

Kilobots simulations with Kilombo

HowTo and examples

Slides largely from «Dallai, Rapicetta - KILOMBO how-to and sample functions»



Key resources for this teaching

Linux (and the *GCC* compiler) or **OSX**

Kilombo (runs on Linux or OSX) <https://github.com/JIC-CSB/kilombo>

- see the README for installation instructions
- Detailed installation instructions, that may be necessary for some distributions, are at <http://jic-csb.github.io/kilombo/> (check also known bugs and patches if you find installation mistakes)

Any **editor** for C programming on Linux (no recommendations - any editor is fine including text editors)

Description of the API (web page):

<http://www.kilobotics.com/docs/index.html>

Introduction - Kilombo simulator

A user-controllable robot simulator consists of at least two parts:

- A core implements a physical model of the robots, their interactions and their environment
- A programming interface provides a way for users to programmatically control the robot behavior.

The C program for a Kilobot is compiled using `make` on Kilombo. It uses a simulator library that implements the physical model and provides the same functionalities of the `kilolib` API.

In Kilombo, the parallel execution of the user program is modelled by **sequentially calling the `loop` function for every robot once per simulation step.**

Introduction - Physycal model

The physical model in a robot simulator needs to incorporate the aspects of reality the robots interact with (act or observe).

The simulator keeps track of the **2D position** and the **orientation** of each robot as **time** progresses (in time slices, or time steps).

In each time step, the user program's `loop` function is run **once for each robot**.

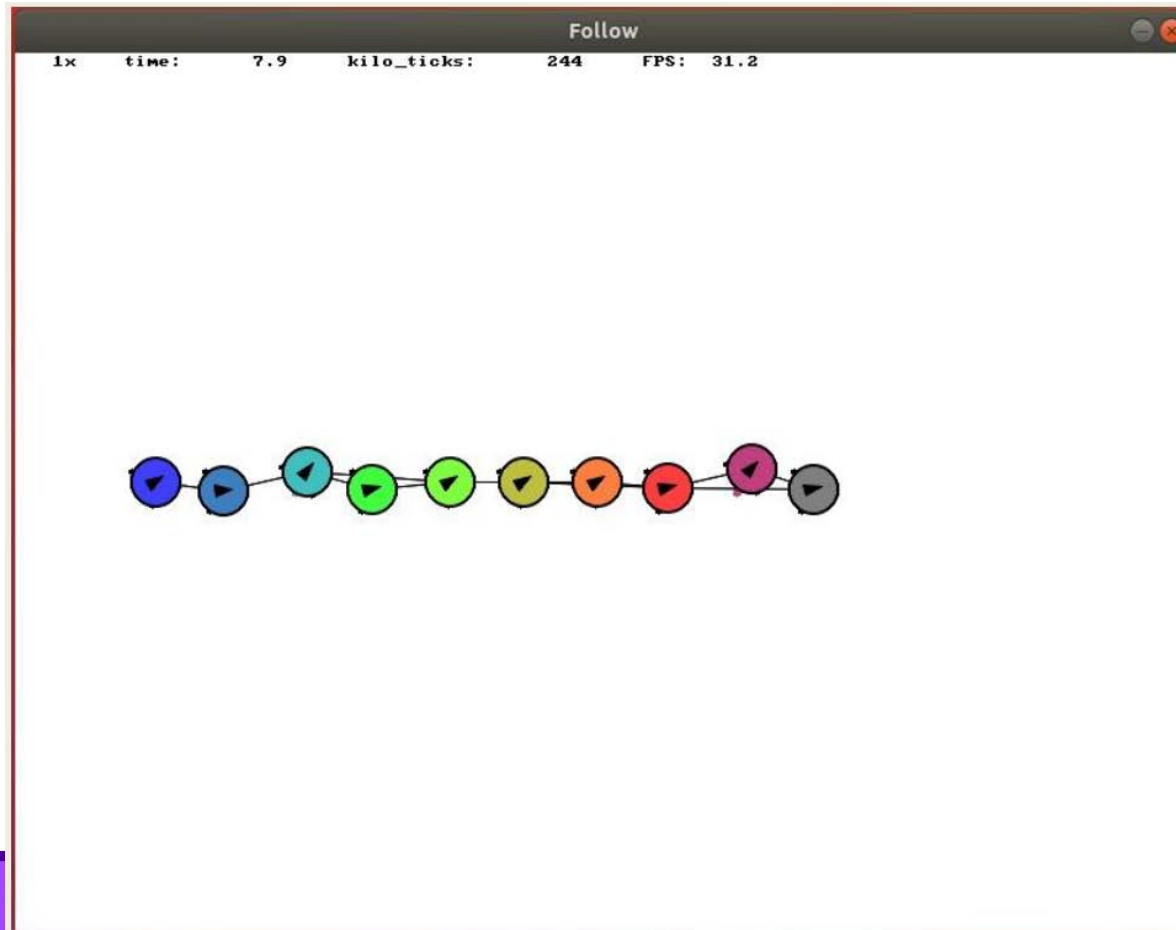
After this, the simulator updates the positions and orientations of the robots, based on their movement state, which the user program controls by turning the two motors on or off:

- if both motors are on (and same power), the robot moves forward with constant velocity;
- if only one motor is on, the robot rotates.

Introduction - user interface

The interface makes it possible to interact with the simulation at run time.

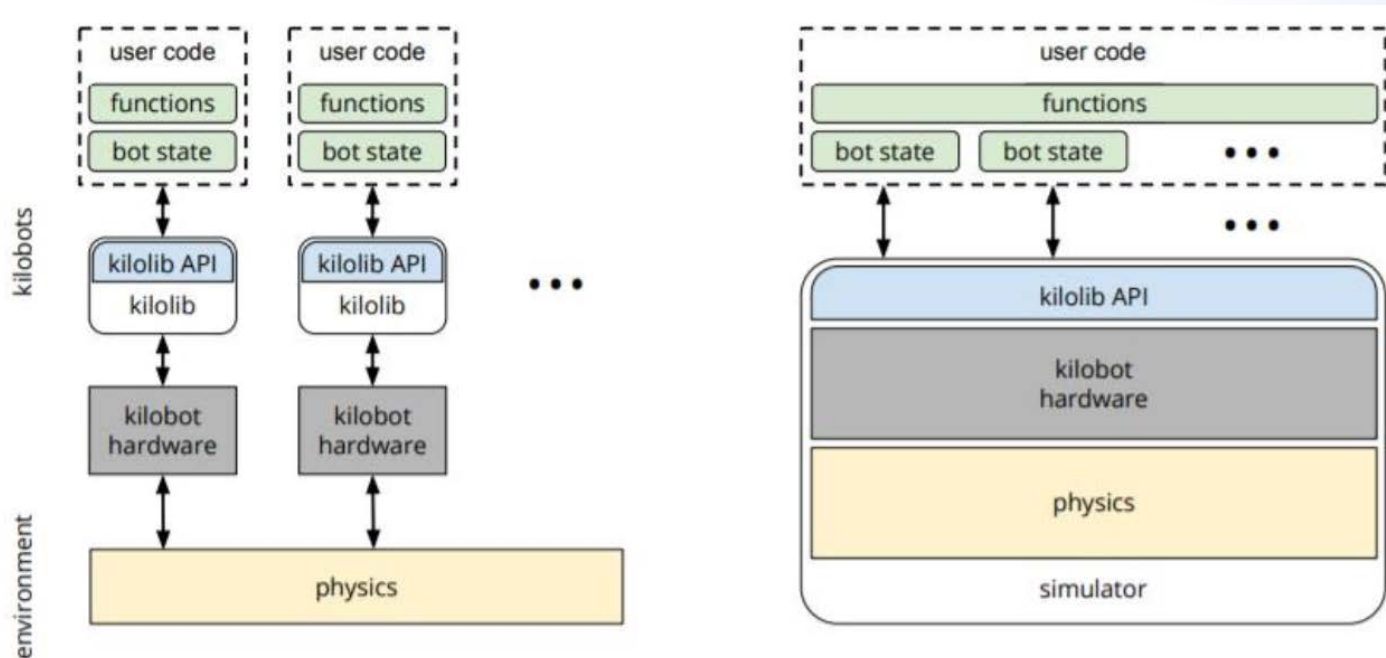
Just run any example from "Kilombo examples" now...



