Student names: Aurélien Morel, Lucas Wälti, Luca Kiener

Instructions: Update this file (or recreate a similar one, e.g. in Word) to prepare your answers to the questions. Feel free to add text, equations and figures as needed. Hand-written notes, e.g. for the development of equations, can also be included e.g. as pictures (from your cell phone or from a scanner). ***This lab is graded.*** *and needs to be submitted before the* ***Deadline : Wednesday 15-05-2019 Midnight. You only need to submit one final report for all of the following exercises combined henceforth.*** *Please submit both the source file (*.doc/*.tex) and a pdf of your document, as well as all the used and updated Python functions in a single zipped file called final_report_name1_name2_name3.zip where name# are the team member's last names. Please submit only one report per team!*

NOTE : The following exercises on Salamandra robotica are based on the research of [1], [2] and [3].

Swimming with Salamandra robotica – CPG Model

In this exercise you will control a salamander-like robot Salamandra robotica for which you will use Python and the dynamics simulator Webots. Now you have an opportunity to use what you've learned until now to make the robot swim (and eventually walk). In order to do this, you should implement a CPG based swimming controller, similarly to the architecture shown in Figure 1.

In the folder Webots you will find sub folders containing the simulated world file and Python codes describing the controller. ***NOTE :*** Do not change the relative positions of files within those folders.

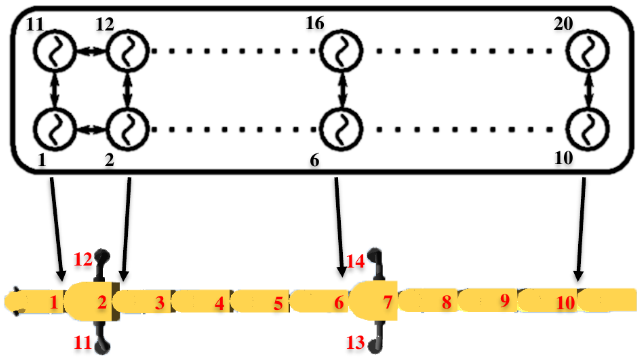


Figure 1: A double chain of oscillators controlling the robot's spine.

Code organization

- **Webots::worlds::cmc_salamander_#.wbt** - These are the world files which describe the worlds and allow to run the simulations. You can run a simulation by running this file with Webots. It also automatically loads the pythonController. Note that each of these files will also run the appropriate exercise_#.py such that you can run each exercise separately. Note that the simulation of the exercises may close immediately as they are not implemented yet. Only cmc_salamander_9_example.wbt will run for a few seconds before closing.
- **Webots::controllers::pythonController::exercise_#.py** - To be used to implement and answer the respective exercise questions. Note that exercise_example.py is provided as an

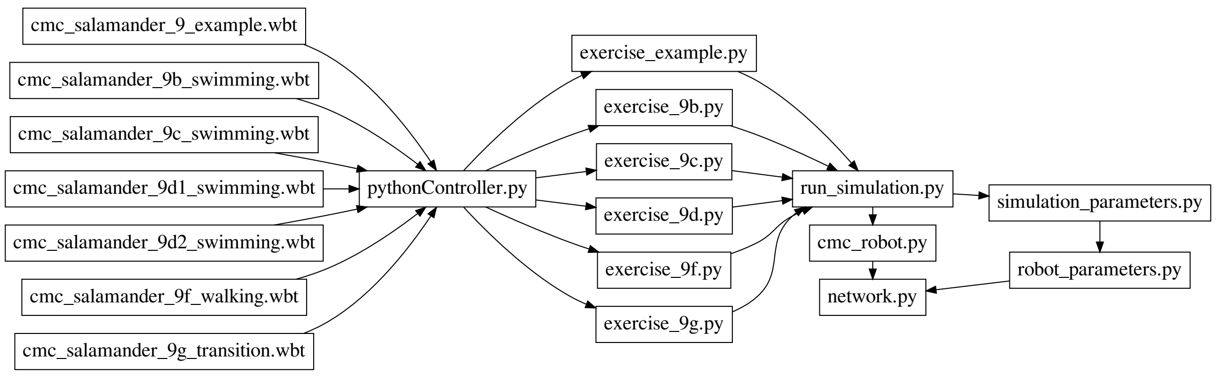

cmc_salamander_9_example.wbt
cmc_salamander_9b_swimming.wbt
cmc_salamander_9c_swimming.wbt
cmc_salamander_9d1_swimming.wbt
cmc_salamander_9d2_swimming.wbt
cmc_salamander_9f_walking.wbt
cmc_salamander_9g_transition.wbt
pythonController.py
exercise_example.py
exercise_9b.py
exercise_9c.py
exercise_9d.py
exercise_9f.py
exercise_9g.py
run_simulation.py
cmc_robot.py
network.py
simulation_parameters.py
robot_parameters.py

Figure 2: Exercise files dependencies. In this lab, you will be modifying `exercise1.py` and `pendulum_system.py`

example to show how to run a parameter sweep. Note that network parameters can be provided here.

