

UP AND RUNNING WITH RUNSWIFT

A GUIDE FOR RMIT'S ROBOCUP STANDARD LEAGUE
DEVELOPERS

Table of Contents

Minimum and Recommended System Specifications	2
Potential Areas of Improvement on Host	2
Further Development Work	2
Installation Guide for rUNSWift, Simspark and Roboviz	3
Preamble	3
Section 1 - Install rUNSWift, Simspark and Roboviz	3
Installing the Virtual Machine	3
Setting up the Virtual Machine	5
Setting up Github	6
Installing Dependencies	6
Section 2 – Run the Simulation	7
Offnao Getting Started Guide	9
Developer Guide of Robot Behaviours	11
Introduction	11
Configuration	11
What is the Blackboard?	11
Existing Python Modules	12
Folders	12
Creating a new behaviour	13
How does python interact with C++ vice versa	15
Developer Traps to Avoid	17
Don't try to SSH into a Simulated Robot	17
Offnao's Featureless Features	17
Irrelevant Logging	17

Minimum and Recommended System Specifications

	Minimum Host System Requirements	Recommended Host System Requirements
Host OS	Windows, Linux, Mac OS X, Solaris and OpenSolaris	MacOS High Sierra
Processor	2.6Ghz Dual Core	2.6Ghz Quad Core with SMT or equivalent
Memory	8GB	16GB or higher
Free Disk Space	35GB	50GB
Screen Resolution	1024x768	1920x1080
Native VT-x/VT-d Support	Untested without VT-x/VT-d	Turned on
Notes	This specification has been designed with the specified operating system running on the host. It may work for other operating systems but it is untested, please view Oracle VM VirtualBox User Manual for a list of all supported host operating systems.	

Installation Guide for rUNSWift, Simspark and Roboviz

Preamble

This installation guide explains how to set up a virtual machine that can run rUNSWift. This covers either setting up the virtual machine from scratch or downloading and using an image of an already setup virtual machine (pre-built VDI).

Install from scratch

In the case of installing a new virtual machine from scratch, please follow the steps starting at [Section 1 - Install rUNSWift, Simspark and Roboviz](#).

Install using pre-built image

If you want to setup the pre-built VirtualBox Disk Image (VDI), it can be downloaded from here (11GB):

https://drive.google.com/open?id=1NkRU0_Pc2tCodKehz9uFYxsa8qFy5oPh

The password for the pre-built image is "naosoccer18"

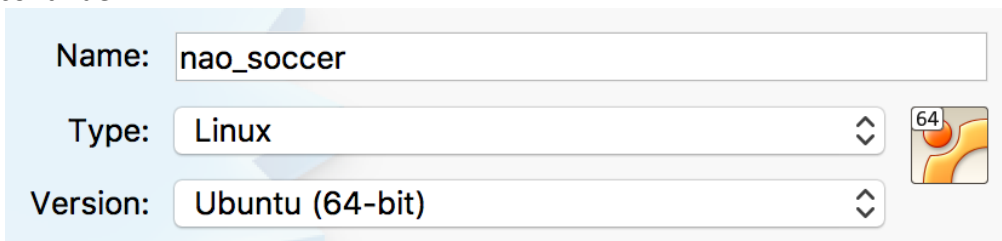
Follow the guide paying attention to notes for using the pre-built image.

NOTE: The operating system that currently works without any issues is Ubuntu 14.04.2 LTS 64-bit (the long-term release one). Attempts to setup the rUNSWift environment on 14.04.3 and later versions have failed thus far.

Section 1 - Install rUNSWift, Simspark and Roboviz

Installing the Virtual Machine

1. Download [Oracle VM Virtualbox Manager](#) for macOS or other operating systems in the following link: <http://download.virtualbox.org>
2. Download [Ubuntu 14.04.2 LTS 64-bit](#) (the long-term release one).
3. Click on New in virtual box, set up the operating system as following, then click on continue:



Name:

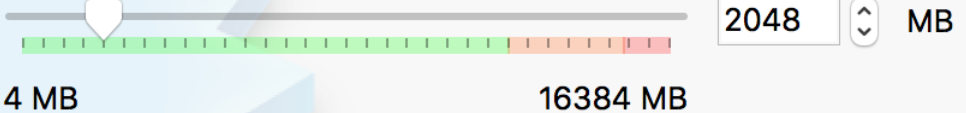
Type:

Version:

4. In the next page, choose memory size, click on continue (Recommend 4 GB memory):

Select the amount of memory (RAM) in megabytes to be allocated to the virtual machine.

