Laboratory of Embedded Control Systems

Lego Mindstorms NXT Motor Parameter Identification

Authors:
Nelson K. Psenjen
Yared S.Wondimu

Supervisors:
Prof. Luigi Palopoli
Prof. Daniele Fontanelli

June 19, 2014

Contents

1 Introduction

An experiment to determine the parameters for a `Lego Mindstorms NXT` motor was carried out and the results are presented here. Different power values for the motor are set and the tacho count is recorded, filtered and analysed for each power value.

Since the motor is already known to act as a second order system, a grey box approach was used to identify the damping factor, ξ and the natural frequency, ω_n.

Also, since for each power value we have an estimate of ξ_{est} and ω_{est} which results into an overdetermined system, a simple average of the ξ_{est} and ω_{est} values is done in order to determine the the final ξ and ω_n values.

Finally a set of verification data is used to check the accuracy of our model and from it, performance indices are calculated.

2 Data Collection

This section explains how the data needed for the identification of `Lego Mindstorms NXT` parameters was collected. It also includes a brief description of the code used.

2.1 Code

Here a brief description of the code used to collect the data including the various modules used is given.

2.1.1 `Lego Mindstorms NXT` Client

The `Lego Mindstorms NXT` client module is programmed using `nxtOSEK RTOS` and is written in C. It uses bluetooth to communicate with a server running on a PC to receive commands for controlling the motor and sending back the results. The data packets exchanged is of type `bt_packet_t` and it contains the following fields:

- **size** : This stores the size of the packet. Its of type `uint32_t`
- **packets** : This is an array of size **MAX_REQ** which holds individual requests send to/from the `Lego Mindstorms NXT` client. Its of type `bt_req_t`. The `bt_req_t` type contains the following fields:
  - **operation** : The operation that's to be performed by `Lego Mindstorms NXT`. This operation can either be to set the **motor power** or to fetch the current **Tacho count** of the motor.
  - **port** : The `Lego Mindstorms NXT` port onto which the request should be fowarded to.
  - **data** : An array of size **MAX_NUM_DATA** which contains data sent to/from the `Lego Mindstorms NXT`.
    For setting the motor power, only the first index is used and it contains the power value to be set. However for fetching the tacho count, the first index contains the tacho count itself while the second index contains the timestamp when data was fetched.

2.1.2 Server

The server runs on the **PC** and it uses an **AF_UNIX** socket to receive requests from a **Controller**, also running on the PC which issues the commands to be fowarded `Lego Mindstorms NXT`. Its sole purpose is to act as a gateway between the `Lego Mindstorms NXT` and the **Controller**.

2.1.3 Controller

The controller application runs on the PC and it uses an **AF_UNIX** socket to communicate to the server. It allows for the possibility to automatically set a given range of power values and store the tacho count data for each given power in a text file.
Its main functions include:

- Decoding of command line arguments. These arguments include:
  - Port number: This is th port number on which the motor is connected to the `Lego Mindstorms NXT`. its preceded with **-p**.
  - Power range: This is the range of the power values we wish to set to the motor. It contains three values delimited by **':'**. These values are the minimum power, maximum power and the step size(ie. Difference between each successive power value).Its preceded by **-m** .e.g. -m MIN_POWER:MAX_POWER:STEP_SIZE
  - Number of samples : This value specifies the number of samples to be fetched for each power value set to the motor. Its preceded by **-m**.
- Settting the motor power. It prepares packets to be sent to the `Lego Mindstorms NXT` for altering the power values of the motor.
- Fetching the tacho count. The controller prepares and sends packets for feching the motor tacho counts and then logs the data in a text file. The format of the text file is as follows:
  ⟨tacho count⟩⟨timestamp ⟩⟨power value ⟩

Between each subsequent power change, the motor power is set back to 0 and setting the next power value is delayed by 5 seconds to allow for the motor to rest and increase the accuracy of the data.

2.2 Collecting the Data

Five sets of three thausand samples(3000) each was collected for ten different power values. In order to minimize the noise in the data, the five sets of data for each power value was averaged to produce the values to be used for analysis and identification of the motor parameters. The power values used is from ten to a hundred respectively.
The averaging was done using a scicos script and the results saved in text files: one for each power value in the same format as the original files.

