Pentapod

Jochen Alt



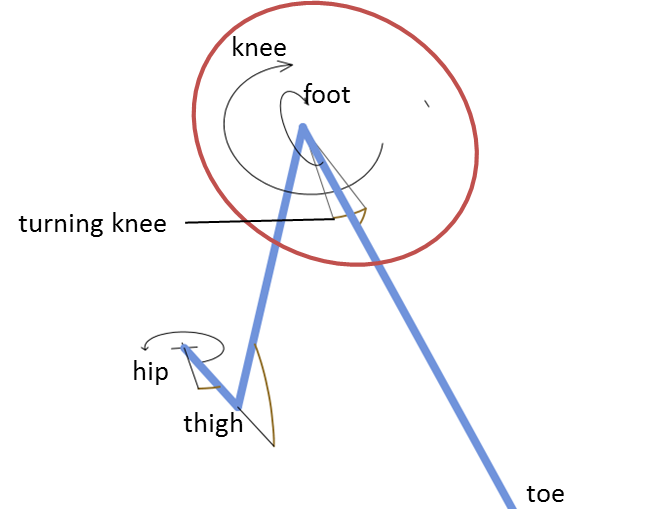
General Thoughts

When seeing a video about a walking Hexapod no one screams wow, that’s unbelievable. There must be more. For one, you could cut off one leg. Having 5 legs only is uncommon (which is good), and might be a challenge to develop a gait with one missing leg where you cannot steal something from nature. And, you can spend 1/6 of the money for something else.

What else, not yet sufficient to get a wow. So, I tried to give it a personality, this time it should be something shy and chicken-hearted, but how to express that, if you’re an insect with one missing leg? Hiding from anything that’s moving could be something. And this was the solution to the previous question what you can do with the price of one leg: Buy a Lidar sensor and put it on its head to identify suspicious subjects and escape in holes.

Typical Hexapods have 3-5 DOF per leg, and most of them have only one DOF that moves the leg from the left to the right while all other DOFs move up and down. I never go the reason behind, since when looking at most of the hexapods you get the impression of a “brutal” gait since the entire leg including the thigh (or “femur”) is moving while \*not\* trying to minimize the mass that is to be accelerated. There’s just one hexapod I am aware of (“Weaver”) that spends one DOF for turning a leg such that the foot can move forward and backward without moving the femur. Unfortunately, this DOF is not used during walking but only to compensate standing on a ramp and levelling the body’s orientation.

In order to minimize the moved mass, it seems natural to introduce a knee, what leads to a 4 DOF leg design with a turning possibility:

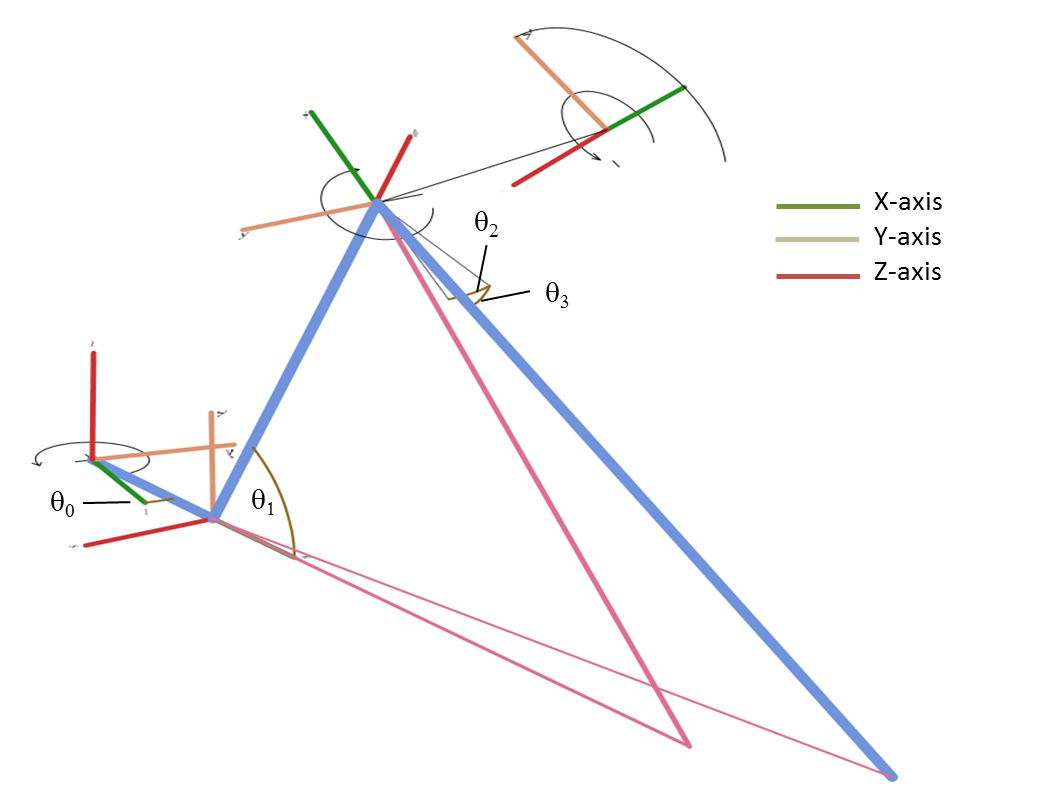
The blue lines illustrate the leg consisting of the joints hip, thigh, knee and foot.