The recommended memory size is **1024 MB**.



4 MB 16384 MB

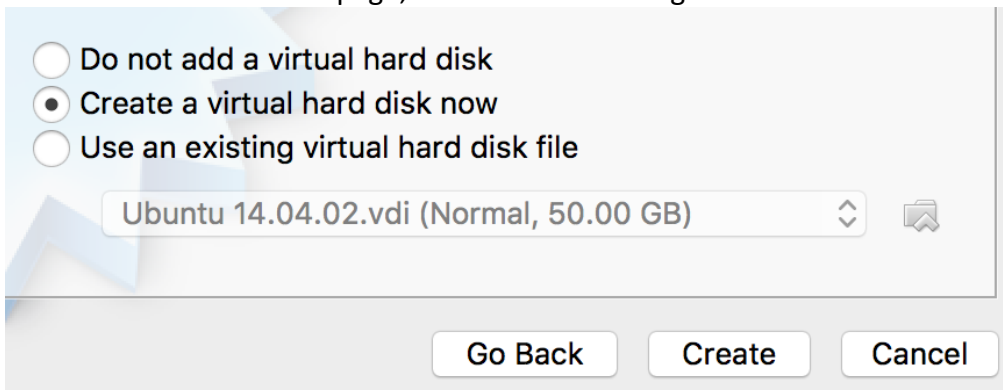
2048 MB

5. Setting up Hard Disk

Follow 5.1 if building from scratch, follow 5.2 if building from pre-built VDI

5.1 Building from scratch

In the Hard Disk selection page, choose the following one:



☐ Do not add a virtual hard disk

☒ Create a virtual hard disk now

☐ Use an existing virtual hard disk file

Ubuntu 14.04.02.vdi (Normal, 50.00 GB)

Go Back Create Cancel

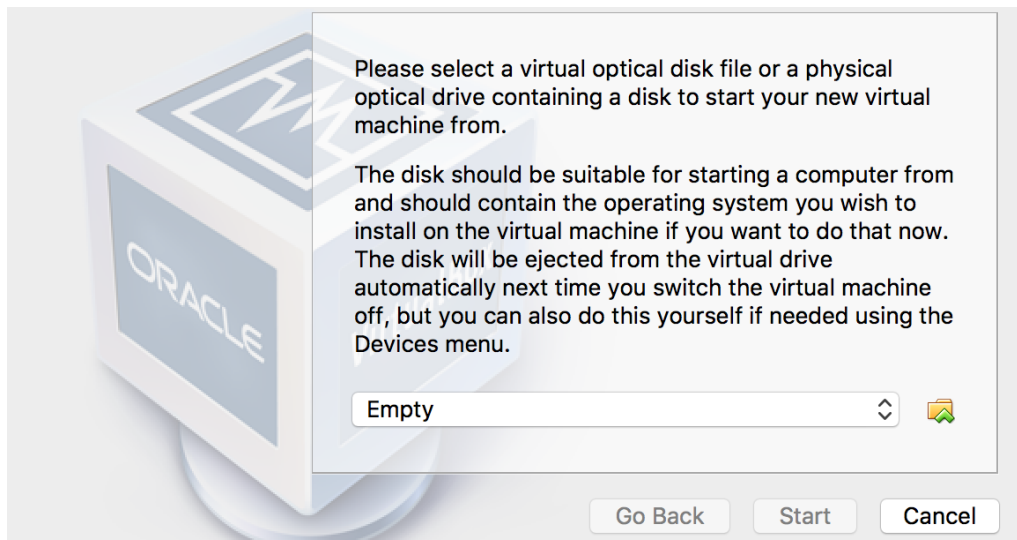
And choose VDI (VirtualBox Disk Image) -> Fixed size -> Choose hard disk size (recommend 30GB hard disk) and click on create.

