Realization of a Physic Simulation for a Biped Robot

Stéphane Mojon
stephane.mojon@epfl.ch - Student in Computer Science
Swiss Federal Institute of Technology Lausanne (EPFL)

27th of June 2003
Semester project realized under the direction of Professor Auke Jan Ijspeert
School of Computer and Communication Sciences - EPFL

1. Introduction 5

2. Goals 6

2.1. Creating the simulation and the model of the robot 6
2.2. Testing the simulation with a simple walk 6

3. Open Dynamics Engine (ODE) 7

3.1. What ODE really is 7
3.1.1. The bodies 7
3.1.2. The joints 8
3.2. The first model based on ODE 8
3.3. Hinge joints versus universal joints 9
3.4. The controller used 9
3.5. An attempt to tune the shape of the feet 11

4. Webots 12

4.1. The main features of Webots 12
4.1.1. The description of the robot and its world 12
4.1.2. How to control the robot 13
4.1.3. The simulation 13
4.1.4. The transfer to a real robot 13
4.2. Webots and ODE 14
4.3. Why we transferred the model to Webots 15
4.4. If you are interested in Webots 15
4.5. Thanks to Olivier Michel for adapting Webots to our needs 15

5. The model of the robot 16

5.1. Sony's SDR4X prototype 16
5.2. From the real world to the virtual model 17

6. Control of the walk 18

6.1. A simple open loop control based on sinus 18
6.2. Using a genetic algorithm to optimize the parameters 19
6.2.1. The genome used 20
6.2.2. The creation of the first generation 20
6.2.3. The fitness function 21
6.2.4. The selection method 21
6.2.5. The recombination operator 21
6.2.6. The mutation operator 22
6.2.7. Introducing elitism 23
6.3. The shape of the search space 23
6.4. The implementation using Webots 23
6.4.1. One program for the robot, one program for the supervisor 23
6.4.2. Webots and the problem of the initial conditions 24

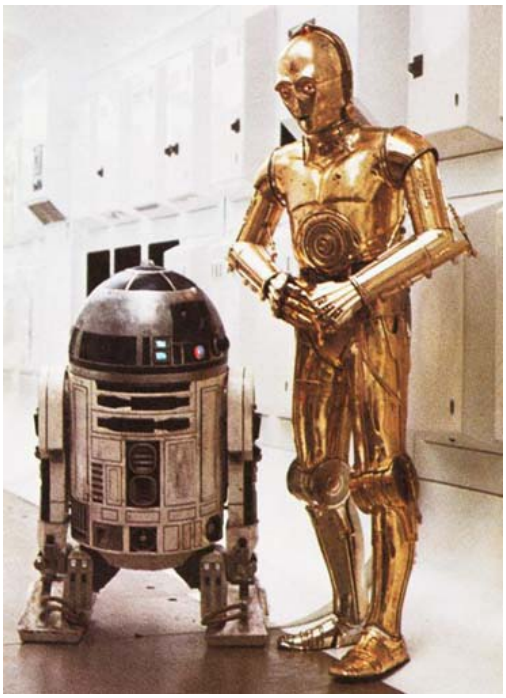
7. Analysis of the results 26

7.1. The ability to walk 26
7.2. Was the search space well explored? 27
7.3. The robustness of the walk 27
7.4. Improving the robustness by adding noise during the learning phase 28
7.4.1. Adding noise into the controller 28
7.4.2. Adding an external force as noise 29
7.4.3. Did the noise improve the robustness? 30

8. Conclusion 31

1. Introduction

In his history, the human being has always been fascinated by its image: The cave men were painting their hunting scene on the stone, Leonardo da Vinci studied the human body with his work the Man of Perfect Proportions. But the limit of the technology always stand as a barrier between the human being and his will to create an artificial representation of himself.
We could think that today the situation would be different; we are now able to send someone on the moon and to play with the matter at the atomic level. But all this great technology is still useless when we are speaking about building a robot that would behave like a human being. For the moment, the only robots able to think and to move like us are the ones described by science fiction. It would be totally stupid to try to compare the robots we are building today with the cleverness and the sense of humor of a robot like C3PO, in Star Wars. But the topic of this document is not about making a thinking robot with an artificial intelligence; that's far beyond our competency. No, our interest is somewhere else.
Let's come back to the fabulous C3PO. Behind his impressing ability to think, he has another incredible quality: He is able to walk! The fact of walking is so common for us that we don't even notice that when we are watching the movie. But if we take the time to think about it, the biped locomotion is already an incredibly complex problem. First of all, the right motion and coordination must be found to create the motion and to preserve the equilibrium. Then, walking also means being able to prevent a fall if suddenly the equilibrium is lost. This second task is probably the most difficult one, because a fall is never the same as the previous and because it happens very quickly.
But the difficulty of the problem makes it very interesting. So let's explore how we could make our robots walk in C3PO's footprints.



C3PO (on the right) and his friend R2D2 (on the left)

2. Goals

As explained in the introduction, biped locomotion is a very difficult problem in itself. Such a problem cannot be solved in a single step. We have to divide our main goal in many sub-goals. So, here are the two very first goals that are going to be discussed in this document. For sure, these two goals are not going to lead us to a nice looking and robust walk. But they will establish a good base on which next research will be possible.

2.1. Creating the simulation and the model of the robot

It seams stupid to say, but to make a robot walk we first need one. The obvious solution could be to buy one. A few years ago, we wouldn't even have dreamt about such a solution. The only biped robots available at this time were the prototypes made by some big companies like Honda. But the purpose of these first robots, like the Honda robot, was mostly to impress the media and the consumer. These projects were mostly used for the marketing and for the promotion of the company. But slowly the situation is changing and we begin to find some real robots available on the market. Sony confirmed this tendency by launching their robot Aibo, which is the first robot design with commercial goals. But unfortunately, Aibo is a quadruped robot, and its biped cousin, Sony's SDR4X, is still a prototype.
Finally, even if the biped robots now exist, it's still very difficult to get one of them. Therefore the only solution we have is to simulate a biped robot in software. But the difficulty to find a real robot is not the only motivation to create a simulation. A simulation will always be more flexible and easy to use than a real robot. That transforms the simulation in a powerful testing and prototyping tool.
But if we want the simulation to be usable, it has to be as close as possible to the reality. It means that the simulated robot must be, at least, able to fall. In other words the simulation must respect the laws of physic.

That leads us to the first and main goal of this project: Creating a simulation of a biped robot that includes the laws of physics.

2.2. Testing the simulation with a simple walk

The second goal of this project stands as a first attempt to address the biped locomotion problem. But this first set of tests is mainly needed to confirm that the simulation is usable and fits our needs. There are so many parameters to set in the model of the robot (like the mass of each part of the robot or the friction coefficient of the feet) that doing a test is the only way to verify the correctness of our decisions.

So, the second goal of the project is the generation of a first and simple walk using the simulation. If this goal is reached, it will mean that the simulation is a good approximation of the reality and a good base for studding the biped locomotion.

3. Open Dynamics Engine (ODE)

Our simulation must include the laws of physic to approximate the reality. But physics is a huge field, and of course only a small part of it is needed to make a robot walk. So, let's see exactly what we need and how it can be simulated.

From the physic's point of view, a robot is simply an assembly of many rigid bodies connected together with some joints. Each one of these bodies can interact with its neighbors: a force or a torque applied on one body will also affect its connected neighbors, and all the bodies must be able to bounce into each other. Finally all the bodies have to be affected by a force of gravitation, otherwise the robot couldn't fall.
In others words, we need a tool for simulating articulated rigid body dynamics. One could easily imagine that creating such a tool is already a huge project in itself. Hopefully there are many fields in which rigid body simulations are required. For example, the video games industry is very interested in this technology to make theirs games more and more realistic.
This increasing demand has triggered the apparition of many libraries dedicated to the simulation of rigid bodies. Most of these libraries are still expensive products, but there is one library that is completely free and open source: This library is called ODE.

