Технически университет - София
Филиал Пловдив

Дипломна работа

Тема: Проектиране на управление за робот манипулатор с пет степени на свобода

Студент: Станислав Георгиев

Специалност: АИУТ

Образователна степен: ОКС Бакалавър

Съдържание

Увод

Целта на дипломната работа е да се реализира управление на 5 звенен робот с помощта на ROS (Robot operating system). За целта ще използваме показания на фигурата робот. След изпълнение на дипломната работа този робот трябва да приема произволна поза и да я изпълнява.

За показване на самата реализация сме разделили дипломната работа в пет глави като 4 и 5 глава са свързани с физическия робот и читателите които се интересуват само от симулацията могат да ги пропуснат.

В първа глава се разглеждат теоретически постановки на кинематиката, като представяне на позиция и ориентация, хомогенни трансформации и т.н. Отново читателите коiто са запознати с основната теория на кинематиката могат да пропуснат тази глава. Не е препоръчително да се пропуска 1.3.2. тъй като там сме показали как се моделира робота при ROS. За да можем да използваме останалите функции на ROS е нужно да създадем модел на нашия робот. Моделирането става лесно с помощта на URDF файл както ще видим в 1.3.2. За целта трябва да сме инсталирали ROS и Moveit както е показано в Приложение 1.

След като имаме готов модел на нашия робот можем да преминем към глава 2, където ще покажем как да конфигурираме Moveit. Moveit е ROS пакет който след като го конфигурираме ще ни предостави интерфейс с помощта на който можем да управляваме робота. За целта в 2.2. сме показали как от направения в 1.3.2. модел конфигурираме и генерираме пакет който по нататък ще използваме. След това в 2.3. сме използвали python интерфейса на Moveit за да напишем програма която да изчислява правата и обратната кинематика.

В глава 3 след изпълнение на глава 1 и 2 можем да използваме python интерфейса на Moveit за да изпълним най-различни задачи. За целта на демонстрацията ще напишем програма която на всеки 20 секунди ще генерира произволна поза и ще я изпълнява , но за сега само на симулация. За да може лесно да се смени заданието, сме разделили изпълнението на две програми една за изчисление и изпълнение на позата и друга за задаване на позата. Така лесно след това можем да пренапишем програмата за задаване на заданието. В тази глава ще намерите описание как се стартира програмата и как се използва python интерфейса за писане на програми за управление на робота. Но засега само на симулация, реалното изпълнение ще стане след глава 4 и 5.

В глава 4 ще видим как да конфигурираме Raspberry Pi който след изпълнение на програмите в глава 3 да изпрати изчислените ъгли към драйвера. За целта в 4.2. сме написали програма която взима тези ъгли и ги изпраща по SPI в обхвата [-90°; 90°] цели числа. След което в 4.3. сме показали как можем да стартираме реалното изпълнение на робота паралелно със симулацията. В приложение 2 сме показали как да инсталираме Ubuntu на Raspberry Pi и как да използваме SSH.

В глава 5 след като вече сме изчислили и изпратили заданието ни трябва драйвер който да управлява моторите. За целта може да се използва Arduino но ние сме избрали да реализираме специализиран драйвер за целта. Тук ще видим как се реализира хардуерно и софтуерно вградена система която приема ъглите по SPI и ги преобразува в сигналите необходими за управление на моторите.

<u>Необходим условия и знания за читателя са:</u>

- Базови познания за Linux и ROS , като използване и писане на команди под Linux, познание за основните концепции и идеи в ROS.
- Основи на програмирането. За реализиране на дипломната работа са използвани два програмни езика C и Python.
- Електроника.

Глава 1 Кинематика на робота. Представяне на кинематиката на робота в ROS.

Кинематиката се занимава с изучаване на движението на телата без да се интересува от силите породили това движение. Тъй като всеки робот е създаден за движение кинематиката е основния аспект при дизайн, анализ, управление и симулация на роботи. Основния фокус на кинематиката е представянето на позиция и ориентация и изчисляване на правата и обратна задача на кинематиката.

Тази глава ще представи най-често използваните представяния на позиция и ориентация на едно тяло, кинематиката на често срещаните ставни съединения и геометричното им представяне с помощта на параметрите на Денавит-Хартенберг. Ще бъде разгледано представянето на 5-звенен сериен робот в ROS(Robot Operating System). Тези представяния ще бъдат използвани за изчисления на правата и обратна кинематика с помощта на ROS Moveit.

Целта на тази глава е да представи основните означения използвани за описание на един робот. Да запознае читателя с представяне на робот в ROS и изчисляване на правата и обратна кинематика с помощта на ROS Moveit.

1.1. Позиция и ориентация

Минималния брой координати необходими за представянето на едно тяло в Евклидовото пространство са шест. Тук ще се разгледат различни методи за представянето на позиция и ориентация.

Една координатна система се състои от база O_i, и три взаимно перпендикулярни вектора (x_i, y_i, z_i), които са фиксирани към едно тяло. Позата на едно тяло се представя относително към друго тяло, за да може да се изрази като позата на една координатна система отнесена към друга координатна система.

1.1.1. Позиция и отместване

Позицията на базата на една координатна система i, към координатна система j, може да се представи с 3x1 вектор.

$$ {}^{j}p_i = \begin{pmatrix} {}^{j}p_i^x \\ {}^{j}p_i^y \\ {}^{j}p_i^z \end{pmatrix} \qquad (1.1.1) $$

Компонентите на този вектор са Декартовите координати на базата на система i - O_i в система j, които за проекциите на вектора ${}^{j}p_i$, към съответните оси.

Транслацията е преместване при което нито една от точките на тялото не остава в стартовата си позиция и всички прави линии в тялото остават паралелни на стартовите си ориентации. Транслацията на едно тяло може да се представи като комбинация от позицията преди и след преместването. Обратното също важи тоест, позицията на едно тяло може да се представи като транслация на тялото от стартова позиция (базата) към сегашната му позиция. Което показва, че всяко представяне на позиция може да се разгледа като представяне на преместване и обратно.

1.1.2. Ориентация и ротация

Ротацията е преместване при което поне една точка от тялото остава в стартовата си позиция и не всички линии в тялото остават паралелни на началната си ориентация.Както и при позицията и транслацията , така и тук всяко представяне на ориентацията може да се използва за представяне на ротация.

За разлика от позицията ориентацията може да се представи по няколко начина тук ще бъдат разгледани някой от тях.

Ротационни матрици

Ориентацията на координатна система i, спрямо j, може да се представи като се изразят векторите (x_i, y_i, z_i), спрямо (x_j, y_j, z_j). В резултат се получава 3x3 матрица известна под името ротационна матрица. Компонентите на тази матрица означена с ${}^{j}R_{i}$, скаларните произведения на векторите на двете координатни системи.

$$ {}^{j}R_{i} = \begin{pmatrix} x_i.x_j & y_i.x_j & z_i.x_j \\ x_i.y_j & y_i.y_j & z_i.y_j \\ x_i.z_j & y_i.z_j & z_i.z_j \end{pmatrix} \quad (1.1.2) $$

Тъй като базисните вектори (x_i, y_i, z_i) , (x_j, y_j, z_j) са единични вектори скаларното произведение е равно на косинуса между тях.

Елементарна ротация на координатна система спрямо i спрямо z_j оста на ъгъл Θ се изразява с матрицата:

$$ R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad (1.1.3) $$

докато същата ротация около y_j се представя с матрицата:

$$ R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \quad (1.1.4) $$

а ротация около x_j :

$$ R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \quad (1.1.5) $$

Ротационните матрици се комбинират с просто умножение на матрици така, че ориентацията на система i, спрямо система k може да се получи с израза.

$$ {}^{k}R_{i} = {}^{k}R_{j}\,{}^{j}R_{i} \quad (1.1.6) $$

${}^{j}R_{i}$ е ротационна матрица която трансформира вектор представен в координатна система i, към вектор представен в координатна система j. Тази матрица предоставя представяне на ориентацията на система i, спрямо система j както и представяне на ротация от система i към система j .

Ойлерови ъгли

За представяне на ротацията с минимален брой параметри се използват Ойлерови ъгли. С тяхна помощ ориентацията на система i, спрямо система j може да се представи като вектор от три ъгъла $(\alpha, \beta, \gamma)^{\mathrm{T}}$. При този вид представяне всяко следващо завъртане зависи от предишното затова редът на изпълнение на завъртанията трябва да бъде обозначен. Например символа $(\alpha, \beta, \gamma)^{\mathrm{T}}$ може да се използва за означение на Z-Y-X Ойлерови ъгли. Тоест предполагайки, че системи i и j съвпадат α е завъртането по z оста след което се изпълнява β което е завъртане по y оста и последно γ, което е завъртане по x оста.

Съществуват 12 различни комбинации за ротация които са разделени в две групи:

- Правилни Ойлерови ъгли(Proper Euler angles) (z-x-z, x-y-x, y-z-y, z-y-z, x-z-x, y-x-y).
- Ъгли на Тайт-Брайън(Tait-Bryan angles) (x-y-z, y-z-x, z-x-y, x-z-y, z-y-x, y-x-z).

Ъглите на Тайт-Брайън още се наричат ъгли на Кардан или **roll, yaw, pitch angles**. На таблица 1.1.1. са показани дванайсетте различни комбинации и съответните им ротационни матрица.

В таблицата се използват следните означения:

- 1,2,3 означават ъглите (α, β, γ).
- X, Y, Z означават ротациите по съответните оси x,y,z (например X_1- означава ротация по x на ъгъл α.
- s, c означават синус и косинус (например s_1 означава $\sin\alpha$).

Таблица 1.1.1.[2]

Proper Euler angles	Tait-Bryan angles
$X_1Z_2X_3 = \begin{bmatrix} c_2 & -c_3s_2 & s_2s_3 \\ c_1s_2 & c_1c_2c_3 - s_1s_3 & -c_3s_1 - c_1c_2s_3 \\ s_1s_2 & c_1s_3 - c_2c_3s_1 & c_1c_3 - c_2s_1s_3 \end{bmatrix}$	$X_1Z_2Y_3 = \begin{bmatrix} c_2c_3 & -s_2 & c_2s_3 \\ s_1s_3 + c_1c_3s_2 & c_1c_2 & c_1s_2s_3 - c_3s_1 \\ c_3s_1s_2 - c_1s_3 & c_2s_1 & c_1c_3 + s_1s_2s_3 \end{bmatrix}$
$X_1Y_2X_3 = \begin{bmatrix} c_2 & s_2s_3 & c_3s_2 \\ s_1s_2 & c_1c_3 - c_2s_1s_3 & -c_1s_3 - c_2c_3s_1 \\ -c_1s_2 & c_3s_1 + c_1c_2s_3 & c_1c_2c_3 - s_1s_3 \end{bmatrix}$	$X_1Y_2Z_3 = \begin{bmatrix} c_2c_3 & -c_2s_3 & s_2 \\ c_1s_3 + c_3s_1s_2 & c_1c_3 - s_1s_2s_3 & -c_2s_1 \\ s_1s_3 - c_1c_3s_2 & c_3s_1 + c_1s_2s_3 & c_1c_2 \end{bmatrix}$
$Y_1X_2Y_3 = \begin{bmatrix} c_1c_3 - c_2s_1s_3 & s_1s_2 & c_1s_3 + c_2c_3s_1 \\ s_2s_3 & c_2 & -c_3s_2 \\ -c_3s_1 - c_1c_2s_3 & c_1s_2 & c_1c_2c_3 - s_1s_3 \end{bmatrix}$	$Y_1X_2Z_3 = \begin{bmatrix} c_1c_3 + s_1s_2s_3 & c_3s_1s_2 - c_1s_3 & c_2s_1 \\ c_2s_3 & c_2c_3 & -s_2 \\ c_1s_2s_3 - c_3s_1 & c_1c_3s_2 + s_1s_3 & c_1c_2 \end{bmatrix}$
$Y_1Z_2Y_3 = \begin{bmatrix} c_1c_2c_3 - s_1s_3 & -c_1s_2 & c_3s_1 + c_1c_2s_3 \\ c_3s_2 & c_2 & s_2s_3 \\ -c_1s_3 - c_2c_3s_1 & s_1s_2 & c_1c_3 - c_2s_1s_3 \end{bmatrix}$	$Y_1Z_2X_3 = \begin{bmatrix} c_1c_2 & s_1s_3 - c_1c_3s_2 & c_3s_1 + c_1s_2s_3 \\ s_2 & c_2c_3 & -c_2s_3 \\ -c_2s_1 & c_1s_3 + c_3s_1s_2 & c_1c_3 - s_1s_2s_3 \end{bmatrix}$
$Z_1Y_2Z_3 = \begin{bmatrix} c_1c_2c_3 - s_1s_3 & -c_3s_1 - c_1c_2s_3 & c_1s_2 \\ c_1s_3 + c_2c_3s_1 & c_1c_3 - c_2s_1s_3 & s_1s_2 \\ -c_3s_2 & s_2s_3 & c_2 \end{bmatrix}$	$Z_1Y_2X_3 = \begin{bmatrix} c_1c_2 & c_1s_2s_3 - c_3s_1 & s_1s_3 + c_1c_3s_2 \\ c_2s_1 & c_1c_3 + s_1s_2s_3 & c_3s_1s_2 - c_1s_3 \\ -s_2 & c_2s_3 & c_2c_3 \end{bmatrix}$
$Z_1X_2Z_3 = \begin{bmatrix} c_1c_3 - c_2s_1s_3 & -c_1s_3 - c_2c_3s_1 & s_1s_2 \\ c_3s_1 + c_1c_2s_3 & c_1c_2c_3 - s_1s_3 & -c_1s_2 \\ s_2s_3 & c_3s_2 & c_2 \end{bmatrix}$	$Z_1X_2Y_3 = \begin{bmatrix} c_1c_3 - s_1s_2s_3 & -c_2s_1 & c_1s_3 + c_3s_1s_2 \\ c_3s_1 + c_1s_2s_3 & c_1c_2 & s_1s_3 - c_1c_3s_2 \\ -c_2s_3 & s_2 & c_2c_3 \end{bmatrix}$

Кватернион

Кватернионът $\boldsymbol{\epsilon}$ се дефинира като $\boldsymbol{\epsilon} = \epsilon_0 + \epsilon_1 i + \epsilon_2 j + \epsilon_3 k$ (1.1.7).

