Lab 8: Solutions

2 Localization with positioning systems

1. **(Q)**: The code we use in this lab might seem a bit longer and more complex than what you are used to. Therefore, in order to ease your task, we have already provided the whole code structure. You will only modify the `main()` function as well as some `init()` and `get()` functions.
   Take some time to read the `main()` in *controller.c*. What are the five main steps in the while loop, and can you describe a bit their goal? Read the file *odometry.h* and the lines 36 to 76 in *controller.c*. What do contain the structures `simulation_t`, `measurement_t` and `pose_t`?

   Answer:
   1. Perception / Measurement
   The robot must interpret its sensors to extract meaningful data.

   2. Localization
   The robot must determine its position in the environment.

   3. Cognition / Action
   The robot must decide how to act to achieve its goals (based on previous information).

   4. Motor Controls
   The robot must modulate its motor outputs to achieve the desired trajectory.

   5. Logging
   This is a necessary step to understand how the variables are evolving with time.

   Generally, it is a good practice to store meaningful data inside a structure in C. It might be easier to maintain, debug, and understand your code. It is also a good practice to organize your code when you start a project:
   - `simulation_t`*: contains the* `WbDeviceTag` *objects used in the simulation (sensors, actuator, ...).*
   - `measurement_t`*: contains the measurement obtained by the sensors.*
   - `pose_t`*: contains the 2D pose of the robot (i.e. x, y and orientation / heading).*

2. **(I)**: At the top of the file `controller.c`, set VERBOSE_GPS to true, to print the GPS values in the terminal. Then, compile your robot controller.

   Answer: -

3. **(S)**: Run the simulation. **Click on the window of the world** (this will let Webots capture your keyboard inputs)**,** then use your keyboard to move the robot in the arena. The corresponding action and keyboard keys are listed in Table 2.

Table 1: Keyboard keys and corresponding actions

Keyboard key	Simulation actions
R	The robot starts to move
S	The robot stops
U	Increases the speed
D	Decreases the speed
Up Arrow	The robot moves forward
Down Arrow	The robot moves backward
Left Arrow	The robot turns left

Right Arrow	The robot turns right

Hint: You need to press R first for the robot to start moving.

Answer: -

4. **(Q)**: From the values obtained in the terminal, can you deduce in which direction point the GPS uses the x, y, and z axes with respect to {World} frame? Can you move the robot to the origin of the GPS coordinates (i.e., at least, to reach the point where two of the three values are zeros)? *Hint: Look at line 195 to see the print command.*

   Answer: *The GPS uses the same convention for the x-, y- and z-axis as the world frame. The origin of the GPS frame is in the center of the arena.*

5. (**I**): The GPS does not provide any information about the orientation (heading) of the robot. One option is to use the delta position vector (difference between previous GPS values and actual ones) and to compute its angle, as can be seen in `controller_get_heading()`.

   Uncomment the indicated lines in the function `controller_get_pose()` to fill the pose structure with the GPS positions and heading. The pose should be written in `pose_t` structure with respect to the reference frame {A}, which coincides with the {Body} frame at the beginning of the motion, detailed in Figure 3.

   Note: The variable `_pose_origin` contains the position and orientation of the reference frame's origin A expressed in the world frame (same as GPS values). Its fields are `x`, `y,` and `heading`.

   Set the VERBOSE_POSE to true and compile.

   Answer: -

6. (**S**): Run the simulation by again moving the e-puck robot through the keyboard and try to follow the square. Does the pose behave as you expected? If not, try to correct your code.

   Answer: -

7. (**Q**): Is the heading angle still correct when you move your robot backward or rotate it on itself? Can you propose a sensor to get the orientation of the robot directly?

   Answer:
   If you move your robot backward, the computed heading has a shift of π w.r.t to the real one.
   If your robot rotates on itself, the delta position in the x and y directions becomes really close to 0 meters. Therefore, a small error in position can have a large impact on the computed angle. You should see a large oscillation of the orientation.
   Alongside the GPS, it is common to add a magnetometer (compass) to measure the bearing (heading) of the robot directly.