3.1. What ODE really is

ODE stands for Open Dynamic Engine. On his web site (http://q12.org/ode), ODE's author Russell Smith gives the following description of his library:

> *"ODE is a free, industrial quality library for simulating articulated rigid body dynamics - for example ground vehicles, legged creatures, and moving objects in VR environments. It is fast, flexible, robust and platform independent, with advanced joints, contact with friction, and built-in collision detection."*

This describes very well the main features included in ODE. But to complete this description here is a more accurate definition of the two main concepts available in ODE: The bodies and the joints.

3.1.1. The bodies

As explained in ODE's manual, a rigid body has the following properties:

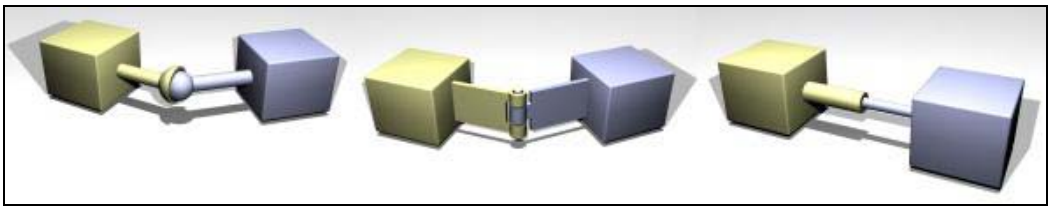
- A position vector that correspond to the body's center of mass.
- A Linear velocity.
- An orientation of a body.
- An angular velocity vector, which describes how the orientation changes over time.

The previous properties are changing over time, but others are usually constant over time:

- The mass of the body.
- The position of the center of mass (relative to the body).
- An inertia matrix that describes how the body's mass is distributed around the center of mass.

3.1.2. The joints

The rigid bodies described in the previous section can be connected together by a large variety of joints, like the ball and socket joint, the hinge joint or the universal joint.

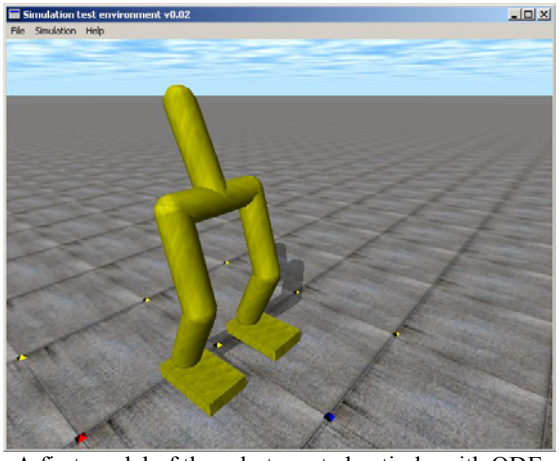


Some of the joints available in ODE.

But the only type of joints that is going to be used in the simulation is the hinge joint that allows a rotation around one degree of freedom. The explanation of this choice will be discussed latter in this chapter.

3.2. The first model based on ODE

ODE is very well designed, and allowed to create very quickly a first prototype of our robot. The picture below shows this first model. It is has no arms but has already 11 degrees of freedom (1 for the back, 2 for each hip, 1 for each knee and 2 for each ankle)

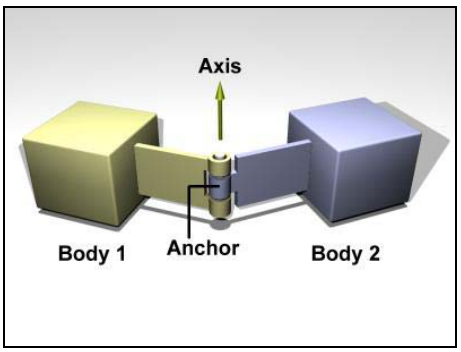


A first model of the robot created entirely with ODE

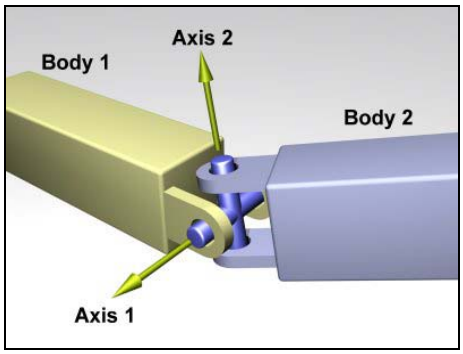
But this first model is still a puppet and has no "muscle". It now needs controllers that will bring each of its joints in a predefined position by applying a torque.

3.3. Hinge joints versus universal joints

In the first model of the robot, the only joint used is the hinge joint. This choice could seam surprising and we could argue that a universal joint would be more appropriate for the hip and the ankle. Even if we look at the documentation, we notice that the universal joint has been recently introduced into ODE to simulate an articulation like the hip. But the universal joint suffers of his novelty. All the functions needed to get the state in which the universal joint is, are not yet implemented. For example there is no direct way to get the angle between two bodies connected by a universal joint.
These functions, and some others useful ones, are available for the hinge joint. To control the motion of the joints, these functions are going to be very useful. It's why the only joint used to create the robot is the hinge.
The only tradeoff is that the hip's joint has to be created with to successive hinge joints rather than a unique universal joint. That makes the creation of the model a little bit more difficult to create. But at the end the behavior of the robot is not affected, because two hinge joints placed perpendicularly at the same position are completely equivalent to a universal joint.



The hinge joint with a unique degree of freedom



The universal joint with two degrees of freedom

3.4. The controller used

Until now, the robot is only a puppet that falls on the ground like a dead body. Thus, the final step to complete the simulation is to add some "strength" to our robot. In this case, strength means that each joint should be able to follow a trajectory over time. In robotics this is called a control problem: In input, a controller receives the desired position of the joints it controls. And according to this input, the controller decides which force must be applied on the joint to reach the desired position.
One of the most used controller is called the PID controller. PID is the abbreviation used for Proportional, Integrative and Derivative. But the purpose of this section is not to explain all about control, the real goal is to explain the choices we made when

we implemented the controller. It is why the following description of the PID controller is very brief.

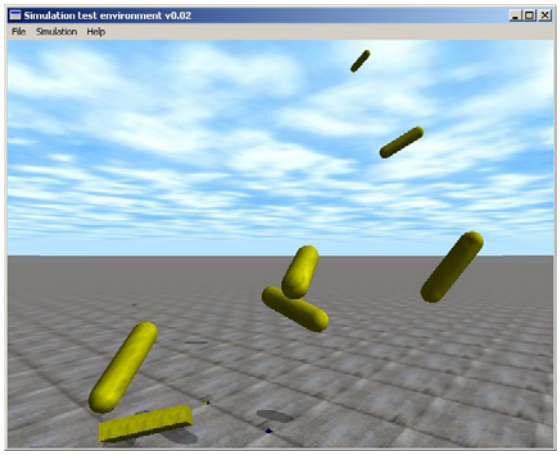
To reach a position $\tilde{\theta}$, from the position θ, the torque $\vec{\tau}$ that has to be applied on the joint is given by the following formula:

$$\|\vec{\tau}\| = \alpha \cdot (\tilde{\theta} - \theta) - \beta \cdot \dot{\theta} + \gamma \cdot \int_0^t (\tilde{\theta} - \theta) \cdot dt$$

In this formula there are three terms:

- The proportional one, weighted by the parameter α.
- The derivative one, weighted by the parameter β.
- And the integrative one, weighted by γ (often this term is not used).

This kind of controller has been implemented in the simulation. But finding the right values for the three parameters α β and γ is not trivial. And these values are critical: If they are too small, the controller won't be able to follow the given trajectory. And if they are too big it is even worse because the robot can literally explode. The explosion comes form an instability problem: If we apply a too big force to one body, the connected bodies won't be able to "follow the motion" of the first one. To correct that, ODE will add a huge elastic force between the two bodies. That leads to an unstable situation and the robot explodes. But this kind of instability is not a bug in ODE because ODE cannot prevent the user to create an unstable situation.



