`is_gui`**`()`

Return whether Soar is running in GUI mode.

Set by the client when it first loads.

Returns `True` if running in GUI mode, `False` if headless.

Return type bool

`soar.hooks.`**`raw_print`**`(`**args*, ***kwargs*`)`

Allows a brain to print without the `'>>>'` prepended.

All arguments and keyword arguments are passed to `print()`.

`soar.hooks.`**`sim_completed`**`(`*obj=None*`)`

Called by the brain to signal that the simulation has completed.

Set by the controller before the brain's `on_load()` function is called.

Parameters **`obj`** (*`optional`*) – Optionally, an object to log to the logfile after the simulation has completed.

`soar.hooks.`**`tkinter_hook`**`(`*widget*, *linked=True*`)`

Hook a Tkinter widget created by a brain into Soar, so that the UI is aware of it.

This function is redefined by the client before a brain is ever loaded.

Brains that import this hook can expect that their methods will always run in the main thread when running in GUI mode, as Tkinter is not thread-safe. Otherwise, this is not guaranteed.

If not running in GUI mode, importing and using this hook simply returns its argument unchanged and does nothing.

Parameters

- **`widget`** – The Tkinter widget to attach to Soar. This may also be some object that supports a `destroy()` method.
- **`linked`** (*`bool`*) – If `True`, the widget will be destroyed whenever the controller reloads.

Returns The window or object that was attached.

`soar.hooks.`**`tkinter_wrap`**`(`*widget*, *linked=True*`)`

Make Soar's UI aware of a Tk widget, and wrap its callables so that they run on the main thread.

This function is redefined by the client before a brain is ever loaded.

This is a 'softer' version of *tkinter_hook*–it does not force all brain methods to run on the main thread.

Rather, it attempts to find and wrap widget callables so that, when called, they will execute on the main thread.

If not running in GUI mode, importing and using this hook simply returns its argument unchanged.

Parameters

- **widget** – The Tk widget to attach to Soar. This should support a `destroy()` method.
- **linked** (*bool*) – If `True`, the widget will be destroyed whenever the controller reloads.

Returns The widget, with its callables wrapped so that they will run on the main thread.

5.7 Command Line Reference

Allows use of Soar from the command line by passing arguments to *soar.client.main()*

```
usage: soar [-h] [--headless] [--nosleep] [--logfile LOGFILE]
            [-s step duration] [-b brain] [-w world] [--options OPTIONS]

optional arguments:
  -h, --help          show this help message and exit
  --headless          Run in headless mode
  --nosleep           Run quickly, without sleeping between steps
  --logfile LOGFILE   Log file to write to
  -s step duration    The duration of a controller step
  -b brain            Path to the brain file
  -w world            Path to the world file
  --options OPTIONS   Options to pass to the robot, as a JSON deserializable dictionary
```

5.8 soar.robot

Soar robot modules, containing the BaseRobot class, example robot subclasses, and bundled implementations like PioneerRobot.

5.8.1 soar.robot.arcos

ARCOS (Advanced Robot Control and Operations Software) Client.

Classes and functions for communicating with an ARCOS server running on an Adept MobileRobot platform (typically Pioneer 2 and 3).

`soar.robot.arcos.`**`ADSEL = 35`**
Select the A/D port number for reporting ANPORT value in standard SIP.

class `soar.robot.arcos.`**`ARCOSClient`**(*timeout=1.0*, *write_timeout=1.0*, *allowed_timeouts=2*)
Bases: `object`

An ARCOS Client communicating over a serial port with an ARCOS server.

Parameters

- **timeout** (*float*) – The time to wait while receiving data before a timeout occurs, in seconds.

- **`write_timeout`** (*`float`*) – The time to wait while sending data before a timeout occurs, in seconds.
- **`allowed_timeouts`** (*`int`*) – The number of timeouts to tolerate before the update coroutine closes the port.

`standard`
dict – The last standard Server Information Packet (SIP) received, or `None`, if one hasn't been received yet.

`config`
dict – The last CONFIGpac SIP received, or `None`, if one hasn't been received yet.

`encoder`
dict – The last ENCODERpac SIP received, or `None`, if one hasn't been received yet.

`io`
dict – The last IOpac SIP received, or `None`, if one hasn't been received yet.

`standard_event`
`threading.Event` – Set whenever a standard SIP is received.

`config_event`
`threading.Event` – Set whenever a CONFIGpac SIP is received.

`encoder_event`
`threading.Event` – Set whenever an ENCODERpac SIP is received.

`io_event`
`threading.Event` – Set whenever an IOpac is received.

`sonars`
list – A list of the latest sonar array values, updated whenever a standard SIP is received.

`connect`(*forced_ports=None*)
Attempt to connect and sync with an ARCOS server over a serial port.

Returns if successful.

Parameters **`forced_ports`** (*`list, optional`*) – If provided, a list of serial ports to try connecting to. Otherwise, the client will try all available ports.

Raises *`ARCOSError`* – If unable to connect to any available ports.

`disconnect`()
Stop the ARCOS server and close the connection if running.

`pulse`()
Continually send the PULSE command so that the robot knows the client is alive.

`receive_packet`()
Read an entire ARCOS Packet from an open port, including header and checksum bytes.

Returns The entire packet as a list of bytes, including header and checksum bytes.

Return type list

Raises

- *`Timeout`* – If at any point a timeout occurs and fewer bytes than expected are read.
- *`InvalidPacket`* – If the packet header, checksum, or packet length are invalid.
- *`ARCOSError`* – If something went wrong reading from the serial port.

send_command(*code*, *data=None*, *append_null=True*)
Send a command and data to the ARCOS server.