2.2.1 Data Filtering

The data is filtered using an exponential low pass filter.

$$y(n) = \alpha x(n) + (1 - \alpha)x(n - 1) \tag{1}$$

An α value of 0.575 is used to perform the filtering.
The filtered data is them used in for parameter identification.

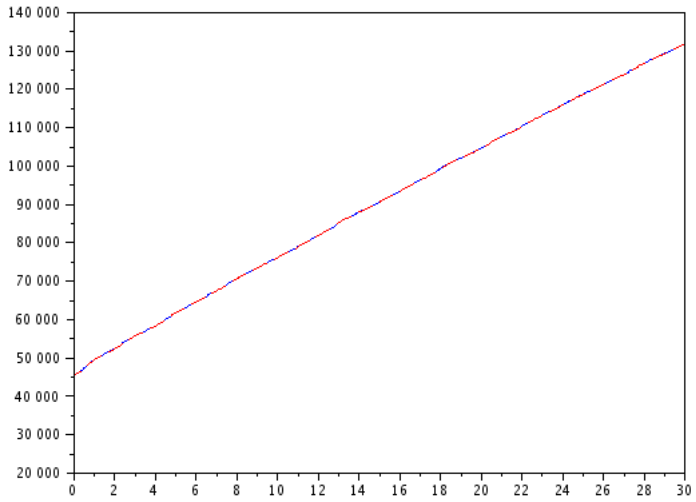

Figure 1: A graph showing filtered tacho count data with power value=10

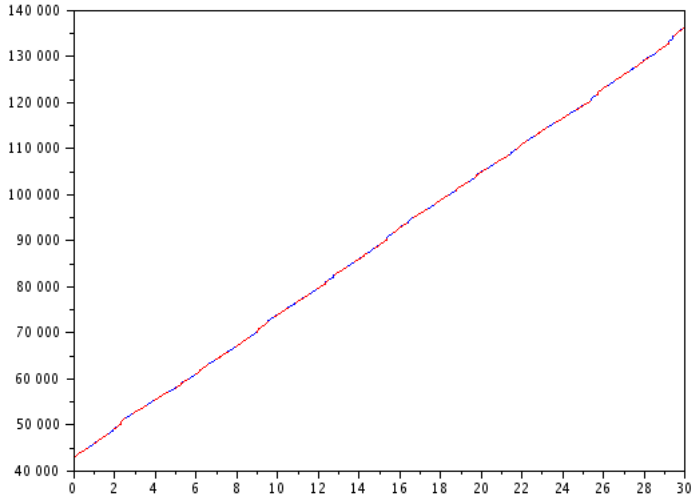

Figure 2: A graph showing filtered tacho count data with power value=100

3 System Identification

A grey box model time domain approach was used to identify the unknown free parameters of the motor. This is because we know beforehand that the motor acts as a second order system whose output given a step input is:

$$\frac{q}{s(\frac{s^2}{\omega_n^2} + \frac{2\xi}{\omega_n}s + 1)} \tag{2}$$

The parameters to be identified include:

- The natural frequency, ω_n
- The damping factor, ξ

These parameters can be calculated from the estimates of the settling time and the overshoot.

3.1 Overshoot

The overshoot is the maximum value reached by the time evolution of the output with respect to the steady state calue. It is calculated as follows:

$$O = \frac{|\chi_{max} - \chi(\infty)|}{|\chi(0) - \chi(\infty)|} \tag{3}$$

Where:

- $\chi(0)$ is the initial value
- χ_{max} is the maximum value reached
- $\chi(\infty)$ is the steady state value

Overshoots for the different power values were estimated and plotted as below.

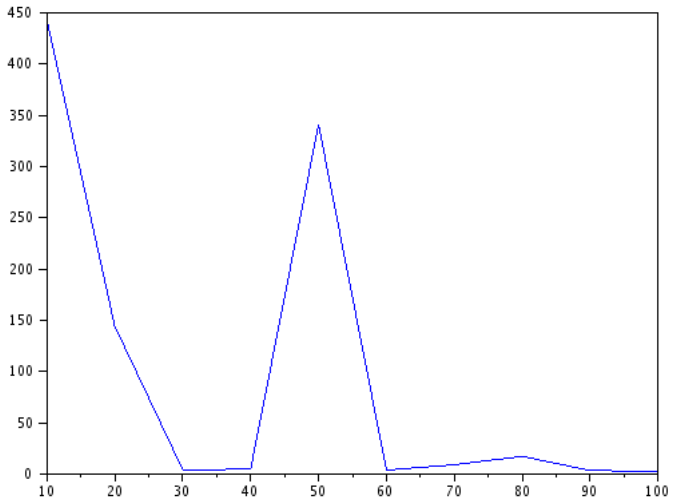

Figure 3: A graph showing the changes in the overshoot(y-axis) with power(x-axis)