Usually the explosion is very violent…

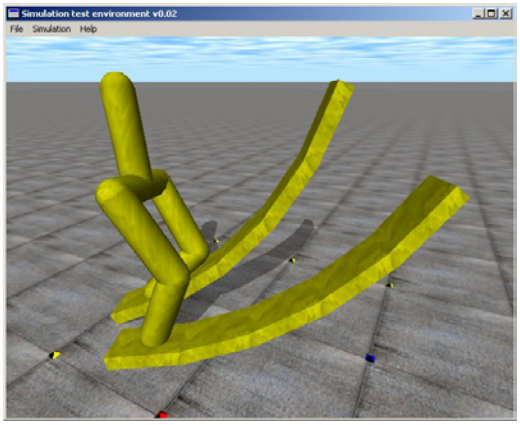
To prevent these explosions a solution could be to use the controller that is implemented directly in ODE. This controller is more careful when applying the forces on the joints, and limits the danger of falling into unstable situations. But this controller has also a big default: It controls the angle rate of a joint, and not directly its position. We can bypass this problem and use the angle rate $\dot{\theta}$ to control the angle θ like this:

$$\dot{\theta} = \alpha \cdot (\tilde{\theta} - \theta)$$

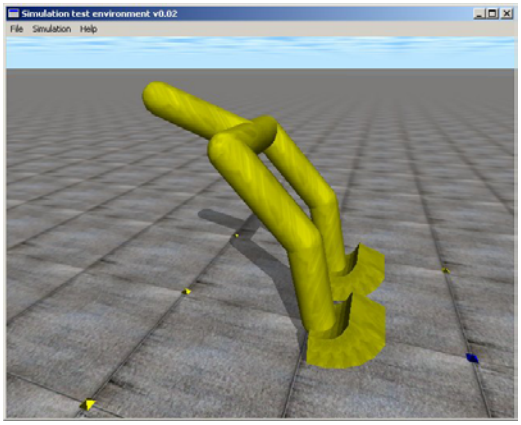
$\dot{\theta}$ is computed with this formula and the passed to ODE's controller. It works because the angle rate $\dot{\theta}$ and the torque τ are linked together by Newton's law. But, this solution is less elegant than the PID controller, mostly because it hides totally the problem. And the user must convince himself that the controller is doing a good job.
So each method has its advantages and its tradeoffs. The choice between the two depends of the user's. But, for example, the second solution (ODE's controller) is the one that is used in the simulation engine Webots (the explanation of what Webots is comes in the next chapter).

3.5. An attempt to tune the shape of the feet

A lot of studies have already been done on the biped locomotion problem. And one of the major conclusion that emerges from these studies is that the shape of feet is a key point.
It's why we included in the simulation, the possibility of tuning the shape of the feet. The parameters used to do the tuning were: the curvature, the length, the width and the thickness of the feet. But as we will see in the next chapter, the model of the robot was moved toward the software Webots. And the ability to change the shape of the feet was not easily transferable to Webots.
It's why the idea of studying the impact of the feet's shape on the walk, has been abandoned in this project.



A robot with big curved feet.



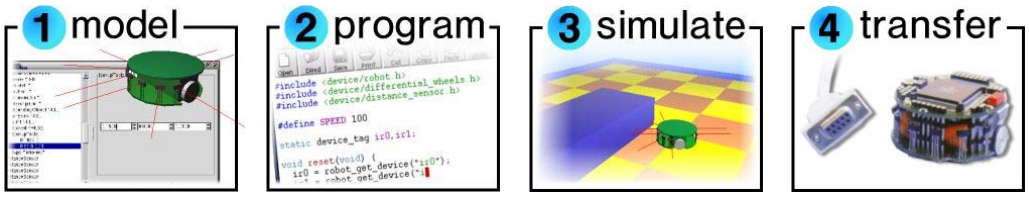
The same robot with smaller feet.

4. Webots

Webots is a 3D mobile robotics simulation software that provides a fast and convenient way of modeling, programming and simulating mobile robots. As we will see in this chapter, Webots is a very powerful tool with many built-in capabilities.

4.1. The main features of Webots

Creating a robot into Webots can be separated in four different phases:



Each phase is going to be explained with more details in the following sections. In each section, a comparison will also be made between our previous simulation made directly with ODE, and Webots.

4.1.1. The description of the robot and its world

As we did when we used directly ODE, the first thing to do when using Webots is to build the model of the robot. That mainly means deciding what will be the shape of the robot, and placing all the joints and motors that will allow the robot to move.
This is done very easily with the help of a nice graphical user interface (GUI). That's already a big advantage comparing to ODE. With ODE the user had to code the shape of the robot completely by hand. But now, with webots the user can simply add the elements one by one with the GUI and directly see the result on the screen. That makes the conception of the model easier and faster.

Another advantage is the way Webots uses to store the model of the robot and its environment. With the previous version of the simulation, made with ODE, the model was hard-coded directly in the C program. But Webots uses the WRML standard to store all the environment of the simulation. That allows the models created, to be easily modified and reused.
A GUI displays and allows to edit the WRML tree describing the robot world (the robot and its environment). This tree can contain a large variety of different nodes. Each node represents one of the primitive offered by Webots to create the world. For example, one subset of these nodes are used to describe all the sensors that can bee directly plugged into a robot; here is the list of these sensors:

- Distance sensors (Infrared & Ultrasound)
- Range finders
- Light sensors

- Touch sensors
- Global Positioning Sensor (GPS)
- Cameras (1D, 2D, color, black and white)
- Receivers
- Position sensors for the joints
- Incremental encoders for wheels

The number of these primitives furnished by Webots is huge. That allows us to create a very complete and accurate robot's model. So, the next step is to see how we can control and give life to a robot.

4.1.2. How to control the robot

In the previous section the model of the robot was built with a nice looking GUI. But to determine the behavior of the robot, the only solution is still to program the controller by hand. But, the code will no more contain the part needed to build the model. Now, all the elements previously defined in the world will be accessible in the code through the Webots' API.
The code can be written in C, C++ or in Java. But the functions available are the same for the three languages. So the choice is mostly a matter of taste. For this project, the controllers were programmed in C.
Once the controller is written and compiled, the binary file generated must simply be put in a particular directory and Webots will use it.

4.1.3. The simulation

To summarize the situation we now have a WRML file describing the world, and a C program describing the behavior of the robot. No more is needed to launch a simulation and to see our robot moving in his 3D virtual environment. We just need to launch Webots, open the file containing the world and then click the "play" button. It's as simple as that.
The simulation can of course run in real time mode. But Webots can also make the simulation run as fast as it can, by using all the resources of the machine on which it is running. On a modern computer, Webots can run up to 20 times faster than the real time speed. This is highly appreciated when the simulation is used for a learning purpose. The idea of making a robot learn a behavior will be discussed in detail later in this document. But every learning method needs a lot of simulation's time; because the robot needs a huge number of tries before it discovers a good solution. Therefore, if the simulation runs 20 times faster, the robot will learn 20 times faster.

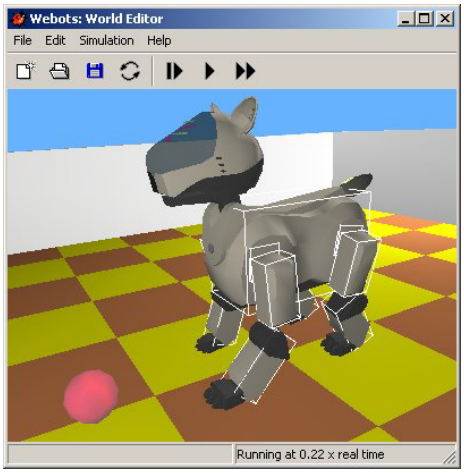
4.1.4. The transfer to a real robot

This last step is probably the most impressing feature present into Webots. Once the simulation runs well and fits the will of the user, the program doing the control can easily be uploaded to the real robot. Of course this feature is limited to a very limited set of robots, like the Khepera or the Koala robot. But for a programmer who wants to develop a real application on one of these robots, the advantage huge. He can do everything, from the prototyping to the deployment on the real robot, by using a single software: Webots.

Of course for the biped robot used in our simulation, this feature is completely useless, because we don't even have a real robot corresponding to our model. But in the future who knows. In its last version, Webots already includes a model of Sony's Aibo robot. The transfer capabilities are no yet present for this robot but it should be added in the future. So, if the industry continues its effort to make the robotic more and more accessible, perhaps one day Webots will also be able to make a real biped robot Walk.