Parameters

- **code** – The command code. Must be in *soar.robot.arcos.command_types*.
- **data** (*optional*) – The associated command argument, assumed to be of the correct type. For commands that take a string argument, a `bytes` object may also be used.
- **append_null** (*optional*) – If `True`, append a null byte to any `str` or `bytes` object passed as a command argument.

Raises

- *Timeout* – If the write timeout of the port was exceeded.
- *ARCOSError* – If an error occurred writing to the serial port.

send_packet(**data*)
Send arbitrary data to the ARCOS server.

Adds the packet header and checksum. Thread-safe.

Parameters ***data** – A tuple or iterable of bytes, whose values are assumed to be between 0 and 255, inclusive.

Raises

- *Timeout* – If the write timeout of the serial port was exceeded.
- *ARCOSError* – If something went wrong writing to the serial port.

start()
Open the ARCOS servers, enable the sonars, and start the pulse & update coroutines.

sync(*tries=4*)
Try to sync with an ARCOS server connected over an open serial port.

Returns the raw robot identifying information packet sent after `SYNC2` if successful.

Parameters **tries** (*int, optional*) – The number of failures to tolerate before timing out.

Raises *Timeout* – If the number of tries is exhausted and syncing was not completed.

update()
Continually receive and decode packets, storing them as attributes and triggering events.

static wait_or_timeout(*event*, *timeout=1.0*, *timeout_msg="*)
Wait for an event to occur, with an optional timeout and message.

Parameters

- **event** (*Event*) – The event to wait for. Expected to be one of the attribute events of this class.
- **timeout** (*float, optional*) – How long to wait for the event before timing out.
- **timeout_msg** (*str*) – The message to pass if a timeout occurs.

Raises *Timeout* – If the event has not occurred by the specified time.

exception soar.robot.arcos.**ARCOSError**
Bases: *soar.errors.SoarIOError*

Umbrella class for ARCOS-related exceptions.

`soar.robot.arcos.`**`BUMPSTALL = 44`**
Stall robot if no (0), only front (1), only rear (2), or either (3) bumpers make contact.

`soar.robot.arcos.`**`CLOSE = 2`**
Close servers and client connection. Also stops the robot.

`soar.robot.arcos.`**`CONFIG = 18`**
Request a configuration SIP.

`soar.robot.arcos.`**`DCHEAD = 22`**
Adjust heading relative to last setpoint; +- degrees (+ is counterclockwise).

`soar.robot.arcos.`**`DHEAD = 13`**
Turn at *`SETRV`* speed relative to current heading; (+) counterclockwise or (-) clockwise degrees.

`soar.robot.arcos.`**`DIGOUT = 30`**
Set (1) or reset (0) user output ports. High bits 8-15 is a byte mask that selects the ports to change; low bits 0-7 set (1) or reset (0) the selected port(s).

`soar.robot.arcos.`**`ENABLE = 4`**
Enable the motors, if argument is 1, or disable them if it is 0.

`soar.robot.arcos.`**`ENCODER = 19`**
Request a single (1), a continuous stream (>1), or stop (0) encoder SIPS.

`soar.robot.arcos.`**`E_STOP = 55`**
Emergency stop. Overrides acceleration, so is very abrupt.

`soar.robot.arcos.`**`HEAD = 12`**
Turn at *`SETRV`* speed to absolute heading; +-degrees (+ is counterclockwise).

`soar.robot.arcos.`**`IOREQUEST = 40`**
Request a single (1), a continuous stream (>1), or stop (0) IO SIPS.

`exception` `soar.robot.arcos.`**`InvalidPacket`**
Bases: *`soar.robot.arcos.ARCOSError`*

Raised when a packet's checksum is incorrect.

`soar.robot.arcos.`**`MOVE = 8`**
Translate forward (+) or backward (-) mm absolute distance at *`SETV`* speed.

`soar.robot.arcos.`**`OPEN = 1`**
Start the ARCOS servers.

`soar.robot.arcos.`**`POLLING = 3`**
Change sonar polling sequence. Argument is a string consisting of sonar numbers 1-32 (as single bytes).

`soar.robot.arcos.`**`PULSE = 0`**
Reset server watchdog (typically sent every second so that the robot knows the client is alive).

`soar.robot.arcos.`**`ROTATE = 9`**
Rotate counter- (+) or clockwise (-) degrees/sec at *`SETRV`* limited speed.

`soar.robot.arcos.`**`RVEL = 21`**
Rotate (degrees/sec) counterclockwise (+) or clockwise (-). Limited by *`SETRV`*.

`soar.robot.arcos.`**`SAY = 15`**
Play up to 20 duration, tone sound pairs through User Control panel piezo speaker. The argument is a string whose first byte must be the number of (duration, tone) pairs to play, followed by each (duration, tone) pair. Duration is in 20 millisecond increments. A value of 0 means silence. The values 1-127 are the corresponding MIDI notes, with 60 being middle C. The remaining values are frequencies computed as `(tone - 127)*32`, ranging from 1-4096 in 32Hz increments.

soar.robot.arcos.**SETA = 5**
Set translation acceleration, if positive, or deceleration, if negative, in mm/sec^2.

soar.robot.arcos.**SETO = 7**
Reset local odometry position to the origin `(0, 0, 0)`.

soar.robot.arcos.**SETRA = 23**
Change rotation (+) acceleration or (-) deceleration in degrees/sec^2

soar.robot.arcos.**SETRV = 10**
Set maximum rotation velocity in degrees/sec. Note that the robot is still limited by the hardware cap.