- **Webots::controllers::pythonController::pythonController.py** - The main robot controller is implemented in pythonController.py. This file is the main file called by Webots during simulations and can call classes and functions from other files to control the robot and log data. This file is mainly used for calling the appropriate exercise_#.py file. Note that by default the simulation will close Webots when it finishes. If you do not want Webots to close, you can comment the following line: `world.simulationQuit(0)`.
- **Webots::controllers::pythonController::run_simulations.py** There is a run_simulation.py function which is provided for convenience to easily run multiple simulations with different parameters. You are free to implement other functions to run simulations as necessary.
- **Webots::controllers::pythonController::cmc_robot.py** - Contains the SalamanderCMC class, which is used for controlling and logging the robot.
- **Webots::controllers::pythonController::experiment_logger.py** - Contains the codes for logging the simulation. Feel free to modify this file to extend the logging capabilities. Note that the logging makes use of Numpy.savez to save the data.
- **Webots::controllers::pythonController::network.py** - This file contains the different classes and functions for the CPG network and the Ordinary Differential Equations (ODEs). You can implement the network parameters and the ODEs here. Note that some parameters can be obtained from pythonController.py to help you control the values.
- **Webots::controllers::pythonController::robot_parameters.py** - This file contains the different classes and functions for the parameters of the robot, including the CPG network parameters. You can implement the network parameters here. Note that some parameters can be obtained from SimulationParameters class in simulation_parameters.py and sent by exercise_#.py to help you control the values (refer to example).
- **Webots::controllers::pythonController::simulation_parameters.py** - This file contains the SimulationParameters class and is provided for convenience to send parameters to the setup of the network parameters in robot_parameters.py. All the values provided in SimulationParameters are actually logged in cmc_robot.py, so you can also reload these parameters when analyzing the results of a simulation.
- **Webots::controllers::pythonController::solvers.py** - This features fixed time-step solvers which will can are used by network.py for solving the ODE at each time-step. Feel free to switch

between the Euler and the Runge-Kutta methods. *You do not need to modify this files.*

- **Webots::controllers::pythonController::run_network.py** - By running the script from Python, Webots will be bypassed and you will run the network without a physics simulation. Make sure to use this file for question 9a to help you with setting up the CPG network equations and parameters and to analyze its behavior. This is useful for debugging purposes and rapid controller development since starting the Webots simulation with physics takes more time.
- **Webots::controllers::pythonController::plot_results.py** - Use this file to load and plot the results from the simulation. This code runs with the original pythonController provided.
- **Webots::controllers::pythonController::parse_args.py** - Used to parse command line arguments for run_network.py and plot_results.py and determine if plots should be shown or saved directly. *You do not need to modify this files.*
- **Webots::controllers::pythonController::save_figures.py** - Contains the functions to automatically detect and save figures. *You do not need to modify this files.*
- **Webots::controllers::pythonController::exercise_example.py** - Contains the example code structure to help you familiarize with the other exercises. *You do not need to modify this files.*

Prerequisites

Make sure you have successfully installed Webots by following the instructions outlined in Lab 7

Complete the tutorial and practice examples of Webots as outlined in Lab 7

Open the `Webots::worlds::cmc_salamander_9_example.wbt` file in Webots. This should launch the Salamandra robotica model in simulation world

Running the simulation

Now when you run the simulation, the Salamandra robotica model should float on the water with no errors in the Webots console dialog. At this point you can now start to work on implementing your exercises.

Questions

The exercises are organized such that you will have to first implement the oscillator network model in run_network.py code and analyze it before connecting it to the body in the Webots world. Exercise 9a describes the questions needed to implement the oscillator models. After completing exercise 9a you should have an oscillator network including both the spinal CPG and limb CPG.

Using the network implemented in exercise 9a you can explore the swimming, walking and transition behaviors in the Salamandra robotica model using Webots and complete the exercises 9b to 9g.

9a. Implement a double chain of oscillators along with limb CPG's

Salamandra robotica has 10 joints along its spine and 1 joint for each limb. The controller

$$\dot{\theta}_i = 2\pi f + \sum_j r_j w_{ij} sin(\theta_j - \theta_i - \phi_{ij}) \tag{1}$$

$$\dot{r}_i = a(R_i - r_i) \tag{2}$$

$$q_i = r_i(1 + cos(\theta_i)) - r_{i+10}(1 + cos(\theta_{i+10})) \text{ if body joint} \tag{3}$$

with θ_i the oscillator phase, f the frequency, w_ij the coupling weights, ϕ_{ij} the nominal phase lag (phase bias), r_i the oscillator amplitude, R_i the nominal amplitude, a the convergence factor and q_i the spinal joint angles.