Real kilobots vs simulators

The user program in a real robot interacts with the outside world solely through the kilolib API.

In the simulator, each robot runs its own instance of the user program. The simulator implements the kilolib API functions and connects them to the simulator representation of the physical world.



Initialization,
execution,
messaging,
motion, and
random number generation

work with the same function calls in the simulator and in the real Kilobot.

Main API are the same, but Kilombo exposes them in a `kilombo.h`, instead of `kilolib.h`

- On Kilombo, `#include "kilombo.h"`
- On Kilobot, `#include "kilolib.h"`

Some relevant variables in kilolib.h and kilombo.h

kilo_message_rx: a callback triggered every time a message is successfully received and decoded. The callback has two parameters, a pointer to the message decoded, and a pointer to the distance measurements from the message originator.

kilo_message_tx: a callback triggered every time a message is scheduled for transmission. Returns a pointer to the message that should be sent; if the pointer is null, no message is sent.

kilo_message_tx_success: this callback is triggered every time a message is sent successfully. It receives no parameters and returns no values. **No guarantees it has been received.**

kilo_ticks: clock value, a 32-bit unsigned positive integer. It is initialized to zero whenever the program run or when the kilobot is first turned on. It is incremented approximately 32 times per second.

kilo_turn_left: an 8-bit positive integer which is the calibrated duty-cycle for turning left.

kilo_turn_right: an 8-bit positive integer which is the calibrated duty-cycle for turning right.

kilo_straight_left, kilo_straight_right → **BUGGED ON KILOMBO**

kilo_uid: the kilobot identifier

Some of the main functions

~~delay(): pauses the program for the specified amount of time.~~

get_temperature(): read the temperature of the kilobot.

get_voltage(): read the amount of battery voltage.

kilo_init(): initialize kilobot hardware.

kilo_start(setup, loop): start the event loop.

set_color(): set the output of the RGB led.

set_motors(uint8_t left, uint8_t right): set the power of each motor.

spinup_motors(): turn motors at full-speed for 15ms, to overcome the effects of static friction. **Use no more than twice per second.** It is equivalent to:

```
set_motors(255, 255);
```

```
delay(15);
```

Kilombo global variables - 1

- The simulator handles all robots in a single program, so a global or static variable ends up being common to all robots.
 - A workaround implemented in the simulator is to keep all global variables inside a single structure.
- The type of this user data has to be announced to the simulator by using the macro REGISTER_USERDATA.

```
typedef struct {  
    int N_Neighbors;  
    ...  
} USERDATA;  
REGISTER_USERDATA(USERDATA)
```

The REGISTER_USERDATA macro resolves the declaration:

```
USERDATA *mydata
```

Variables can be accessed using e.g., `mydata->N_Neighbors`.

The simulator ensures that `mydata` points to the data of the currently running kilobot.

Local variables can be used in the usual way.

Clock and timestamp

Time is measured in `kilo_ticks`, incremented 32 times per second, on real kilobots as well as the simulator.

The simulator does not implement the `'delay()'` function. The function exists, but returns immediately.

- To sleep `X` microtick: → **NOT WORKING ON SIMULATOR** (simulation steps are full executions of the loop)

```
waitTime= X + kilo_ticks
```

```
while (kilo_ticks < waitTime){ NULL };
```

- Another approach: either consider that simulation starts from 0, or you need to define workarounds e.g.,

```
if(kiloticks < waitTime) { return;} else { ... }
```

Note that in programs for real kilobots, the kilolib API documentation states that it is best to use `delay()` only for short times, like when spinning up motors, and that for timing the bot's behaviour, one should instead use the global variable `kilo_ticks`.