Удобно е да се дефинира конюгованото на кватерниона като

$$\tilde{\boldsymbol{\epsilon}} = \epsilon_0 - \epsilon_1 i - \epsilon_2 j - \epsilon_3 k \ , \quad (1.1.8)$$

така, че

$$\boldsymbol{\epsilon}\tilde{\boldsymbol{\epsilon}} = \epsilon_0^2 + \epsilon_1^2 + \epsilon_2^2 + \epsilon_3^2 \ . \ (1.1.9)$$

Единичния кватернион се дефинира така, че да изпълнява условието $\boldsymbol{\epsilon}\tilde{\boldsymbol{\epsilon}} = \mathbf{1}$. Често ϵ_0 се нарича скаларна част на кватерниона а $(\epsilon_1, \epsilon_2, \epsilon_3)$ векторна. Единичния кватернион се използва за означаване на ориентацията, като връзката му с ротационната матрица се дава с уравнение (1.1.10).

$${}^{j}R_i = \begin{pmatrix} 1-2(\epsilon_2^2+\epsilon_3^2) & 2(\epsilon_1\epsilon_2-\epsilon_0\epsilon_3) & 2(\epsilon_1\epsilon_3+\epsilon_0\epsilon_2) \\ 2(\epsilon_1\epsilon_2+\epsilon_0\epsilon_3) & 1-2(\epsilon_1^2+\epsilon_3^2) & 2(\epsilon_2\epsilon_3-\epsilon_0\epsilon_1) \\ 2(\epsilon_1\epsilon_3-\epsilon_0\epsilon_2) & 2(\epsilon_2\epsilon_3+\epsilon_0\epsilon_1) & 1-2(\epsilon_1^2+\epsilon_2^2) \end{pmatrix} \quad (1.1.10)$$

1.1.3. Хомогенна трансформация

Досега беше разгледано представяне на позиция и ориентация отделно. С помощта на хомогенни трансформации , векторът на позицията и матрицата на ориентацията могат да се представят в едно означение. Всеки вектор ${}^{i}r$, изразен спрямо координатна система i, може да се изрази относно координатна система j, ако са известни позицията и ориентацията на система i, спрямо система j. Позицията може да се изрази чрез (1.1.1) , а ориентацията чрез (1.1.2). Така за вектора ${}^{j}r$, изразен спрямо координатна система j, се получава:

$${}^{j}r = {}^{j}R_i\,{}^{i}r + {}^{j}p_i \quad (1.1.11)$$

В матрична форма уравнението може да се запише като:

$$\begin{pmatrix} {}^{j}r \\ 1 \end{pmatrix} = \begin{pmatrix} {}^{j}R_i & {}^{j}p_i \\ 0 & 1 \end{pmatrix}\begin{pmatrix} {}^{i}r \\ 1 \end{pmatrix} \quad (1.1.12)$$

където

$${}^{j}T_i = \begin{pmatrix} {}^{j}R_i & {}^{j}p_i \\ 0 & 1 \end{pmatrix} \quad (1.1.13)$$

Матрицата ${}^{j}T_i$ трансформира вектор представен относно координатна система i, към вектор представен относно координатна система j. Обратната матрица ${}^{j}T_i^{-1}$, трансформира вектор от координатна система j към координатна система i.

$$ {}^{j}T_{i}^{-1} = {}^{i}T_{j} = \begin{pmatrix} {}^{j}R_{i}^{T} & -{}^{j}R_{i}^{T}\,{}^{j}p_{i} \\ 0 & 1 \end{pmatrix} \qquad (1.1.1.14) $$

Хомогенната трансформация на просто завъртане около дадена ос понякога се означава с ***Rot,*** така например ротация на ъгъл Θ около z оста е

$$ Rot(z,\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (1.1.15) $$

Подобно, хомогенната трансформация на просто преместване по ос понякога се означава с ***Trans,*** така например преместване на разстояние *d* по x се означава с

$$ Trans(x,d) = \begin{pmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} . \qquad (1.1.16) $$

Хомогенните трансформации са удобни при нужда от компактен запис, но не са ефикасни при изчисления.

Обобщение

До тук бяха разгледани различни представяния на позицията и ориентацията на едно тяло спрямо друго съответно на една координатна система спрямо друга. Първо беше разгледана форма за представяне на позиция съответно транслация ${}^{j}p_{i}$, след това бяха разгледани няколко представяния на ориентация съответно ротация, (ротационни матрици, Ойлерови ъгли, Кватерниони) съществуват и други представяния на ориентацията но тук бяха разгледани само най-често използваните и тези които ще са необходими по-нататък. С помощта на хомогенни трансформации видяхме как позицията и ориентацията могат да бъдат представени в едни запис.

По нататък ще се покаже как могат да бъдат използвани тези означения за да се опише един сериен робот тип ръка с 5 степени на свобода. С помощта на показаните до тук означения роботът ще бъде описан и симулиран в ROS. Но първо ще разгледаме нужните за описанието ставни съединения и тяхната кинематика.

1.2. Кинематика на става

Кинематичното описание на един робот използва много идеализации. Звената които изграждат един робот са приети за идеални твърди тела , формата на които е перфектна. Тези звена се свързват посредством стави който предоставят перфектна връзка между две звена без луфт между тях. Геометрията на повърхностите който се намират във връзка определят степените на свобода между двете звена, или кинематиката на ставата.

Ставата е съединение между две тела което ограничава относителното им движение. Две тела които се намират във връзка създават проста става. Повърхностите на две тела които са в контакт имат възможността да се движат едно спрямо друго, което позволява относително движение между двете тела. Модела на ставата представя движението на едно звено отнесено към движението на друго звено. Движението се изразява чрез променливите на движение на ставата, други елементи от модела на една става включват ротационната матрица, вектора на позицията, степените на свобода.

Съществуват много видове ставни съединения, като ротационни, линейни, сферични, спирални(винтови), цилиндрични и т.н тук ще бъдат разгледани само ротационните и линейните тъй като те ще бъдат нужни по-нататък за представяне на робота, за повече информация относно останалите вижте Springer Handbook of Robotics [1].

1.2.1. Ротационна става(Revolute joint)

Ротационната става позволява ротация на едно тяло спрямо друго. Ротацията може да бъде представена с един ъгъл на завъртане. Ротационната става е става с една степен на свобода. Когато ротацията е около z оста на ъгъл θ ротационната матрица и вектора на преместване са съответно:

$$ {}^{j}R_i(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad {}^{j}p_i = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \quad (1.2.1) $$

1.2.2. Линейна става(Prismatic joint)

Линейната става позволява транслация на едно тяло спрямо друго. Позицията на едно тяло спрямо друго може да се определи посредством дистанцията между две точки на права паралелна на посоката на движение. Тази става също има само една степен на свобода. Когато преместването е по z оста на разстояние d ротационната матрица и вектора на преместване са съответно

$$ {}^{j}R_i(\theta) = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \qquad {}^{j}p_i = \begin{pmatrix} 0 \\ 0 \\ d \end{pmatrix} \quad (1.2.2) $$

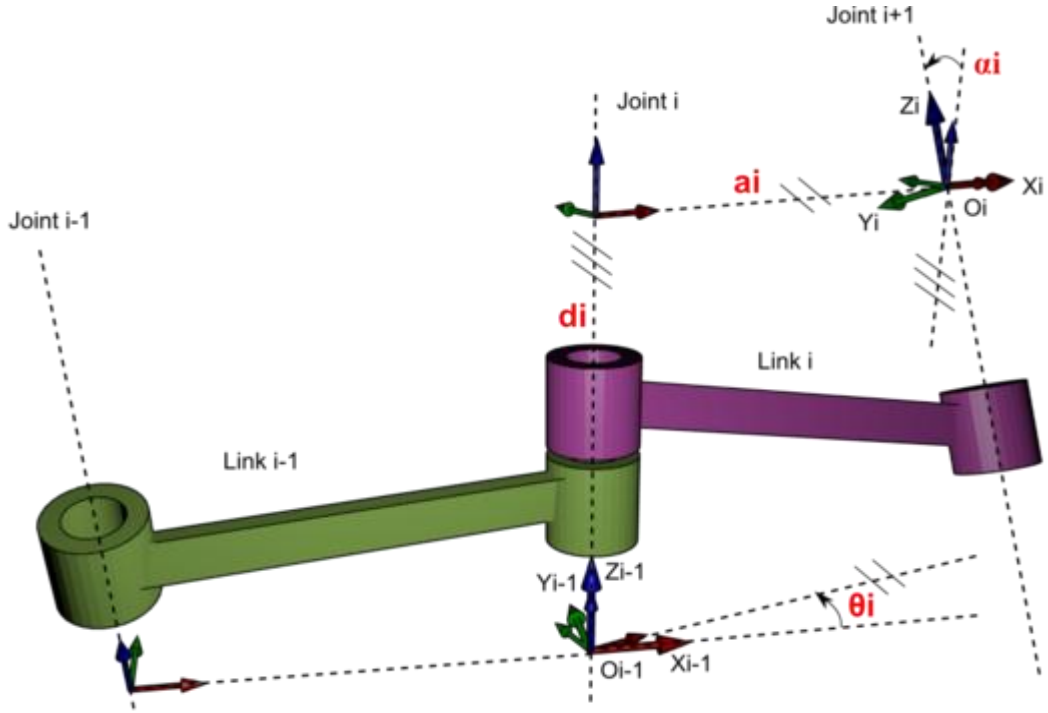
1.3. Геометрично Представяне

Геометрично един робот се представя като към всяко негово звено се закачи координатна система . Всяка координатна система закачена към дадено звено може да се опише спрямо друга координатна система с разгледаните дотук означение.

1.3.1. Параметри на Денавит–Хартенберг

Но най-често се използват означенията на Денавит–Хартенберг. При този вид означения са нужни само 4 параметъра са да се опише една координатна система спрямо друга. Четирите параметъра са два описващи състоянието на звеното, дължината му a_i, и завъртането му α_i, и два параметъра описващи ставата , отместването на ставата d_i, и завъртането и θ_i. Тази икономия се постига при разполагане на осите така, че оста x на едно тяло пресича и е перпендикулярна на оста z на следващото тяло.

Има четири различни начина за разполагане на координатните системи. Всеки има своите предимства и недостатъци. При класическия метод на Денавит–Хартенберг , става i, се разполага между звена i и $i+1$, така че да е извън звено i. Също отместването d_i, и завъртането на θ_i на става i, се измерват относно става $i-1$, поради това индексите на параметрите не съвпада с означенията на ставите.



Фиг. 1.3.1. Четирите параметъра при класическите DH параметри означени в червено $\theta_i, d_i, a_i, \alpha_i$. С помоща на тези параметри може да се премине от координатна система $O_{i-1}X_{i-1}Y_{i-1}Z_{i-1}$ към $O_iX_iY_iZ_i$.[3]

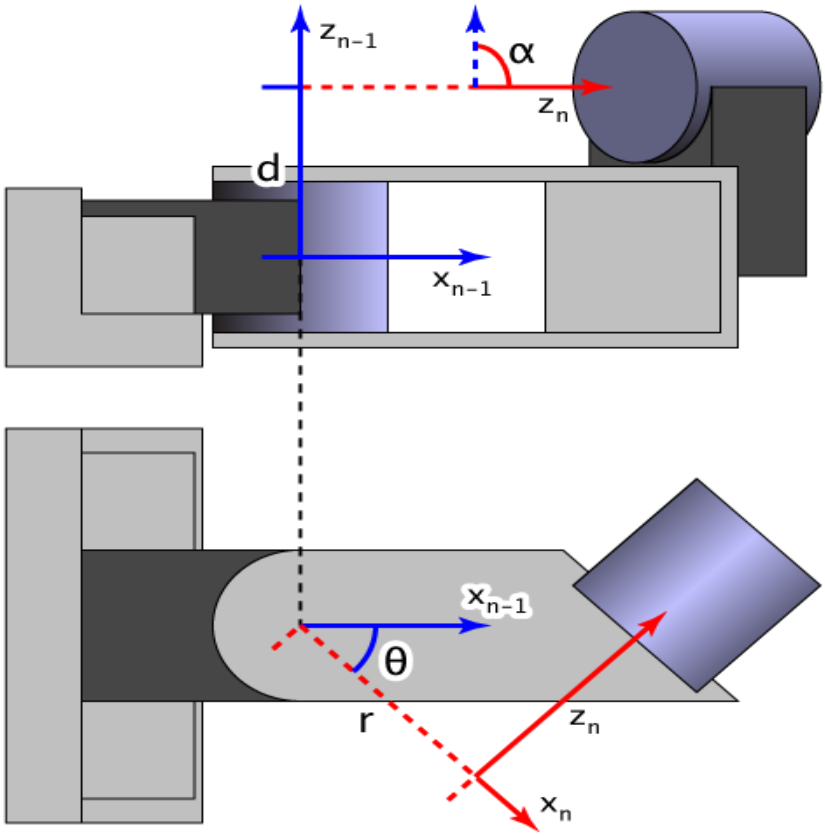
d_i - отместване по оста Z_{i-1} .
θ_i - ъгъл на завъртане по оста Z_{i-1}.
a_i – отместване по оста X_i.
α_i - ъгъл на завъртане по оста X_i.

Привеждането на система i, към координатна система $i-1$ се получава като първо се приложи преместване по оста Z_{i-1}, на разстояние d_i, ротация по оста Z_{i-1} на ъгъл θ_i, транслация по оста X_i, на разстояние a_i и ротация по оста X_i, на ъгъл α_i. Чрез прилагане на индивидуалните трансформации:

$$^{i-1}T_i = Trans_{z_{i-1}}(d_i).Rot_{z_{i-1}}(\theta_i).Trans_{x_i}(a_i).Rot_{x_i}(\alpha_i), \quad (1.3.1)$$

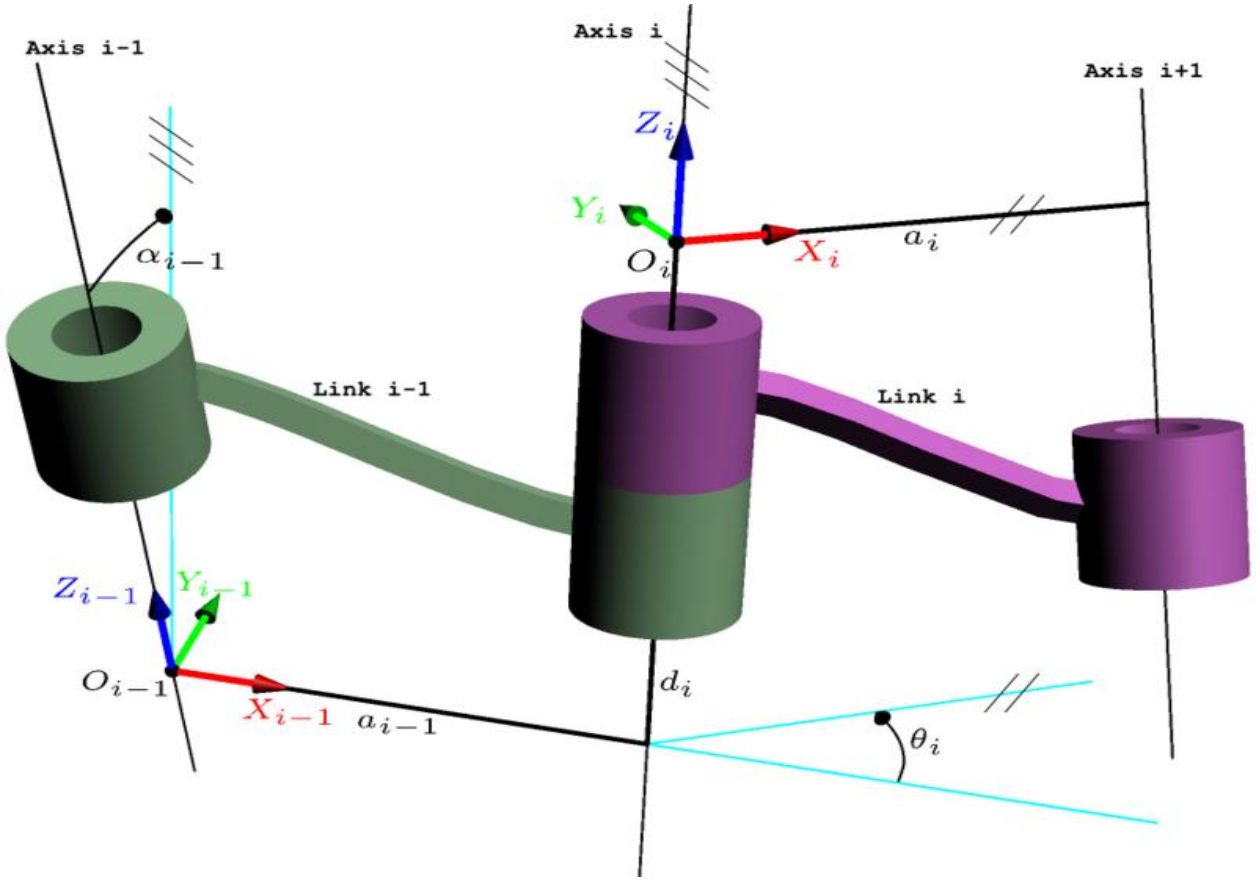
може да се получи еквивалентната хомогенна трансформация (1.3.2).

$$^{i-1}T_i = \begin{pmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_n \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.3.2)$$



Фиг.1.3.2. Илюстрация на параметрите на Денавит-Хартенберг.[3]

Често се използват и модифицираните Д-Х параметри. Разликата с класическите е разположението на координатните системи. В сравнение с класическите системата O_{i-1} е поставена на оста $i-1$, а не на оста i, както може да се види при сравнение на фиг.1.3.1. с фиг.1.3.3.