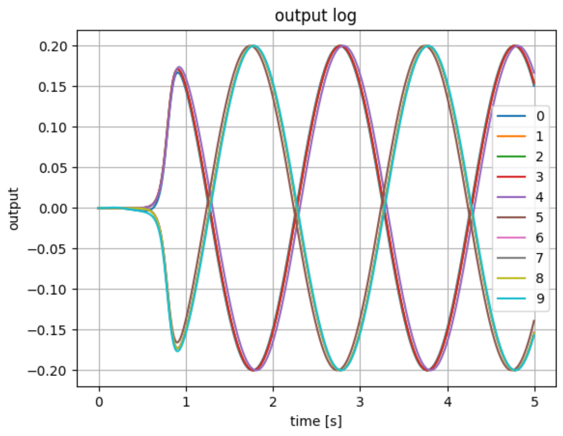
1. Implement the double chain oscillator model using the functions `network.py::network_ode`. Test your implementation by running the network using `run_network.py`. For the network parameters check lecture slides (pay attention to different number of segments). You can also find more information in [3] (especially in the supplementary material). You can set all the network parameters in the `robot_parameters.py::RobotParameters`. To facilitate your work, you could start by only implementing the network for the body oscillators ($i = [0, ..., 19]$) and ignoring the leg oscillators ($i = [20, ..., 23]$). Refer to network::RobotState and robot_parameters.py::-RobotParameters for the dimensions of the state and the network parameters respectively.

2. Implement the output of your CPG network to generate the spinal joint angles according to equation 3. Implement this in the function `network.py::motor_output`. Verify your implementation in by running the Python file `run_network.py`. Use the functions in `plot_results.py` to report your spinal joint angles q_i.

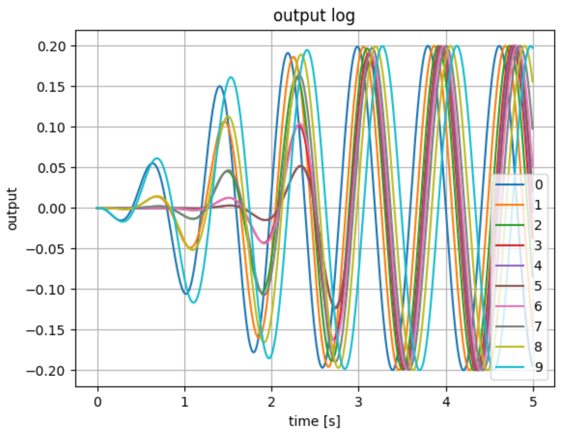
3. Implement a drive and show that your network can generate swimming and walking patterns similarly to [3].

   Hint: The state for the network ODE is of size 48 where the first 24 elements correspond to the oscillator phases θ_i of the oscillators and the last 24 elements correspond to the amplitude r_i. The initial state is set in the init of network.py::SalamanderNetwork.

Results: Figure 3 illustrates the difference between the walking and swimming regimes. During walking, standing waves are generated along the body, while they propagate for swimming, generating thrust through the liquid. The only difference in the command of the network is the drive level, low for walking (typically 2.5) and high for swimming (typically 5). We see that for both drives, the system takes some time to converge to the desired gait and then keeps the gait really stable.



(a) Spinal joint angles q_i during walking.



(b) Spinal joint angles q_i during swimming.

Figure 3: Spine angles (0 to 9) during walking (3a) and swimming (3b) with a fixed desired amplitude of 0.2 [rad].

9b. Effects of amplitude and phase lags on swimming performance

Now that you have implemented the controller, it is time to run experiments to study its behaviour. How does phase lag and oscillation amplitude influence the speed and energy? Use the provided run_simulation.py::run_simulation() to run a grid search to explore the robot behavior for different combinations of amplitudes and phase lags. Use plot_results.py to load and plot the logged data from the simulation. Feel free to extend the logging in cmc_robot.py to show additional measurements if necessary. Include 2D/3D plots showing your grid search results and discuss them. How do your findings compare to the wavelengths observed in the salamander?