soar.robot.arcos.**SETV = 6**
Set maximum translational velocity in mm/sec. Note that the robot is still limited by the hardware cap.

soar.robot.arcos.**SONAR = 28**
1=enable, 0=disable all the sonar; otherwise bits 1-3 specify an array from 1-4 to enable/disable.

soar.robot.arcos.**SONARCYCLE = 48**
Change the sonar cycle time, in milliseconds.

soar.robot.arcos.**SOUNDTOG = 92**
Mute (0) or enable (1) the user control piezo speaker.

soar.robot.arcos.**STOP = 29**
Stop the robot without disabling the motors.

soar.robot.arcos.**SYNC0 = 0**
The initial synchronization packet.

soar.robot.arcos.**SYNC1 = 1**
The second synchronization packet.

exception soar.robot.arcos.**Timeout**
Bases: *soar.robot.arcos.ARCOSError*

Raised when no packet is read after a certain interval.

soar.robot.arcos.**VEL = 11**
Translate at mm/sec forward (+) or backward (-), capped by *SETV*.

soar.robot.arcos.**VEL2 = 32**
Set independent wheel velocities; bits 0-7 for right wheel, bits 8-15 for left in 20mm/sec increments.

soar.robot.arcos.**command_types = {0: None, 1: None, 2: None, 3: <class 'str'>, 4: <cla**
The argument type of every supported ARCOS command.

soar.robot.arcos.**decode_packet**(*packet*)
Decode a SIP (Server Information Packet) into a field-indexable dictionary.

Returns A dictionary with field names as keys and values as corresponding numbers. The `'TYPE'` key holds a value of `'STANDARD'`, `'CONFIG'`, `'ENCODER'`, or `'IO'`, corresponding to the packet type.

Return type dict

Raises *InvalidPacket* – If a packet's fields could not be decoded.

soar.robot.arcos.**packet_checksum**(*data*)
Calculate and returns the ARCOS packet checksum of a packet which does not have one.

Parameters **data** (*list*) – A list of data bytes.

Returns The packet checksum.

Return type int

5.8.2 soar.robot.base

Soar BaseRobot class, intended as a parent class for nontrivial/useful robots.

All robots usable in Soar should either subclass from BaseRobot or, if this is not possible, reproduce its behaviors.

class `soar.robot.base.`**`BaseRobot`**(*polygon*, ***options*)

Bases: *`soar.sim.world.WorldObject`*

`collision`(*other*, *eps=1e-08*)

Determine whether the robot collides with an object.

Supported objects include other robots, and subclasses of *`soar.sim.world.Polygon`* and *`soar.sim.world.Wall`*.

Parameters

- **`other`** – A supported `WorldObject` subclass with which this object could potentially collide.
- **`eps`** (*`float, optional`*) – The epsilon within which to consider a collision to have occurred, different for each subclass.

Returns A list of `(x, y)` tuples consisting of all the collision points with `other`, or `None` if there weren't any.

Return type list

`delete`(*canvas*)

Delete the robot from a canvas.

Parameters **`canvas`** – An instance of *`soar.gui.canvas.SoarCanvas`*.

`draw`(*canvas*)

Draw the robot on a canvas.

Canvas items are preserved. If drawn more than once on the same canvas, the item will be moved and not redrawn.

Parameters **`canvas`** – An instance of *`soar.gui.canvas.SoarCanvas`*.

`move`(*pose*)

Move the robot to the specified `(x, y, theta)` pose.

Parameters **`pose`** – An *`soar.sim.geometry.Pose`* or 3-tuple-like object to move the robot to.

`on_load`()

Called when the controller of the robot is loaded.

The behavior of this method should differ depending on the value of *`simulated`*; if it is `False`, this method should be used to connect with the real robot. If a connection error occurs, a *`soar.errors.SoarIOError`* should be raised to notify the client that the error was not due to other causes.

`on_pause`()

Called when the controller is paused.

`on_shutdown`()

Called when the controller of the robot is shutdown.

If interacting with a real robot, the connection should be safely closed and reset for any later connections.

on_start()

Called when the controller of the robot is started.

This method will always be called by the controller at most once per controller session, before the first step.

on_step(*step_duration*)

Called when the controller of the robot undergoes a single step of a specified duration.

For BaseRobot, this tries to perform an integrated position update based on the forward and rotational velocities. If the robot cannot move to the new position because there is an object in its way, it will be moved to a safe space just before it would have collided.

Subclasses will typically have more complex `on_step()` methods, usually with behavior for stepping non-simulated robots.

Parameters **step_duration** (`float`) – The duration of the step, in seconds.

on_stop()

Called when the controller of the robot is stopped.

pos

set_robot_options(***options*)

Set one or many keyworded, robot-specific options. Document these options here.

Parameters ****options** – `BaseRobot` does not support any robot options.

to_dict()

Return a dictionary representation of the robot, usable for serialization.

type = 'BaseRobot'

A base robot class, intended to be subclassed and overridden.

Any robot usable in SoaR should supplement or re-implement this class' methods with the desired behavior.

type

str – A human readable name for robots of this type.

world

The instance of `soar.sim.world.World` (or a subclass) in which the robot resides, or `None`, if the robot is real.

simulated

bool – Any BaseRobot subclass should consider the robot to be simulated if this is `True`, and real otherwise. By default, it is `False`.

pose

An instance of `soar.sim.geometry.Pose` representing the robot's `(x, y, theta)` position. In simulation, this is the actual position; on a real robot this may be determined through other means.

polygon

A `soar.sim.world.Polygon` that defines the boundaries of the robot and is used for collision.

fv

float – The robot's current translational velocity, in arbitrary units. Positive values indicate movement towards the front of the robot, and negative values indicate movement towards the back.

rv

float – The robot's current rotational velocity, in radians/second. Positive values indicate counter-clockwise rotation (when viewed from above), and negative values indicate clockwise rotation.