The turning knee (later on **) allows to move the toe to the left and the right without moving the tibia. This should allow more efficient movements since the tibia moves less compared to a classical design without a turning knee.

In order to allow a smooth gait and to have the capability to walk in uneven terrain, there’s the need knowing the distance of a toe to the ground. Although it might be sufficient to know if the toe touches the ground, but this would surprise the gait controller all the time, knowing the distance should result in a smoother gait control. So, I decided to go with a cheap laser distance controller per leg.

Kinematics

Kinematics is about computation of the toe’s point out of the joint angles and vice versa. First is simple, latter is tricky. The coordinate systems are illustrated as follows, such that we can derive the Denavit Hardenberg.



A Denavit Hardenberg transformation from anglei to anglei+1 is given via

1. rotation around the z-axis by joint angle
2. translation along the z-axis by *d*, and
3. translation along the x-axis by *a*
4. rotating around the x-axis by 

So, the Denavit Hardenberg parameters are:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Joint | a[°] | a[mm] | d[mm] |  |
| hip | 90° | d0 | 0 |  |
| thigh | 90° | d1 | 0 |  |
| knee | *(90° +)*  | 0 | 0 |  |
| lower leg | 0 | d3 | 0 |  |

According to the coordinate system above, has an offset of 90°. The general definition of a Denavit-Hardenberg (*DH*) transformation is

|  |
| --- |
|  |

which is a homogeneous matrix with two rotations (x,z) and two translations (x,z).

Combined with the DH parameters, the following DH matrixes define the transformation from one joint to its successor:

|  |
| --- |
|  |

Forward Kinematics

With the DH transformation matrixes at hand, computation of the leg’s pose out of the joint angles is straight forward. The matrix representing the toes pose is

|  |  |
| --- | --- |
|  | (6‑8) |

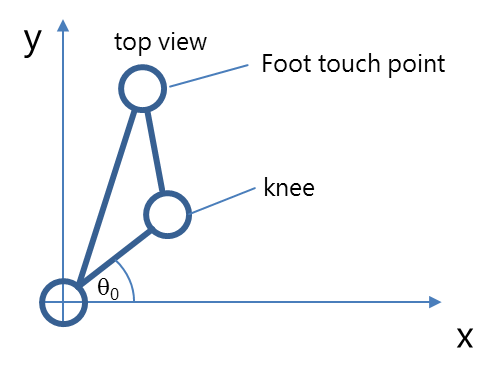
By multiplying the transformation matrix with the origin (as homogeneous vector), we get the absolute coordinates of the toe point (*TP*) centre point in world coordinate system (i.e. relative to the legs’s base).

|  |  |
| --- | --- |
|  | (6‑9) |

That was easy. The tricky part comes now.