- **Hint 1:** To use the grid search, check out the function run_simulation.py::run_simulation() and the example provided in exercise_example.py. This function takes the desired parameters as a list of SimulationParameters objects (found in simulation_parameters.py) and runs the simulation. Note that the results are logged as simulation_#.npz in a specified log folder. After the grid search finishes, the simulation will close, you can remove this feature by commenting world.simulationQuit(0) in pythonController.py::main().
- **Hint 2:** An example how to load and visualise grid search results is already implemented in plot_results.py::main(). Pay attention to the name of the folder and the log files you are loading. Before starting a new grid search, change the name of the logs destination folder where the results will be stored. In case a grid search failed, it may be safer to delete the previous logs to avoid influencing new results by mistake.
- **Hint 3:** Estimate how long it will take to finish the grid search. Our suggestion is to choose wisely lower and upper limits of parameter vectors and choose a reasonable number of samples. To speed-up a simulation, make sure to run Webots in a fast mode.
- **Hint 4:** Energy can be estimated by integrating the product of instantaneous joint velocities and torques. Feel free to propose your own energy metrics, just make sure to include the justification.

Results: *The energy was computed as proposed, hence as the integral of the product of the torques and joint speeds.*

The results of the grid search are shown in Figure 5. The energy used by the salamander follows a simple relationship: it increases with amplitude and decreases with phase lag. On the contrary, the speed of the robot seems to act like a 2D bell curve: it reaches a maximum for a critical amplitude

(0.27) and phase lag (0.6). In that sense, if the salamander wants to only maximize its speed, it should choose these optimal parameters. However, in some cases, the robot should be able to keep a good speed while minimizing its energy consumption. In order to do this, it has two solutions: reducing the amplitude, or increasing the phase lags. By looking closely at 5b, we see that around the maximum, the absolute value of the derivative with respect to amplitude is way bigger than the one with respect to phase lag. Consequently, to optimize both speed and energy consumption, it would be better to gently increase the phase lags.

It would be interesting to compare our results with real data acquired from a living salamander to investigate how the animal optimizes its gait. One possible experiment would be to see the difference in gait between normal conditions, where the animal is expected to optimize both speed and energy consumption, and stressful conditions where the salamander should have a maximum speed without thinking about its consumption (for example escaping a predator).

The wavelength along the salamander spinal cord is inversely proportional to the phase lags between its joints. First, we have seen that the maximum speed is reached for phase lags of 0.6, really close to $2*\pi/10$: the robot will have a wavelength of one body length when reaching its peak speed. Also,if the salamander wants to minimize its energy consumption while keeping some speed, it will have bigger phase lags as seen before, and thus have a smaller wavelength. Therefore, we could expect a body length longer than the wavelength (see figure 4).

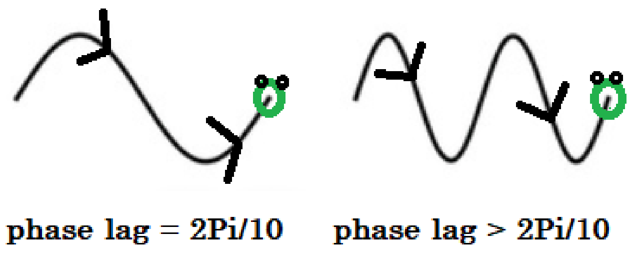
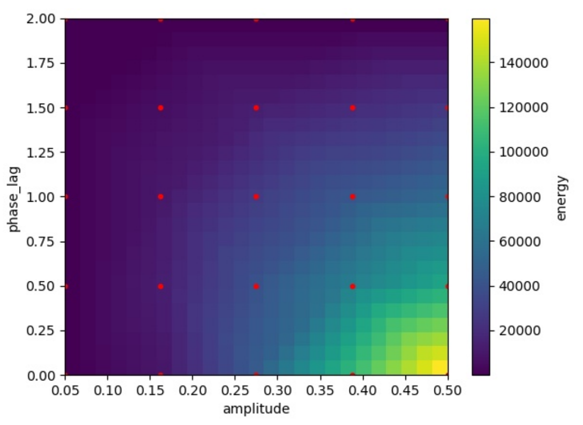
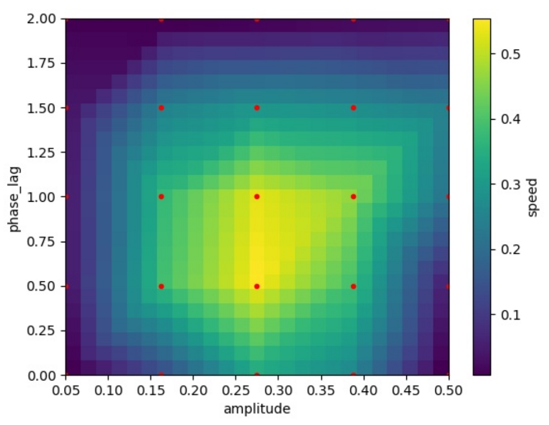


Figure 4: Increasing the phase lags between the body joints shorten the wavelength.



(a) Grid search for energy, exploring amplitude and phase lag.