Фиг.1.3.3. Модифицирани Д-Х параметри.[3]

При използване на тези означения координатна система i, може да се приведе към координатна система $i-1$, като първо се приложи ротация около оста X_{i-1}, на ъгъл α_{i-1}, транслация по оста X_{i-1} на разстояние a_{i-1}, ротация около Z_i, на ъгъл θ_i и транслация по оста Z_i, на разстояние d_i. Чрез прилагане на индивидуалните трансформации:

$$^{i-1}T_i = Rot_{x_{i-1}}(\alpha_{i-1}).Trans_{x_{i-1}}(a_{i-1}).Rot_{z_i}(\theta_i).Trans_{z_i}(d_i), \quad (1.3.3)$$

може да се получи еквивалентната хомогенна трансформация (2.4).

$$^{i-1}T_i = \begin{pmatrix} \cos\theta_i & -\sin\theta_i & 0 & a_{i-1} \\ \sin\theta_i \cos\alpha_{i-1} & \cos\theta_i \cos\alpha_{i-1} & -\sin\alpha_{i-1} & -d_i \sin\alpha_{i-1} \\ \sin\theta_i \sin\alpha_{i-1} & \cos\theta_i \sin\alpha_{i-1} & \cos\alpha_{i-1} & -d_i \cos\alpha_{i-1} \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.3.4)$$

Симулация на 5 звенен робот в Матлаб

За да покажем връзката между Д-Х параметрите и хомогенните трансформации ще използваме Robotics Toolbox да моделираме примерен 5-звенен робот с помощта на който ще разгледаме отделните хомогенни трансформации на някой звена.

Този модел не е реален и няма връзка с реалния модел на робота който ще използваме и е описан в 1.3.2 . Той ще послужи за по-добро описание на параметрите на Денавит Хартенберг .

Robotics Toolbox е свободен за ползване и е разработен от Питър Корк. Пакета може да бъде изтеглен от http://petercorke.com/wordpress/toolboxes/robotics-toolbox. На сайта могат да се намерят и подробни инструкции за използване на пакета. След като се инсталира пакета както е обяснено в сайта може да се конструира симулация на робот.

За пример ще бъде зададен робот с помощта на Денавит-Хартенберг параметрите и ще бъдат разгледани някой от функциите които могат да се използват.

С помощта на командата *Link* се създава едно звено от робота параметрите и подробно описание на функцията може да се намери в помощната страница на Матлаб(*help Link)*. С командата *SerialLink* се създава серийно свързан робот тип ръка. Приема аргументи от тип масив от *Link,* които описват робота. Метода *robot.plot(q)* изкарва прозореца показан на фиг. 1.3.4. С матрицата *q* се задават ъглите на ставите θ_i.

```
L(1) = Link('d', 0, 'a', 0, 'alpha', pi/2, 'offset', pi, 'qlim', [-pi/2, pi/2]);
L(2) = Link('d', 0, 'a', 0.2, 'alpha', 0, 'offset', pi/2, 'qlim', [-pi/2, pi/2]);
L(3) = Link('d', 0, 'a', 0.2, 'alpha', 0, 'qlim', [-pi/2, pi/2]);
L(4) = Link('d', 0, 'a', 0, 'alpha', pi/2, 'offset', pi/2, 'qlim', [-pi/2, pi/2]);
L(5) = Link('d', 0.15, 'a', 0, 'alpha', 0, 'qlim', [-pi/2, pi/2]);
q = [pi/4 pi/4 pi/4 pi/4 pi/4];
robot = SerialLink(L);
robot.plot(q)
```

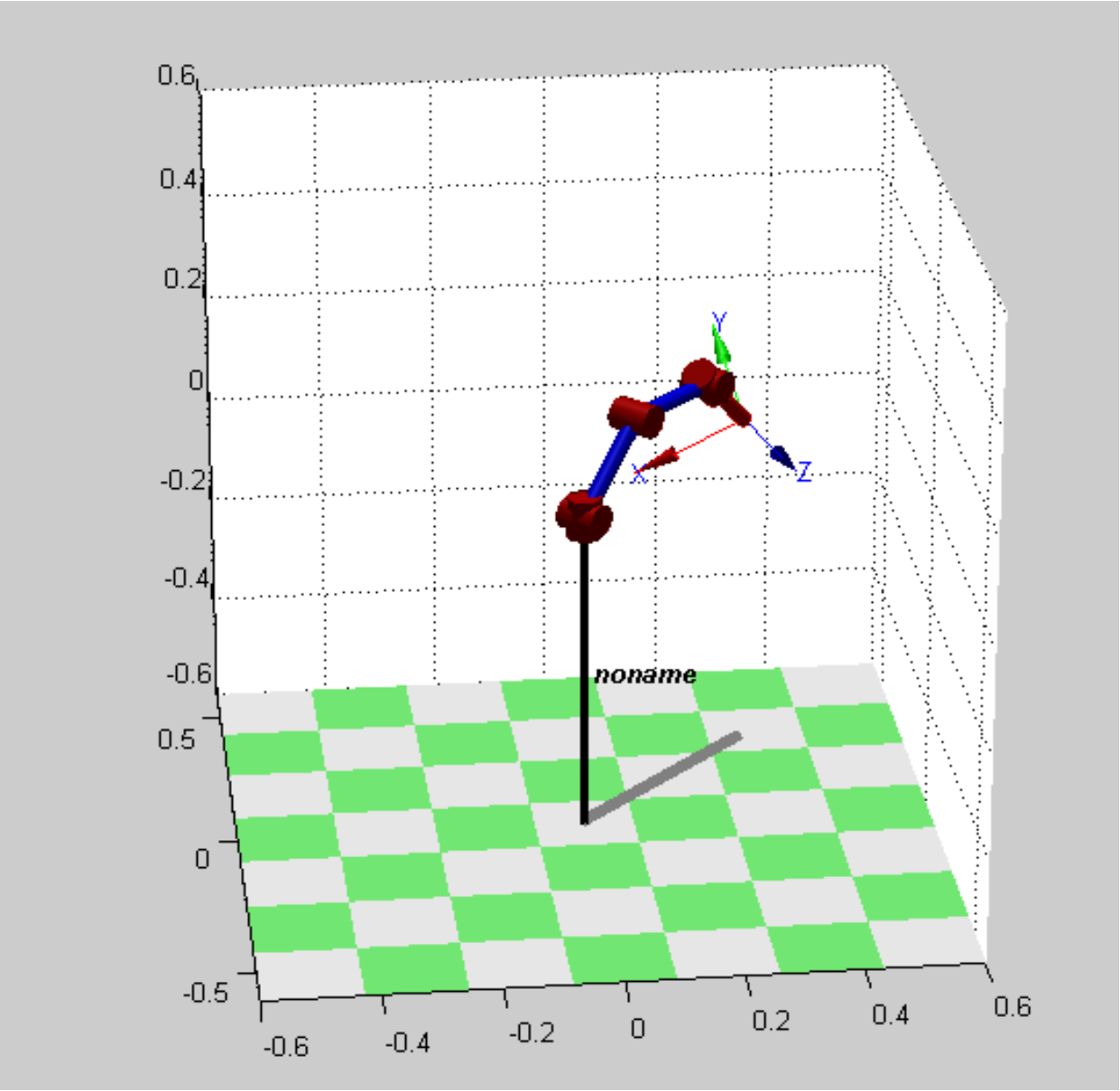
С помощта на метода *A* (*robot.A(i,q)*), може да се види хомогенната трансформация на отделните звена например за звено 1 се използва *robot.A[1,q]*. Например за звено 2 спрямо звено 1 хомогенната трансформация е

$$ {}^{1}T_{2} = \begin{matrix} -0.7071 & -0.7071 & 0 & -0.1414 \\ 0.7071 & -0.7071 & 0 & 0.1414 \\ 0 & 0 & 1.0000 & 0 \\ 0 & 0 & 0 & 1.0000 \end{matrix} \quad (1.3.5) $$

За да се види хомогенната трансформация на дадено звено не спрямо предходното му трябва да се изредят всички звена от желаното начално до крайното желано. Например за

хомогенната трансформация на звено 5 спрямо 2 се използва *robot.A([2 3 4 5],q)* в резултат на което се получава

$$ {}^{1}T_{5} = \begin{matrix} 0.5000 & -0.5000 & -0.7071 & -0.4475 \\ -0.5000 & 0.5000 & -0.7071 & 0.0354 \\ 0.7071 & 0.7071 & 0.0000 & 0 \\ 0 & 0 & 0 & 1.0000 \end{matrix} \quad (1.3.6) $$



Фиг.1.3.4. Прозорец след извикване на функцията `robot.plot(q)`.

Хомогенната трансформация на звено 1 *robot.A([1],q)* спрямо основната координатна система е:

$$ {}^{0}T_{1} = \begin{matrix} -0.7071 & 0.0000 & -0.7071 & 0 \\ -0.7071 & 0.0000 & 0.7071 & 0 \\ 0 & 1.0000 & 0.0000 & 0 \\ 0 & 0 & 0 & 1.0000 \end{matrix} . \quad (1.3.7) $$

От уравнения (1.3.5) и (1.3.7) ясно може да се види връзката между параметрите на Денавит–Хартенберг и хомогенните трансформации (1.3.2).

За уравнение (1.3.7) тъй като:

$d_1 = 0$ - елемент 3x4 от матрица (1.3.7) е равен на 0,

$a_1 = 0$ - елементи 1x4 и 2x4 са нули,

$\alpha_1 = pi/2$ - елемент 3x2 е 1 а елемент 3x3 е 0,

$\theta_1 = \frac{pi}{4} + pi$, - pi идва от факта че за звено 1 сме добавили отместване на $\theta_1 = pi$ (`'offset', pi`) при дефинирането му, затова елемент 1x1 е равен на -0.7071 и елемент 2x1 също е равен на -0.7071.

$$ {}^{1}T_{2} = \begin{matrix} -0.7071 & -0.7071 & 0 & -0.1414 \\ 0.7071 & -0.7071 & 0 & 0.1414 \\ 0 & 0 & 1.0000 & 0 \\ 0 & 0 & 0 & 1.0000 \end{matrix} \quad (1.3.5) $$

За уравнение (1.3.5),(**тук звено 2 е представено спрямо звено 1**) тъй като

$d_2 = 0$ - елемент 3x4 от матрица (1.3.5) е равен на 0,

$a_2 = 0.2$ и $\theta_2 = \frac{pi}{4} + \frac{pi}{2}$ (`'offset', pi/2`) - елемент 1x4 е равен на -0.1414 а 2x4 е 0.1414,

$\alpha_2 = 0$ елемент 3x2 е 0 а елемент 3x3 е 1,

$\theta_2 = \frac{pi}{4} + \frac{pi}{2}$, - pi/2 идва от факта че за звено 2 сме добавили добавили отместване на $\theta_2 = pi/2$ (`'offset', pi`) при дефинирането му, затова елемент 1x1 е равен на -0.7071 а елемент 2x1 е равен на 0.7071.

От тук се вижда, че в този toolbox се използват класическите параметри на Денавит-Хартенберг.

1.3.2. *Моделиране на 5 звенен робот в ROS*

За да се представи визуално един робот в ROS се използва URDF, той представлява XML език с помощта на който може да се моделира робота. След малко ще разгледаме основните конструкции който се използват за изграждане на робота. За примери ще бъдат използвани части от URDF модела на робота който ще използваме в останалите глави, целия URDF може да бъде намерен на посочения линк.

https://github.com/st1na/robot_arm/blob/master/software/ros_packages/robot_urdf/urdf/robot_arm_hand.urdf

За тази глава е необходим пакета *robot_urdf*, който може да бъде намерен на посочения долу линк .

https://github.com/st1na/robot_arm/tree/master/software/ros_packages/robot_urdf.

За да бъде изграден пакета с модела на робота трябва да се създаде работна директория(workspace) името на която е произволно, но тук ще я наречем *robot_urdf_ws*, в работната директория създаваме директория *src*, и се местим в нея. След като вече се намираме в директория *src* на работната директория , стартираме следните команди.

```
git clone https://github.com/st1na/robot_arm_ros_packages
mv robot_arm_ros_packages/* . & sudo rm -r robot_arm_ros_packages
sudo rm -r robot_moveit_config code_samples
```

След като вече сме изтеглили пакетите трябва да инициализираме работната директория това става с командата **catkin_init_workspace** . След това се местим в работната директори (*robot_urdf_ws*) и стартираме командата **catkin_make** . След като **catkin_make** приключи използваме командата **source devel/setup.bash** . След като вече сме конфигурирали работната директория можем да стартираме симулацията и да разгледаме модела на робота. За да стартираме модела трябва да се намираме в работната директория/*src/robot_urdf(robot_urdf_ws/src/robot_urdf)* и да стартираме командата.

```
roslaunch urdf_tutorial display.launch model:=urdf/robot_arm_hand.urdf
```

С цел изясняване на начина по който се моделира с помощта на URDF ще бъдат разгледани три звена свързани с две стави една ротационна и една фиксирана. Примера може да бъде изтеглен от посочения линк.

https://github.com/st1na/robot_arm/blob/master/software/ros_packages/robot_urdf/urdf/urdf_example.urdf

urdf_example.urdf

```
<?xml version="1.0"?>
<robot name="robot">

	<material name="red">
		<color rgba="0.8 0 0 1"/>
	</material>

	<material name="blue">
		<color rgba="0 0 0.8 1"/>
	</material>

	<material name="white">
		<color rgba="1 1 1 1"/>
	</material>

	<material name="black">
		<color rgba="0 0 0 1"/>
	</material>

	<link name="base_link">
		<visual>
			<geometry>
				<box size="0.58 0.46 0.25"/>
			</geometry>
			<origin rpy="0 0 0" xyz="0 0 0"/>
			<material name="red"/>
		</visual>
	</link>

	<link name="link1">
		<visual>
			<geometry>
				<cylinder radius="0.045" length="0.05"/>
			</geometry>
			<material name="white"/>
			<origin rpy="0 0 0" xyz="0 0 -0.025"/>
		</visual>
	</link>

	<joint name="joint1" type="revolute">
		<parent link="base_link"/>
		<child link="link1"/>
		<limit effort="1000.0" lower="0" upper="3.1415" velocity="0.5"/>
		<origin rpy="1.5707 0 0" xyz="-0.1 0.23 0"/>
		<axis xyz="0 0 1"/>
	</joint>

	<link name="link2">
		<visual>
			<geometry>
				<cylinder radius="0.1" length="0.02"/>
			</geometry>
			<material name="white"/>
			<origin rpy="0 0 0" xyz="0 0 -0.01"/>
		</visual>
	</link>
```

```
    <joint name="joint2" type="fixed">
        <parent link="link1"/>
        <child link="link2"/>
        <origin rpy="0 0 0" xyz="0 0 -0.05"/>
    </joint>

</robot>
```

За да се разгледа модела представен в горния пример, след като е създаден пакета, трябва да сме в работната директория/*src/robot_urdf* и да стартираме командата:

roslaunch urdf_tutorial display.launch model:=urdf/urdf_example.urdf

След което ще видим прозорец с 3D модел създаден в горния пример.Тъй като вече виждаме модела можем по-ясно да разгледаме горния пример.

Звено се създава като му зададем име `<link name="base_link">`, визуални параметри като форма и разположение – като разположението на първото звено се взима спрямо глобалната координатна система. Това са задължителните параметри за дефиниране на едно звено тоест за да се дефинира едно звено то задължително трябва да има форма и разположение, цвета не е задължителен параметър като по подразбиране е червен.