Inverse Kinematics

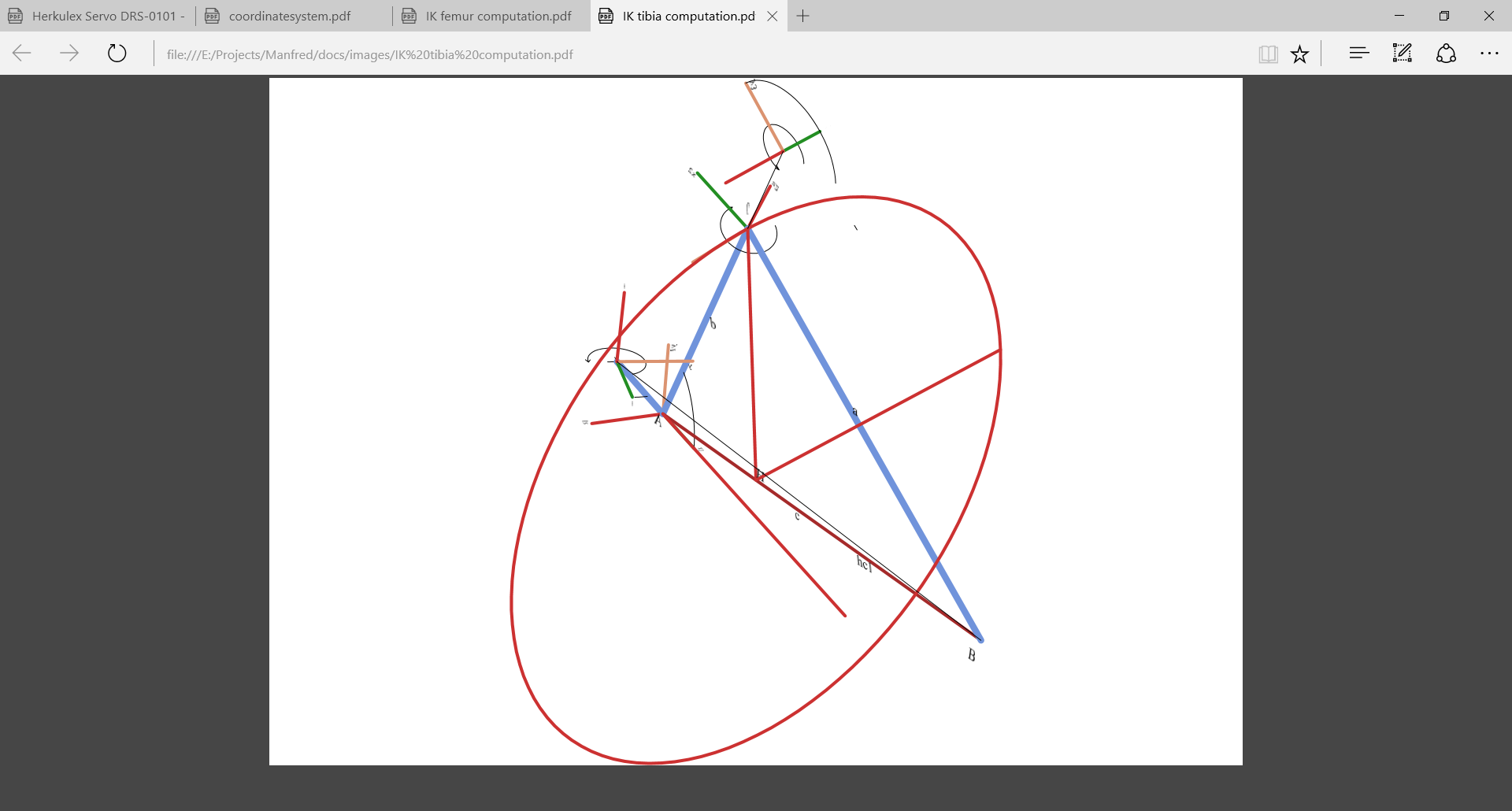
Inverse kinematics denotes the computation of all joint angles out of the toe’s position (*TP*). Since the leg has four joints, it is becomes clear that there is an infinite number of solutions for that, so I need to predefine one angle with an arbitrary definition. Having the objective in mind of moving the higher limbs of the leg as little as possible, I arbitrarily chose ** and set it as angle bisector of the toe to the hip (from bird’s perspective):



We get

Later on, we will need the coordinates of end of the first limb (A) which is

Computation of the second angle ** at point A requires a geometric analysis. The leg is denoted in blue, all construction lines are red.