4.2. Webots and ODE

From the beginning of this chapter, the comparison is made between Webots and the first simulation made with ODE. But we must mention that Webots also uses ODE. It is why our first simulation looks so ridiculous when compared to Webots. In fact Webots creates ODE's bodies and joints directly from the WRML description of the world. So by using Webots we also incontinently use ODE. But this is totally transparent and the user has nothing to know about ODE to create a simulation with Webots.
Thus, we could ask why we spend so much time to learn how to use ODE and to create the first simulation. There are two good reasons that motivated this choice.
First, when this project started, the version of Webots that uses ODE was still in development. It's only since the recent release of the version 4 that ODE is included into Webots. Second, knowing ODE helps to understand the behavior of the robots into Webots. For example, knowing ODE was very useful while debugging a robot that behaves abnormally because of ODE.
Finally, the time spent on ODE was very useful and allowed to gain a better global understanding of the project.



The Aibo robot simulated into Webots
The box is white represent the bodies used by ODE

4.3. Why we transferred the model to Webots

After all the previous explanations, the reason why we transferred the project to Webots should be pretty obvious. It can be summarized in two words: flexibility and Modularity.

- Flexibility for the well-done GUI that allows to quickly modify the WRML description of the world.
- Modularity for nice separation that is made between the model, stored in a WRML file, and the controller expressed in C or Java.

In conclusion, it would be stupid to code everything by hand with ODE when we have a nice tool like Webots that include ODE.

4.4. If you are interested in Webots

Webots is a software commercialized by Cyberbotics Ltd. If you are interested in this product you can go on the Cyberbotics' web site at the following address:

http://www.cyberbotics.com

On this site you will find a demo version of Webots and all the information needed to purchase the product.

4.5. Thanks to Olivier Michel for adapting Webots to our needs

Parallel to the creation of the simulation, the creator and programmer of Webots Olivier Michel, was working on the version 4 of Webots. That allowed us to interact a lot with the development of Webots. Each week we met and we exposed to Olivier what we would need into Webots to progress in the project. He always listens to our requests and did his best to satisfy them. And on the other hand our suggestions were helping him to improve Webots.

Without this collaboration this project certainly would not be what it is today. So I simply want to thank Oliver for all the work he did for us.

Thanks

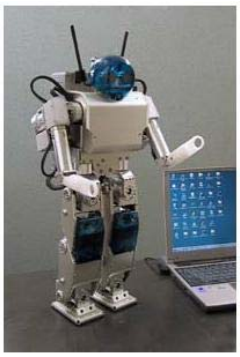
5. The model of the robot

Until now, we always spoke about simulating a generic model of biped robot. This way of doing would give us a complete freedom on the model. But having a lot of freedom also means that we would have a lot of decision to take: How many degrees of freedom should have the robot? Were should we place all the joints? What should be the size and the weight of the robot?
A priori we have no idea of what the answers of these questions should be. We could only try to guess what the good answers should be. But if we make the wrong suppositions, that could lead to a model with a reduced ability to walk. With such an uncertainty the rest of the project can be seriously affected. For example, if the robot is not able to walk with a particular algorithm, it would be impossible to tell if the problem comes from the algorithm or from the robot.
To solve this problem we finally decided to base our model on an existing biped robot. Those who made a real robot were facing the same problem we are having, and they certainly did a lot of research to determine what the good solution should be. So it would be stupid to do the same research on our own. Therefore we will choose an exiting robot and suppose that it is well designed.

Here are some existing humanoid robots:


The ASIMO robots by Honda


The Sony Dream Robot


The HOAP robot by Fujitsu


The Pino robot by Sony

5.1. Sony's SDR4X prototype

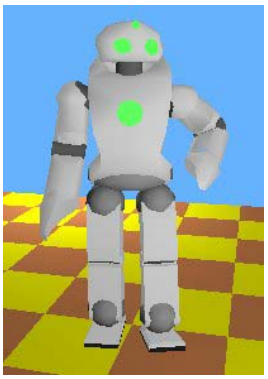
Even if we are going to copy the model of a real robot, one question remains: Which robot are we going to chose? All the existing robots are prototypes, and most of them are not even planned to be commercialized. So, most of these robots are kept secrets and it is very difficult to collect technical data about them. It's why we decided to base our model on one of the most popular and promising robot:
The Sony Dream Robot (SDR). This robot is still a prototype but it has been shown in a lot of shows around the world. Therefore, it is not too difficult to find some pictures and videos of it. Sony even reveled some technical details about the robot and its capabilities. And finally, with the experience gathered by Sony on the Aibo robot, we

have good reasons to think that the SDR will be a well-designed robot. That makes the SDR the perfect candidate for our simulation.

5.2. From the real world to the virtual model

On Internet we already found many useful information about the SDR, like its high, its weight and also a schema showing were are placed all the degree of freedom. But a lot of parameters were sill unknown. That forces us to make some reverse engineering based on all the pictures and videos collected. The most difficult job was to determine to position of some joints, hidden in the robot's trunk. But with some patience and many hours spent observing the real SDR, we finally obtained a nice looking and well-imitated virtual model.
On the following pictures we can compare the existing SDR (on the left) and its virtual cousin (on the right).

The Sony Dream Robot
in the real world



The Sony Dream Robot
simulated into Webots

Thanks to Oliver Michel who made the 3D model that we can see on the picture above.

6. Control of the walk

We now have to robot, so let's try to make it walk. Therefore, in this chapter we are going to explain which algorithm was used to make the virtual robot walk. But, as it was already explained in the goals of this project, the techniques used here are not going to revolutionize the biped locomotion. It's why the algorithm explained below could seam a little bit naive and inefficient. But the biped locomotion is such a complicated problem; we have to approach it step by step. Thus, this is our first try, made to gain a better understanding of the challenge that represents the biped locomotion.

6.1. A simple open loop control based on sinus

The basic idea of the technique use here is very simple. Each joint of the robot will simply follow a trajectory described by a sinus function. If we call the joint's position θ, the value of θ over time will be given by the following formula:

$$\theta(t) = A \cdot \sin(\omega \cdot t + \varphi) + \theta_0$$

This equation has four parameters:

- A : The amplitude of the sinus.
- ω : The frequency of the sinus.
- φ : The dephasing of the sinus.
- θ_0 : A constant value around which the sinus oscillate.

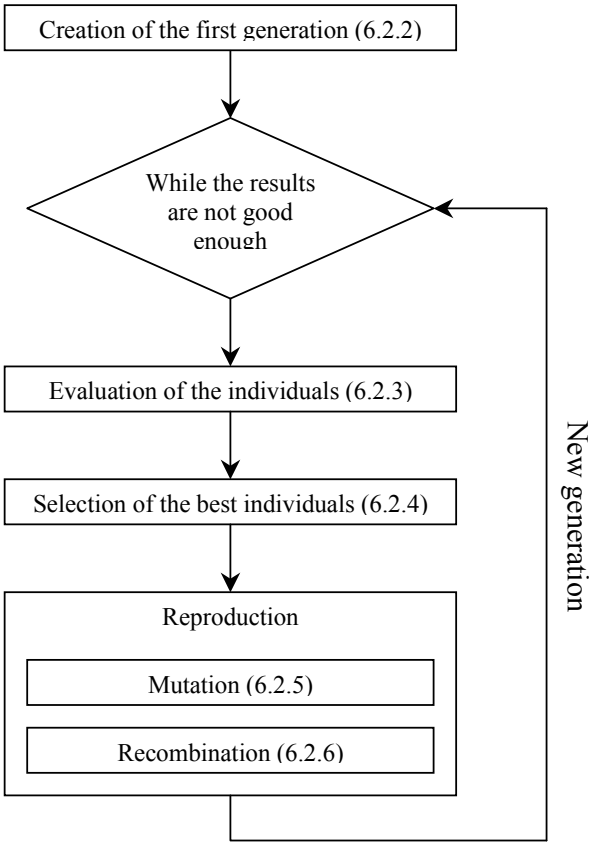
Each joint present in the robot will be associated with one set of these four parameters. If we consider all the 19 joints present in our robots, that makes a total of 57 parameters to define. Optimizing a system of dimension 57 is going to be incredibly difficult. So we should reduce the number of parameters, to make the optimization less difficult.
First, we can reduce the number of parameters by fixing the value of ω. This will simply fix the frequency of the motion. We can also hope that this restriction will tend to optimize the movements made and not the speed of the motion.
Second, the walking motion is perfectly symmetric and alternates regularly between the right and the left. So we can use this symmetry to suppress the parameters that control one half of the body. For example the motion of the left knee will be the motion of the right knee dephased by pi. If we call A_{left_knee}, ω_{left_knee}, φ_{left_knee} *and* $\theta_{0\,left\_knee}$ the parameters controlling the left knee, the position θ_{right_knee} of the right knee will be given by this formula:

$$\theta_{right\_knee}(t) = A_{left\_knee} \cdot \sin(\omega_{left\_knee} \cdot t + \varphi_{left\_knee} + \pi) + \theta_{0left\_knee}$$