(b) Grid search for speed, exploring amplitude and phase lag.

Figure 5: Influence of phase lag and amplitude on energy (5a) and speed (5b).

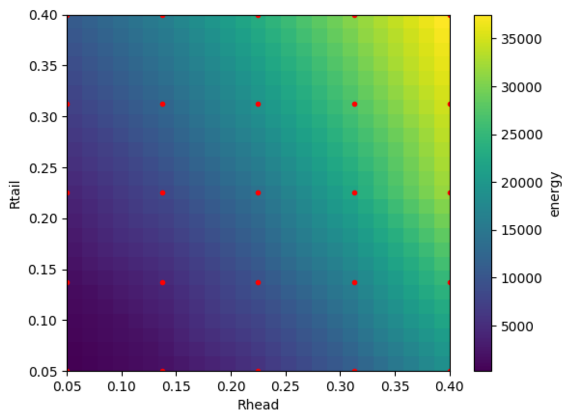
9c. Amplitude gradient

1. So far we considered constant undulation amplitudes along the body for swimming. Implement a linear distribution of amplitudes along the spine, parametrized with two parameters: amplitudes of the first (Rhead) and last (Rtail) oscillator in the spine (corresponding to the first and last motor). To do so, you can add a parameter amplitudes=[Rhead, Rtail] in simulation_parameters.py::SimulationParameters. Don't forget to modify robot_parameters.py::RobotParameters::set_nominal_amplitudes() and interpolate the amplitude gradient between values Rhead and Rtail within the function. Note that you can then provide this amplitudes parameter from exercise_9b.py.
2. Run a grid search over different values of parameters Rhead and Rtail (use the same range for both parameters). How does the amplitude gradient influence swimming performance (speed, energy)? Include 3D plots showing your grid search results. Do it once, for frequency 1Hz and total phase lag of 2π along the spine.
3. How is the salamander moving (with respect to different body amplitudes)? How do your findings in 2) compare to body deformations in the salamander? Based on your explorations, what could be possible explanations why the salamander moves the way it does?

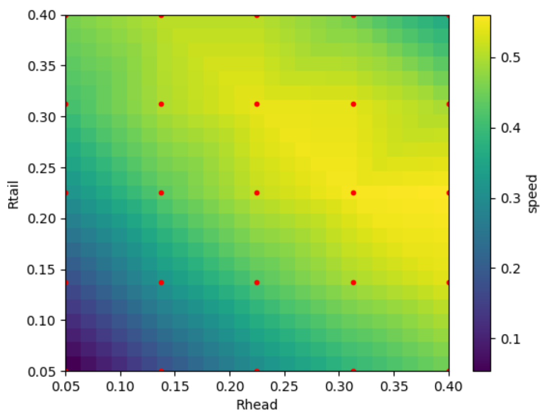
Results: Running the grid search, we see that the energy consumption is proportional to the sum of the head amplitude and the tail amplitude (the mean amplitude). The speed also follows a direct relationship with the mean, but in a bell shape: speed reaches a maximum for a value of the mean and then decreases: too high amplitudes lead to bad swimming.

In that sense, the head and the tail should have high amplitudes to reach higher speeds, but the mean should be kept under a maximum threshold. However, this also means higher energy requirements. Reducing one of the two amplitudes reduces the energy consumption. The head amplitude can therefore be less than that of the tail to reduce the energy, despite loosing speed.

In the real salamander the lateral displacement (amplitude) increases from head to tail. Thus the salamander can compensate having the head more steady by increasing the tail displacement, while having the same energy consumption and speed as a configuration with mean amplitude for both head and tail. A steady head is important for the salamanders visual system to be able to exactly locate prey and danger, and it is still able to move thanks to high amplitudes in the tail.



(a) Grid search for energy.



(b) Grid search for speed.

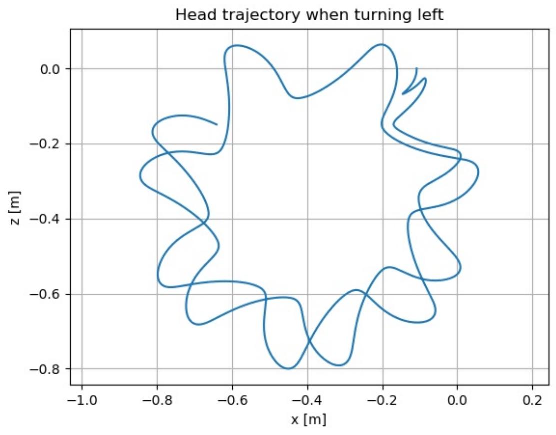
Figure 6: Influence of Rhead and Rtail on energy (6a) and speed (6b).