We consider the triangle from A, B and C. The two lines and are of fixed length. So, the point C is upon the circle with the centre H and the radius of the triangle’s height. Additionally, C is defined as function of ** and **, so we should be able to derive ** by intersecting the circle with *C( ,)*.

The only thing we need to do is to express that in terms of coordinates. First, we compute the length of a, b and c:

Now that the triangle is defined, we can compute the height by Herons formula:

The base of the height H is defined by

Now we need to define the circle *K* with radius *h* and centre *H*. This is done by

with S orthogonal to beginning from *H* and T orthogonal to S and :

So, with the arbitrary assumption and the length

we get

(This equation could be simplified, but this way programming is easier by computing the y coordinate and deriving the x coordinate)

There are two possibilities for *S*, representing two configuration with knee up and knee down. We always take the healthy one where the knee is above the toe point. Finally, T is defined by its orthogonality to *S* and its length :

Having the circle defined, we need to intersect it with the possible positions of C:

Hereby denotes . We consider only the equations of x and y coordinates and solve these for . Equating gives

This needs to be solved by in order to get point C. Unfortunately, we have sin and cos in the equation, but luckily with the same parameter. Wikipedia helps with sinusoids:

This is used to solve the equation above for 

a =

Out of  we get C by , out of C we compute **by considering the z-coordinate of C:

which results in

The first angle is always the hardest, time for a beer.

We leave the knee-turn-angle **aside for a while and continue with the tibia **. This is done by considering the triangle ABC, and the angle at the point C represents **. In a fully elongated leg ** is 0.

Therefore,

The last angle ** is computed by use of

So, let’s have a closer look into the transformation matrix and check if there are some useful equations considering that we already have all other angles. Annoying multiplication results in

Since we need to compare this to the toe point, it is not necessary to compute the full matrix, the right column is sufficient. We are lucky, the third line has only one expression that depends on **, so we get

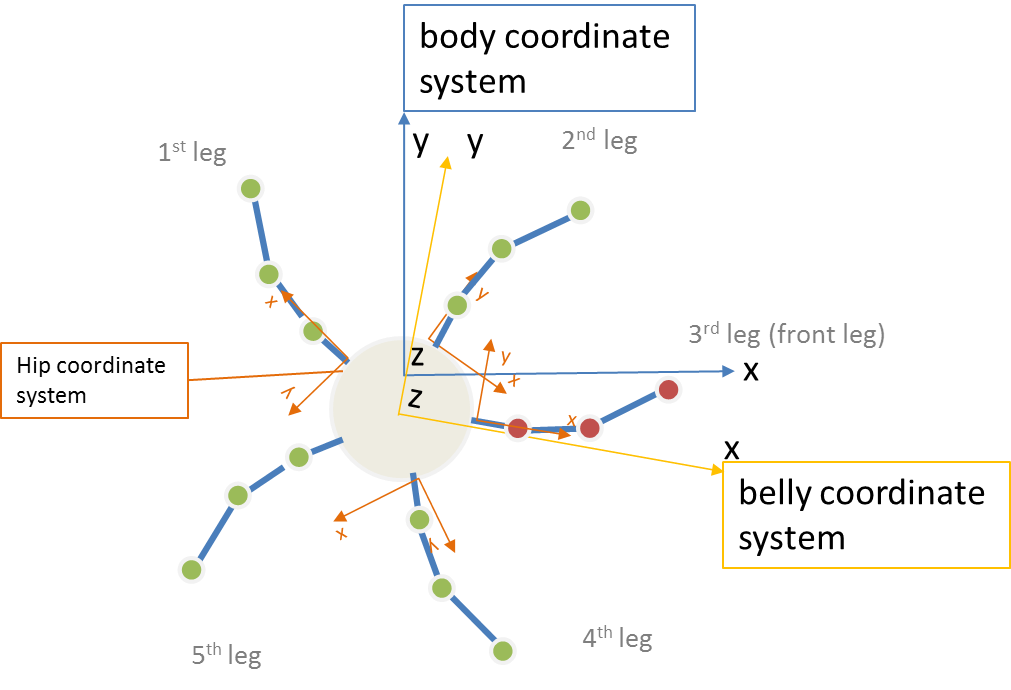
Again, *arcsin* results in two solutions, so we need the other coordinates as well to check which solution is valid.