В случая на първото звено формата(geometry) е от тип `<box size="0.58 0.46 0.25"/>`, кутия с размери 0.58 по X 0.46 по Y и 0.25 по Z. Съществуват два типа форми box(кутия) и cylinder(цилиндър). Като цилиндъра както може да се види от двете звена link1 и link2, се дефинира с два параметъра радиус и дължина като радиуса се задава по X оста а дължината по Z оста `<cylinder radius="0.1" length="0.02"/>.`

Ставата има два параметъра име и тип, тук ще бъдат разгледани само стави от тип ротационна(revolute) `<joint name="joint1" type="revolute">` и фиксирана(fixed) `<joint name="joint2" type="fixed">.` За да се дефинира ставата е необходимо да се дефинира звеното към което е закачена ставата това става с таг parent: `<parent link="link1"/>,` и звеното което е закачено на ставата това става с таг child: `<child link="link2"/>.` Таг origin дефинира положението и ориентацията на ставата спрямо звеното parent. В случая на става joint1 ставата е изместена по X на -0.1(1 см в отрицателната посока), по Y на 0.23(2.3 см в положителна посока) и завъртяна спрямо координатната система на link1 на 90º около X оста: `<origin rpy="1.5707 0 0" xyz="-0.1 0.23 0"/>.` Както може да се види за дефиниране на ориентация тук се използват XYZ Ойлерови ъгли, както бяха разгледани в таблица 1.1.1.

Тъй като вече разгледахме как се дефинира положението на става, сега може да разгледаме как се дефинира положението на звено закачено към дадена става. С помощта на същия таг като при ставите положението и ориентацията се дефинират с origin, като тук позата на звеното се задава спрямо ставата към което то е закачено. Например при звено link1 позата се задава спрямо става joint: `<origin rpy="0 0 0" xyz="0 0 -0.025"/>,` като тук звено link1 е изместено по Z оста -0.025(0.25 см в отрицателна посока) спрямо става joint1.

За ротационна става е задължителен още един таг за дефиниране това е таг limit , с негова помощ се дефинира диапазона на ротация на ставата в случая на става joint0 диапазона е от 0º до 180º: `<limit effort="1000.0" lower="0" upper="3.1415" velocity="0.5"/>.`

Таг axis не е задължителен но с негова помощ може да се дефинира оста около която ще се върти ставата, по подразбиране това е Z оста: `<axis xyz="0 0 1"/>.`

1.4. Права кинематика

Задачата на правата кинематика за един робот манипулатор изграден от последователно свързани звена е да се намери позицията и ориентацията на крайното изпълнително устройство относително базовата координатна система знаейки позицията на всички стави и геометричните параметри на всички звена. Често координатната система фиксирана на изпълнителното устройство се нарича координатна система на инструмента, и въпреки, че е фиксирана спрямо последното звено на манипулатора тя най-често има константно отклонение по позиция и ориентация спрямо него. Координатната система спрямо която трябва да се изпълни тази задача е фиксирана и често се намира в основата на робота, но тя също може да бъде отместена .

По общо задачата на правата кинематика може да се дефинира като намиране на относителна позиция и ориентация на две координатни системи разположени на звената на робота, като е известна геометричната структура на робота и стойностите на променливите параметри на ставите , равни по брой на степените на свобода на механизма.

На практика, задачата на правата кинематика е да определи позицията на координатната система фиксирана на изпълнителното устройство спрямо координатната система фиксирана в основата (основната координатна система) . За робот от последователно свързани звена това е тривиална задача, която се решава като просто се умножат хомогенните трансформации между съседните звена. Например за един робот манипулатор с пет степени на свобода с хомогенни трансформации описващи всяко звено спрямо съседното зададени под формата ${}^{j}T_{i}$ задачата се свежда до решаване на (1.4.1). ${}^{j}T_{i}$ са матрици от вида (1.3.2).

$$ {}^{0}T_{6} = {}^{0}T_{1}\,{}^{1}T_{2}\,{}^{2}T_{3}\,{}^{3}T_{4}\,{}^{4}T_{5}\,{}^{5}T_{6} \qquad (1.4.1) $$

1.5. Обратна Кинематика

Задачата на обратната кинематика за едни робот изграден от последователно свързани звена е да намери стойности за променливите на ставите при зададена позиция и ориентация на изпълнителния механизъм(инструмента). Това означава да се намерят стойностите на променливите на ставите знаейки хомогенната трансформация между точките представляващи интерес. Това често се свежда до решаване на няколко нелинейни уравнения. За да съществува

решение, позицията и ориентацията на инструмента трябва да принадлежат на работното пространство на робота манипулатор.

Съществуват алгебрични, геометрични и числови методи за изчисляване на обратната кинематика. Тук няма да ги разгледаме подробно а отново ще разгледаме функции в Матлаб от Robotics Toolbox за да видим какви проблеми възникват при робот с 5 степени на свобода. За по-подробно обяснение теорията на методите за решаване на обратната кинематика вижте *R.M. Murray, Z. Li, S.S. Sastry: A Mathematical Introduction to Robotic Manipulation (CRC, Boca Raton 1994).*

За да видим правата и обратната кинематика за конфигурацията на 5-звенен робот показан на фиг.1.3.4. ще използваме кода.

```
L(1) = Link('d', 0, 'a', 0, 'alpha', pi/2, 'offset', pi, 'qlim', [-pi/2, pi/2]);
L(2) = Link('d', 0, 'a', 0.2, 'alpha', 0, 'offset', pi/2, 'qlim', [-pi/2, pi/2]);
L(3) = Link('d', 0, 'a', 0.2, 'alpha', 0, 'qlim', [-pi/2, pi/2]);
L(4) = Link('d', 0, 'a', 0, 'alpha', pi/2, 'offset', pi/2, 'qlim', [-pi/2, pi/2]);
L(5) = Link('d', 0.15, 'a', 0, 'alpha', 0, 'qlim', [-pi/2, pi/2]);
q = [pi/4 pi/4 pi/4 pi/4 pi/4];
robot = SerialLink(L);
robot.plot(q)
% Изчисляване на правата кинематика за стойност на ъглите на всички стави 45°
T = robot.fkine(q);
% Задаване на начална конфигурация на ставите за обратната кинематика
q0 = [ pi/2 pi/2 pi/2 pi/2 0];
% Изчисляване на обратната кинематика
a = robot.ikine(T,q0, [1 1 1 1 1 0], 'alpha', 0.05, 'ilimit', 1000);
tmp = mod(a+pi/2,pi);
out = tmp+pi*(a>0&tmp==0)-pi/2

% Задаване на начална конфигурация на ставите за обратната кинематика
q0 = [ 0 0 0 0 0];
a = robot.ikine(T,q0, [1 1 1 1 1 0], 'alpha', 0.02, 'ilimit', 1000);
tmp = mod(a+pi/2,pi);
out = tmp+pi*(a>0&tmp==0)-pi/2
```

С помощта на кода първо ще изчислим правата кинематика на робота при стойности на ъглите на ставите равни на 45° след това с изчислената хомогенна трансформация ще изчислим обратната кинематика при което тя трябва да върне стойности на ъглите равни на 45°.

За хомогенната трансформация изчислена с помощта на `T = robot.fkine(q)` се получава:

$$ {}^{0}T_{6} = \begin{pmatrix} -0.8536 & -0.1464 & 0.5000 & 0.3164 \\ 0.1464 & 0.8536 & 0.5000 & 0.3164 \\ -0.5000 & 0.5000 & -0.7071 & 0.0354 \\ 0 & 0 & 0 & 1.000 \end{pmatrix} \quad (1.5.1) $$

При начална конфигурация `q0 = [ pi/2 pi/2 pi/2 pi/2 0]` стойността на ъглите получени от метода *ikine* е: `q1 = [ 0.7854 0.7854 0.7854 0.7854 0.6584]`, а при начална конфигурация `q0 = [ 0 0 0 0 0]` е: `q2 = [ 0.7854 -0.8389 -1.0764 -0.3987 0.7986]`.

Ако приложим *fkine* метода за тези две конфигурации на ъглите q1 и q2 хомогенните трансформации са съответно:

$$ {}^{0}T_{6} = \begin{pmatrix} -0.8286 & -0.2534 & 0.5000 & 0.3164 \\ 0.0372 & 0.8652 & 0.5000 & 0.3164 \\ -0.5593 & 0.4326 & -0.7071 & 0.0354 \\ 0 & 0 & 0 & 1.000 \end{pmatrix} \quad (1.5.2) $$

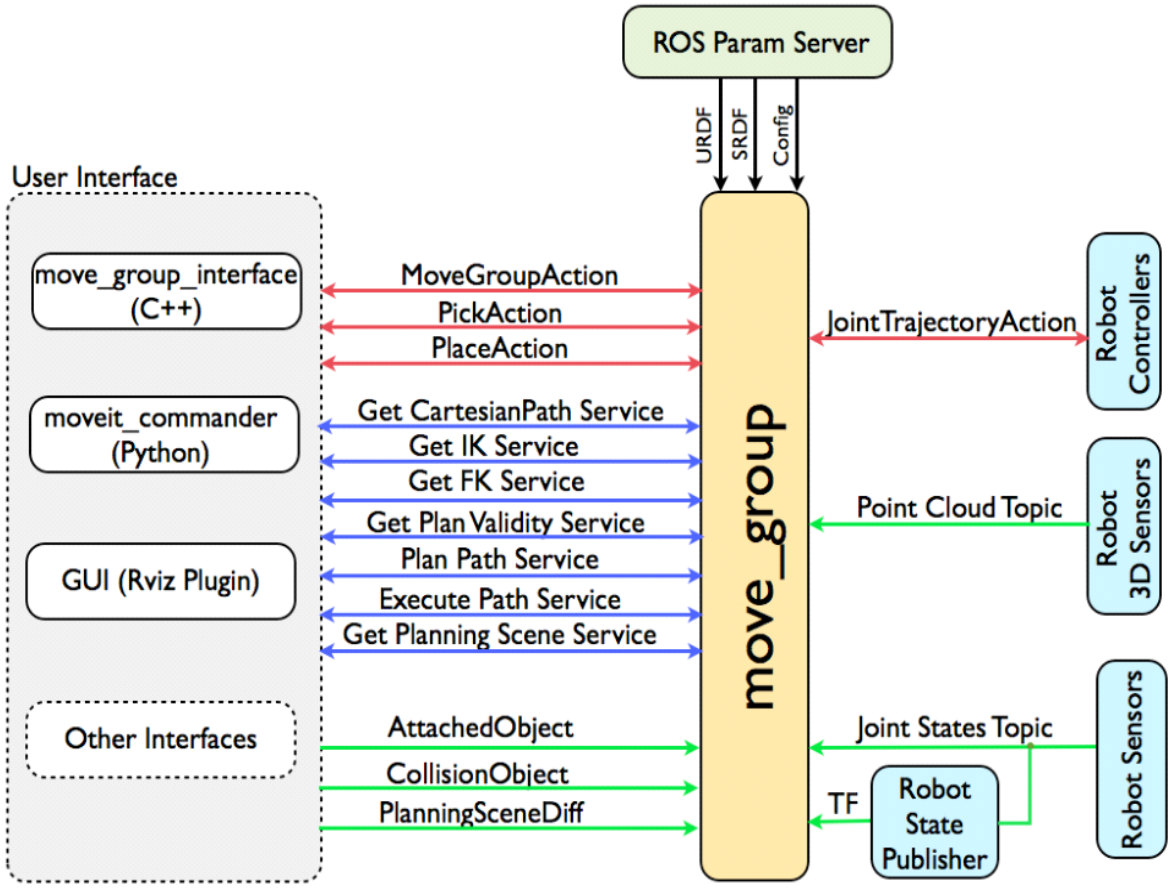
$$ {}^{0}T_{6} = \begin{pmatrix} -0.8404 & -0.1506 & -0.5207 & -0.3164 \\ 0.1728 & 0.8361 & -0.5207 & -0.3164 \\ 0.5138 & -0.5276 & -0.6766 & -0.0354 \\ 0 & 0 & 0 & 1.000 \end{pmatrix} \quad (1.5.3) $$

Както може да се види от 1.5.2. и 1.5.3. хомогенната трансформация която се получава за конфигурацията от ъгли изчислени от *ikine* метода не съвпада с желаната такава. Това често се случва при използване на общи числени методи за роботи със степени на свобода <6. Същия недостатък притежава и kdl inverse kinematics плъгин който ще използваме за изчисляване на обратната кинематика за моделирания в 1.3.2. робот.

За да избегнем това при моделирането на робота сме дефинирали четири допълнителни стави с ъгли на изменение в границите [-0.3°; 0.3°] който няма да окажат голямо влияние върху изчисленията, но благодарение на тях ще можем да използваме kdl inverse kinematics плъгин. Тези стави са link2_bar1, link2_bar1_bar2, vj0 и vj1.

Глава 2 Конфигуриране на робота в ROS.

Moveit е пакет в ROS с помощта на който може да се изчислява права, обратна кинематика , да се планира движение и др. На фиг.2.1. е показана архитектурата на Moveit пакета. След като разгледаме архитектурата на Moveit пакета , ще видим как с помощта на Moveit Setup Assistant и URDF файла за нашия робот(1.3.2) можем да конфигурираме и използваме пакета. В тази глава ще видим как след като сме конфигурирали Moveit можем да изчислим правата и обратната кинематика за нашия робот.



Фиг. 2.1 Архутектура на Moveit пакета [4]

2.1. Основни концепции в Moveit

На фиг.2.1. е показана системната архитектура за главния възел в Moveit наречен move_group. Този възел свързва отделните плъгини за всички компоненти който използва Moveit, като плъгини за кинематиката, планирането на движение и д.р, и предоставя методи чрез който можем да използваме тези плъгини. С помощта на това разделяне можем да използваме най-различни плъгини, например за изчисляване на обратната кинематика тук ще използваме kdl kinematics plugin , но както ще видим след малко можем да конфигурираме Moveit да използва най-различни плъгини за кинематиката, така ако някой от плъгините не е подходящ за нашата задача можем да го сменим с друг или да си реализираме наш.

Потребителски интерфейс

Потребителите могат да достъпят услугите който предоставя move_group по три начина:

- C++ интерфейс
- Python интерфейс
- С помощта на Rviz чрез графичен интерфейс.

Главно ще използваме Python интерфейс, като в глава 3 ще видим как с негова помощ ще зададем позиция и ориентация която робота ще изпълни след това.