9d. Turning and backwards swimming

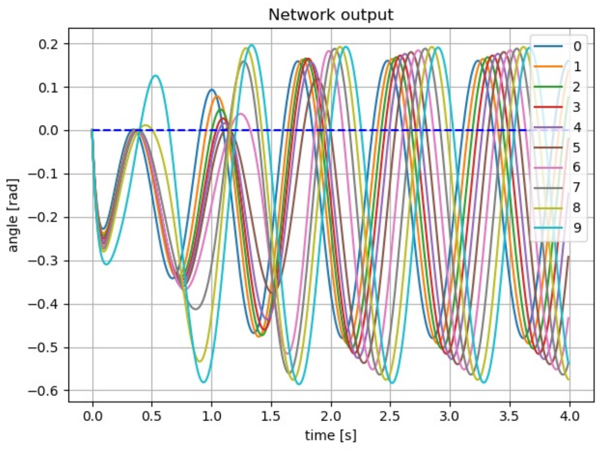
1. How do you need to modulate the CPG network (network.py) in order to induce turning? Implement this in the Webots model and plot example GPS trajectories and spine angles.
2. How could you let the robot swim backwards? Explain and plot example GPS trajectories and spine angles.

Results: In order to turn, one side of the spine needs to be more stimulated than the other. We achieve this by requiring higher amplitudes on one side than the other. This unbalance generates stronger movements on one side, inducing a turn. This is shown in figure 7.

To swim backward, we need to change the direction of propagation of the waves along the body. By simply applying a negative phase lag to the spine, the robot swims backward as shown in figure 8.

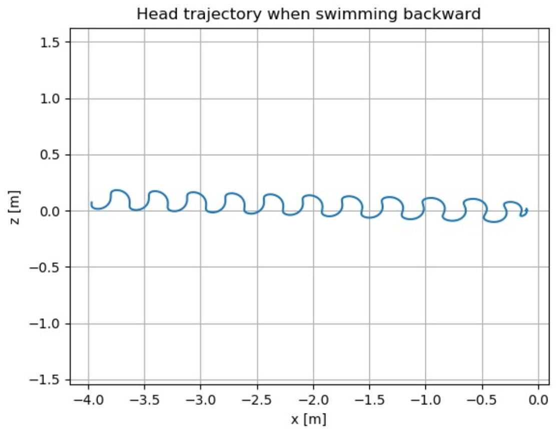


(a) Trajectory of the robot's head in the world while swimming and turning left.

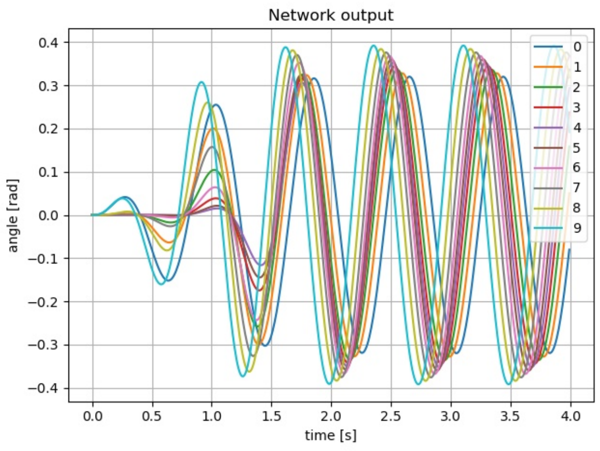


(b) Spine motor commands. The blue line indicates the zero, clearly showing the unbalance in the amplitudes between both sides of the spine.

Figure 7: Illustration of how the robot can turn while swimming.



(a) Trajectory of the robot's head in the world while swimming backward. The robot is looking toward $\vec{x}$ but swims backward toward $-\vec{x}$.



(b) Spine motor commands. The motor commands show clearly the tail motor commands leading the head, which generates backward swimming.

Figure 8: Illustration of how the robot can swim backwards.

9e. Cancelled

9f. Limb – Spine coordination

In this next part you will explore the importance of a proper coordination between the spine and the limb movement for walking.

1. Change the drive to a value used for walking and verify that the robot walks
2. Analyze the spine movement: What are your phase lags along the spine during walking? How does the spine movement compare to the one used for swimming?
3. Notice that the phase between limb and spine oscillators affects the robot's walking speed. Run a parameter search on the phase offset between limbs and spine. Set the nominal radius R to 0.3 [rad]. Include plots showing how the phase offset influences walking speed and comment the results. How do your findings compare to body deformations in the salamander while walking?
4. Explore the influence of the oscillation amplitude along the body with respect to the walking speed of the robot. Run a parameter search on the nominal radius R with a fixed phase offset between limbs and the spine. For the phase offset take the optimal value from the previous sub-exercise. While exploring R, start from 0 (no body bending).