That’s it. Surprisingly complex for a leg with only 4 degrees of freedom.

This is implemented in [LegKinematics.cpp](https://github.com/jochenalt/Pentapod/blob/master/code/Kinematics/src/LegKinematics.h).

Body Kinematics

Attaching 5 legs to a body implies to compute the leg kinematics depending on each hip. Additionally, we might want to translate and rotate the belly in certain limit. Since the chapter on leg kinematics computes the angles out of the toe in the hip coordinate system, we need to translate each leg’s toe point into the hip’s coordinate system.

The pentapod’s pose is given in the body’s coordinate system, which origin is on the ground right below the body button. Since the belly can translate or rotate, the next coordinate system is the belly coordinate system which origin is the belly button. When the pentapod is in the default position, the belly coordinate system is translated in the z-axis only by the height of the belly. Finally, we have 5 hip coordinate systems which are x-translated by the distance of the belly to the hip and z-rotated by , where *n* is the number of the leg.

We define the transformation matrix *Belly* that defines the belly coordinate system out of the body coordinate system, that is a 3D rotation matrix plus a translation along the belly coordinates:

Per leg we have an own transformation matrix which is a rotation in the xy-pane around z

where ,

Having a point in one coordinate system and watching it from another one is done by multiplying it with the inverse transformation matrix. So, the toe point from the hips coordinate system *toehip* is computed out of the toe point from the body’s coordinate system by

Computing-wise, the inverse matrix is done by Gauss or similar approaches with a complexity of , which might be bad for the performance. Luckily, the inverse of a symmetric rotation matrix is the transposed matrix, and the rest can be computed by

which is much simpler.

All this is implemented in [BodyKinematics.cpp](https://github.com/jochenalt/Pentapod/blob/master/code/Kinematics/src/BodyKinematics.h).