3.2 Settling Time, T_s

The settling time, T_s is assumed to be the time the output reaches within a given α value of the steady state value, $\chi(\infty)$ and forever stays there.
$\alpha = 0.01$ was used. It was calculated as,

$$st\_est = t(index) - StepValue/2; \tag{4}$$

where,

- $t(index)$ is the time when the output first reaches the interval.
- $StepValue$ is the interval between two consecutive outputs.

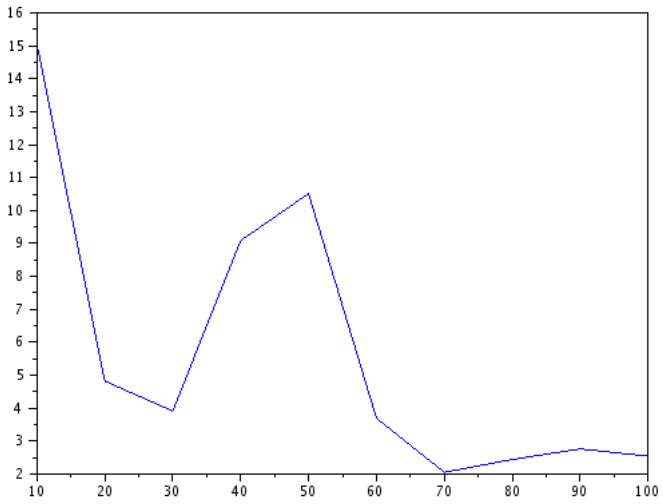

Figure 4: A graph showing the changes in the settling time estimates(y-axis) with power(x-axis)

3.3 Damping Factor, ξ

The damping factor, ξ estimate is calculated as:

$$\xi = \sqrt{\frac{\log(ov)^2}{\pi + \log(ov)^2}} \tag{5}$$

Where ov is the overshoot.

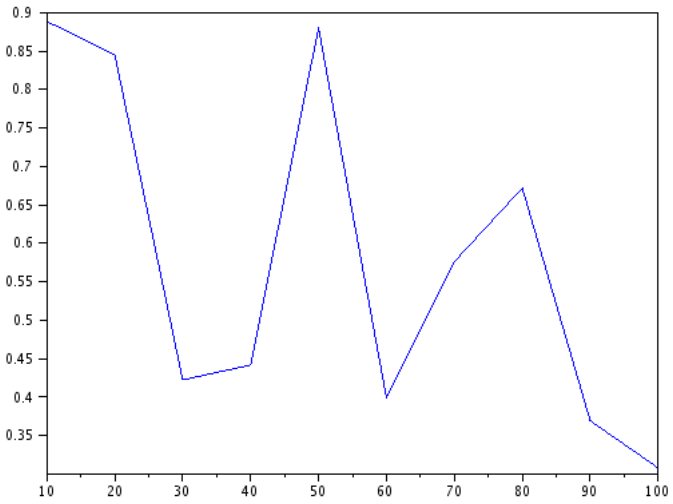

Figure 5: Changes in the ξ estimate value(y-axis) with power(x-axis)

3.4 Natural Frequency, ω_n

The natural frequency, ω_n is calculated as,

$$\omega_n = \frac{\log(\alpha) - \log(nb)}{-\xi st} \tag{6}$$

where,

- $$nb = \frac{1}{\sqrt{1-\xi^2}} \tag{7}$$
- st is the settling time estimate.