Include plots showing how the oscillation radius influences walking speed and comment on the results.

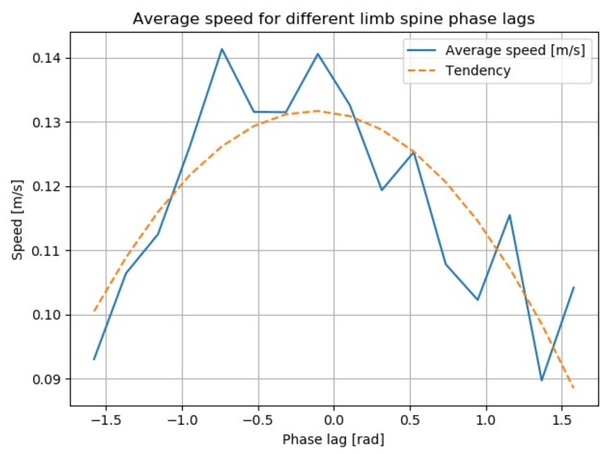
Results: With lower drives, the robot starts walking instead of swimming. There are no longer any propagating waves along the body. Since the spine oscillators are strongly coupled to the legs, standing waves appear along the body, in synchronization with the legs.

However, a good synchronization between the limbs and the body is important. Figure 9a shows this effect. Note that we provide the average speed by dividing the measured distance reached by the simulation time (10 seconds). Regarding the amplitude of the spine oscillations, they should be quite large. Figure 9b illustrates this hypothesis. If the amplitudes are too high, the performances become unreliable and the speeds degrade. We show the global effect of those parameters by applying an interpolation of the data collected with a quadratic polynomial. It allows to show the global influence of each parameter.

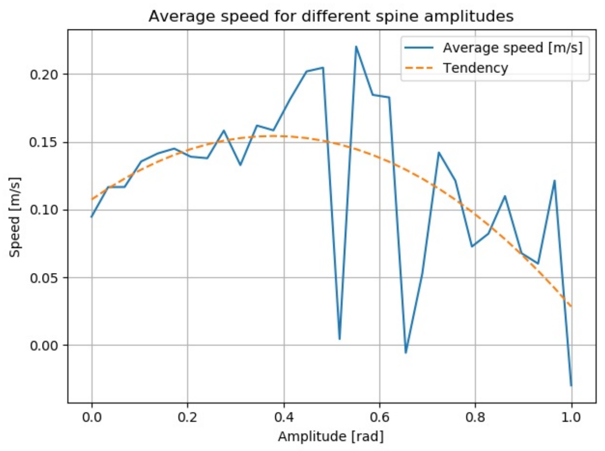
9g. Land-to-water transitions

1. In this exercise you will explore the gait switching mechanism. The gait switching is generated by a high level drive signal which interacts with the saturation functions that you should have implemented in 9a. Implement a new experiment which uses the x-coordinate of the robot in the world retrieved from a GPS reading (See `self.gps.getValues()` in cmc_robot::log_iteration() for an example). Based on the GPS reading, you should determine if the robot should walk (it's on land) or swim (it reached water). Depending on the current position of the robot, you should modify the drive such that it switches gait appropriately.
2. Run the Webots simulation and report spine and limb angles, together with the x coordinate from the GPS signal. Record a video showing the transition from land to water and submit the video together with this report.
3. (BONUS) Achieve water-to-land transition. Report spine and limb angles, the x-coordinate of the GPS and record a video.

Hint: Use Webots' internal video recording tool to easily record videos.



(a) Walking distance for different phase lags. The best option seems to not have any phase lag at all, as illustrated by the tendency curve.



(b) Walking distance for different body amplitudes. The distance increases steadily up until about 0.2 [rad]. The results are then no longer reliable or degrade.

Figure 9: Walking distances under 10 seconds for different limb/body phase lags (9a) and spine amplitudes (9b).

Results: Figures 10, 11 and 12 present the transition from walking to swimming by showing the spine angles, the x-coordinate and the leg angles respectively. The achieved change of gait is really quick and converges impressively fast, especially knowing that it is only driven but a high level change of drive.

A video illustrating the transition is provided along with this document.