With these two operations, we reduced the number of parameters to 33. This is still a lot. But a genetic algorithm should be able to find a good solution, into this huge search space.
Before going any further in the explanation of the genetic algorithm used, just a word about this sinus based controller: We can see by looking at the formula above that the control is done totally in an open loop way. It means that the controller has absolutely no feedback of what it is actually doing. Each joint will follow its sinus function even if the robot begins to fall. Therefore this controller won't be robust at all. A small perturbation like a bump on the ground and the robot will probably fall. But let's first generate a simple walk. We will discuss the robustness latter.

6.2. Using a genetic algorithm to optimize the parameters

The genetic algorithm is an optimization method often used in robotics. The idea of the genetic algorithms comes from the Darwinian evolution: At the beginning a random population of programs is created. Then each individual of this population is tested and obtains a score called the fitness. In fact the fitness represents how well an individual is performing the desired task. Then the individuals who obtained the best fitness are selected to reproduce themselves and thus generate the next generation. Then generation after generation the individuals are getting better and better.

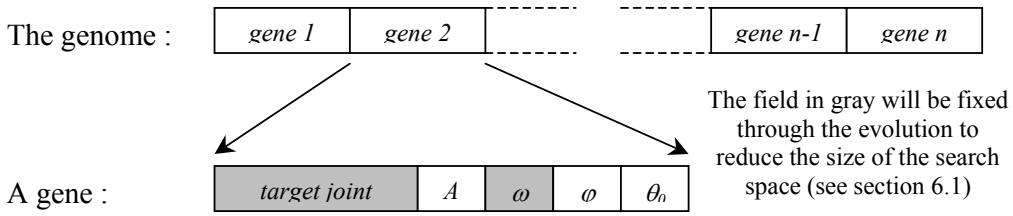


The schema above describes roughly the execution of a genetic algorithm. Each box of the schema represents one stage of the algorithm. Each stage is explained in more detail in the section designated by the numbers in parenthesis.

6.2.1. The genome used

In the real evolution, each individual develops itself according to its DNA. In fact the DNA is the carrier of the information in the evolution. It's through the DNA that the parent can transmit their strengths and weaknesses to their offspring. So, no DNA means no evolution. This statement is also true in an artificial evolution. It's why our genetic algorithm also needs a virtual DNA. Usually in artificial evolution, the virtual DNA is called the genome.
Therefore the first decision to take, when implementing a generic algorithm, is to design the genome that will be used. In the case of the robot the genome will be pretty simple: It will be a sequence of genes. And each gene will contain the four parameters A ω φ and θ_0, described in section 6.1, plus one symbol called *target joint* that indicates which joint is actually controlled by the gene.

Our genome: A sequence of n genes, each gene controlling the *target joint* joint:



Each robot of each generation will have a virtual DNA like the one shown above. In the algorithm used for this project the genome contains 11 genes (1 for each joint minus the symmetric joints). Then this genotype will be interpreted and the robot will move.

6.2.2. The creation of the first generation

Before launch an evolution we need to generate a first generation of individual. Of course we have no idea of what this first generation should be. It's why usually the first generation is created totally at random. The random generation should also insure that the initial population is well distributed all over the search space.
But for the walk problem, it's not possible to generate the first generation at random; because it generates some ample motions with big amplitude. These motions automatically make the robot fall. Therefore all the individuals of the first generation fall, and there will be no way to tell which robot was good. So there is no way to select the best robots for the reproduction and the evolution is will never start.
To avoid this problem we have to be sure that most of the robots of the first generation won't fall. It means that the robots of the initial generation must all have motions of small amplitude. This was done by setting the two parameters A and φ, of the first generation, to a value near from zero. Of course this trick has the big

disadvantage to concentrate all the individual in narrow region of the search space, but normal the evolution should make the individuals move to some other regions of the search space.

6.2.3. The fitness function

The fitness function is the way to grade all the individuals of a generation, thus it must reflect the ability of an individual to perform the required task. This is the reason why the fitness function is one of the most important parts of a genetic algorithm: The fitness function is the engine of the evolution and must guide the individuals to the good solutions. A bad fitness function can completely ruin the evolution. Usually a good fitness function should be implicit and not too directive, otherwise there is the risk to introduce a bias in the evolution.
The fitness function used in this genetic algorithm works on the following principle:

- A good robot should walk far.
- But a good robot should also not fall.

Therefore, each individual will be tested during a fixed amount of time. At the end of the evaluation time the distance covered by a robot is measured. If the robot was still standing when the evaluation terminated, its fitness is directly the distance measured. But if the robot falls or if the distance is negative the robot receives very low fitness.
Finally we have to make sure that the evaluation of an individual last long enough. Making short evaluation looks interesting because it would accelerate the algorithm. But short evaluations can also bias the evolution. For example, if the evaluation is really short, an individual will perhaps have a good fitness by jumping as far as he can. In this case the fitness will of course be good, but it won't estimate correctly the ability to walk.

6.2.4. The selection method

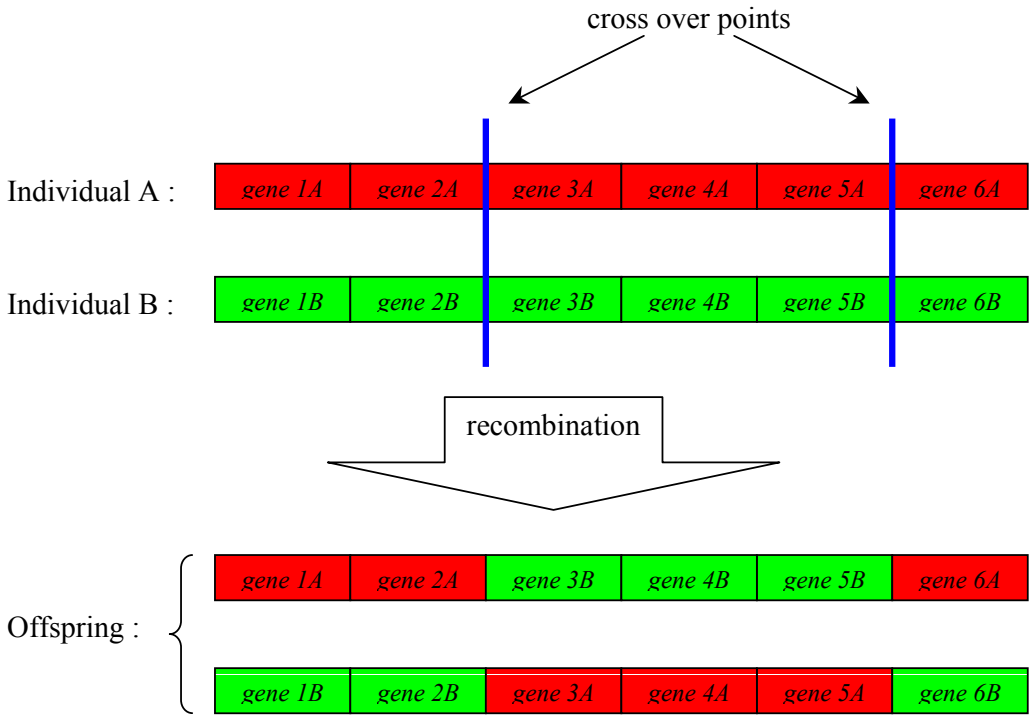
There are several selection methods used in genetic algorithms. Buts the most popular one is call the Roulette Wheel method. Here is how it works:
Each individual is a slot of the wheel like the ones present in the casinos. The size of the slot is proportional to the fitness of the individual. Then the selection game starts: The wheel is launch and the individual on which the wheel stops is selected for the reproduction. The game continues as long as new individuals are required for the next generation.
With this method the individuals with a big fitness will more likely be selected, but the worst ones will still have a chance to reproduce themselves.