5.2 Using pre-built VDI image

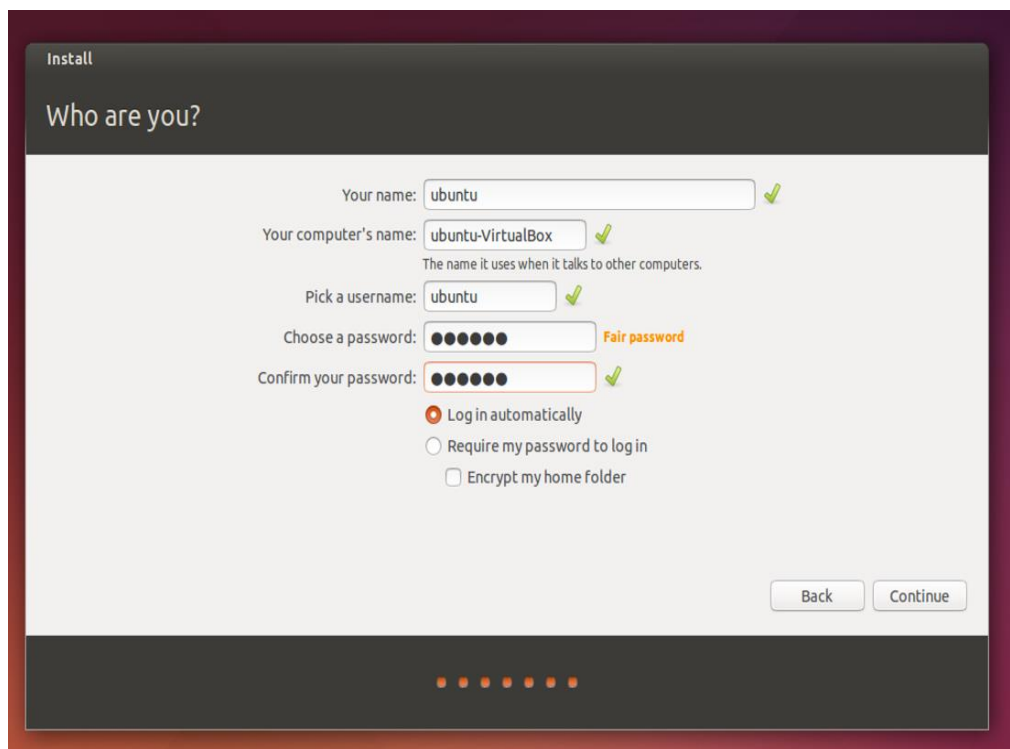
In the Hard disk selection page choose 'Use an existing virtual hard disk file' and then select the image you downloaded from

https://drive.google.com/open?id=1NkRU0_Pc2tCodKehz9uFYxsa8qFy5oPh

- Choose the ISO Ubuntu 14.04.2 LTS 64-bit you just downloaded and click on start then follow the instructions to install Ubuntu. **NOTICE: The username of this virtual machine MUST BE "Ubuntu".**



NOTE: If using the pre-build VDI image you will already have Ubuntu installed. The password for the VDI's Ubuntu installation is "**naosoccer18**" without the quotes.



Setting up the Virtual Machine

7. *OPTIONAL: Click on Devices -> Insert Guest Additions CD image in ubuntu can improve frame rates and performance depending on your machine.*

8. Install git and gitk by following command:

```
sudo apt-get install git gitk
```

Setting up Github

9. Generate a new ssh key by the following command:

```
ssh-keygen -t rsa
```

Please leave the filename and passphrase empty by press ENTER.

10. Open the key by following command and copy the string in the key you just created:

```
gedit ~/.ssh/id_rsa.pub
```

11. In the [Github key management page](#), choose “New SSH Key” and paste the string in Key text area, then click on Add SSH Key.

12. Set up your git settings to correspond to your GitHub account:

```
git config --global user.name "Firstname Lastname"  
git config --global user.email "your_email@youremail.com"
```

Installing Dependencies

NOTE: Skip this section and move to 'Section 2 – Run the simulation' if you are using the prebuilt VDI.

13. Install dependencies including CMake 2.8.12.2:

```
sudo apt-get install cmake zlib1g-dev libglib2.0-dev
```

Extra libraries for 64 bit.

```
sudo apt-get install libc6-i386 lib32z1-dev libstdC++6:i386  
libbz2-1.0:i386 libfontconfig:i386 libglib2.0-0:i386  
libsm6:i386 libxrandr2:i386 libfontconfig1:i386
```

If the installation is aborted abnormally, you can close this terminal and open a new terminal to continue.