Конфигурация

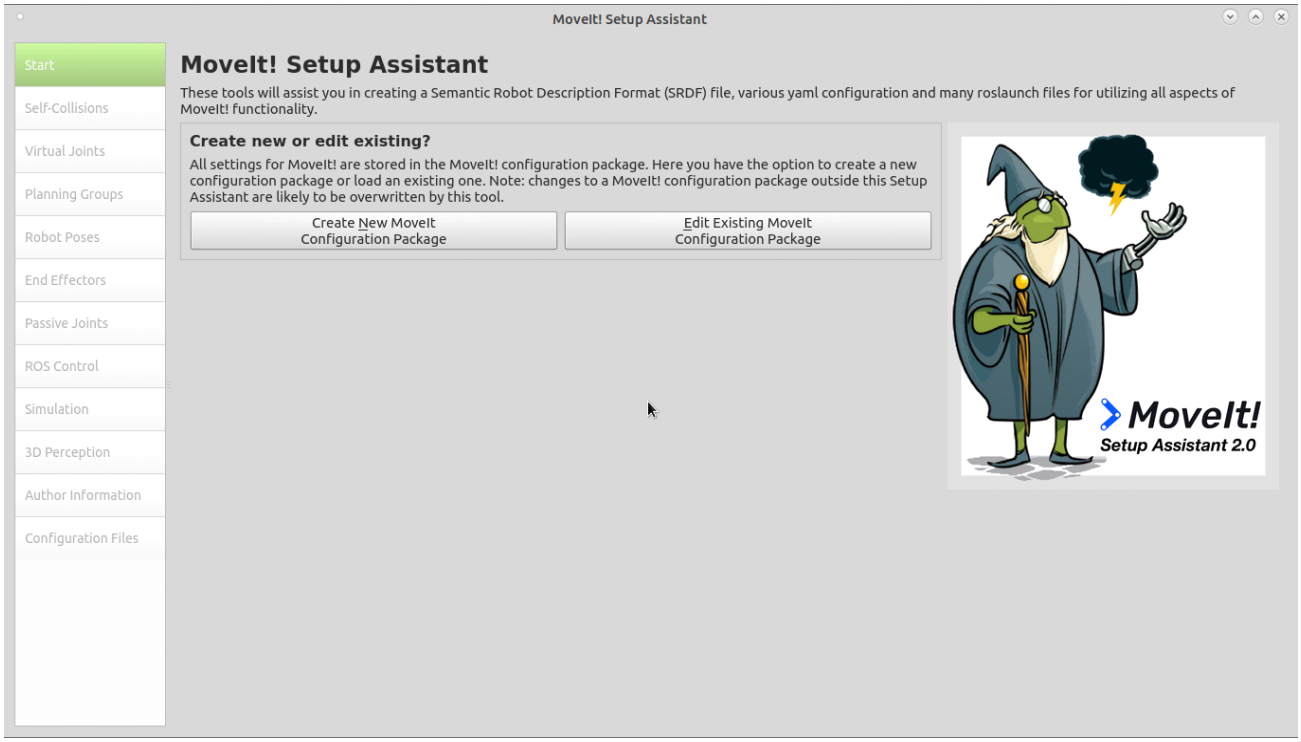
move_group е ROS възел, който използва ROS param server за да получи три вида информация. Повече информация за ROS param server може да намерите в Приложение 1.

- URDF – move_group получава информация за описанието на робота от URDF файла
- SRDF – този файл се създава от Moveit Setup Assistant, той е по-пълно описание на робота.
- Moveit configuration – Това са конфигурационни файлове свързани с кинематиката, ограниченията на ставите, планирането на движение и др. Тези файлове се генерират от Moveit Setup Assistant и се съдържат в config директорията на Moveit config пакета.

2.2. Moveit Setup Assistant

След като сме създали URDF файла можем да използваме Moveit Setup Assistant да конфигурираме нашия робот. За нашия робот след като изтеглим robot_urdf пакета и го създадем както е обяснено в приложение 1 можем да стартираме Moveit Setup Assistant с командата **roslaunch moveit_setup_assistant setup_assistant.launch** или да изтеглим пакета от https://github.com/st1na/robot_arm, конфигурирания пакет се намира в *software/ros_packages/robot_moveit_config.* Ако изтеглите конфигурирания пакет може да пропуснете тази част. Подробна информация за Moveit Setup Assistant може да се намери тук http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/setup_assistant/setup_assistant_tutorial.html. Преди да стартираме горната команда трябва да се намираме в работната директория(workspace) където се намира robot_urdf и да стартираме командата *source devel/setup.bash*

След като стартираме Moveit Setup Assistant излиза следния прозорец.



Фиг. 2.2.1. Moveit Setup Assistant стартов прозорец

След това избираме Create **New Moveit Configuration Package**, след което избираме **Browse** и от там избираме нашия URDF файл , в случая *robot_arm_hand.urdf* , след това избираме **Load Files.** След това избираме **Self-Collisions** и избираме **Generate Collision Matrix.** Избираме **Virtual Joints** и създаваме една виртуална става както е показано на фиг. 2.2.2. след което избираме **Save.**

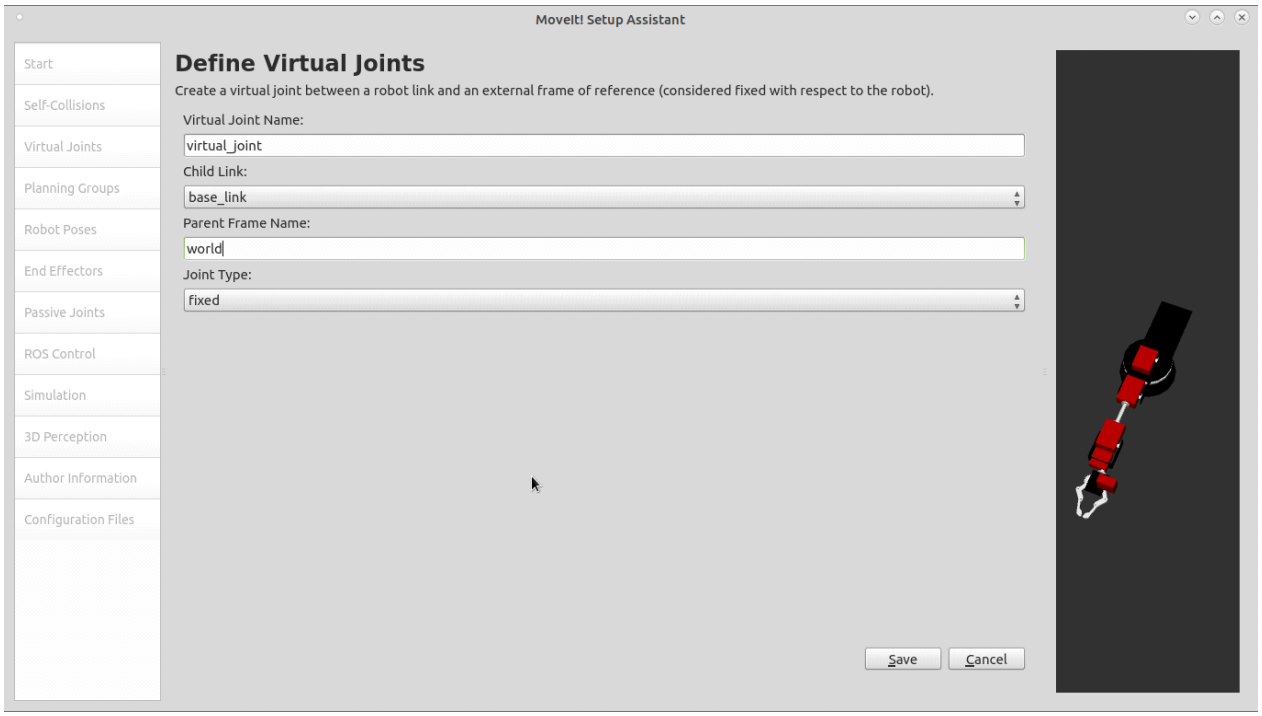
След това избираме **Planning Groups→ Add Group** и създаваме две групи една за ръката и една за инструмента. При избиране на **Add Group** излиза следния прозорец показан на фиг.2.2.3. От тук трябва да създадем първо една група за ръката след което отново да стартираме **Add Group** и да създадем друга група за инструмента.

Групата за ръката има следните параметри:

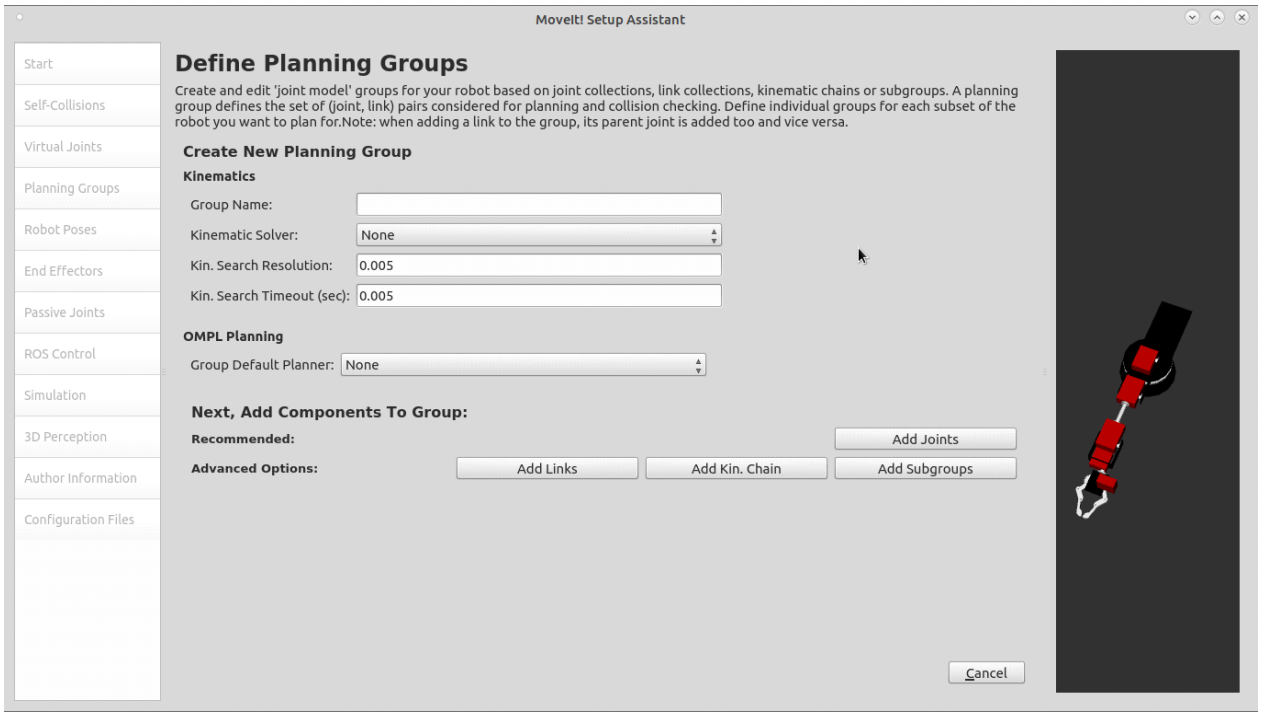
- *Group Name: robot_arm*
- *Kinematic Solver: kdl_kinematic_plugin/KDLKinematicPlugin*
- *Kin. Search Resolution: 0.005*
- *Kin. Search Timeout(sec): 0.005*
- *Group Default Planner: None*

След това избираме **Add Joints** и избираме следните стави: **virtual_joint, base_link_black, base_black_white, joint0, base_link1, joint1, link1_rot2_rot1, link1_rot2_frame1, link1_frame1_frame3, joint2, link2_rot2_rot1, link2_rot1, link2_bar1, link2_bar1_bar2, link2_bar2_bar3, link2_bar3_link3, joint3, link3_rot2_rot1, link3_rot2_frame1, link3_frame1_link4,**

joint4, link4_rot2_rot1, vj0, vj1. Някой от ставите не се избират защото не изпълняват условиенто да са сериино свързани, те са използвани в URDF файла с декоративна цел.



Фиг.2.2.2. Създаване на виртуална става.



Фиг.2.2.3. Planning Group прозорец

Групата за инструмента има следните параметри:

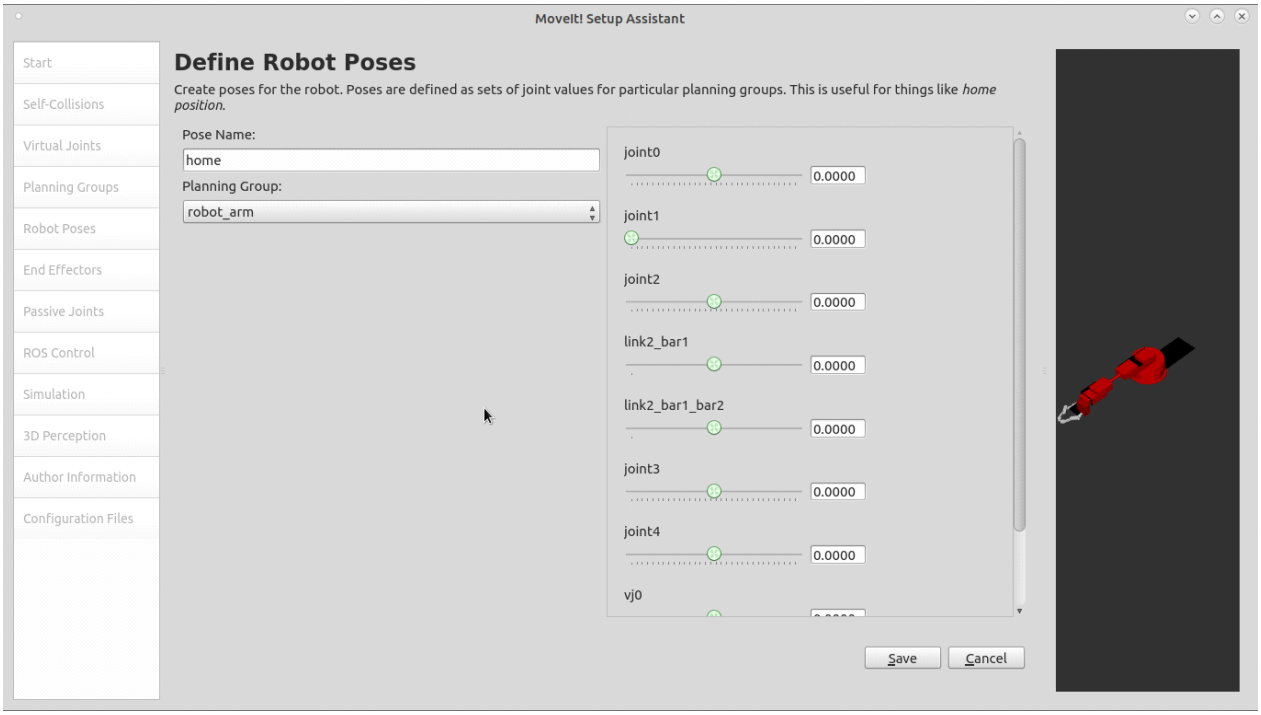
- *Group Name: robot_hand*
- *Kinematic Solver: None*
- *Kin. Search Resolution: 0.005*
- *Kin. Search Timeout(sec): 0.005*
- Group Default Planner: None

След това избираме **Add Joints** и избираме следните стави: **arm_hand, gripper, gripper_left.**

Robot Poses не е задължително но тук може да тестваме дали всичко е наред с модела като проверим дали всички стави се движат както е показано на фиг. 2.2.4.

При **End Effectors** избираме **Add End Effector** със следните параметри:

- *End Effector Name: hand*
- *End Effector Group: robot_hand*
- *Parent Link: vl1*



Фиг.2.2.4. Добавяне на пози за робота

Пропускаме **Passive Joints** и избираме **Ros Control→ Add Controller** с параметри:

- Controller Name: arm_position_controller
- Controller Type: poisition_controller/JointPositionController

След това избираме **Add Planning Group Joints** и избираме **robot_arm.**

Избираме **Simulation→Generate URDF.** След това **3D Perception: Point Cloud.**

Записваме име и имейл след като изберем **Author Information,** след това избираме **Gonfiguration Files** и запазваме пакета в папка която сме създали с името *robot_moveit_config* в работната директория където се намира пакет *robot_urdf.* Например ако работната директория се казва robot_moveit_ws, трябва да запазим пакета в папка *src/robot_moveit_config*. След което избираме **Generate Package** и сме готови.

След като приключим трябва да създадем пакета като в работната директория извикваме командата *catkin_make.* След това отново трябва да заредим setup.bash в терминала и можем да проверим дали всичко е наред с помощта на командата

roslaunch robot_moveit_config demo.launch rviz_tutorial:=true. Тази команда ще покаже графичния интерфейс който споменахме в 2.1. , как се използва този интерфейс няма да се разглежда тъй като ние ще използваме Python интерфейса. Информация за това как се използва този интерфейс може да намерите тук:
http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/quickstart_in_rviz/quickstart_in_rviz_tutorial.html.