Parameters

- **polygon** – A *soar.sim.world.Polygon* that defines the boundaries of the robot and is used for collision.
- ****options** – Arbitrary keyword arguments. This may include Tkinter keywords passed to the `WorldObject` constructor, or robot options supported as arguments to *set_robot_options*.

5.8.3 soar.robot.names

Generate a neutral name from an arbitrary serial number string.

`soar.robot.names.`**`has_all`**`()`

`soar.robot.names.`**`is_prime`**`(`*n*`)`

`soar.robot.names.`**`name_from_sernum`**`(`*sernum*`)`

`soar.robot.names.`**`test_collisions`**`()`

5.8.4 soar.robot.pioneer

A PioneerRobot class, for representing a real or simulated Pioneer 3 robot.

See the MobileRobots documentation for more information.

`class` `soar.robot.pioneer.`**`PioneerRobot`**`(`***options*`)`

Bases: *soar.robot.base.BaseRobot*

`SONAR_MAX = 1.5`

An abstract, universal Pioneer 3 robot. Instances of this class can be fully simulated, or used to communicate with an actual Pioneer3 robot over a serial port.

type

str – Always `'Pioneer3'`; used to identify this robot type.

simulated

bool – If `True`, the robot is being simulated. Otherwise it should be assumed to be real.

pose

An instance of *soar.sim.geometry.Pose* representing the robot's `(x, y, theta)` position. In simulation, this is the actual position; on a real robot this is based on information from the encoders.

world

An instance of *soar.sim.world.World* or a subclass, or `None`, if the robot is real.

FV_CAP

float – The maximum translational velocity at which the robot can move, in meters/second.

RV_CAP

float – The maximum rotational velocity at which the robot can move, in radians/second.

SONAR_MAX

float – The maximum distance that the sonars can sense, in meters.

arcos

An instance of *soar.robot.arcos.ARCOSClient* if the robot is real and has been loaded, otherwise `None`.

serial_device
str – The device name of the serial port the robot is connected to, if it is real and has been been loaded, otherwise `None`.

Parameters `**options` – See *set_robot_options*.

analogs
Get the robot's 4 analog inputs, so long as it is real and not simulated, as a 4 tuple.

delete(*canvas*)
Delete the robot from a canvas.

Parameters **canvas** – An instance of *soar.gui.canvas.SoarCanvas*.

draw(*canvas*)
Draw the robot on a canvas.

Canvas items are preserved. If drawn more than once on the same canvas, the item will be moved and not redrawn.

Parameters **canvas** – An instance of *soar.gui.canvas.SoarCanvas*.

fv
`float` The robot's current translational velocity, in meters/second.

Positive values indicate movement towards the front of the robot, and negative values indicate movement towards the back.

Setting the robot's forward velocity is always subject to *soar.robot.pioneer.PioneerRobot.FV_CAP*. On a real robot, this is further limited by the hardware translational velocity cap.

get_distance_left()
Get the perpendicular distance to the left of the robot.

Returns The perpendicular distance to the left of the robot, assuming there is a linear surface.

Return type float

get_distance_left_and_angle()
Get the perpendicular distance and angle to a surface on the left.

Returns `(d, a)` where `d` is the perpendicular distance to a surface on the left, assuming it is linear, and `a` is the angle to that surface.

get_distance_right()
Get the perpendicular distance to the right of the robot.

Returns The perpendicular distance to the right of the robot, assuming there is a linear surface.

Return type float

get_distance_right_and_angle()
Get the perpendicular distance and angle to a surface on the right.

Returns `(d, a)` where `d` is the perpendicular distance to a surface on the right, assuming it is linear, and `a` is the angle to that surface.

on_load()
Called when the controller of the robot is loaded.

The behavior of this method should differ depending on the value of *simulated*; if it is `False`, this method should be used to connect with the real robot. If a connection error occurs, a *soar.errors.SoarIOError* should be raised to notify the client that the error was not due to other causes.

on_shutdown()
Called when the controller of the robot is shutdown.

If interacting with a real robot, the connection should be safely closed and reset for any later connections.

on_start()
Called when the controller of the robot is started.

This method will always be called by the controller at most once per controller session, before the first step.

on_step(*duration*)
Called when the controller of the robot undergoes a single step of a specified duration.

For BaseRobot, this tries to perform an integrated position update based on the forward and rotational velocities. If the robot cannot move to the new position because there is an object in its way, it will be moved to a safe space just before it would have collided.

Subclasses will typically have more complex *on_step()* methods, usually with behavior for stepping non-simulated robots.

Parameters **step_duration** (*float*) – The duration of the step, in seconds.

on_stop()
Called when the controller of the robot is stopped.

play_notes(*note_string*, *bpm=120*, *sync=False*, *_force_new_thread=True*)
Play a string of musical notes through the robot's piezoelectric speaker.