6.2.5. The recombination operator

As shown in the schema of section 6.2 the reproduction includes two different operations: the mutation and the recombination. Each one has a different purpose but both required.
Let's begin with the more intuitive one: the recombination. This operation is the equivalent of the crossing over, in the animal reproduction. In other words, the DNA of two different individuals will be mixed together to create one new individual.

The number of crossing over point is fixed and their position is defined randomly for each recombination. But to preserve the quality of an individual, a cross over point will always be located between two genes of the genotype.

The following schema illustrates well the crossing over:



From the point of view of the search, this operator is the one that will make the individual converge to the good solutions. But if we use only the recombination operator during the reproduction, the search will easily be caught in some local minimum of the search space. It's why we introduced the mutation operator.

6.2.6. The mutation operator

As explained in the previous section the recombination operator is used to converge to the good solutions. But we also want the algorithm to explore some new regions of the search space. The mutation operator can do this. Once again the idea of mutation is directly taken from the biology: As it happens with the DNA, some parts of the genotype are modified randomly. These random changes will sometimes bring an advantage to an individual. With his new advantage the individual will have more chances to reproduce himself and to transmit his advantage to the next generations.
The mutation operator used in this project is very simple: It selects at random one of the three modifiable parameters of a gene (A φ or θ_0) in the whole genome and then adds or removes a little value to it.
Finally we just have to make sure that this operator will not force the robot to make an impossible move like breaking the articulation of the knee. The model of the robot has no limit set for its joints. It's why we have to take care of this problem at the level of

the genes. Otherwise the evolution will perhaps find an efficient way to walk, but totally non-realistic according the physical limits of the real robot. So if we notice that a mutation brings a joint in an unreachable position, the strength of the mutation will be diminished.

6.2.7. Introducing elitism

The description of the algorithm is almost over, we just need to take one final decision about the evolution. This decision concerns the replacement policy of the population. In other words, we have to decide how many individual will survive from one generation to the next one, and how the new individual will be added to the population. Once again many policies exist and are commonly used. But none of them is better; each one is adapted to different situations.
So we made the choice to conserve only the best individual from a generation to the next one. This will at least preserve one set of good genes in the population if suddenly there is a very bad generation.

6.3. The shape of the search space

The analysis of the results will come in the next chapter. But one observation can already be made on this particular problem of walking. This problem is incredibly hard and not that well adapted to the generic algorithms. It primarily comes from the shape of the search space. As we already noticed with the problem of the initial population, the set of the good solutions covers a very small portion of the entire search space. The search space is also really steep, and ridiculous change can transform a running robot into a mass rolling on the floor.
But finally the difficulty of the search makes it interesting.

6.4. The implementation using Webots

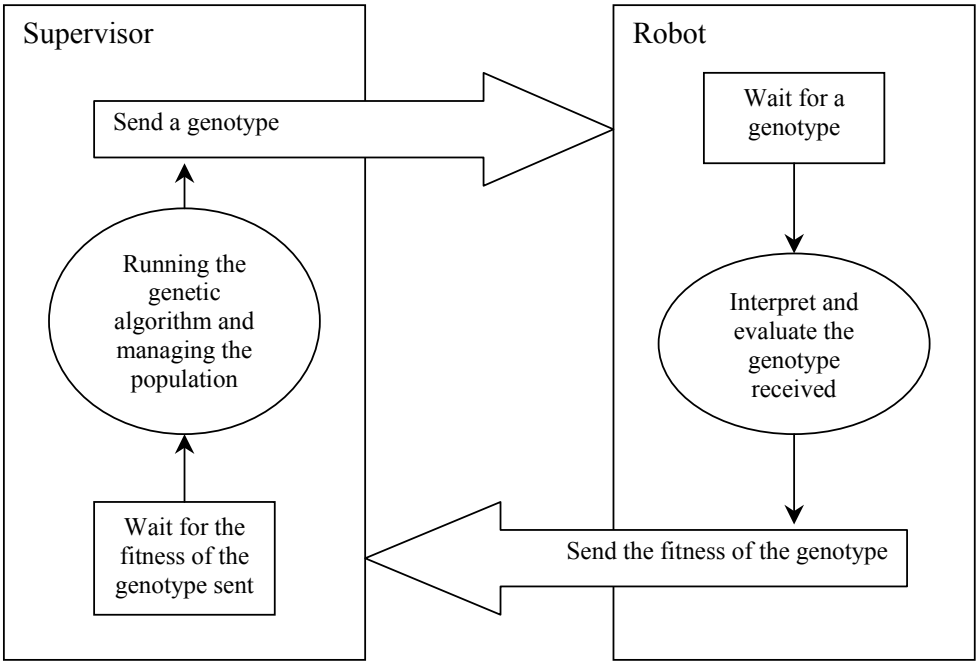
The theoretical part concerning the genetic algorithm is over. So, here comes the time to think about the implementation. The way the algorithm will be implemented is going to be exactly the same that we would use to train a real robot. There will be two programs running: One taking charge of the evolution like a supervisor and the second will control the robot.

6.4.1. One program for the robot, one program for the supervisor

In Webots, each robot present in the world contains a field (a WRML node in fact) that defines which program must be used to control the robot. So Webots launches on different program for each robot present in the scene. But that's not all. We can also introduce another kind of actor in the scene, called a supervisor. This supervisor is like a god in the scene: not visible on the screen, it has the power to manipulate all the objects present in the world. This supervisor also has an associated program. So we will use one of these supervisors to run the genetic algorithm and to manipulate the

population. On the other hand the robot will only receive its genotype from the supervisor and will interpret it.

The follow schema shows the execution cycle and the responsibilities of the two programs:



On the left of the schema there is the supervisor, which is in charge of the learning side of the simulation. And on the left there is the robot's controller, which is only here to interpret the genomes it receives. Both are running separately and alternatively.
The two big arrows represent the communication between the two programs. Usually communication between programs or processes is not a trivial in computer science. But hopefully for us, Webots provides a nice communication interface that hides to the user all the underling mechanisms. The robot and the supervisor must simply both be equipped with a receiver and an emitter; and they can communicate. Then any data can be sent through an emitter and the received with the corresponding receiver.

6.4.2. Webots and the problem of the initial conditions

Very often, doing an artificial evolution on a real robot is very difficult. Usually the robot must always start the simulation at the same place in the same position. But during its evaluating the robot will move and change its position. That means that at the end of each evaluation the robot must be sent back to its initial position. But we are working in a simulation so this problem should not exist. That seams logic but Webots and ODE are imitating the reality so well that the problem occurs also in the simulation.