3 Odometry using an accelerometer

8. (**S**): The controller leveraged in the previous part is again used here. Set VERBOSE_ACC to true and compile the robot's controller. Start the simulation on Webots and do not move the robot. You should see the values of the accelerometer in the terminal. Why are those values not zero, what do these values mean (e.g., what force is involved)?

Answer: *We see the effect of gravity on the accelerometer. This force causes a bias on each axis of the accelerometer. To obtain the actual acceleration of the robot, one should remove these values. Take note that the effect of gravity on each accelerometer axis changes with the robot's actual orientation.*

9. (**B**): We want to focus on a thought experiment for a moment now: Imagine your e-puck is falling (no friction with air), and the accelerometer returns zero values for all three axes. With regard to the previous question (static conditions), did you expect those values? What is an accelerometer really measuring (recall the schematic representation of the accelerometer sensor mentioned in the lecture)?

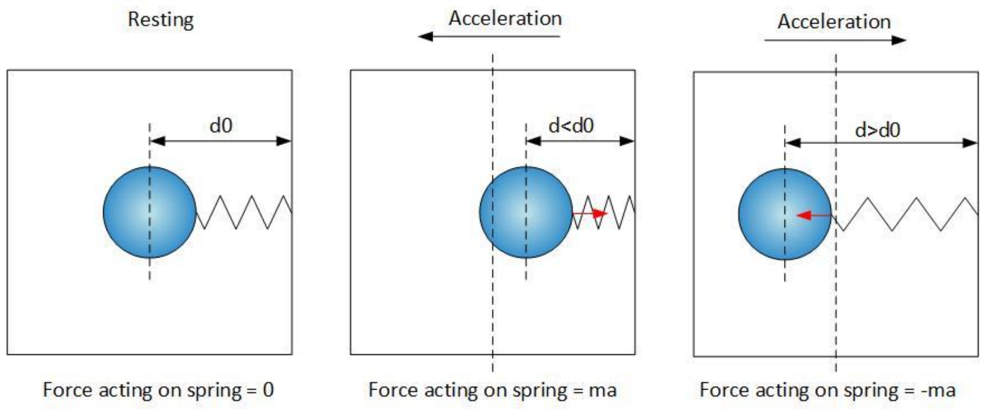
Answer: *A really good explanation comes from [1]:*

"A typical accelerometer could be modeled as a mass on a spring. The position of the mass, when the spring is not compressed or stretched, is defined as zero and corresponds to zero reported acceleration.

When the spring is compressed or stretched, the displacement of the mass from its zero position is measured (using one of many available techniques) and reported as a positive or negative acceleration.

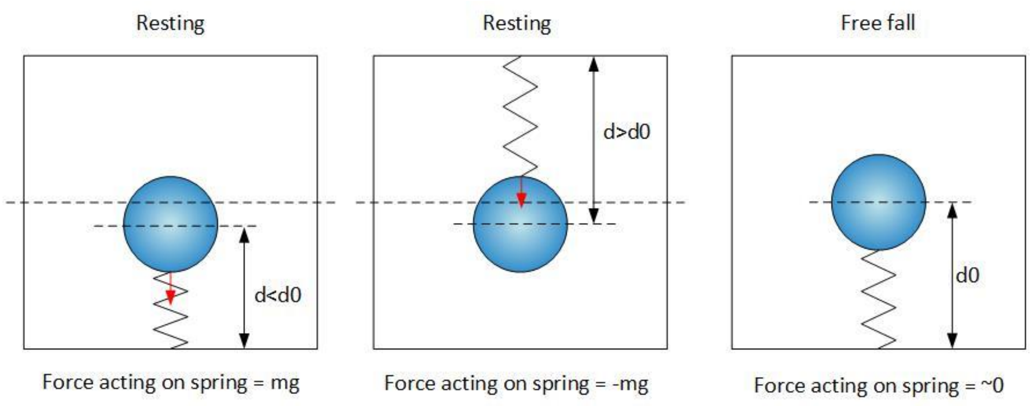
So, what is reported is a displacement of the mass, which may or may not correspond to the acceleration of the accelerometer that we would measure externally.