14. Retrieve our repository and run the build, close all shells when the build is completed:

```
git clone git@github.com:RMIT-RoboCup-Standard-League/PP1-Nao-Soccer.git rUNSWift  
(^^ the line above is all one line in the terminal)
```

```
cd rUNSWift
```

```
sudo bin/build_setup.sh
```

Please view the message after build, if the build process terminated with files existing errors, run the last command again. During the process of running buid_setup.sh and sim_setup.sh, the size of required resources are quite large and

you may terminate the processes before it completed, so you will need to remove the .zip or .tar.gz and their extracted folders and start the command again.

(OPTIONAL) Once you have completed git setup, you can synchronize your branch with the latest change in the remote branch on Github by the command:

git fetch

git merge origin master

15. Install Roboviz simulation monitor and close all shells after the installation is completed:

```
rUNSWift/bin/sim_setup.sh
```

16. Install Simspark simulation server:

```
sudo add-apt-repository universe
sudo add-apt-repository multiverse
sudo apt-add-repository ppa:gnurubuntu/rubuntu
sudo apt-get update
sudo apt-get install rcssserver3d
```

17. Change access permission of all files in rUNSWift folder:

```
sudo chmod 777 -R rUNSWift
```

18. Build a new instance for simulation purpose only, do not run the build on a NAO:
sim_build

If all setup steps have been completed, you can start a new simulated instance following section 2 “Run the Simulation”, if any error occurs, please go to Troubleshooting in GitHub (<https://github.com/RMIT-RoboCup-Standard-League/PP1-Nao-Soccer/wiki/Troubleshooting>).

Section 2 – Run the Simulation

1. Open a new terminal and initiate the simulation server:

```
rcssserver3d
```

2. Open a new terminal and initiate the internal monitor:

```
roboviz.sh
```

Alternatively, you can use ‘rcsoccersim3d’ to open both the server and the simspark monitor for better performance.

3. Open a new terminal and start a new simulated instance (ensure the state in Roboviz monitor is set to BeforeKickOff by press O):


```
sudo rUNSWift/build-release/robot/runswift [-n num] [-T team]
[-s behaviour]
```

e.g. `sudo rUNSWift/build-release/robot/runswift -n 1 -T 22 -s WalkKickDribble`
where num is the player number, team is the team number and behaviour is the simulated robot's behaviour. A list of behaviours can be found [here](#).

4. Change access permissions for the rUNSWift log folder. NOTICE: This folder will only be created after you have already run `rcssserver3d`, `roboviz.sh` and `runswift`:

```
sudo chmod 777 -R /var/volatile/runswift
```

Once you have done this step, you can remove `sudo` in the front of `runswift` command in step 3 because the permission of creating log files in that folder has been changed.

5. The '`sim_run`' command can initiate multiple Simswift instances on a specified team, however it redirects Simswift output to `/dev/null`, so no robot output will be seen in console (to see rUNSWift output messages, use '`runswift`' command for a single instance):

```
sim_run [team] [number of robots]
```

e.g. `sim_run 18 5`

If you want to add behaviours for robots, please run `runswift` command in step three of this section.

6. You should be able to find the simulated robots in Roboviz after all steps, and a game controller need to be initiated by the command:

```
gamecontroller
```

7. To remove all simulated instances in Roboviz or Simspark monitor you can use the following command:

```
sim_kill
```

END OF INSTALLATION GUIDE

Offnao Getting Started Guide

To connect to simulated robots, we need to follow these steps:

1. Start the simulation server in a new terminal by typing:

```
rcsserver3d
```

2. Create a simulated instance in a new terminal:

```
sudo rUNSwift/build-release/robot/runswift
```

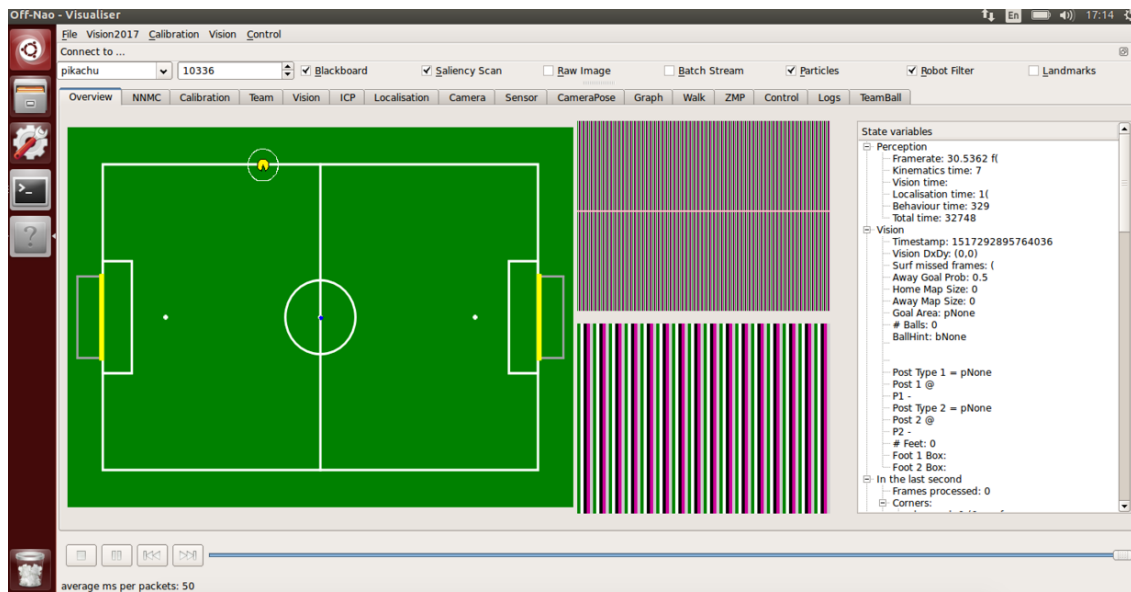
3. Open Offnao:

```
offnao
```

4. Choose File -> Connect to ... And enter the port number shown in console before (If no arguments are added for runswift, the default is 10336).



5. Choose a name in the dropdown menu on the left e.g. pikachu (this step is to refresh the frame so Offnao can connect to the port).
6. Hit the red record button in the bottom left corner and start the recording.



Developer Guide of Robot Behaviours

Introduction

Behaviour development is implemented in python then executed by robot C++. Runswift will be constantly checking for any changes in python files during runtime. By breaking up the development this way you can constantly iterate on your python behaviour and test it without having to edit or recompile any C++. You don't even need to reset runswift for your changes to take effect.

All python modules are kept in:

[rUNSWift/image/home/nao/data/behaviours/](#)

Configuration

Each robot will inevitably have some differences in both its physical characteristics and desired behaviours e.g. goalie vs striker.

The [runswift.cfg](#) file stored in [rUNSWift/image/home/nao/data/](#) is the recommended way to store these robot specific configurations.

To see the full list of what can be set in this config file look at options.cpp stored in [rUNSWift/robot/utis/options.cpp](#). This is also where all the default options are set if no arguments are given in runswift.cfg

There is a lot in options.cpp but for our purposes the most relevant is the behaviour.skill option. This option sets what will be the first behaviour to execute when the robot spins up. The default will be gamecontroller. If we want to run another behaviour as default such as WalkInACircle, you can change this by adding the following to runswift.cfg

```
[behaviour]
skill=WalkInACircle
```

Note: You should include the class name here not the file name e.g WalkInACircle.py will not work.

This is useful as you can then load the same code onto all robots and then just define what role a robot will take through this option in the config file.

NOTE: You can also set this and other options as a command line argument e.g

```
runswift -s WalkInACircle
```

What is the Blackboard?

[Blackboard.cpp](#)

The Blackboard is fundamental in how the runswift code communicates between threads and modules. The Blackboard is essentially a data structure created to store global data. Every thread and class/module has access to the same Blackboard and so it is used to pass messages between threads and classes/modules rather than trying to return everything up a chain of classes and back down to the target. The Blackboard is broken into subsections categorising the data into a related thread or module e.g. BlackboardBehaviours(). The

blackboard object is created in [main.cpp](#)

The initial values in the blackboard are created from the config file runswift.cfg found in rUNSWift/image/home/nao/data/

If an option is not set in the config file the default values found in [rUNSWift/robot/utlis/options.cpp](#) are used.

The implementation of Blackboard.cpp uses the C++ boost library [Boost.Program options](#). If you want to understand how the blackboard works in more detail have a read through these docs.