Parameters

- **note_string** (*str*) – A space-delimited list of notes. Notes should all be in the form `n/m(name)[#|b][octave]`. Ex: `'1/4C'` produces a quarter note middle C. `'1/8A#7'` produces an eighth note A# in the 7th MIDI octave. `'1/4-'` produces a quarter rest. All of the MIDI notes in the range 1-127 can be played.
- **bpm** (*int*) – The beats per minute or tempo at which to play the notes.
- **sync** (*bool*) – By default `False`, this determines whether notes are sent one by one, with synchronization performed by the function itself, or all at once.
- **force_new_thread** (*bool*) – By default `True`, this determines whether to force the execution of this function to occur on a new thread.

play_song(*song_name*)
Play one of a number of songs through the robot's piezoelectric speaker.

Parameters **song_name** (*str*) – The song to play. Must be one of `'reveille'`, `'daisy'`, or `'fanfare'`.

rv
`float` The robot's current rotational velocity, in radians/second.

Positive values indicate counterclockwise rotation (when viewed from above) and negative values indicate clockwise rotation.

Setting the robot's rotational velocity is always subject to *soar.robot.pioneer.PioneerRobot.RV_CAP*. On a real robot, this is further limited by the hardware rotational velocity cap.

set_analog_voltage(*v*)
Sets the robot's analog output voltage.

Parameters **v** (*float*) – The output voltage to set. This is limited to the range of 0-10V.

set_robot_options(***options*)

Set Pioneer3 specific options. Any unsupported keywords are ignored.

Parameters

- ****options** – Arbitrary robot options.
- **serial_ports** (*list, optional*) – Sets the serial ports to try connecting to with the ARCOS client.
- **ignore_brain_lag** (*bool*) – If `True`, a step will always be assumed to be 0.1 seconds long. Otherwise, whatever duration the controller tells the robot to step is how long a step lasts.
- **raw_sonars** (*bool*) – If `True`, sonar values will not be recast to `None` when no distance was returned. `5.0` will be returned instead.

sonars

(`list` of `float`) The latest sonar readings as an array.

The array contains the latest distance sensed by each sonar, in order, clockwise from the robot's far left to its far right. Readings are given in meters and are accurate to the millimeter. If no distance was sensed by a sonar, its entry in the array will be `None`, unless the robot option `raw_sonars` has been set to `True`, in which case its entry will be `5.0`.

to_dict()

Return a dictionary representation of the robot, usable for serialization.

This contains the robot type, position, sonar data, and forward and rotational velocities.

`soar.robot.pioneer.`**gen_tone_pairs**(*note_string*, *bpm=120*)

Given a string of musical notes separated by spaces and a tempo, generate a corresponding list of (duration, tone) pairs corresponding to each note.

Parameters

- **note_string** (*str*) – A space-delimited list of notes. Notes should all be in the form `n/m(name)[#|b][octave]`. Ex: `'1/4C'` produces a quarter note middle C. `'1/8A#7'` produces an eighth note A# in the 7th MIDI octave. `'1/4-'` produces a quarter rest. All of the MIDI notes in the range 1-127 can be played.
- **bpm** (*int*) – The beats per minute or tempo to use to calculate durations.

Returns A list of (duration, tone) tuples where duration is in seconds, and tone is the corresponding MIDI number. Musical rest has a tone of 0.

5.9 soar.sim

Simulation modules, containing general 2D geometry methods, graphical world objects with collision detection, etc.

5.9.1 soar.sim.geometry

Geometry classes, for manipulating points, collections of points, lines, and normalizing angles.

class `soar.sim.geometry.`**Line**(*p1*, *p2*, *eps=1e-08*, *normalize=False*)

Bases: `object`

A line in the `(x, y)` plane defined by two points on the line.

Parameters

- **p1** – An `(x, y)` tuple or *`Point`* as one of the points on the line.
- **p2** – An `(x, y)` tuple or *`Point`* as another point on the line.
- **eps** (*float, optional*) – The epsilon within which to consider a line horizontal or vertical, for precision purposes.
- **normalize** (*bool, optional*) – If `True`, normalize the internal vector representation to be a unit vector.

`distance_from_line`(*p*)

Determine the (signed) distance of a point from the line.

Parameters **p** – An `(x, y)` tuple or *`Point`* to measure distance from.

Returns The signed distance from the line.

Return type float

`has_point`(*p*, *eps=1e-08*)

Determine whether a point lies on the line.

Parameters

- **p** – The `(x, y)` tuple or *`Point`* to check.
- **eps** (*float, optional*) – The distance to tolerate before a point is considered not to be on the line.

Returns `True` if the point is on the line, `False` otherwise.

Return type bool

`intersection`(*other*, *eps=1e-08*)

Determine whether two lines intersect.

Parameters

- **other** – The *`Line`* to find the intersection with.
- **eps** (*float, optional*) – The smallest absolute difference to tolerate before the lines are considered to be converging.

Returns The *`Point`* of intersection, or `None` if the lines are parallel (based on epsilon).

class `soar.sim.geometry.`**`LineSegment`**(*p1*, *p2*, *eps=1e-08*)

Bases: *`soar.sim.geometry.Line`*

A line segment in the `(x, y)` plane defined by two endpoints.

Parameters

- **p1** – An `(x, y)` tuple or *`Point`* as the first endpoint.
- **p2** – An `(x, y)` tuple or *`Point`* as the second endpoint.
- **eps** (*float, optional*) – The minimum absolute difference in x or y before the line is considered horizontal or vertical.

`has_intersect`(*other*)

Determine whether one segment intersects with another.

Parameters **other** – Another *`LineSegment`*.

Returns `True` if the segments have an intersection, `False` otherwise.

Return type bool

has_point (*p*, *eps=1e-08*)

Determine whether a point lies on the line segment.

Parameters

- **p** – The `(x, y)` tuple or *Point* to check.
- **eps** (*float, optional*) – The distance to tolerate before a point is considered not to be on the line segment.