2.3. Права и обратна кинематика

Сега можем да разгледаме как можем да изчислим правата и обратната кинематика за нашия робот. За целта ще използваме fi_kine.py script в пакета code_samples. След като стартираме **roslaunch robot_moveit_config demo.launch rviz_tutorial:=true** в отделен терминал отиваме в папка robot_moveit_ws/src/code_samples/src/ и стартираме скрипта с командата **./fi_kine.py**. Изхода на командата е показан на фиг. 2.3.1.

```
[ INFO] [1556470437.827846165]: Loading robot model 'robot'...
[ INFO] [1556470439.209593248]: Ready to take commands for planning group robot_arm.
Kinematics Tutorial
============ Press `Enter` to execute a movement using a joint state goal :
[1.5707963267948966, 1.5707963267948966, 1.5707963267948966, 1.5707963267948966, 1.5707963267948966]

Pose is:
position:
  x: -0.130175081483
  y: -1.31015315388
  z: 1.02506356199
orientation:
  x: 0.707101189855
  y: 0.70711236945
  z: -2.2940746634e-05
  w: 6.12379479332e-05
================================================================================
============ Press `Enter` to execute a movement using a pose goal :
position:
  x: -1.91319177157
  y: 1.0260209944
  z: -0.0542995273767
orientation:
  x: -0.862638826051
  y: -0.418320070496
  z: 0.177931618426
  w: 0.221817297733

Joint values are:
-0.430407291647
2.39889752351
1.46110524194
0.270091555727
0.239056297646
```

Фиг. 2.3.1. Права и обратна кинематика с помощта на ROS

Глава 3 Симулация на робота.

Тук ще покажем как с помощта на Moveit Python интерфейса можем да зададем позиция и ориентация след което да изчислим обратната кинематика и да видим как се симулират резултатите. Тук ще разгледаме как от зададена позиция и ориентация се изчисляват ъглите и се публикуват към тема(topic). След това в глава 4 ще видим как да конфигурираме Raspberry Pi да изпраща изчислените ъгли по SPI към драйвер, който ще бъде разгледан в глава 5.

3.1. Симулация на робота.

Конфигуриране на работната директория

Тук ще използваме пакетите code_samples, robot_moveit_config, robot_urdf, който могат да бъдат изтеглени от https://github.com/st1na/robot_arm_ros_packages. За да бъдат изградени всички пакети трябва да се създаде работна директория(workspace) името на която е произволно, но тук ще я наречем *robot_moveit_ws*, в работната директория създаваме директория *src*, и се местим в нея. След като вече се намираме в директория *src* на работната директория , стартираме следните команди.

```
git clone https://github.com/st1na/robot_arm_ros_packages
mv robot_arm_ros_packages/* . & sudo rm -r robot_arm_ros_packages
chmod +x code_samples/src/*
```

След като вече сме изтеглили пакетите трябва да инициализираме работната директория това става с командата **catkin_init_workspace** . След това се местим в работната директори (*robot_moveit_ws*) и стартираме командата **catkin_make** . След като **catkin_make** приключи използваме командата **source devel/setup.bash** . След като вече сме конфигурирали работната директория можем да стартираме симулацията и да разгледаме кода който ще използваме за задаване и изчисление на ъглите както при симулацията така и при реалния робот.

Симулация

Стартирането на симулацията става с командата

roslaunch robot_moveit_config simulation_start.launch.

Тази команда стартира launch файла *simulation_start.launch*. Чрез него се стартира симулатора Rviz и още два скрипта get_input.py и joints_calc.py.

simulation_start.launch

```
<launch>

      <include file="$(find robot_moveit_config)/launch/demo.launch">
      </include>

      <node pkg="code_samples" name="get_input" type="get_input.py"
output="screen">
      </node>

      <node pkg="code_samples" name="joints_calc" type="joints_calc.py"
output="screen">
      </node>


</launch>
```

3.2. Възли get_input и joints_calc.

Възела get_input е python скрипт който с цел демонстрация е написан да генерира произволна поза на всеки 20 секунди и да я изпраща към joints_calc възела. Възела joints_calc изчислява ъглите от зададената поза след което rviz показва изпълнението на движението. Двата скрипта се намират в пакет *code_samples* папка *src.* Сега ще разгледаме как са реализирани get_input и joints_calc възлите. Както споменахме по-горе те използват python интерфейса на Moveit.

С помощта на възела *get_input ce* задава позата която трябва да се изпълни. Това става с функцията **publish_random_pose()** . Но първо трябва да сме инициализирали *move_group* обекта, това се случва още в началото с командата

move_group = moveit_commander.MoveGroupCommander("robot_arm").

След като сме инициализирали move_group извикваме функцията **publish_random_pose().**

```
def publish_random_pose():
    # Publish every 20 seconds
    rate = rospy.Rate(0.05)
    pub = rospy.Publisher('pose_goal', Pose, queue_size=10)

    while not rospy.is_shutdown():
        pose_goal = random_pose()
        # Check if z is less than 0
        while pose_goal.position.z < 1 or pose_goal.position.x<0:
            pose_goal = random_pose()

        pub.publish(pose_goal)
        print pose_goal
        rate.sleep()
```

Първо се създава тема(topic) pose_goal към който този възел ще публикува данни от тип Pose. Това става с **pub = rospy.Publisher('pose_goal', Pose, queue_size=10).** След което се създава един безкраен цикъл **while not rospy.is_shutdown(),** който избира една произволна поза **pose_goal = random_pose()** проверява дали координата z<1 или координата x<0 и ако и двете условия не са изпълнени публикува позата **pub.publish(pose_goal)** , ако едно от двете условия е изпълнено се генерира нова поза чийто x и z координати отново се проверяват и ако те отново изпълняват едно от двете условия цикъла се повтаря докато не се генерира поза с координата z<1 и координата x<0. С помощта на **rate = rospy.Rate(0.05)** се задава честота с която да се повтаря цикъла.

Възела joints_calc се абонира към две теми едната е *pose_goal* темата която разгледахме по горе а другата е *joint_states,* тя дава ъглите на ставите на робота. Първо ще разгледаме как става самото абониране към темата *pose_goal* и изчисляването на ъглите от получените координати. Абонирането става с помощта на **self.pose_goal_sub = rospy.Subscriber ('pose_goal', geometry_msgs.msg.Pose, self.pose_callback),** след като се получи информация от *pose_goal* темата се извиква функция **pose_callback(self, pose_goal)**, която на свой ред извиква функцията която изчислява ъглите на ставите **compute_joints(self, pose_goal)**.

```
def compute_joints(self, pose_goal):
    # compute the joints from the pose
    move_group.set_goal_position_tolerance(0.01)
    move_group.set_goal_joint_tolerance(0.01)
    move_group.set_goal_orientation_tolerance(0.01)
    move_group.set_planning_time(5.0)

    move_group.set_pose_target(pose_goal)
    plan = move_group.go(wait=True)
    move_group.stop()
```

Първите няколко ред задават конфигурационни параметри изчисляването на ъглите и изпълнението им на симулацията става с помощта на тези два реда:

move_group.set_pose_target(pose_goal)
plan = move_group.go(wait=True) .

След като робота изпълни зададената поза ъглите които съответсват на желаната поза се публикуват на тема *joint_states* затова след като вече сме се абонирали на тази тема **self.joint_state_sub=rospy.Subscriber('joint_states', sensor_msgs.msg.JointState, self.joint_callback)** се извиква функцията **joint_callback(self, joint_states)** която преобразува ъглите в обхвата [-90°; 90°] и ги публикува на тема *goal_joints.* Следващия възел *send_joints* получава тези ъгли и ги изпраща по SPI, той няма да бъде разгледан тук , тъй като не се използва при симулацията а само при реалното изпълнение . Той ще бъде разгледан в глава 4 след като видим как да конфигурираме Raspberry Pi.

```
    def joint_callback(self, joint_states):
        self.robot_joints = self.joint_remove_virtual(joint_states)
        self.robot_joints = self.joint_map(self.robot_joints)
        self.robot_joints = self.joint_radtodeg(self.robot_joints)
        # Send joints in range [-90, 90] deg
        send_joints(self.robot_joints)
```

След като ъглите вече са получени първо трябва да премахнем тези виртуалните които дефинирахме в URDF описанието за да работи kdl_kinematic_plugin.Това става с функция **joint_remove_virtual(self, joint_states).**

```
    def joint_remove_virtual(self, joint_states):
        # joint_states name and position have
        # the same indexes, get the virtual joints
        # indexes and remove them
        virtual_joints_names = ['link2_bar1', 'link2_bar1_bar2', 'vj0',
'vj1', 'gripper_left']
        joint_states.position = list(joint_states.position)
        for i in virtual_joints_names:
            virtual_joint_index = joint_states.name.index(i)
            del joint_states.name[virtual_joint_index]
            del joint_states.position[virtual_joint_index]

        return joint_states
```

Тя намира звената с имена *'link2_bar1', 'link2_bar1_bar2', 'vj0', 'vj1', 'gripper_left'* премахва стойностите им от листа със стойности на ъглите на ставите и връща лист с ъгли който отговарят на реалните който робота ще изпълни. За gripper_left не сме говорили това е става която имитира gripper и се използва да имитира реалното движение на инструмента на робота.

След като вече са останали само реалните стави ги привеждаме в обхват [-pi/2; pi/2] с функцията **self.robot_joints = self.joint_map(self.robot_joints)**, след което ги преобразуваме в градуси с функцията **self.robot_joints = self.joint_radtodeg(self.robot_joints)** и ги изпращаме към goal_joints темата **send_joints(self.robot_joints).**

```
def send_joints(goal_joints):
    goal_joints_pub =
rospy.Publisher('goal_joints',sensor_msgs.msg.JointState,
queue_size=10)
    goal_joints_pub.publish(goal_joints)
```

Тази функция създава тема *goal_joints* след което публикува изчислените и преобразувани ставни ъгли към тази тема.

Глава 4 Конфигуриране на управляващия модул.

Тъй като вече сме разгледали как да използваме и конфигурираме симулацията е време да видим как да конфигурираме Raspberry Pi за реалния робот. За целта ще се използва Ubuntu Mate 18.04 и ROS Melodic. Инсталирането на Ubuntu Mate 18.04 за Raspberry Pi 3B+ е разгледано в приложение 2, инсталирането на ROS Melodic става по същия начин както и на PC с Ubuntu 18.04, както е разгледано в приложение 1. Тук ще разгледаме как след като вече сме изчислили ставните ъгли, както беше показано в глава 3, ги изпращаме към драйвера по SPI.

Изграждането на пакетите става по същия начин както беше показано в глава 3. Тук изграждането става на Raspberry Pi след като сме се логнали с помощта на SSH . Преди да изградим пакетите трябва да изтеглим и инсталираме библиотеката необходима за използване на SPI интерфейса. Библиотека се казва spidev и се инсталира по следния начин. Първо проверяваме дали системата е актуализиране.

```
sudo apt-get update
sudo apt-get upgrade
```

След което стартираме командата.

```
sudo apt-get install python-dev python3-dev
```

След което трябва да изтеглим spidev и да го инсталираме, това става със следните команди.

```
cd ~
mkdir python_packages
cd python_packages
git clone https://github.com/doceme/py-spidev.git
cd py-spidev
make
sudo make install
```

След като сме инсталирали нужните библиотеки за използване на SPI можем да изградим пакета и да стартираме робота с помощта на командата.

```
roslaunch robot_moveit_config robot_start.launch
```

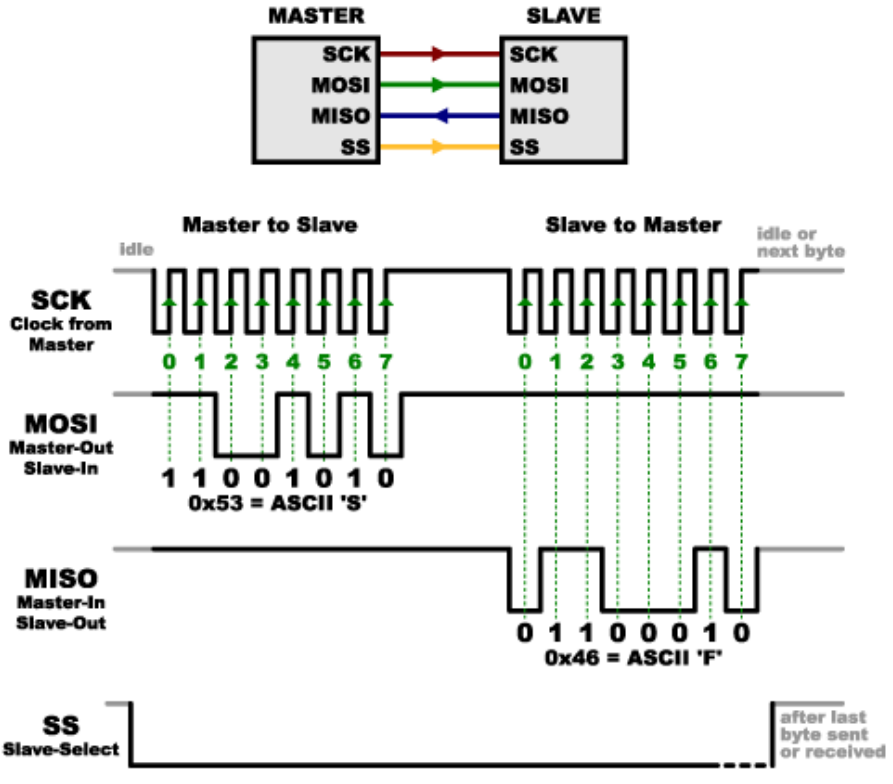
Сега ще дадем кратко описание на SPI протокола след което ще видим как с помощта на инсталираната горе библиотека лесно можем да изпратим изчислените ъгли към драйвера който ще разгледаме в глава 5.

4.1. SPI протокола

SPI е протокола със синхронна сериина комуникация използван предимно при къси разстояние, при вградените системи. Той се базира на master-slave архитектура с едно master устройство. Master устройството дава началото на пакетите(frame) за четене и запис. Избирането на slave устройството с което ще комуникира master устройството става чрез индивидуални сигнали наречени slave select(SS) или chip select(CS). SPI е full duplex протокол с два отделни сигнала за изпращане (MOSI) и получаване на данни (MISO). Основни недостатък на този протокол е че при увеличаване на устройствата се увеличават сигналите тъй като всяко устройство се нуждае от отделен CS сигнал. Сигналите които се използват са както следва:

- SCLK - Serial Clock (синхронизиращия clock между устройствата)
- MOSI – Master Out Slave In (изходни данни от master устройството)
- MISO – Master In Slave Out (изходни данни от slave устройството)
- CS – Chip Select (често този сигнал е с ниско активно ниво, избира slave устройството)

При SPI SCLK сигнала се генерира от master устройството, както вече споменахме при този вид комуникация може да има само едно устройство което да е master. Когато изпращаме данни от master към slave , данните се изпращат с MOSI сигнала, ако slave устройството трябва да върне данни, master устройството ще продължи да генерира определен брой импулси на сигнала SCLK, и slave устройството ще върне данни с помощта на MISO сигнала. CS сигнала се държи във високо ниво, за всички slave устройства, като преди да стартираме изпращането трябва да сменим състоянието на CS сигнала на slave устройството на което искаме да изпратим данни, на ниско ниво и след като приключим да върнем сигнала във високо ниво. Фиг. 4.1. илюстрира комуникацията между две устройства.