The diagrams below illustrate how it works in x or y directions, where gravity does not play a role.


Resting
Acceleration
Acceleration
d0
d<d0
d>d0
Force acting on spring = 0
Force acting on spring = ma
Force acting on spring = -ma

Whenever the body of the accelerometer accelerates, it pushes or pulls the spring, which, in turn, pushes or pulls the mass. The force required to accelerate the mass, ma, will cause the spring to compress or stretch, and the resulting displacement of the mass will be measured and reported as a positive or a negative acceleration.

For the z direction, illustrated in the diagrams below, the dynamics are different.


Resting
Resting
Free fall
d<d0
d>d0
d0
Force acting on spring = mg
Force acting on spring = -mg
Force acting on spring = ~0

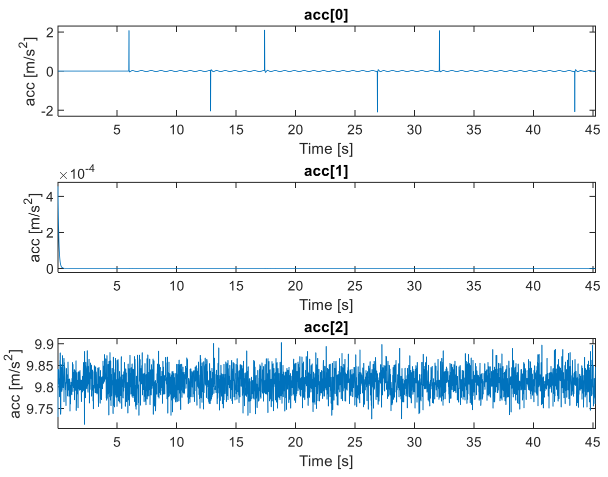
At rest, the spring is compressed or stretched to counter the weight of the ball.

In a free fall, when all parts of the accelerometer experience the same acceleration, the mass accelerates due to gravity - not due to the push or the pull of the spring. Therefore, the spring does not experience any forces; hence, there is no compression or stretching and no displacement, and the reported acceleration is zero.

This description would not be entirely accurate if the resistance of the air were considered. Since such resistance would slightly reduce the acceleration of the falling accelerometer, the spring would have to compress or stretch a bit to adjust (reduce) the acceleration of the mass accordingly."

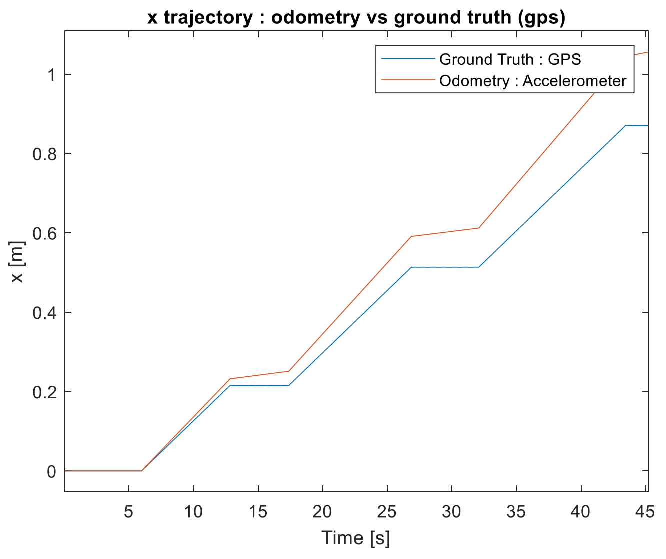
10. (**S**): Uncomment the lines (122 – 127 and 141) in `main()`. This will force the robot to stay static for five seconds. This time is used to estimate the bias of the accelerometer. The resulting mean values for the acceleration are stored in the table *_meas.acc_mean* in the file *controller.c*. Compile the robot controller and start the simulation. Let your robot move forward in the x-axis direction using the keyboard (press R). Stop it before it reaches the end of the arena (press S).
A file *data.csv* has been created inside the controller folder. This file contains the simulation logs. Run MATLAB code *plot_main.m* (only part A) to plot the values of the accelerometer. What is the frame of the accelerometer? Which indexes of the accelerometer array (*acc*) correspond to the x-, y- and z-axis of frame A (see Figure 2)?