Returns `True` if the point is on the line segment, `False` otherwise.

Return type bool

intersection (*other*, *eps=1e-08*)

Find the intersection(s) between two line segments, or this line segment and a *Line*.

Parameters

- **other** – The other *LineSegment* to find intersections with.
- **eps** (*float, optional*) – The epsilon or tolerance to pass to the *Line* intersection and *has_point* checks.

Returns Either a list of *Point* (s) representing all of the intersections, or `None`, if there weren't any. Also returns `None` if the segment and a *Line* are exactly parallel.

class `soar.sim.geometry.`**Point** (*x*, *y*)

Bases: `object`

Represents a point in the x, y plane.

Points can be interpreted and treated as x, y tuples in most cases.

x

float – The x coordinate of the point.

y

float – The y coordinate of the point.

Parameters

- **x** (*float*) – The x coordinate of the point.
- **y** (*float*) – The y coordinate of the point.

add (*other*)

Vector addition; adds two points.

Parameters other – An `(x, y)` tuple or an instance of *Point*.

Returns The *Point* that is the sum of this point and the argument.

angle_to (*other*)

Return the angle between two points.

Parameters other – An `(x, y)` tuple or an instance of *Point*.

Returns Angle in radians of the vector from self to other.

Return type float

copy ()

Returns a copy of the *Point*.

distance(*other*)
Calculate the distance between two points.

Parameters **other** – An `(x, y)` tuple or an instance of *Point*.

Returns The Euclidean distance between the points.

Return type float

is_near(*other*, *eps*)
Determine whether the distance between two points is within a certain value.

Parameters

- **other** – An `(x, y)` tuple or an instance of *Point*.
- **eps** (*float*) – The epilson within which to consider the points near one another.

Returns `True` if the points are withing `eps` of each other, `False` otherwise.

Return type bool

magnitude()
The magnitude of this point interpreted as a vector.

Returns The magnitude of the vector from the origin to this point.

Return type float

midpoint(*other*)
Return a new *Point* that is the midpoint of self and other.

Parameters **other** – An `(x, y)` tuple or an instance of *Point*.

Returns A *Point* that is the midpoint of self and other.

rotate(*other*, *theta*)
Rotate about other by theta radians (positive values are counterclockwise).

Parameters

- **other** – An `(x, y)` tuple or an instance of *Point*.
- **theta** (*float*) – The number of radians to rotate counterclockwise.

Returns The rotated *Point*.

scale(*value*, *other=(0, 0)*)
Scale the vector from other to self by value, and return the endpoint of that vector.

Parameters

- **value** (*float*) – The value to scale by.
- **other** – An `(x, y)` tuple or a *Point* as the origin to scale from.

Returns The endpoint of the scaled vector, as a *Point*.

sub(*other*)
Vector subtraction; subtracts subtracts two points.

Parameters **other** – An `(x, y)` tuple or an instance of *Point*.

Returns The *Point* that is the difference of this point and the argument.

xy_tuple()
Returns: An `(x, y)` tuple representing the point.

class `soar.sim.geometry.`**`PointCollection`**(*points*, *center=None*)

Bases: `object`

A movable collection of points.

Can be iterated over like a list of *`Point`*. Unlike *`Point`*, PointCollections are mutable–that is, rotating them, translating them, etc. changes the internal point list.

Parameters

- **`points`** – A list of `(x, y)` tuples or *`Point`*.
- **`center`** – An `(x, y)` tuple or *`Point`* as the pivot or center of the collection.

`recenter`(*new_center*)

Re-center the collection.

Parameters **`new_center`** – An `(x, y)` tuple or *`Point`* that will be the collection's new center.

`rotate`(*pivot*, *theta*)

Rotate about other by theta radians (positive values are counterclockwise).

Parameters

- **`pivot`** – An `(x, y)` tuple or a *`Point`*.
- **`theta`** (*`float`*) – The number of radians to rotate counterclockwise.

`scale`(*value*, *origin=(0, 0)*)

Scale each point away/towards some origin.

Parameters

- **`value`** (*`float`*) – The scale amount.
- **`origin`** – An `(x, y)` tuple or *`Point`* from which the collection's points will move away/towards.

`translate`(*delta*)

Translate the collection by the vector delta.

Parameters **`delta`** – An `(x, y)` tuple or *`Point`*, treated as a vector and added to each point in the collection.

class `soar.sim.geometry.`**`Pose`**(*x*, *y*, *t*)

Bases: *`soar.sim.geometry.Point`*

A point facing a direction in the xy plane.

Poses can be interpreted and treated as `(x, y, theta)` tuples in most cases.

Parameters

- **`x`** – The x coordinate of the pose.
- **`y`** – The y coordinate of the pose.
- **`t`** – The angle between the direction the pose is facing and the positive x axis, in radians.

`copy`()

Returns a copy of the Pose.

`is_near`(*other*, *dist_eps*, *angle_eps*)

Determine whether two poses are close.

Parameters

- **other** – An `(x, y, t)` tuple or *Pose*.
- **dist_eps** (*float*) – The distance epilson within which to consider the poses close.
- **angle_eps** (*float*) – The angle episilon within which to consider the poses close.

Returns `True` if the distance between the point portions is within `dist_eps`, and the normalized difference between the angle portions is within `angle_eps`.

Return type bool

point()
Strips the angle component of the pose and returns a point at the same position.

Returns The `(x, y)` portion of the pose, as a *Point*.

rotate(*pivot*, *theta*)
Rotate the point portion of the pose about a pivot. Leaves the theta portion unchanged.