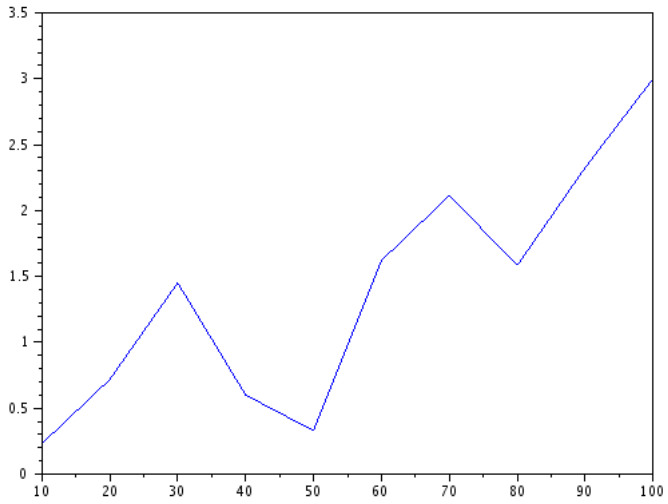

Figure 6: Changes in the ω_n estimate value(y-axis) with power(x-axis)

An estimate of ω, ξ settling time and the overshoot for each power value is presented below.

Power	ξ	ω_n	set time(s)	Overshoot
10	0.888824	0.230329	15.055	442.951331
20	0.845369	0.718232	4.825	144.275678
30	0.422576	1.451258	3.915	4.326551
40	0.441914	0.600584	9.085	4.70033
50	0.88039	0.329319	10.515	341.038526
60	0.399064	1.620381	3.695	3.924681
70	0.576415	2.114344	2.055	9.171027
80	0.672154	1.583978	2.445	17.321917
90	0.369445	2.325918	2.765	3.46885
100	0.308024	3.000851	2.545	2.765275

Figure 7: A table showing the values for ξ, ω_n, settling time and overshoot for the different power values

3.5 Parameter Approximation

A simple average is chosen to compute the values of ξ and ω_n and they are therefore computed as follows:

$$\xi = \frac{1}{N}\sum_{i=1}^{N}\xi_{i,est} \tag{8}$$

$$\omega_n = \frac{1}{N} \sum_{i=1}^{N} \omega_{i,est} \tag{9}$$

The following values were arrived at:

- $\xi = 0.5804175$
- $\omega_n = 1.3975257$

4 Model Verification

In order to test our model, a set of verification data **disjoint** from the training data are used. Three different sets of power values is used to test the model against the following performance indices:

- Integral squared error (ISE) = $\sum_{t=0}^{N}(y_{model}(t) - y_{est}(t))^2$
- Integral absolute error (IAE) = $\sum_{t=0}^{N}|y_{model}(t) - y_{est}(t)|$
- Integral time squared error (ITSE) = $\sum_{t=0}^{N} t(y_{model}(t) - y_{est}(t))^2$
- Integral time absolute error (ITAE) = $\sum_{t=0}^{N} t|y_{model}(t) - y_{est}(t)|$

Where y_{est} is the signal used to in verification and y_{model} is the signal generated by our model

The output, y_{model} is generated using the estimated model parameters. Its time response is given by:

$$y_{model}(t) = [q + 2N\mathrm{e}^{-\xi\omega_n t}\cos(\omega_n\sqrt{1-\xi^2}t + \phi)]\mathbf{1}(t) \tag{10}$$

With $N = \frac{q}{2\sqrt{1-\xi^2}}$ and $\phi = \arctan(\frac{\xi}{-\sqrt{1-xi^2}})$.
q is the final value of $y_{est}(t)$ and is given by:

$$\lim_{t\to+\infty} y_{est}(t) = q \tag{11}$$

Which is the estimated steady state value of $y_{est}(t)$.

The table below summarizes performance indices for the 3 sets of verification data used.

Power	ISE	IAE	ITSE	ITAE
30	81089.78	21302.75	1203.23	1458.55
80	42799.34	12731.58	4023.19	6826.76
100	54420.04	13518.735	3664.89	678.25

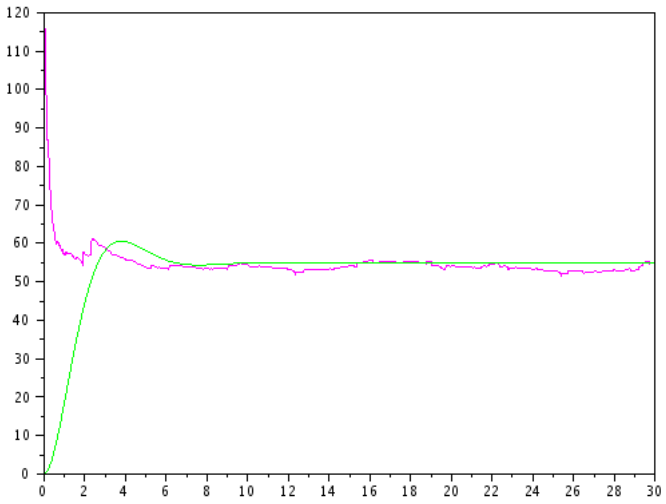

Figure 8: A graph showing the signals $y_{model}(green)$ and y_{est} for the power value=100