Existing Python Modules

For a visual representation please view the [Software Architecture Diagram](#)

Note: The diagram is simplified to make it a more useful aid in understanding the overall structure of the code. For a full understanding its recommended you have a dig through the directories .

[Constants.py](#)

Stores constants to be used by python modules.

[Global.py](#)

A list of helper functions that can be used by python modules.

[World.py](#)

Acts as a connection to environment variables through blackboard. This wraps up the blackboard and is passed through to behaviourTasks objects.

[Actioncommand.py](#)

A list of readymade functions that return actioncommand data structures. These are used to make action requests of the robot e.g. walk(). These are used in behaviourTasks.

[Task.py](#)

Defines the interface used for behaviourTasks objects and TaskState objects. behaviourTasks objects are where most of your development will take place. This is where we programme what we want the robot to do.

TaskStake objects are used as subroutines in behaviourTasks, helping us break complicated behaviours down into its sub-components.

[Sensors.py](#)

Group of objects that return sensor information to be used in python modules.

[Behaviour.py](#)

This is the main entry point from the robot into python. It loads a behaviourTask object to run, which is based of what is set in the behaviour.skill option in the config file runswift.cfg. Then calls 'behaviourToBeRun'.tick() to execute that behaviour.

Folders

This is where you store behaviours being developed
[../roles](#) - store high level behaviours e.g. goalie

[../skills](#) - store mid level behaviours used in high level behaviours

[../tests](#) - store you guessed it test behaviours used for testing

Note: it doesn't really matter where you store your behaviour as long as it's in one of these folders.

Creating a new behaviour

1. Create 'aNewBehaviour.py' and store it in one of the roles, skills or tests folders.

2. Create a new behaviourTask class. Let's use StandStraight.py found in the tests/ folder as an example.

```
from Task import BehaviourTask
import actioncommand

# Creating a new BehaviourTask object
class StandStraight(BehaviourTask):

    # Setup and decide what class should be run first.
    def init(self):
        pass # We want 'StandStraight' to run so we will leave
            # this empty. More on this later

    # This function decides what class we should run next
    def transition(self):
        pass # Again we will leave this empty

    # This is the function that decides what behaviour to
    # execute.
    def _tick(self):
        self.world.b_request.actions.body =
            actioncommand.standStraight()
```

NOTE: transition() always executes before _tick. See the function tick() in Task.py to see why. If you don't want this to happen use the function tick() in your behaviourTask instead of _tick()

3. If we want to make a more complicated behaviour, we can break the code up into sub-behaviours and each behaviour can be broken down further into TaskStates. Using the transition() function we can change what class/behaviour we should run next. Either a different behaviourTask entirely or one of the taskStates. Lets use WalkAround found in tests/ folder as an example of changing between TaskState objects.

```
from Task import BehaviourTask, TaskState
from util.Timer import Timer
import actioncommand

# The parent class. Notice the only thing defined here is init()
# _tick() and transition() is still defined but we are just using
```

```

# the implementation of these functions found in the super class
# BehaviourTask
class WalkAround(BehaviourTask):
def init(self):
    # Set the next class to run. current_state is initially
    # set to null in the super class BehaviourTask.__init__ found
    # in Task.py
    self.current_state = WalkForwardState(self)

# A subroutine class to break up our BehaviourTask class.
# TaskStates are very similar to BehaviourTasks but lack the
# initialisation done in BehaviourTask objects.
class WalkForwardState(TaskState):
    def init(self):
        self.timer = Timer(3000000).start()

    # Here after each execution of tick we check back to see if
    # the timer has finished and if so return trunState. What
    # this does is change the current_state variable to the class
    # trunState. Why this works is explained further down in
    # this manual
    def transition(self):
        if self.timer.finished():
            return TurnState(self.parent)
        return self

    # The action to be executed
    def tick(self):
        self.world.b_request.actions.body =
            actioncommand.walk(forward=200)

# Another subroutine class
class TurnState(TaskState):

    def init(self):
        self.timer = Timer(3000000).start()

    # Another example of transition
    def transition(self):
        if self.timer.finished():
            return WalkForwardState(self.parent)
        return self

    # The action to be executed
    def tick(self):
        self.world.b_request.actions.body =
            actioncommand.walk(turn=1)