Parameters

- **pivot** – A *Point*, subclass (like *Pose*), or `(x, y)` tuple as the pivot/axis of rotation.
- **theta** (*float*) – The number of radians to rotate by. Positive values are counterclockwise.

transform(*other*)
Return a new pose that has been transformed (translated and turned).

Parameters **other** – A *Pose* or 3-tuple-like object, by which to translate and turn.

Returns A new *Pose* equivalent to translating `self` by `(other[0], other[1])` and rotating by `other[2]`.

xyt_tuple()
Returns: An `(x, y, t)` tuple representing the pose.

soar.sim.geometry.**ccw**(*p1*, *p2*, *p3*)
Determine whether the turn formed by points p1, p2, and p3 is counterclockwise.

Parameters

- **p1** – An `(x, y)` tuple or *Point* as the start of the turn.
- **p2** – An `(x, y)` tuple or *Point* as the midpoint of the turn.
- **p3** – An `(x, y)` tuple or *Point* as the end of the turn.

soar.sim.geometry.**clip**(*value*, *m1*, *m2*)
Clip a value between two bounds.

Parameters

- **value** (*float*) – The value to clip.
- **m1** (*float*) – The first bound.
- **m2** (*float*) – The second bound.

Returns A clipped value guaranteed to be between the min and max of the bounds.

Return type float

soar.sim.geometry.**normalize_angle_180**(*theta*)
Normalize an angle in radians to be within `-pi` and `pi`.

Parameters **theta** (*float*) – The angle to normalize, in radians.

Returns The normalized angle.

Return type float

`soar.sim.geometry.`**`normalize_angle_360`**(*theta*)

Normalize an angle in radians to be within `0` and `2*pi`.

Parameters **`theta`** (*`float`*) – The angle to normalize, in radians.

Returns The normalized angle.

Return type float

5.9.2 soar.sim.world

Soar World and WorldObject classes/subclasses, for simulating and drawing worlds.

`class` `soar.sim.world.`**`Block`**(*p1*, *p2*, *thickness=0.002*, ***options*)

Bases: *`soar.sim.world.Polygon`*

An arbitrarily thick wall centered on a line.

Useful when infinitely-thin lines are causing issues with collision detection.

Parameters

- **`p1`** – An `(x, y)` tuple or *`soar.sim.geometry.Point`* as the first endpoint of the line segment.
- **`p1`** – An `(x, y)` tuple or *`soar.sim.geometry.Point`* as the second endpoint of the line segment.
- **`thickness`** (*`float`*) – The thickness of the block to expand out from the line on which it is centered.
- **`**options`** – Tkinter options.

`borders`

`draw`(*canvas*)

Draw the object on a canvas.

Parameters **`canvas`** – A Tkinter Canvas or a subclass, typically a *`soar.gui.canvas.SoarCanvas`*, on which the object will be drawn.

`class` `soar.sim.world.`**`Polygon`**(*points*, *center=None*, ***options*)

Bases: *`soar.sim.geometry.PointCollection`*, *`soar.sim.world.WorldObject`*

A movable polygon with Tkinter options.

Parameters

- **`points`** – A list of `(x, y)` tuples or *`soar.sim.geometry.Point`*.
- **`center`** – An `(x, y)` tuple or *`soar.sim.geometry.Point`* as the pivot or center of the collection.
- **`**options`** – Tkinter options.

`borders`

`collision`(*other*, *eps=1e-08*)

Determine whether the polygon intersects with another *`WorldObject`*.

Parameters

- **other** – Either a *Polygon* or a *Wall* as the other object.
- **eps** (*float, optional*) – The epsilon within which to consider a collision to have occurred.

draw(*canvas*)
Draw the object on a canvas.

Parameters canvas – A Tkinter Canvas or a subclass, typically a *soar.gui.canvas.SoarCanvas*, on which the object will be drawn.

class soar.sim.world.**Ray**(*pose*, *length*, *eps=1e-08*, ***options*)
Bases: *soar.sim.world.Wall*

A ray of a specified length, origin and direction.

Parameters

- **pose** – An (x, y, theta) tuple or *soar.sim.geometry.Pose* as the origin of the ray.
- **length** (*float*) – The length of the ray.
- ****options** – Tkinter options.

class soar.sim.world.**Wall**(*p1*, *p2*, *eps=1e-08*, ***options*)
Bases: *soar.sim.world.WorldObject*, *soar.sim.geometry.LineSegment*

A straight wall with Tk options and collision detection.

Note that these walls are infinitely thin, so on-edge collision cannot occur.

Parameters

- **p1** – An (x, y) tuple or a *soar.sim.geometry.Point* as the first endpoint of the wall.
- **p1** – An (x, y) tuple or a *soar.sim.geometry.Point* as the second endpoint of the wall.
- **eps** (*float*) – The epsilon within which to consider a wall vertical or horizontal.
- ****options** – Tkinter options.

collision(*other*, *eps=1e-08*)
Determine whether two walls intersect.

Parameters

- **other** – A *Wall*, or other LineSegment subclass.
- **eps** (*float*) – The epsilon within which to consider two parallel lines the same line.

Returns A list of (x, y) tuples consisting of the intersection(s), or None if the segments do not intersect.

draw(*canvas*)
Draw the object on a canvas.