Answer:
The accelerometer frame is the same as the body frame.

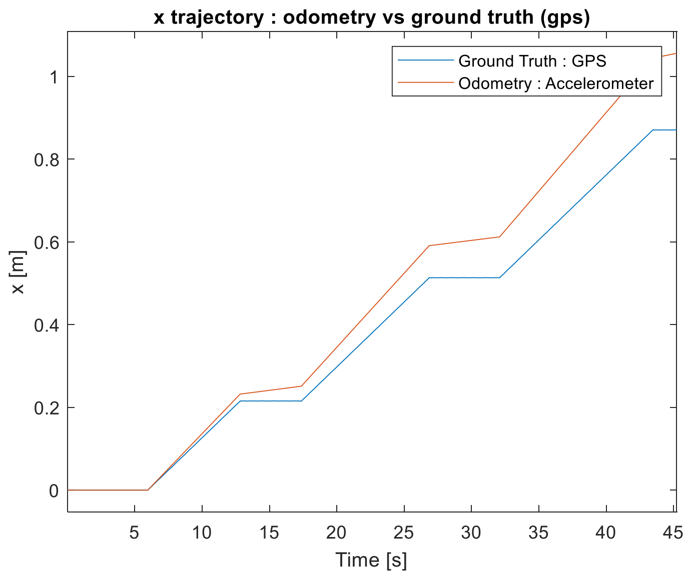


11. (**S**): Look at the odometry equations leveraging the accelerometer implemented in the MATLAB function `odo_acc.m` and see if they match with the equations you defined in the previous question. Use the code *plot_main.m* (only <u>part B</u>), to plot the odometry. What happens if you don't remove the mean (bias) before the integration?

Answer:
You should obtain results <u>similar</u> to the following.
Odometry plot:



Replace `acc(1)-acc_mean(1)` by `acc(1)` in line 7 of `odo_acc.m` to omit bias:



Since the bias is very small in the x direction, both figures are very similar. However, if the bias is large, then the odometry quickly diverges if the bias is not removed. Additionally, the odometry continues to increase even if the robot stays immobile. The explanation is that the odometry error is integrated across time, and the resulting double integration of a constant (i.e., odometry error due to accelerometer bias) is a parabola. Finally, one can notice first that removing the offset improves the quality of the odometry, and second, the odometry values only start to increase when the robot moves.