5 System Design

A APPENDIX

A.1 Scilab Scripts

A.1.1 Model Parameters estimation

This script automates the analysis task for each data set. It passes all the data files listed in *dataFiles* to another script, *analysis.sce* which does the actual analysis and finally saves the parameter estimates for each data set in *parameters.dat*

```
clear;
// global variables
global fname; // used by analysis.sce
global imgname;
global StepValue;
global DATA_PATH;
DATA_PATH = '../data/';
global IMAGES_PATH;
IMAGES_PATH = '../images/';

global CURRENT_POWER; //Current power being analysed

//parameters
global st_est; //settling time
global omega_est; //omega_est
global OverShoot; //overshoot
global xi_est;
global omega_n;
global xi_n;

//list of data files
dataFiles = ['10.dat','20.dat','30.dat','40.dat','50.dat','60.dat','70.dat','80

//get size of dataFiles(Failed to get a better way than this :-))
_ff = size(dataFiles);
fsize =_ff(2);


Mfd = mopen('../data/params.dat', 'w');
if Mfd == -1 then
    error("Failed to open file for reading");
end

//"pass" the files to analysis.sce one at a time
for i=1:fsize
    fname = DATA_PATH+dataFiles(i);
    imgname= IMAGES_PATH+dataFiles(i);
    exec('analysis.sce', -1);
    mfprintf(Mfd,"%d %f %f %f %f\n",CURRENT_POWER, xi_est, omega_est, st_est, OverS
end
```

```
mclose(Mfd);

//Plot the power values
fd =mopen('../data/params.dat','r');
if fd == -1 then
    error("Failed to open file for reading");
end

//read values from file
pData= [];//zeros(fsize,5);
isEOF = 0;
pos = 1;
while isEOF <> 1
    [n, pwr, xi_e, om, st, ov ] = mfscanf(fd,"%d %f %f %f %f");
    if n > 0 then
        pData(pos,1)=pwr;
        pData(pos,4)=st;
        pData(pos,3)=om;
        pData(pos,5)=ov;
        pData(pos,2)=xi_e;
        pos = pos + 1;
    else
        isEOF = 1;
    end
end

//Compute the Least Square Squares Approximation for omega and xi
omega_n = sum(pData(:,3))/fsize;
xi_n = sum(pData(:,2))/fsize;


//plot power vs settling time
hf = scf(3);
clf(hf);
plot(pData(:,1),pData(:,4),'b');
xs2png(hf, IMAGES_PATH+'_power_vs_st.png');


//plot power vs xi
hf = scf(3);
clf(hf);
plot(pData(:,1),pData(:,2),'b');
xs2png(hf, IMAGES_PATH+'_power_vs_xi.png');


//plot power vs OverShoot
hf = scf(3);
clf(hf);
plot(pData(:,1),pData(:,5),'b');
xs2png(hf, IMAGES_PATH+'_power_vs_overshoot.png');
```

```
//plot power vs omega
hf = scf(3);
clf(hf);
plot(pData(:,1),pData(:,3),'b');
xs2png(hf, IMAGES_PATH+'_power_vs_omega.png');
```