```

4. This is likely a bit confusing. To make more sense of what's happening here we need to look through the [Task.py](#) file carefully and look back at the behaviours implemented above. In particular why does changing current_state variable change what class gets executed. Remember how in the class WalkAround we decided to use the super

class BehaviourTasks implementation of `_tick()`, `tick()` and `transition()`.
Well lets have a look at those now as defined in Task.py

```
# By default this transitions the current_state into the next
# but this can be overridden to support more complex (or more
# simple) behaviour.
def transition(self):
    # Because we initialise WalkAround in init() to change the
    # current state to WalkForwardState(self) the below translates to
    # next_state = self.WalkForwardState.transition()
    next_state = self.current_state.transition()
    self.current_state = next_state

# This is what gets called in behaviour.py and acts as the
# main entry point. We can see here that transition is called
# before _tick()
def tick(self, *args, **kwargs):

    # Transition into whatever state.
    self.transition()
    self._tick(*args, **kwargs)

# This function gets called in tick() and so now executes our
# behaviour code.
def _tick(self):

    # Here we can see why changing what class is kept in
    # current_state changes what gets executed. In our above
    # example WalkAround we changed the state
    # to WalkForwardState() so we are executing below
    # self.WalkForwardState().tick()
    self.current_state.tick()
```

How does python interact with C++ and vice versa

Python to C++

PythonSkill.cpp found in `rUNSWift/robot/perception/behaviour/python/` is what looks for changes in python code and executes it in runtime. The python code returns a BehaviourRequest data structure back to PythonSkill.cpp

This returned data structure stores what actions we want the robot to execute. This data is then updated onto the blackboard allowing all threads and modules to access these

requests. In the python code 'world.b_request' stores the robot.BehaviourRequest() as defined in BehaviourRequest.cpp located in rUNSWift/robot/type/BehaviourRequest.cpp

world.b_request.actions.body is the main variable you need to set. This stores an actioncommand data structure to be passed back to the robot and specifies what action we want the robot to take. As defined in rUNSWift/robot/type/ActionCommand.hpp

e.g world.b_request.actions.body = actioncommand.walk(500, 0, 0)

C++ to Python

RobotModule.cpp found in rUNSWift/robot/perception/behaviour/python/ gets compiled into a python module called robot. This is then imported in python files e.g. 'import robot'. This module acts as a wrapper converting python function calls into C++ function calls. Thus allowing the python modules to use some of the functions defined in C++.

Most importantly its through robot we are able to get the implementation of the BehaviourRequest data structure that's used to send data back to the robot.

E.g. self.b_request = robot.BehaviourRequest()

Developer Traps to Avoid

Running runswift on a simulated robot vs a real robot

If you want to test behaviours in the simulator, you can run runswift or sim_run as instructed in the rUNSWift Installation Manual. Since all simulated robots are connected to the Simspark Server, you don't use ssh commands to connect to any simulated robots. If you want to develop behaviours on the real robots, you will need to ssh into the robot to develop the python behaviours and see its effect in real time. To ssh in you will need the robots IP address which you can get by pressing the centre chest button on the robot. You need to get this IP address BEFORE installing runswift on the robot, as runswift overrides this functionality.

Offnao's Featureless Features

Offnao is a debug tool that can test Nao's vision, ball position and colour calibrations etc. You can use it for testing on real robots, but it has over several years accumulated a lot of "features" that simply don't work today. Avoid using Send String function in Offnao to either simulated or real robots. This tool is mostly used for getting feedback on localisation data and to help calibrate real robots.

Irrelevant Logging

If you view the log messages in console when creating runswift, the [network] wireless.iwconfig_flags runswift.cfg setting is dead code that can be removed in a future code release; the [transmitter]address, again from runswift.cfg is likely to be renamed to [network]transmitter_address in a future release. It should work for live robots only.

Flake8 Errors

Flake8 is a python code linter that has been included in the code base. However it seems previous teams did not actually use it. As such if you try to commit code to github with the linter you will get hundreds of formatting warnings that will not allow you to commit. A hacky way to fix these warnings is to comment out the pre-commit file. You can find this file in rUNSWift/.git/hooks/pre-commit

Add # to comment it out.

Ideally the warnings should be fixed to help improve the code quality, this was out of scope for the initial project due to time limitations.