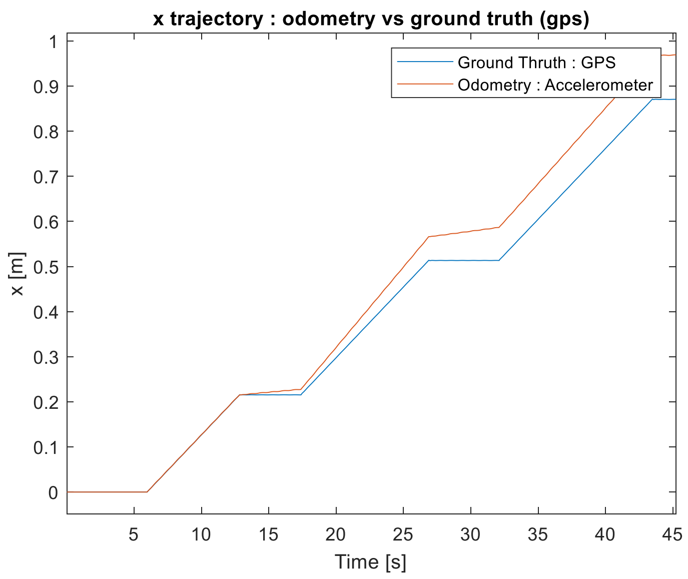
12. (**I**): Implement your equations for the odometry in Webots, within the function `odo_compute_acc()` of the file *odometry.c*. Run the simulation and let your robot run in a straight line. Analyze the resulting logs in MATLAB using *plot_main.m* (only part C). Have you obtained results similar to those previously obtained in MATLAB? If not, try to correct your code or explain why.

Answer:

`odo_compute_acc()` *in odometry.c:*

```
double acc_wx = (acc[0] - acc_mean[0]);

_odo_speed_acc.x += acc_wx *_T;

_odo_pose_acc.x += _odo_speed_acc.x * _T;
```

You should obtain the same results as your MATLAB implementation.



4. Odometry using wheel encoders

In this part, we are interested in continuing our investigation with odometry. This time, we propose to take advantage of the type of locomotion of the e-puck, a differential wheeled vehicle, to design our 2D motion model. In fact, the simulated e-puck provides wheel encoders that return information on the actual position of its wheels. It is worth noticing that there is a main difference between the real e-puck and its simulated version. On the one hand, the real e-puck uses stepper motors, characterized by 1000 steps (increments) per wheel revolution; this type of motor is controlled in an open loop and provides information on the wheel position by itself. On the other hand, the simulated e-puck has one hinge joint (consider it as the rotating shaft) per wheel, to which a rotational motor and a position sensor are attached. The position sensors simulate wheel encoders, a sensing device type that, in reality, is combined with DC motors for closed-loop control; they are physically separated from the motors but anchored to the rotating shaft to obtain the same increment counting functionality.

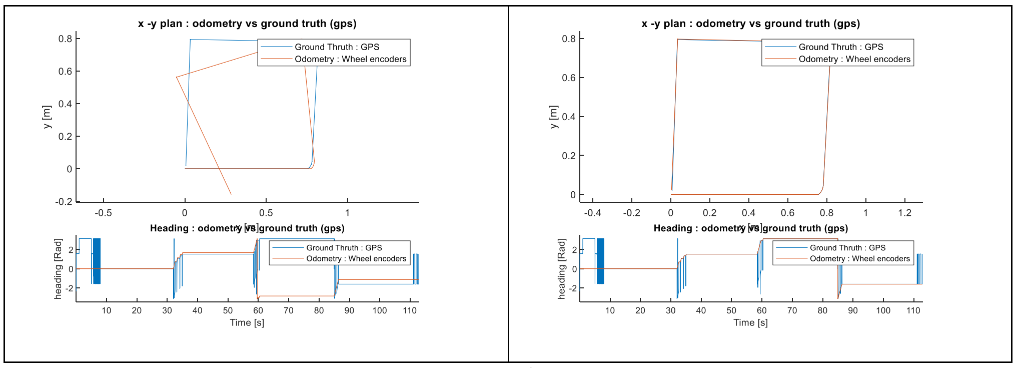
13. (**S**): The same robot controller leveraged in Part 3 is also used for this part. Set VERBOSE_ENC to true and compile the robot's controller. Do a small run in the arena with the e-puck. What is the unit used by the wheel encoders, and how do they convert it into meters? Is it an absolute or a relative measurement of the wheel position?

Answer:
Wheel encoders are given in rad.
We need to multiply the encoder values by the wheel radius.
The simulated e-puck uses absolute encoders.