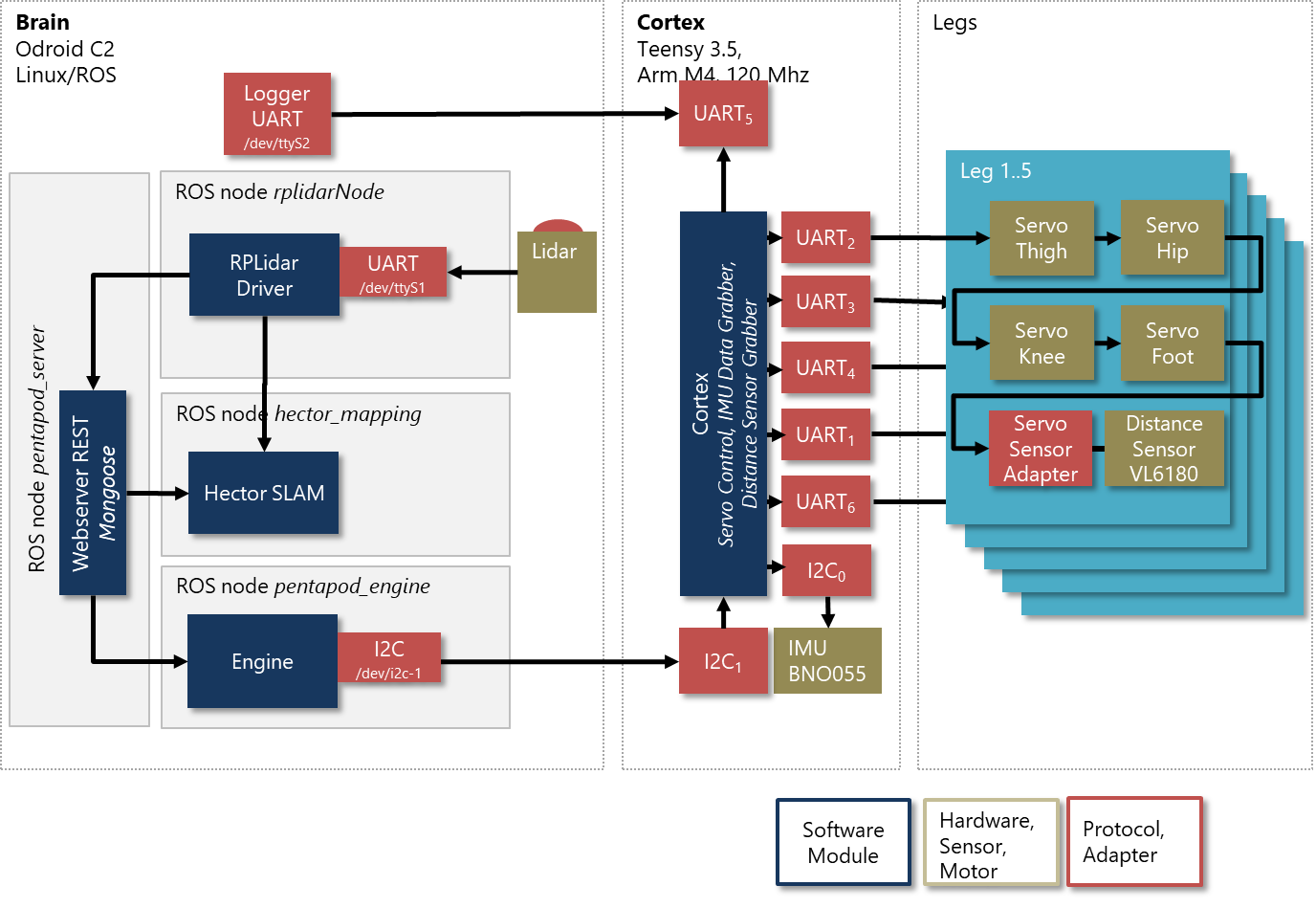
Construction

The construction is built around the following servo motors that have been choosen according to the force in the applied joint:

* Hip: Herkulex DRS-0101, 1.2Nm
* Thigh: Herkulex DRS-0401, 5.2Nm
* Knee: Herkulex DRS-0101, 1.2Nm
* Foot: Herkulex DRS-0201, 2.4Nm