A.1.2 Model Verification

The following script is used to calculate the performance indices.

```
//used to verify the model params
global omega_n;
global xi_n;
global q_n;
global g_n;
global fname;
global imgname;
global CURRENT_POWER; //Current power being analysed

//Get the verification signal

//Open and read file
fd = mopen(fname,'r');
if fd == -1 then
    error("Failed to open file for reading");
end

//read input file containing the data
//format <rev_count, timestamp, power_value>
isEOF = 0;
pos = 1;
i = 0;
mdata = []; //declare an empty matrix
while isEOF <> 1
    [n, rev, ts, pv ] = mfscanf(fd,"%f %f %d");
    if n > 0 then
        mdata(pos,1) = rev;
        mdata(pos,2) = i; //change i to ts to get the real timestamp from the l
        mdata(pos,3) = pv;
        pos = pos + 1;
        i = i + StepValue;
    else
        isEOF = 1;
    end
end
mclose(fd);

//Sanitize the timestamp to start from 0
CURRENT_POWER=mdata(1,3);
```

```
if mdata(1,2) <> 0 then
    ts = mdata(1,2);
    for i=1:length(mdata(:,2))
        mdata(i,2) = mdata(i,2) - ts;
    end
end
//Average values with the same timestamp
avg = [];
t=[];
index = 1;
i = 1;
while i < length(mdata(:,1)) - 1
    tv = mdata(i,1);
    ts = mdata(i,2);
    j = i+1;
    while mdata(j,2) == ts
        tv = tv + mdata(j,1);
        j = j+1;
    end
    tv = tv /(j-i);
    avg(index)=tv;
    t(index)=ts;
    index = index+1;
    i = j;
end

//plot the data
hf = scf(1);
clf(hf,'clear');
plot(t,avg,'b');
xs2png(hf, imgname+'_original.png');


//Apply LowPassFilter
fdata=ExponentialFilter(LpAlpha,avg);

//plot filtered data
 hf = scf(1);
 plot(t,fdata,'r--');
 xs2png(hf, imgname+'_orig+filtered.png');

//Tachometer estimation
y = zeros(1,length(t));
MaxCount = round(length(t)/3);
//MaxCount = 50;
for i=2:length(fdata)
    if i <= MaxCount
        DeltaT = t(i) - t(1);
        y(i-1) = (fdata(i) - fdata(1))/DeltaT*%pi/180;
    else
```

```
        DeltaT = t(i) - t(i-MaxCount);
        y(i-1) = (fdata(i) - fdata(i-MaxCount))/DeltaT*%pi/180;
    end
end
y($) = y($-1);
hf = scf(2);
clf(hf);
plot(t, y, 'm');

//Estimate the Response using the model parameters
v_est =zeros(1,length(t));
StepTime = 1;
q_est = y($);
N = q_est/(2*sqrt(1 - xi_n^2));
phi = atan(xi_n, -sqrt(1 - xi_n^2));
y_est = q_est + 2*N*exp(-xi_n*omega_n*t).*cos(omega_n*sqrt(1 - xi_n^2)*t + phi)
v_est = y_est(1:$)';
hf = scf(2);
plot(t, y_est, 'g');
xs2png(hf, imgname+'_performance.png');
//calculate the perfomance indices
ISE = sum((v_est - y)^2)
IAE = sum(abs(v_est - y))
ITSE = sum(t'.*(v_est - y)^2)
ITAE = sum(t'.*abs(v_est - y))
```

A.2 Data Collection Code

A.2.1 Data Packet Formats

The following code snippets shows the structures used for exchanging the data between the `Lego Mindstorms NXT` and both the **server** and **controller** running on the PC.

```
#ifndef __BT_PACKET_H__
#define __BT_PACKET_H__
#include<stdint.h>

#define MAX_REQ                 1 /*Maximum Number of requests send to/from the
                                    concurrent requests/responses send/received*
#define MAX_NUM_DATA            2 /*Maximum number of data transferred to/from

/**
 * Index Definition for data array in bt_req_t
 * Note only for GET_MOTOR_COUNT operation are both
 * are both indices used
 */
#define VALUE_INDEX             0
#define TIMESTAMP_INDEX         1
```

```
/****PORT Definitions****/
#define PORT_1                  1
#define PORT_2                  2
#define PORT_3                  3
#define PORT_4                  4

#define MOTOR_A                 5
#define MOTOR_B                 6
#define MOTOR_C                 7

/******PORT Operations*****/
#define BT_CLOSE_CONNECTION     102 /*Close Bluetooth Connection > Should not b
#define SET_MOTOR_POWER         1 /*Set Motor power*/
#define GET_MOTOR_POWER         2 /*Get the number of "rotations" of the motor*



/*
 * Defines Bluetooth requests for operations
 * to be performed by the Lego Brick
 */
 typedef struct {
    uint8_t operation;/*Operation to be performed*/
    uint8_t port; /**/
    float data[MAX_NUM_DATA];

} __attribute__((packed)) bt_req_t;


/*
 * Defines a Bluetooth packet send to the to/from the Lego brick
 */

typedef struct {
    uint32_t size; /*size of the packet (Always equal to MAX_REQ */
    bt_req_t packets[MAX_REQ];
} __attribute__((packed)) bt_packet_t;

#endif
```