14. (**S**): Do a complete run around the square with the e-puck. Then, run part D of *plot_main.m* to plot the odometry based on the wheel encoders (note that the odometry is already implemented for you in MATLAB).
Answer: *see next question for example results.*

15. (**S**): Look at the way the odometry is implemented in MATLAB in `odo_enc.m` . Try to reduce the deterministic errors by slightly changing the values of the *WHEEL_RADIUS* and *WHEEL_AXIS*. Your odometry curves should get closer to those obtained with the GPS.

Answer:

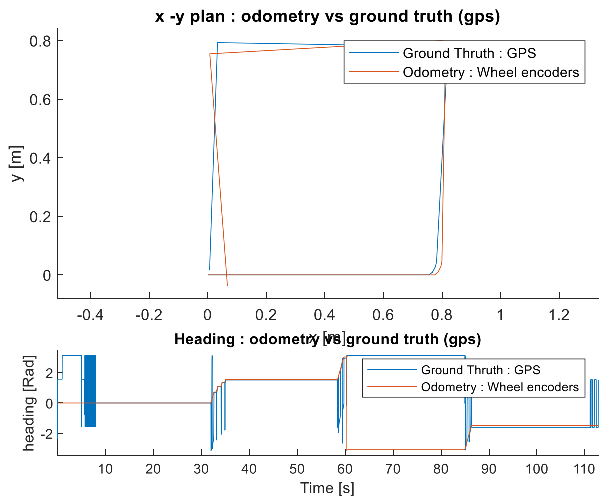


2D Odometry based on wheel encoders. Left : Uncalibrated Odometry. Right : Calibrated Odometry

16. (**I**): Implement your equations for the odometry in Webots in the function `odo_compute_enc()` in file *odometry.c*. Run the simulation and follow the square again. Compare the resulting logs on MATLAB using *plot_main.m* (only part E). As before, you can change the values of the *WHEEL_RADIUS* and *WHEEL_AXIS* to improve your odometry.

Answer:
`odo_compute_encoders()` *in odometry.c:*

```
//  Rad to meter : Convert the wheel encoder units into meters

Aleft_enc  *= WHEEL_RADIUS;

Aright_enc *= WHEEL_RADIUS;

// Comupute speeds : Compute the forward and the rotational speed

double omega = ( Aright_enc - Aleft_enc ) / ( WHEEL_AXIS * _T );

double speed = ( Aright_enc + Aleft_enc ) / ( 2.0 * _T );

//  Compute the speed into the world frame (A)

double speed_wx = speed * cos(_odo_pose_enc.heading);

double speed_wy = speed * sin(_odo_pose_enc.heading);

double omega_w  = omega;


// Integration : Euler method

_odo_pose_enc.x += speed_wx * _T;

_odo_pose_enc.y += speed_wy * _T;

_odo_pose_enc.heading += omega_w * _T;
```



17. (**B**): Decrease the speed (press the D key several times) of the motors and let your robot carry out a complete tour of the square. Compare the logs on MATLAB code *plot_main.m* (only part E). What are your main observations?

    Answer: *You should observe* ***more*** *accurate trajectory estimations.*

18. (**B**): Increase the speed (press the U key several times) of the motors and let your robot carry out a complete tour of the square. Compare the logs on MATLAB code *plot_main.m* (only part E). What are your main observations?

    Answer: *You should observe* ***less*** *accurate trajectory estimations.*

19. (**B**): Which strategies could we use to reduce the error in odometry in terms of motor control? Think about the unmodeled dynamics of our motion models and when they occur.

Answer:

From the lecture course, we know that the following five error sources are involved in our case of study. We can assume that the three first steps are reduced with proper calibration since they are static and deterministic errors.

1. *Limited encoder resolution*
2. *Wheel misalignment and small differences in wheel diameter*
3. *Integration errors*
4. *Variation of the contact point of the wheel*
5. *Unequal floor contact (e.g., wheel slip, nonplanar surface)*

The last two ones are not considered in our basic motion model. By comparing Figures 5 and 6, one can notice the robot underestimates the real value of its orientation while turning on itself.