A note on datatype

A difference between the C compiler used for the kilobots and the C compiler used when compiling with the simulator is the size of datatypes.

- Kilobot works only with 8bit
- It may lead to code working as intended in the simulator while overflowing on the kilobot.
- A solution is to explicitly specify the size of the types, e.g. declaring variables as `uint8_t i`.

Review of differences

Original code	Code adapted for simulator	Comment
<pre>#include <kilolib.h></pre>	<pre>#include <kilombo.h></pre>	The kilombo header file assures automatic detection and compilation for either physical Kilobots or the simulator.
<pre>// Global variables int current_motion = STOP; int distance; int new_message = 0;</pre>	<pre>typedef struct { uint8_t current_motion; uint8_t dist; uint8_t new_message; } USERDATA;</pre>	Global variables are stored in a structure. Use data types with explicit sizes, e.g. uint8_t and uint16_t. Initialization is done in the setup functions.
<pre>// ...</pre>	<pre>REGISTER_USERDATA (USERDATA) // ...</pre>	This defines a pointer mydata, which points to the user data structure.
<pre>void loop() { if (dist < TOO_CLOSE) { set_motion(FORWARD); } // ...</pre>	<pre>void loop() { if (mydata->dist < TOO_CLOSE) { set_motion(FORWARD); } // ...</pre>	Access global variables in the USERDATA structure through the mydata pointer.
<pre>// blink LED once per sec set_color(RED, 0, 1); delay(500); set_color(RED, 0, 0); delay(500); }</pre>	<pre>if (kilo_ticks%31 < 16) set_color(RED, 0, 1); else set_color(RED, 0, 0); }</pre>	Use kilo_ticks for timing rather than delay().

Main functions that you should ALWAYS instantiate

`kilo_init()`: a standard function of the `kilolib.h` library, that initializes the kilobots.

`kilo_start(setup, loop)`:

- `setup` is a function which is called once to initialize the user program
- `loop` is the main body of the program, that is executed (invoked repeatedly).
- ~~you may think to include a `while(true)` inside loop~~ → PLEASE BE CAREFUL!!!

```
int main() {  
    kilo_init();  
    kilo_start(setup, loop);  
    return 0;  
}
```

Body of the code is in a file C

File .c

```
#include <kilombo.h>
#include "headerfile.h"

REGISTER_USERDATA(USERDATA)

void loop() {
    ...
}

void setup() {
}

int main() {
    kilo_init();
    kilo_start(setup, loop);
    return 0;
}
```

File .h

```
typedef struct
{
    ...
} USERDATA;
```

kilombo.json, startposition.json

```
{  
  "botName" : "Orbit bot",  
  "randSeed" : 1,  
  "nBots" : 2,  
  "timeStep" : 0.0416666,  
  "__note" : "0.04166 is 24 FPS",  
  "__timeStep" : 0.03225,  
  "simulationTime" : 0,  
  "commsRadius" : 100,  
  "showComms" : 1,  
  "showCommsRadius" : 0,  
  "distributePercent" : 0.8,  
  "displayWidth" : 640,  
  "displayHeight" : 424,  
  "displayWidthPercent" : 80,  
  "displayHeightPercent" : 80,  
  "displayScale" : 1,  
  "showHist" : 1,  
  "histLength" : 4000,  
  "storeHistory" : 1,  
  "imageName" : "./movie4/f%04d",  
  "saveVideo" : 0,  
  "saveVideoN" : 1,  
  "stepsPerFrame" : 1,  
  "finalImage" : null,  
  "stateFileName" : "simstates.",  
  "stateFileSteps" : 0,  
  "colorscheme" : "bright",  
  "speed" : 7,  
  "turnRate" : 22,  
  "GUI" : 1,  
  "msgSuccessRate" : 0.8,  
  "distanceNoise" : 2  
}
```