Parameters canvas – A Tkinter Canvas or a subclass, typically a *soar.gui.canvas.SoarCanvas*, on which the object will be drawn.

class soar.sim.world.**World**(*dimensions*, *initial_position*, *objects=None*)
Bases: object

A simulated world containing objects that can be simulated stepwise and drawn on a *soar.gui.canvas.SoarCanvas*.

dimensions
tuple – An `(x, y)` tuple representing the worlds length and height.

initial_position
An `(x, y, theta)` or *soar.sim.geometry.Pose* representing the robot's initial position in the world.

objects
list – A list of (*WorldObject*, layer) tuples containing all of the world's objects.

layer_max
int – The highest layer currently allocated to an object in the world.

canvas
An instance of *soar.gui.canvas.SoarCanvas*, if the world is being drawn, otherwise `None`.

Parameters

- **dimensions** (*tuple*) – An `(x, y)` tuple representing the worlds length and height.
- **initial_position** – An `(x, y, theta)` or *soar.sim.geometry.Pose* representing the robot's initial position in the world.
- **objects** (*list*) – The initial *WorldObject* (s) to add to the world

add(*obj*, *layer=None*)
Add an object to the world, with an optional layer specification.

Parameters

- **obj** – A *WorldObject* (or a subclass instance).
- **layer** (*int*) – The layer on which the object is to be drawn. Objects are drawn in order from smallest to largest layer. If this argument is `None`, the object's layer will be set to one higher than the highest layer in the objects list.

delete(*canvas*)
Delete the world from a canvas, by deleting each object at a time.

Parameters canvas – The *soar.gui.canvas.SoarCanvas* from which to delete.

draw(*canvas*)
Draw the world on a canvas.

Objects are drawn in order from the lowest to highest layer if their `do_draw` attribute is True.

Parameters canvas – The *soar.gui.canvas.SoarCanvas* on which to draw the world. How each object is drawn is up to the object.

find_all_collisions(*obj*, *eps=1e-08*, *condition=None*)
Finds all the collisions of a *WorldObject* subclass with objects in the world.

Parameters

- **obj** – A *WorldObject* or subclass instance. Objects in the world must know how to collide with it.
- **eps** (*float, optional*) – An optional epsilon within which to consider a collision to have occurred. What that means differs between *WorldObject* subclasses.
- **condition** (*optional*) – A function to apply to each object in the world that must be `True` in order for it to be considered.

Returns A list of `(world_obj, p)` tuples, where `world_obj` is the object that collided and `p` is the *`soar.sim.geometry.Point`* at which the collision occurred. If multiple collision points occurred with the same object, each will be listed separately. If no collisions occurred, returns `None`.

Return type list

`on_step`(*step_duration*)

Perform a single step on the world's objects.

Parameters **`step_duration`** (*`float`*) – The duration of the step in seconds.

`class` `soar.sim.world.`**`WorldObject`**(*do_draw*, *do_step*, *dummy=False*, ***options*)

Bases: `object`

An object that can be simulated and drawn in a *`soar.sim.world.World`* on a *`soar.gui.canvas.SoarCanvas`*.

Classes that are designed to work with a World in Soar may either subclass from this class or implement its methods and attributes to be considered valid.

`do_draw`

bool – Used by a *`World`* to decide whether to draw the object on a canvas.

`do_step`

bool – Used by a *`World`* to decide whether to step the object in simulation.

Parameters

- **`do_draw`** (*`bool`*) – Sets the value of the *`do_draw`* attribute.
- **`do_step`** (*`bool`*) – Sets the value of the *`do_step`* attribute.
- **`dummy`** (*`bool`*) – Whether the object is a dummy–that is, not intended to be drawn or stepped, but used for some intermediate calcuation (usually collision).
- **`**options`** – Tkinter options. This may include 'tags', for drawing on a canvas, line thickness, etc.

`collision`(*other*, *eps=1e-08*)

Determine whether two *`WorldObject`* (s) collide.

Objects that subclass *`WorldObject`* should implement collision detection for every applicable class from which they inherit.

Parameters

- **`other`** – A supported *`WorldObject`* subclass with which this object could potentially collide.
- **`eps`** (*`float, optional`*) – The epsilon within which to consider a collision to have occurred, different for each subclass.

Returns A list of `(x, y)` tuples consisting of all the collision points with `other`, or `None` if there weren't any.

Return type list

`delete`(*canvas*)

Delete the object from a canvas.

Parameters **`canvas`** – A Tkinter Canvas or a subclass, typically a *`soar.gui.canvas.SoarCanvas`*, from which the object will be deleted.

`draw`(*canvas*)

Draw the object on a canvas.

Parameters **`canvas`** – A Tkinter Canvas or a subclass, typically a *`soar.gui.canvas.SoarCanvas`*, on which the object will be drawn.

`on_step`(*step_duration*)

Simulate the object for a step of a specified duration.

Parameters **`step_duration`** – The duration of the step, in seconds.

5.10 soar.update

Function to check for updates on PyPI, and return a message for the user.

`soar.update.`**`get_update_message`**()

Fetch the HTML of Soar's PyPI page, check for a newer version, and return a notification string.

Returns An empty string if no update is available, a notification message if one is, and an error message if something went wrong.

Return type str

CHAPTER 6

Development

Only stable releases of Soar will be published to PyPI or the releases. Development versions will exist only in the GitHub repo itself, and will be marked with a `.dev<N>` suffix.

Typical versioning will look like the following: `<MAJOR>.<MINOR>.<PATCH>`. Major releases break backward compatibility, minor releases add functionality but maintain backward compatibility, and patch releases address bugs or fix small things.

If you have a specific feature you'd like to see in Soar, or a specific robot type you'd like bundled with the base software, or just want to contribute, consider opening a pull request.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index