Фиг. 4.1. SPI комуникация между две устройства.[5]

4.2. Raspberry Pi SPI

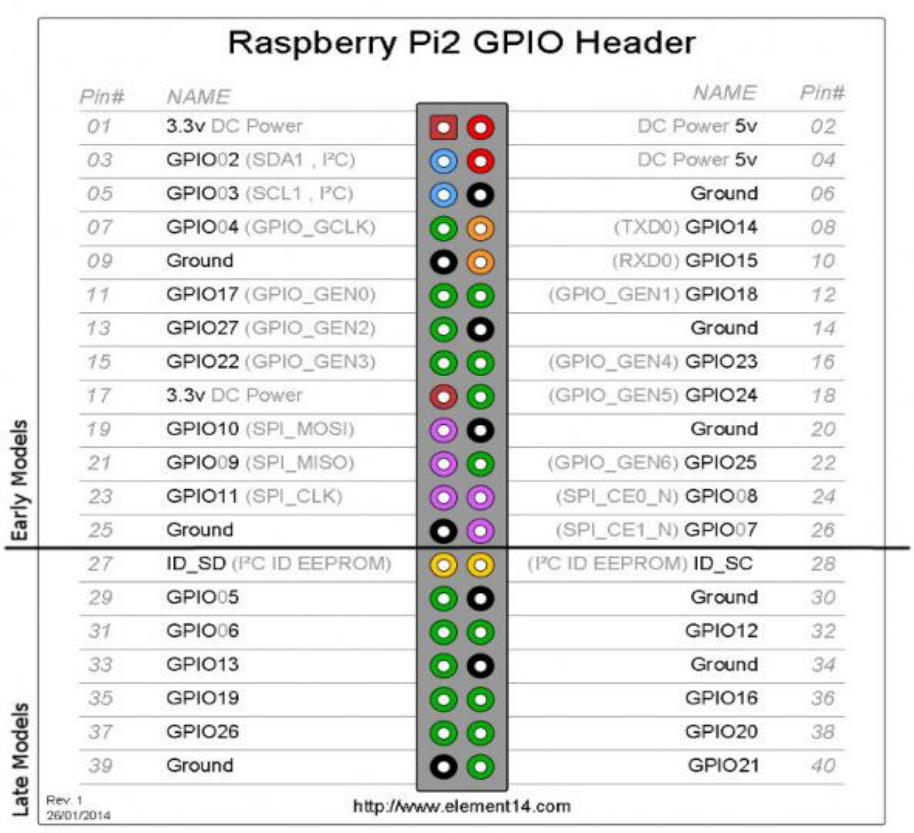
Изпращането на ъглите става с помощта на send_joint възела. Както споменахме в глава 3 изчислените ъгли се публикуват на тема goal_joints затова първо във този възел трябва да се абонираме за тази тема.

```
joint_goal_sub = rospy.Subscriber('goal_joints',sensor_msgs.msg.JointState ,
self.driver_callback)
```

След като вече сме се абонирали при всяко публикуване на ъгли от страна на joints_calc възела се извиква функцията driver_callback, която на свой ред извиква функцията която изпраща ъглите spi_send. Преди да можем да изпратим ъглите трябва да инициализираме SPI това става с функция spi_init.

```
def spi_init(self):
    SPI_SPEED = 100000 # Hz

    bus = 0
    # SS0
    device = 0
    #Enable SPI
    self.spi_handle = spidev.SpiDev()
    self.spi_handle.open(bus, device)
    #Set speed and mode
    self.spi_handle.max_speed_hz = SPI_SPEED
    self.spi_handle.mode=0
```



SPI_SPEED задава скоростта на изпращане, bus = 0 тъй като имаме два SPI интерфейса на Raspberry Pi избираме SPI0 , device = 0 избираме CS0(chip select). На фиг. 4.1. са показани пиновете който ще използваме, за SPI комуникацията.

Фиг. 4.2. Raspberry Pi pinout [6]

След като вече сме инициализирали SPI можем да изпратим ъглите с помощта на **spi_send**.

```
def spi_send(self, joint_goal):
    # Signaling that the last joint is send
    # this value will be used in the driver as well
    END_TRANS = 100

    joints = list(joint_goal.position)
    for i in range(len(joints)):
        joints[i] = int(joints[i])
    print joints

    self.spi_handle.xfer2([ joints[0],joints[1] ])
    self.spi_handle.xfer2([ joints[2],joints[3] ])
    self.spi_handle.xfer2([ joints[4],joints[5] ])
    self.spi_handle.xfer2([END_TRANS])
```

Първо задаваме флаг който ще сигнализира че всички ставни ъгли са изпратени END_TRANS = 100 , тук сме избрали стойност 100 , но това число може да е всичко което е извън интервала [-90;90] и е в обхвата на 8 битов signed integer [-128; 127] тъй като ще изпращаме ъглите като 8 битови signed integers. Преди да изпратим ъглите трябва първо да ги преобразуваме в signed integers, тъй като от pose_goal темата ги получаваме като float64 (64 битови числа с плаваща запетая).

```
joints = list(joint_goal.position)
for i in range(len(joints)):
    joints[i] = int(joints[i])
```

След което ги изпращаме с функцията **spi_handle.xfer2**, тя приема като аргумент лист от две стойности и ги изпраща.

```
self.spi_handle.xfer2([ joints[0],joints[1] ])
self.spi_handle.xfer2([ joints[2],joints[3] ])
self.spi_handle.xfer2([ joints[4],joints[5] ])
self.spi_handle.xfer2([END_TRANS])
```

4.3. Симулация в реално време

Тъй като Raspberry Pi няма достатъчно ресурси да стартираме симулацията, тя трябва да се стартира на потребителското PC, като ъглите се получават по TCP/IP. При ROS получаването на информация от възел на друго устройство по TCP/IP е предвидено като за да използваме тази функция трябва просто да конфигурираме двете устройства. Двете устройства между който ще се извършва комуникацията в случая Raspberry Pi и потребителското PC трябва да се конфигурират като master и slave. Ние ще изберем Raspberry Pi като master и потребителското PC като slave.

4.3.1. Конфигурация на master устройството .

За конфигурацията трябва да стартираме следните командите на Raspberry Pi.

```
export ROS_MASTER_URI=http://MASTER_IP:11311
export ROS_IP=MASTER_IP
```

MASTER_IP – IP адрес на Raspberry Pi.

За по лесно стартиране на робота и конфигуриране на TCP/IP можем да създадем bash script с име robot_start който можем да стартираме с командата. Името на скрипта е произволно.

```
source robot_start
```

Съдържанието на скрипта е следното.

```
#!/bin/bash

MASTER_IP=$(hostname -I | cut -d' ' -f1)
echo ${MASTER_IP}

export ROS_MASTER_URI=http://${MASTER_IP}:11311
export ROS_IP=${MASTER_IP}

ROBOT_WORKSPACE=$(find ~ -type d -name 'robot_moveit_ws')

source $ROBOT_WORKSPACE/devel/setup.bash
roslaunch robot_moveit_config robot_start.launch
```

В този script се предполага че работната директория се казва 'robot_moveit_ws', и се намира в home директорията. Скрипта може да бъде изтеглен от https://github.com/st1na/robot_arm/blob/master/software/bash_scripts/robot_start или със командата .

```
wget
https://raw.githubusercontent.com/st1na/robot_arm/master/software/bash_scripts/robot_start.
```

4.3.2. Конфигурация на slave устройството .

Тези команди трябва да се стартират на потребителското PC.

```
export ROS_MASTER_URI=http://MASTER_IP:11311
export ROS_IP=SLAVE_IP
```

MASTER_IP – IP адреса на Raspberry Pi
SLAVE_IP – IP адреса на потребителското PC

Стартирането на симулацията става с командата.

roslaunch robot_moveit_config moveit_rviz.launch

Като предварително сме заредили необходимия setup.bash от работната директория на пакетите изградени в глава 3.

Отново можем да създадем bash script за стартиране на симулацията с име robot_sim_start който приема един аргумент IP адреса MASTER_IP, за да го стартираме извикваме командата.

robot_sim_start hostname.local

В нашия случай тъй като hostname на Raspberry е robotuser-desktop аргумента е robotuser-desktop.local. Тоест командата придобива вида.

robot_sim_start robotuser-desktop.local

Съдържанието на скрипта е следното.

```
#!/bin/bash

MASTER_IP=$1
SLAVE_IP=$(hostname -I | cut -d' ' -f1)

export ROS_MASTER_URI=http://${MASTER_IP}:11311
export ROS_IP=${SLAVE_IP}

ROBOT_WORKSPACE=$(find ~ -type d -name 'robot_moveit_ws')

source $ROBOT_WORKSPACE/devel/setup.bash:q:
roslaunch robot_moveit_config moveit_rviz.launch
```

Скрипта за потребителското PC за стартиране на симулацията в реално време може да бъде изтеглен от https://github.com/st1na/robot_arm/blob/master/software/bash_scripts/robot_sim_start или с помощта на командата.

wget https://raw.githubusercontent.com/st1na/robot_arm/master/software/bash_scripts/robot_sim_start

Глава 5 Проектиране на драйвер за серво мотори.

След като вече сме изчислили ъглите за желаната поза и можем да ги изпратим по SPI ни трябва драйвер който да получи ъглите и да ги преобразува в необходимите управляващи сигнали за моторите. За нашия робот се използват серво мотори управлението на който ще разгледаме преди да разгледаме как да създадем драйвер за нашия мотор. След като разгледаме необходимите управляващи сигнали за серво моторите вече сме готови да реализираме драйвер който да получи ъглите от Raspberry Pi и да ги преобразува в необходимите управляващи сигнали.

5.1. Серво мотори

Тук няма да разглеждаме конструкцията на серво моторите а само начина по който можем да ги управляваме. Използваните за робота серво мотори са известни още с името RC серво мотори. Тези мотори имат три проводника, два захранващи и един управляващ като

- Жълт или бял проводник – управляващ сигнал
- Червен проводник – захранващо напрежение
- Кафяв или черен проводник – GND

Захранващото напрежение за използваните мотори е между 4.8 и 6 V. Моторите са DS3218 показан на фиг. 5.1.

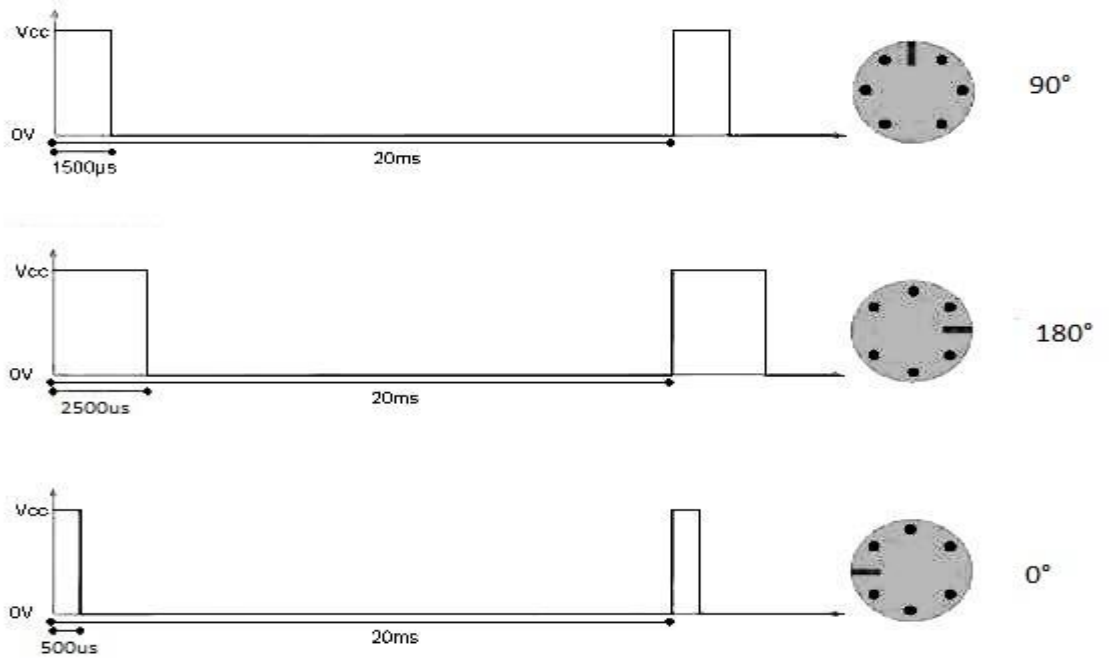
DS3218

Фиг. 5.1.

При RC серво моторите механичния ъгъл на завъртане се определя от ширината на електрически импулс. Сигнала прилича на широчинно импулсна модулация(ШИМ). Обикновено повечето серво мотори очакват да получат импулс всеки 20 ms. Ширината на импулса определя завъртането на мотора, като например в повечето серво мотори импулс с ширина 1.5 ms ще завърти мотора на 90° (neutral position) . Долната граница на импулса (завъртане на 0°) варира в зависимост от серво моторите, като за повечето е между 0.5-1 ms. Горната граница(завъртане на 180°) също варира като за повечето серво мотори е между 2-3 ms.

Позицията на серво моторите не се определя от коефициента на запълване на ШИМ, а от ширината на импулсите. Когато се подаде сигнал към тези серво мотори те ще си задържат позицията дори и при прилагане на външна сила която се опитва да изведе мотора от тази позиция, докато се подават импулси на всеки 20 ms. Максималният момент който може да задържи мотора зависи от мотора. Интервала между който се подават импулсите още се нарича честота на обновяване(refresh rate) и в зависимост от серво мотора може да варира между 40 Hz и 200 Hz, но най-често използваната, както вече споменахме е 50 Hz(20 ms).

Серво моторите които сме използвали очакват да получат импулс с ширина между 0.5-2.5 ms, като 0.5 ms отговаря на завъртане на 0° , 1.5 ms на завъртане на 90° а 2.5 ms на завъртане на 180°. Така например ако подадем импулс с ширина 1 ms получаваме завъртане на 45°. На фиг. 5.2. е илюстриран принципа на управление.



Фиг. 5.2. Управление на RC серво мотори. [7]

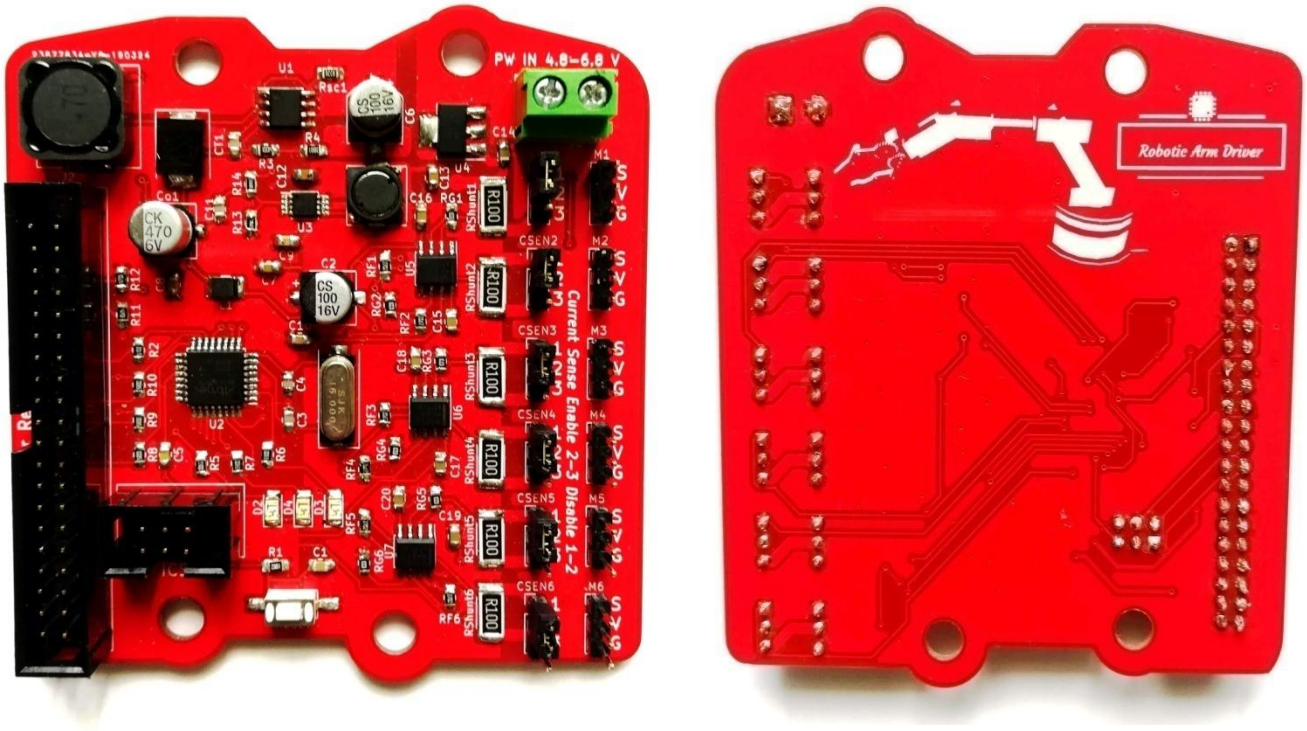
5.2. Управление на серво моторите

След като вече сме разгледали как можем да управляваме серво моторите, трябва да намерим начин да получаваме ъглите на завъртане от Raspberry Pi който са в обхвата [-90° ; 90°] , да ги преобразуваме в импулси с ширина в обхвата [0.5ms; 2.5ms] който да се повтарят всеки 20ms и да ги изпращаме към серво моторите.