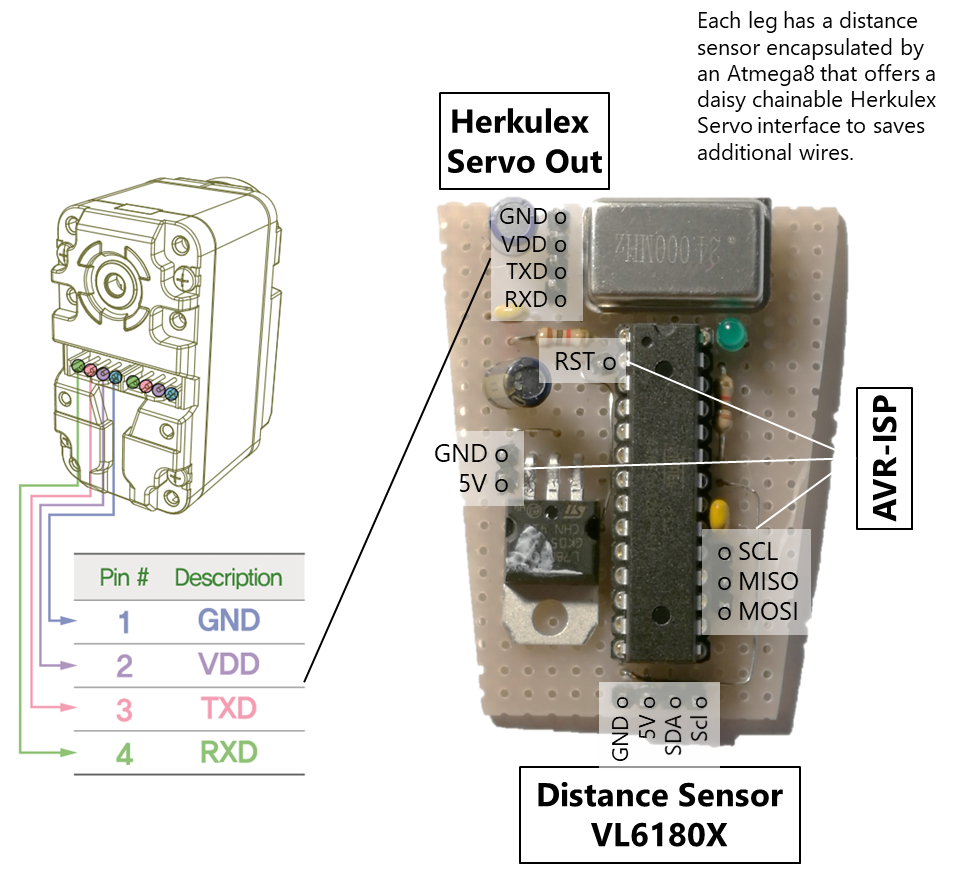
Overview

On the following overview, we can see all major components:

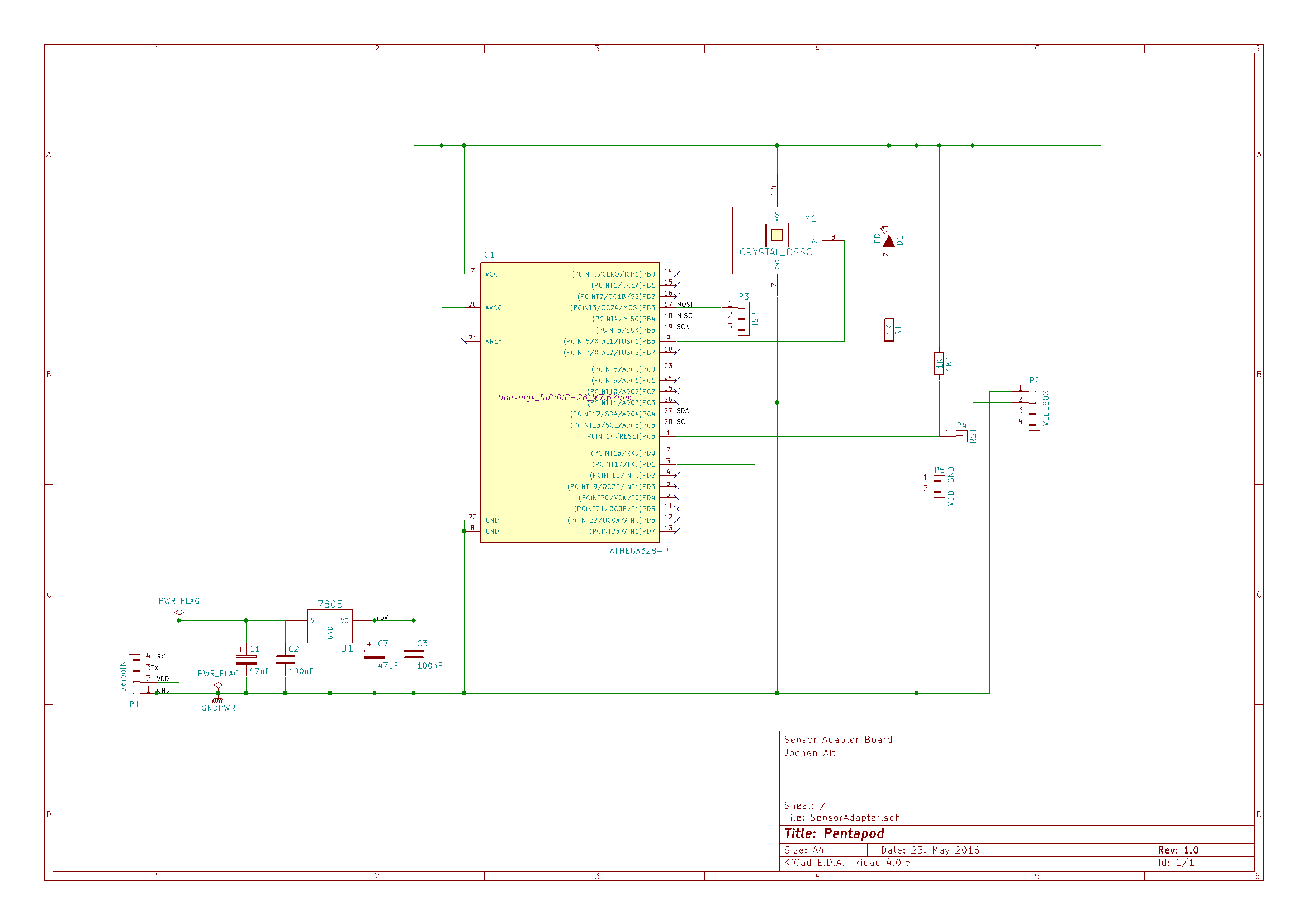


On the left, the high-level components are running on base of ROS on an Odroid C2 board. The most outer node contains a webserver for the communication with the outside world, and orchestrates functionality of all low-level nodes. One is grabbing data from the lidar sensor, one is running the SLAM algorithm, and the last node runs the pentapod-engine which controls the gait, does kinematics and sends according commands to the cortex.

When controlling 20 motors, it is significant how these motors get their commands. An update rate higher than 20 Hz seems to be is desirable to get a smooth movement and a quick response to the distance sensors in the leg. The motors are using a serial line with 115200baud, and one command packet is about 10 bytes long. Thing is, that the HerkuleX protocol returns a 7 byte response after a delay of max 5ms, which means that with one serial line, each motor blocks the daisy chained bus for 7ms which gives you an update rate of 140Hz per motor assuming you have one serial line per motor. 4 in a row (plus the distance sensor) gives you 20Hz in the best case, even without any delays or synchronisation times. So, I patched the Herkulex protocol to work with fire-and-forget requests only (the response is evaluated), assigned one serial line per leg, and controlled all legs by 5 subsequent rounds sending commands to all hips, then all thighs, … . By this, I achieved a communication time of 10ms all in all (100 Hz). Since HerkuleX Motors are working in multiples of 11.2 ms, I choose 3x 11.2ms and 30Hz. Going up to 44 Hz did not improve the movement, so I left some cpu time for other tasks. All this is done on the Teensy controller, which has 6 UARTs and 2 I2C lines. In addition, the Teensy fetches data from the BNO055 IMU in order to keep the Lidar horizontal.

Distance Sensors

Each leg has a cheap laser distance sensor. In order to save wires, it is encapsulated by a Atmega8 controller that behaves like a HerkuleX servo and can be daisy chained to the other servos.

The PCB is small enough to fit into one leg, it contains just a couple of components to serve the Atmega. Its main input is the regular 4-pin socket of a Herkulex server, its output is the socket to connect the VL6180 via I2C. The schematics is this:  


The firmware is a protocol changer, that listens to a serial line, and waits until a package comes in that is intended for the distance sensor. When this package is identified, the last measured distance is returned. The measurement frequency is derived from the request frequency: By averaging the request time, checking the VLS6180 happens right in the middle of two requests.

IMU

The used 9-DOF IMU BNO055 already has a fusion algorithm implemented, so no need to apply the Kalman filter on the teensy. In the beginning, it worked wonderful, until I used it with motors switched on. In total, the motors pull up to 6A served by a switching power supply. This resulted in permanently changing magnetic fields completely confusing the IMUs magnetometer. Luckily, the BNO can be configured to omit the magnetometers data fusing the gyros and accelerators only. So, after making it a 6-DOF device, it worked fine.



The IMU is mounted upside down right below the Lidar, in order to keep the Lidar horizontal. The PCB below is the Slamtec Lidar connector.