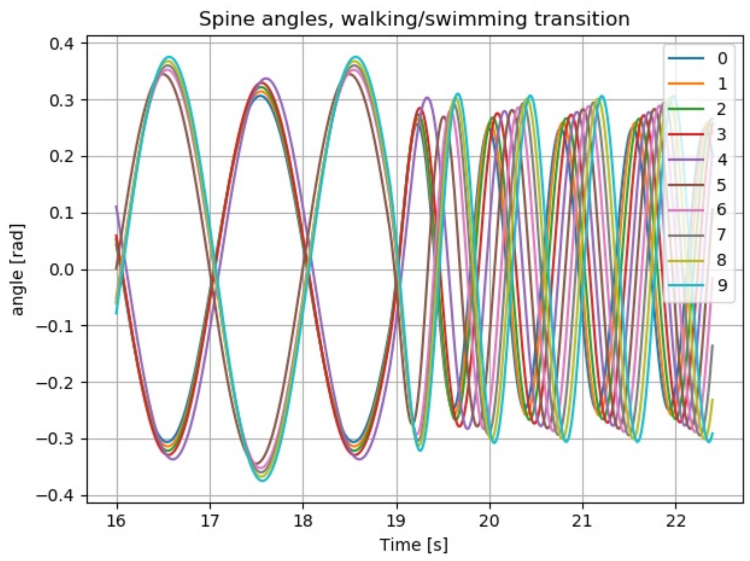


Figure 10: Effect of the transition from walking to swimming on the spine angles.

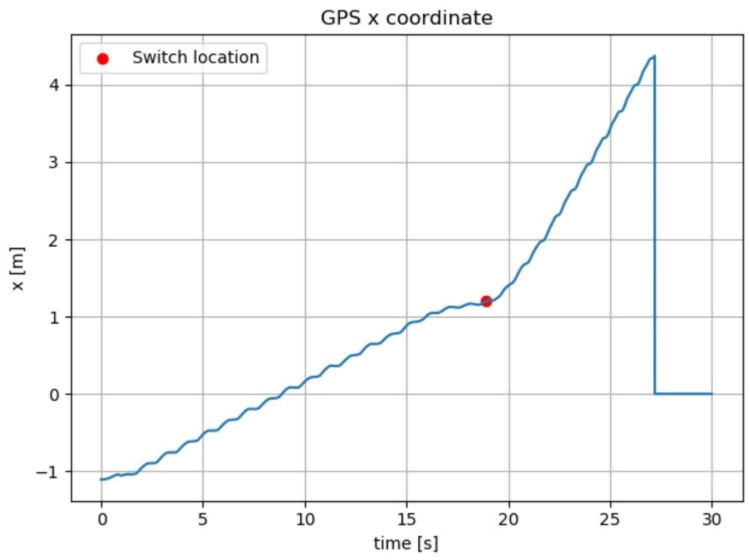


Figure 11: X-coordinate of the robot's head, showing the change in speed between walking and swimming.

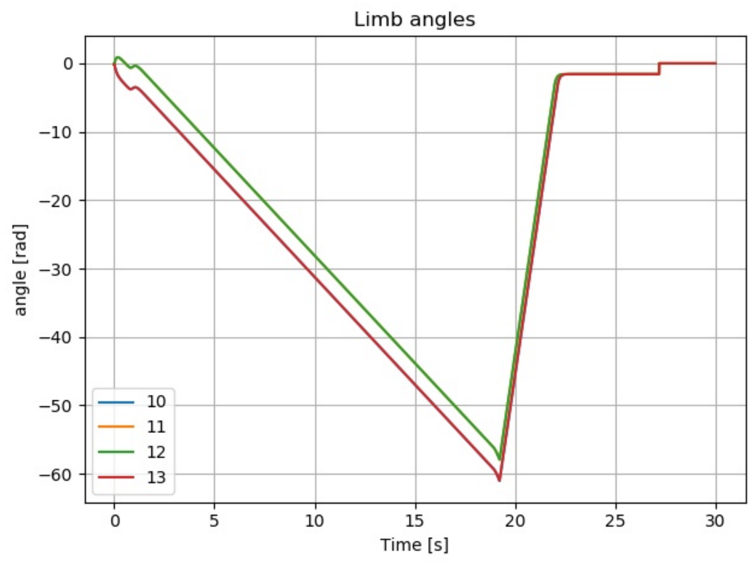


Figure 12: Limb phases with a clear transition at about 19 [s] when switching from walking to swimming gate.

References

[1] A. Crespi, K. Karakasiliotis, A. Guignard, and A. J. Ijspeert, "Salamandra robotica ii: An amphibious robot to study salamander-like swimming and walking gaits," *IEEE Transactions on Robotics*, vol. 29, pp. 308–320, April 2013.

[2] K. Karakasiliotis, N. Schilling, J.-M. Cabelguen, and A. J. Ijspeert, "Where are we in understanding salamander locomotion: biological and robotic perspectives on kinematics," *Biological Cybernetics*, vol. 107, pp. 529–544, Oct 2013.

[3] A. J. Ijspeert, A. Crespi, D. Ryczko, and J.-M. Cabelguen, "From swimming to walking with a salamander robot driven by a spinal cord model," *science*, vol. 315, no. 5817, pp. 1416–1420, 2007.