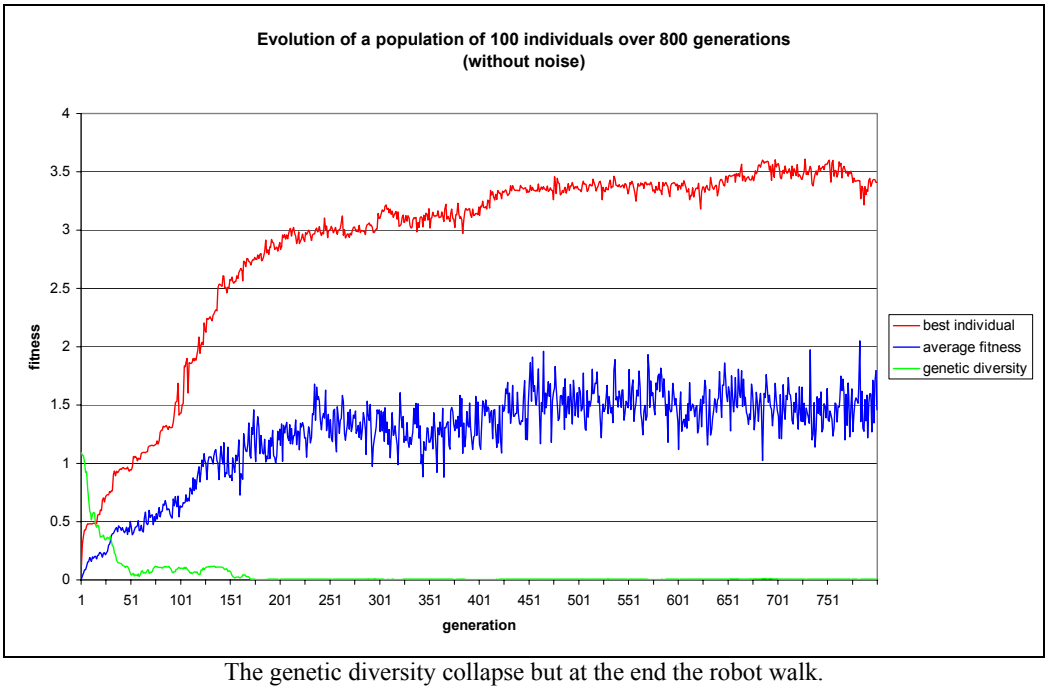
The functions exist into Webots to change the position of a robot. But the source of the problem comes from ODE. As explained in the chapter about ODE, each body has a linear velocity and an angular velocity. So, even if Webots changes the position of the robot, all the bodies that are composing the robot will still have their old velocity. And that will perturb the beginning of the next evaluation. Therefore, between two evaluations we must let the time to the robot to stop his motion.
This solution look acceptable but in fact it introduces another problem related to ODE: A system of rigid bodies simulated with ODE can never be perfectly stable state. For example, there are always some forces that are present between to connected bodies. These forces and some other correction factor must be present to preserve the integrity of the simulation.
So even if the robot stopped its motion, its real state according to ODE is not stable. Therefore there is no way to recreate exactly the same starting condition from one evaluation to the other. Concretely, it means that for one evaluation, an individual can have a good fitness; but if the same individual, with exactly the same genes, is reevaluated, the fitness can change totally: Only because of different starting conditions.
This problem could easily be solved if we add the possibility to manually set to zero the velocity of each ODE body. ODE offers the functions to do that. But these functions are not accessible from Webots yet. So this problem is present in this project, and we can do nothing about it.

7. Analysis of the results

The analysis of the results is the most important part of this project, mostly because the results shown below are far from being perfect. As we will see the robot learned approximately to walk. But this uncertain walk is not the most important result; the experience acquired and the problems we had to face during the experiments are certainly the most valuable outcome of this project. So, let's see what we can discover in the results of our simulation.

7.1. The ability to walk

All the simulations run and analyzed here were all launched with a population of 100 individuals over 800 generations. After this number of generation the population already stopped progressing, so there is no need to continue the evolution any longer. We can clearly see on the graph below that the curve is almost flat for the 200 last generations. It means that the genetic algorithm is trapped in a local minimum of the search space.



The genetic diversity collapse but at the end the robot walk.

The previous graph shows the progress of the population over time. The curve in red represents the fitness of the best individual of each generation. The blue one is the average of a generation. And finally, the green curve measures the genetic diversity of a generation: if the genetic diversity is high, all the individuals of the population are

very different. But when the diversity is near for zero, it means that all the individuals are almost the same.

But the graph also shows that the local minimum found is not a too bad solution. Indeed, the best individual of the last generation has a fitness around 3. Concretely it means that this individual walk a distance of 3 meters during the 20 seconds of simulation. It corresponds to a speed of 0.5 km/h. So to the question: "does the robot walk?" We can answer, "Yes but slowly!"
The walk adopted by the robot is slow, but also not really natural. The robot inclines itself ahead, make big steps and keeps its knees straight. But it's interesting to notice that this walk is dynamic and not static; there is a moment, at each step, when the robot is clearly not in a stable position. It's surprising that a totally open loop control can handle a dynamic walk. The traditional solutions used to make robot walk are usually based on a static walk, which contains no unstable phase. It proves once again that the genetic algorithms are very strong to find solutions that could not be found by hand. Therefore, it is certainly a good idea to use genetic algorithms to solve the biped locomotion problem.

7.2. Was the search space well explored?

The collapse of the genetic diversity is the most surprising result show by the previous graph. After 50 generations, the diversity is already near zero. Why knew that the way we created the first generation was penalizing the genetic diversity. But we didn't expect such a catastrophic behavior during the simulation.
We could imagine that introducing a more aggressive mutation operator in the genetic algorithm would prevent this problem. But it's the opposite: If we increase the mutation rate that will generate a lot of bad individuals that will directly fall. So, only few individual have good fitness and reproduce themselves. Conclusion, the genetic diversity collapses even faster.
This lost of diversity is terrible for a genetic algorithm, because all the solutions are at the same place in the search space. Therefore, when the genetic diversity is closed from zero, the genetic algorithm becomes a stupid greedy search.
Using a replacement policy deferent from the Roulette Wheel method can perhaps diminish this problem. Another solution that preserves the genetic diversity would be to use an island based genetic algorithm. The only difference with the classic algorithm is that there are many population evolving separately on different islands. And sometimes a set of good individuals of an island can be sent to another island. This variant has the advantage to keep each island in a different region of the search space, thus the genetic diversity should stay higher.

7.3. The robustness of the walk

Until now we only spoke about walking or falling. But the reality is not as simple as that: there are some robots that are walking well if they are not disturbed, but as soon

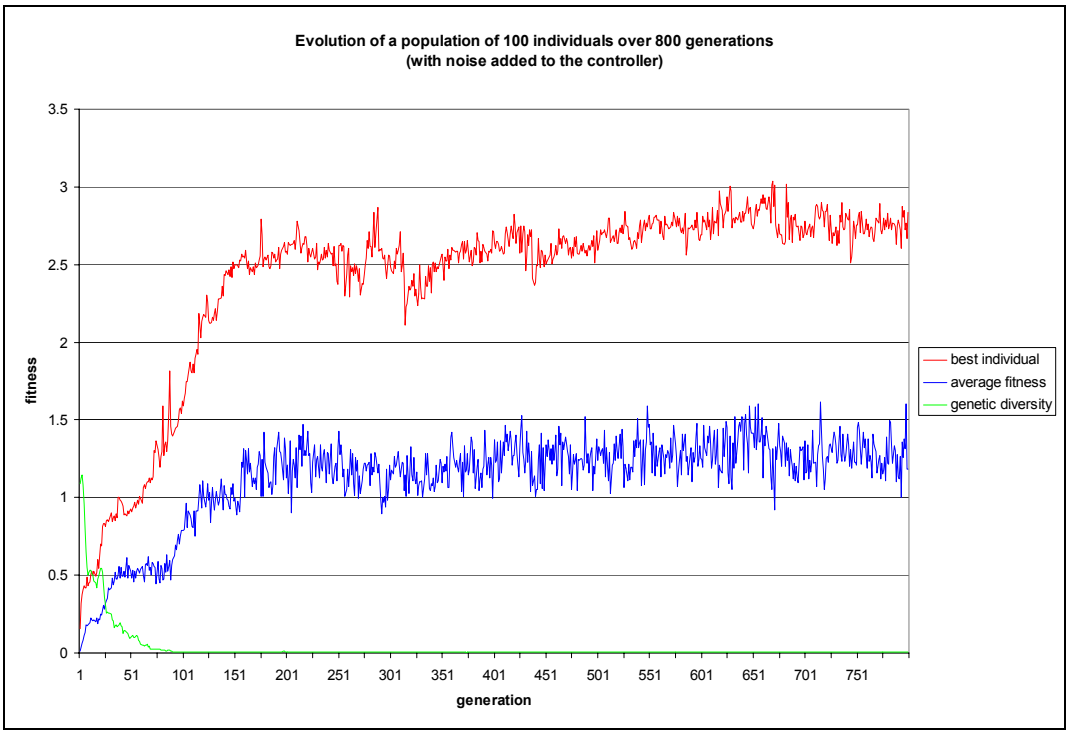
as you push the robot a little bit it falls. In fact, some types of walk are less sensitive than others to perturbations; this is what we call the robustness of the walk.
So, if we watch our simulation from the point of view of the robustness, we must admit that all the solutions found by the evolution are far from being robust. But in some sense it is logic, because there is nothing in our fitness function that rewards the robust walk. So the evolution chooses the easiest solution.