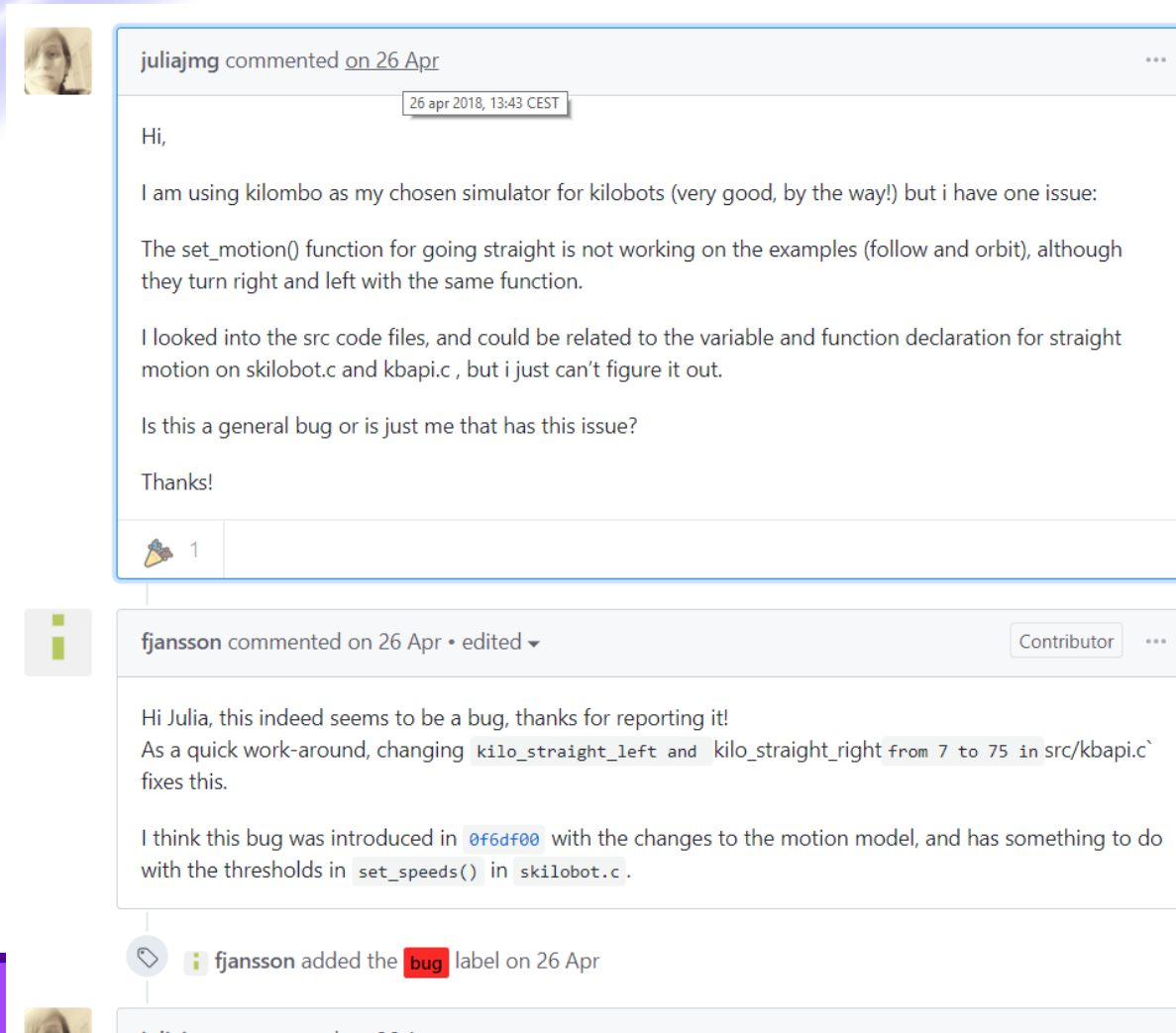
```
{  
  "bot_states": [  
    {  
      "ID": 0,  
      "direction": 5.7269688373804088,  
      "state": {},  
      "x_position": -20.900863330484285,  
      "y_position": 7.4895537075976648  
    },  
    {  
      "ID": 1,  
      "direction": -132.14180267595034,  
      "state": {},  
      "x_position": 36.053018924909047,  
      "y_position": 25.928440210936703  
    }  
  ],  
  "ticks": 7292  
}
```