One partial explanation is that the centripetal acceleration is equal to v^2/R, and the resulting force pulls the robot out of the desired trajectory. Large forward speed coupled with a low rotation radius leads to large centripetal acceleration. This is the reason why when you drive a car, it is better to reduce your speed before an abrupt turn if you want to stay on the track. Additionally, our motion model is purely kinematic; neither the mass nor the inertia of the robot is considered.

The key message here is the way your robot moves impacts your odometry accuracy. An aggressive motion (high acceleration) is likely to result in poor estimation of your robot's pose. The following three points are some possible ways to explore in order to improve your robot's localization. Feel free to implement and test several of them in your course project.

1. *Mind both the robot's motion and motor controllers*
2. *Improve your odometry model*
3. *Fuse different data and information (see Lab09)*

20. (**B**): In the recommended reading material for week 9, Chapter 5 of Introduction to Autonomous Mobile Robot (2004, R. Siegwart and I. Nourbakhsh) describes another motion model for differential-wheeled robots. The equations (5.3 – 5.6) were implemented in MATLAB for you in `odo_enc_bonus.m`. Try to reproduce them in Webots (you can create your function in `odometry.c` to replace the default `odo_compute_encoders()` function). Does the odometry improve when compared to your previous implementation?

Answer:

In *odometry.c* :

- Create a new function `odo_compute_encoders_bonus()`
- Add a new variable `static pose_t _odo_pose_enc_bonus`
- Reset this new variable in the `odo_reset()`
- Add a verbose flag `#define VERBOSE_ODO_ENC_BONUS false`

In *odometry.h* :

- Add the declaration of `odo_compute_encoders_bonus()`

In *controller.c* :

- Create a new variable `static pose_t _odo_enc_bonus`
- Reset this new variable in the `controller_init()`
- Call your new odometry function in `main()`
- Modify both function `controller_init_log()` and `controller_print_log()`

```
/**
 * @brief     Compute the odometry using the encoders. Use the motion model
proposed in bonus question
 *
 * @param      odo         The odometry
 * @param[in]  Aleft_enc   The delta left encoder
 * @param[in]  Aright_enc  The delta right encoder
 */
void odo_compute_encoders_bonus(pose_t* odo, double Aleft_enc, double
Aright_enc)
{
      // Rad to meter
      Aleft_enc  *= WHEEL_RADIUS;
      Aright_enc *= WHEEL_RADIUS;

      // Compute forward speed and angular speed
      double omega = ( Aright_enc - Aleft_enc ) / ( WHEEL_AXIS * _T );
      double speed = ( Aright_enc + Aleft_enc ) / ( 2.0 * _T );

      // Apply rotation (Body to World)

      // smaller integration step for the angle (1/2)
      _odo_pose_enc_bonus.heading += omega * _T / 2.0;
      double a = _odo_pose_enc_bonus.heading;
      double speed_wx = speed * cos(a);
      double speed_wy = speed * sin(a);

      // Integration : Euler method
      _odo_pose_enc_bonus.x += speed_wx * _T;
      _odo_pose_enc_bonus.y += speed_wy * _T;

      // smaller integration step for the angle (2/2)
      _odo_pose_enc_bonus.heading += omega * _T / 2.0;
      memcpy(odo, &_odo_pose_enc_bonus, sizeof(pose_t));

      if(VERBOSE_ODO_ENC_BONUS)

      printf("ODO with wheel encoders (Bonus): %g %g %g\n", odo->x , odo->y , RAD2DEG(odo->heading) );

}
```

You should observe a slight improvement in terms of performance using this new motion model.

5. References

[1] : V.F. (https://physics.stackexchange.com/users/189477/v-f), Why an accelerometer shows zero force while in free-fall, URL (version: 2018-04-28): https://physics.stackexchange.com/q/402645