7.4. Improving the robustness by adding noise during the learning phase

To make the evolution find a more robust solution, we had a very simple idea: We simply added some noise directly in the simulation. Then if we launch the evolution with the noise, it should find individuals that walks and that are able to resist to the noise. But as we will see, all doesn't work so easily in practice.

7.4.1. Adding noise into the controller

The first way to add perturbations into the simulation is to add some noise at the controller level. With noisy controllers, the joints won't follow exactly the trajectory described by the sinus function. The noise was simulated by adding a cumulative error to the sinus function. The value of this error was fixed into a range between 0.01 and –0.01 radian.
The following graph shows an evolution made with noisy controllers:

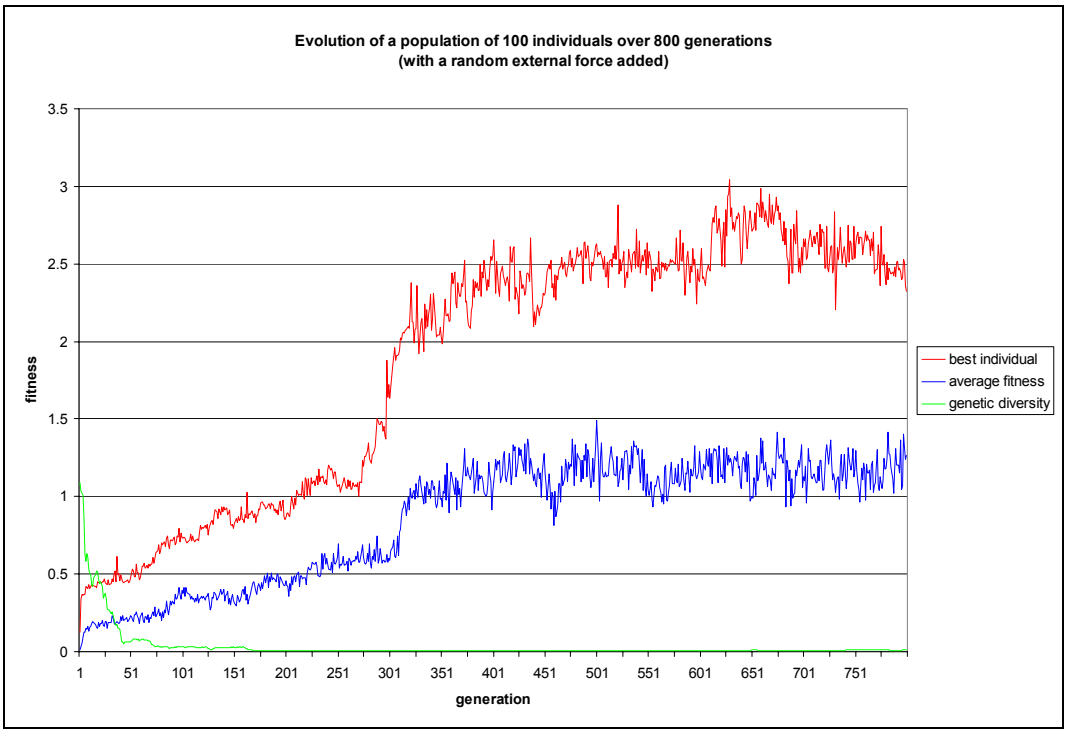


The noisy red curve shows how the noise changes continuously the fitness space.

The analysis of these results is far from being easy. We can first notice that the evolution was a little bit disturbed by the noise, thus the fitness are not as high as previously.
Second the red curve is noisier in this simulation. It suggests that the noise added in the simulation also perturbs our fitness space. In other words, the fitness of the same individual can change a lot between two evaluations. And of course, that perturbs the genetic algorithm.

7.4.2. Adding an external force as noise

The second technique used to add noise to the simulation is to push the robots with a random external force. The following graph shows an evolution made with this external force.



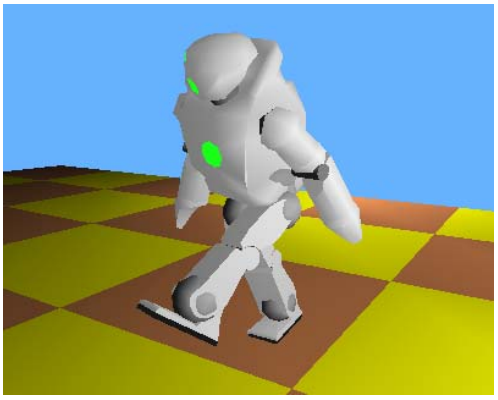
This time the evolution was slower but the noisy red curve is still here

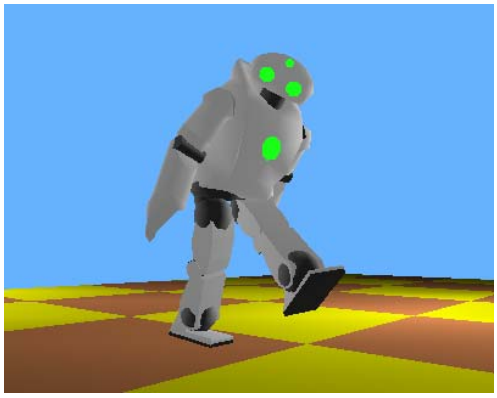
As the previous noisy simulation, this one is not trivial to comment. This time we can notice that the noise makes the evolution slower, and the fitness space is once again disturbed. But it's difficult to say more about it. Even in the simulation, the walk without noise and the one with are looking very similar.

7.4.3. Did the noise improve the robustness?

To have a better idea of the impact of the noise in the evolution, we decided to test the individuals trained with the noise in the ideal situation and vice versa. But unfortunately the results were no good at all: We were hopping that an individual trained in a noisy situation would behave very well without noise. But it wasn't really the case. Some times, it was even the opposite: the robot was always falling when the noise was not present; as if the robot learned the noise. This unexpected behavior makes the situation really difficult to analyze. But we can at least be sure that our try to improve the robustness is a failure. So, we should at least try to understand the reason of this failure.
Perhaps we were not aggressive enough with the noise. We should have added more noise to see a clearer difference between the robots evolved with the noise and the others. But too much noise can also totally prevent the evolution. Once again in this project the limit is thin between not enough and too much.
There is also this problem related to the perturbation of the fitness space. We could limit that by evaluating each individuals many times and computing an average fitness.
But the problem could also come form the simulation itself. Perhaps ODE isn't good at managing noise, because it is not designed to. Therefore, the result could be that the simulation no more matches the reality.

That makes a lot of suppositions. The only thing we can be sure of is that a lot of work still has to be done to find a robust walk.


Our best robot is walking.


But the motion doesn't look natural at all.

8. Conclusion

Here comes the time to conclude this project, and to see if we reached the goals we fixed at the beginning of this document.

The first goal of this project was to create a simulation of a biped robot that includes the laws of physics. With the help of ODE and Webots this first goal was not too difficult to reach: ODE provides all the tools needed to simulate the physic and Webots takes care of the robot itself and all its devices. These two tools made the implementation of the simulation very simple from the coding point of view. That let us the freedom to devote a lot of time to the simulation's tests. These theses helped us to correct our first design and the overall quality of the simulation has been greatly improved. Finally, we can affirm that our first goal is reached: The simulation runs well and the robot's behavior is credible.

The second goal was to do a first approach of the biped locomotion problem. The approach used was very simple, almost naive. However, the results observed are not so bad and the robots finally walks. But the most important here is not the results themselves, because we were not expecting them to be wonderful. The most important is that we learn a lot about the problem by trying to solve it. Of course, there are still many parts of the problem that we don't understand perfectly. But now we are aware of these dark parts. So we can now begin to ask the right question and to look in the right directions.

However we have to be realistic: Our robot is at years light of the brave C3PO. But the robotic is a brand new field of research and our robots is a young child compared to C3PO. As a child our robot falls very often. But as a child it will also grow up and learn better techniques. The biped locomotion of the human being itself is also far from being completely understood.
All these reason make the biped locomotion a dynamic and impassioning field of research. And perhaps, one day this field will discover the robots we were dreaming about while watching Star Wars.