↑
startposition.json

← kilombo.json

kilo_straight_left and kilo_straight_right not working

<https://github.com/JIC-CSB/kilombo/issues/45>



The screenshot shows a GitHub issue discussion. The first comment is from user 'juliajmg' on 26 Apr 2018 at 13:43 CEST. The second comment is from user 'fjansson' on 26 Apr, who has the 'Contributor' label. The issue title is 'kilo_straight_left and kilo_straight_right not working'.

juliajmg commented on 26 Apr

26 apr 2018, 13:43 CEST

Hi,

I am using kilombo as my chosen simulator for kilobots (very good, by the way!) but i have one issue:

The `set_motion()` function for going straight is not working on the examples (follow and orbit), although they turn right and left with the same function.

I looked into the src code files, and could be related to the variable and function declaration for straight motion on `skilobot.c` and `kbapi.c`, but i just can't figure it out.

Is this a general bug or is just me that has this issue?

Thanks!

1

fjansson commented on 26 Apr • edited

Contributor

Hi Julia, this indeed seems to be a bug, thanks for reporting it!

As a quick work-around, changing `kilo_straight_left` and `kilo_straight_right` from 7 to 75 in `src/kbapi.c` fixes this.

I think this bug was introduced in [0f6df00](#) with the changes to the motion model, and has something to do with the thresholds in `set_speeds()` in `skilobot.c`.

fjansson added the **bug** label on 26 Apr

We inspect one quick example....

1. Open kilombo
2. Run the Orbit example
3. Inspect the configuration files
4. Inspect the source code
 1. Check main(args)
 2. Check setup, loop
 3. Check message exchange

(we do one exercise, then we can check more complex examples with information transmission, then we do another exercise if we have time)

Exercise 1

Step 1. Set 2 kilobots (kilobot 0, kilobot 1) in *start_positions.json* file

Step 2. Create a file called *move_and_blink.c* implementing the following instructions sequentially:

Step 2.a move the kilobot 0 forward for 3 seconds;

Step 2.b Set the led color to "Purple" (1, 0, 1) to the kilobot 0;

Step 2.c Turn right the kilobot 1 for 5 seconds;

Step 2.d Set the color to "Red" (1, 0, 1) to the kilobot 1;

Step 2.e Stop the kilobot 1 and move forward (indefinitely) the kilobot 0.

Exercise 2 (need to know how to transmit data)

- Given Step 1 as before, when started, the kilobot 0 sends a "GO" message to Kilobot 1. After acknowledgment of reception, the exercise progresses as from Step 2.a of Exercise 1

Code for Exercise 1

```
#include <kilombo.h>
#include "movement.h"
REGISTER_USERDATA(USERDATA)
void loop() {
  if ( kilo_ticks >= 0 && kilo_ticks < 96 && kilo_uid == 0) {
    set_motors(kilo_turn_left, kilo_turn_right);
  } else if (kilo_ticks >= 96 && kilo_ticks < 256) {
    if (kilo_uid == 0) {
      set_color( RGB (1 , 0 , 1));
    } else {
      set_motors(0, kilo_turn_right);
    }
  } else if (kilo_ticks >= 256) {
    if (kilo_uid == 1) {
      set_color( RGB (1 , 0 , 0));
      set_motors(0,0);
    } else {
      set_motors(kilo_turn_left, kilo_turn_right);
    }
  }
}
```

```
void setup() {}
int main() {
  kilo_init();
  kilo_start(setup, loop);
  return 0;
}
```