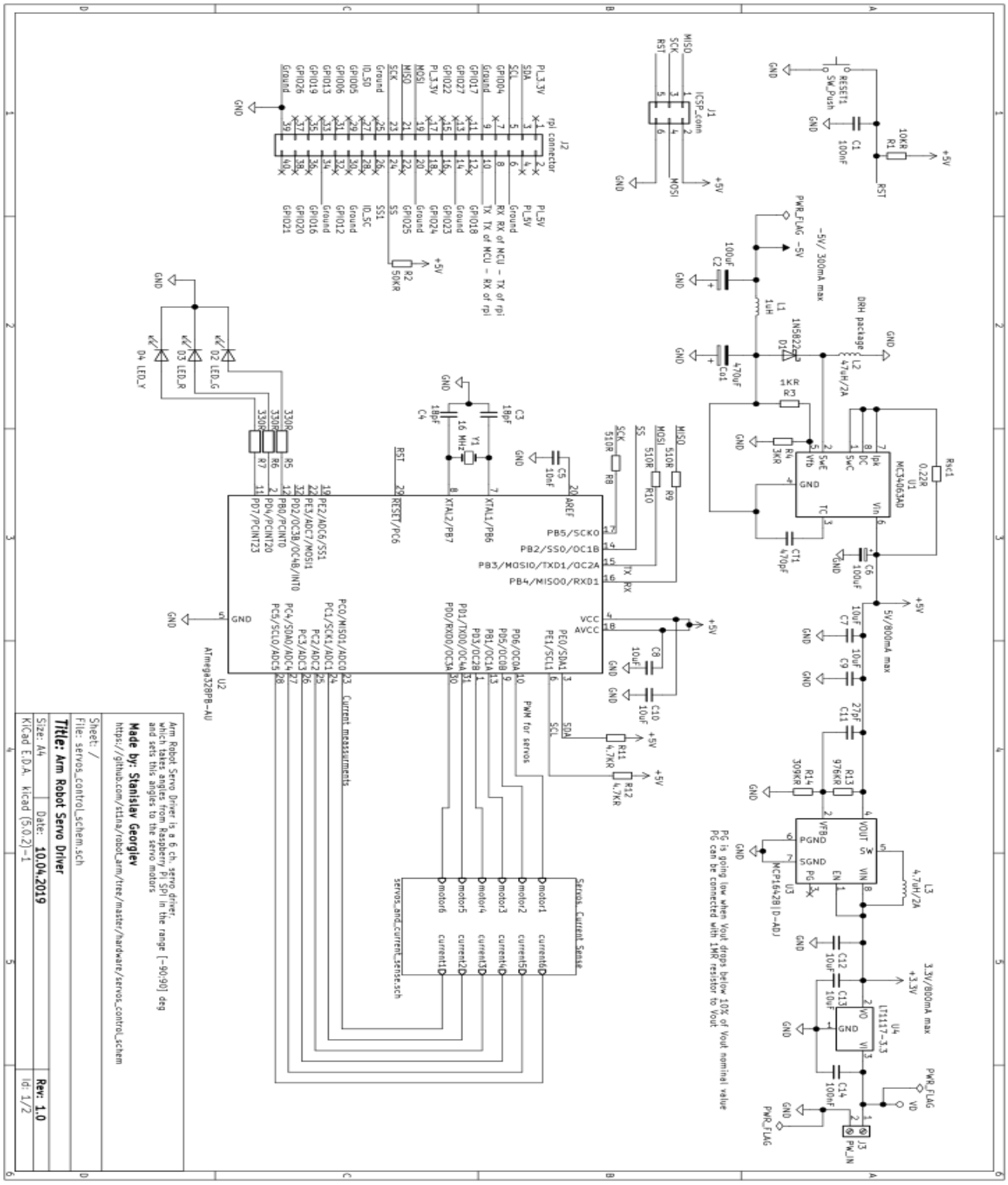
За целта може да се използва Arduino но тук ще покажем как да се реализира платка за управление на моторите при получаване на задание от SPI.

5.2.1. Hardware

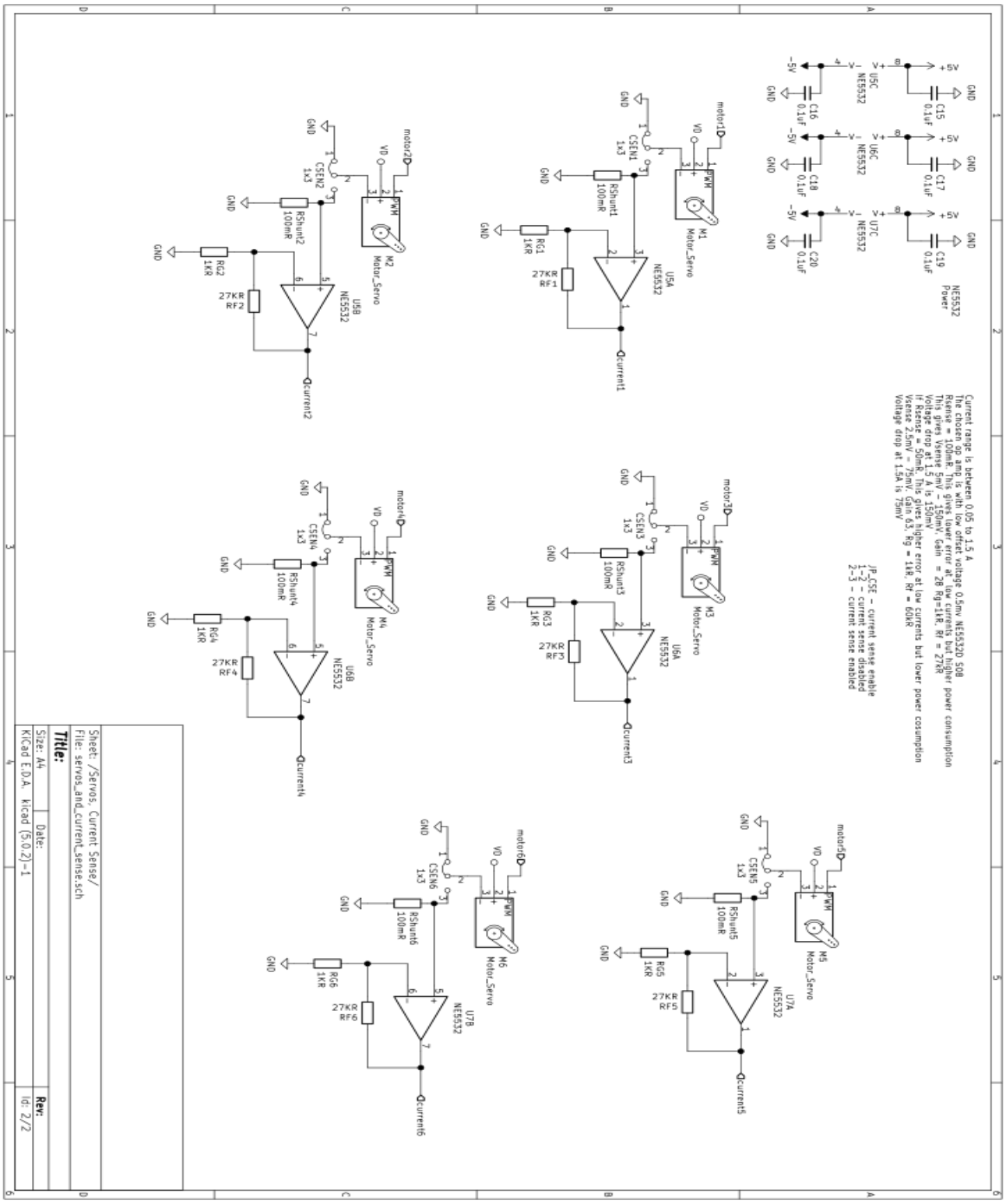
Използваната платка е показана на фиг. 5.3., тук няма да навлизаме в подробности относно изграждането на тази платка. На фиг. 5.4. е показана схемата за тази платка, основната и функция както беше споменато е да получи ъглите по SPI и да ги зададе на моторите. След малко ще разгледаме софтуера за нея. Както може да се види има и възможност за измерване на тока на всяко серво, но тази функционалност все още не е осъществена в софтуера.



Фиг. 5.3. Servo Driver



Фиг. 5.4.а. Схема на Servo Driver



Фиг. 5.4.b. Схема на Servo Driver Current Sense

Схемата заедно с PCB чертежа и gerber файловете за изработване на платката могат да бъдат намерени тук https://github.com/st1na/robot_arm/tree/master/hardware/servos_control_schem . Софтуера за изчертаване на схемата и платката е Kicad 5.0.2-1.

5.2.2. Software

Използвания микроконтролер е Atmel Mega328PB, за програмирането му се използва AVR ISP MKII програматор и Atmel Studio 7. Програмата и source кода който ще ни трябват могат да бъдат намерени на https://github.com/st1na/robot_arm/tree/master/software/robot_arm_driver_rev1_0. Тук ще разгледаме как става самото преобразуване на ъглите и задаването им за изпълнение. Първо ще бъде разгледано как се получават ъглите от SPI и преобразуват в интервала [0°; 180°], след това как се преобразуват в необходимия сигнал, който разгледахме в 5.1, за управление на серво моторите.

```
/*
 * robot_arm_driver_rev1_0.c
 *
 * Created: 05/04/2019 14:07:21
 * Author : st1na
 */

#include <avr/io.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <avr/wdt.h>
#include "Communication.h"
#include "ServoDriver.h"
//#include "Debug.h"

int main(void)
{

	cli();
	wdt_reset();
	wdt_disable();

	uint8_t joints[NUM_JOINTS] = {90, 90, 90, 90, 90, 0};
	initSPISlave();
	//initUART();
	initPWM();

	sei();
	setServoAngles(joints);
    while (1)
    {
		if( checkSPI_status() ){
			getJointsFromSPI(joints);
			setServoAngles(joints);
			clearSPI();
		}
    }
}
```

От горния код се вижда, че първо трябва да се инициализират SPI и ШИМ който ще използваме след инициализацията започва един безкраен цикъл който с помощта на функцията `checkSPI_status()` проверява дали са получени всички ъгли след това ако са получини се извикват функцията `getJointsFromSPI(joints)` която взима ъглите и ги записва в буфер след което с функцията `setServoAngles(joints)` ъглите се задават за изпълнение.

Communication

Тук ще разгледаме как става самото получаване на ъглите и преобразуването им към обхвата [0°, 180°]. Подробности относно регистрите на SPI може да намерите в спецификацията на микроконтролера Atmega 328PB.

След като инициализираме SPI в slave режим на работа с помощта на функцията `initSPISlave()` трябва да създадем функция за прекъсването което сме инициализирали.

```
void initSPISlave(void){
	// Set MISO ouptut, others INPUT
	DDR_SPI |= (1<<MISO);
	DDR_SPI &= (~(1 << MOSI)) |  (~(1 << SCK )) | (~(1 << SS));
	// Enable SPI and interrupt
	SPCR0 = (1<<SPE) | (1<<SPIE);
	process_joints = false;
}
```

Функцията на прекъстането ще се изпълни при всяко получаване на данни от master устройството. Данните са с големина 8 бита и се получават в регистър SPDR0. Прекъсването се обслужва от функцията `ISR(SPI0_STC_vect)`.

```
ISR(SPI0_STC_vect) {
	int8_t joint = SPDR0;
	if(joint_number < sizeof(joints_buffer)){
		// Save joints in buffer
		joints_buffer[joint_number++] = joint;
		// Check if all joint are send
		if(joint == LAST_JOINT_SEND){
			process_joints = true;
		}
	}
}
```

След като получим един от ъглите (както вече споменахме в глава 4.1. ъглите се получават като signed integers с големина 8 бита) го записваме в буфер който съдържа ъглите в интервала[-90°,90°] (joints_buffer) и проверяваме дали всички ъгли са изпратени, това става с флага `LAST_JOINT_SEND=100` който дефинирахме в глава 4.1, ако получената дума е равна на флага(100) , задаваме стойност на флага `process_joints = true`. При което функцията която ползваме за получаване на състоянието

```
bool checkSPI_status(void){
	return process_joints;
}
```

също връща true, при което се изпълнява условието `if( checkSPI_status() )` в безкрайния цикъл и се извиква функция `getJointsFromSPI(joints)`. Тази функция приема като аргумент буфер в който да запише ъглите който ще се използват от функцията `setServoAngles(joints)`.

```
void getJointsFromSPI(uint8_t* joints){
	mapJointsFromSPI();
	for(int i=0; i<NUM_JOINTS; i++){
		joints[i] = u_joints_buffer[i];
	}
}
```

Задачата на тази функция е първо да преобразува ъглите от интервала [-90°; 90°] към интервала [0°; 180°], това става с извикването на функцията `mapJointsFromSPI()`.

```
void mapJointsFromSPI(void){
	int i;
	for(i=0; i<NUM_JOINTS; i++){
	// The sixth joint is the gripper
		if(i == 5){
			u_joints_buffer[i] = joints_buffer[i];
		}
		else{
			// Map joints to [0, 180]
			u_joints_buffer[i] = joints_buffer[i] + 90;
		}
	}
	// Second and fourth joint are reversed
	u_joints_buffer[1] = 180 - u_joints_buffer[1];
	u_joints_buffer[3] = 180 - u_joints_buffer[3];
}
```

Тази функция зависи от конструктивните особености на робота. Преобразуваните от нея ъгли се записват в буфера `u_joints_buffer`.

След като се изпълни функцията записваме ъглите в буфера joints.

```
	for(int i=0; i<NUM_JOINTS; i++){
		joints[i] = u_joints_buffer[i];
	}
```

Servo Driver

След като `getJointsFromSPI(joints)` приключи се извиква функцията `setServoAngles(joints)`. Преди да можем да използваме тази функция трябва да сме инициализирали ШИМ изходите това става с функцията `initPWM()`. Тъй като Atmega328PB разполага само с три 16 битови таймера можем да използваме хардуерна ШИМ останалите три ШИМ сигнала ще бъдат генерирани софтуерно с помощта на прекъсване и 8 битов таймер. В инициализиращата функция инициализираме трите 16 битови таймера като ШИМ изходи с период на ШИМ 20 ms, това става с регистри ICRx. Тъй като сме задали, че таймера се увеличава с едно на

всеки 0.5 µs за да получим период от 20ms(20000µs), в регистри ICRx трябва да зададем 40000. След което коефициента на запълване се задава в регистри OCRxA, като в header файла ServoDriver.h сме направили макроси за тях. Коефициента на запълване за трите мотора който използват хардуерна ШИМ (мотор 3, 5 и 6) се задават като число между 250 и 1250 в регистри OCRxA което отговаря на микросекунди между 500 и 2500.

```
#define servo3_us OCR1A
#define servo5_us OCR4A
#define servo6_us OCR3A
```

За останалите три ШИМ сигнала инициализираме 8 битов таймер така, че да създава прекъсване на всеки 16.4 ms.

```
void initPWM(){
	// Motor 1,2,4 will use software PWM

	// Set PWM fast mode period in ICP
	TCCR1A |= (1<<WGM11);
	TCCR1B |= (1<<WGM13) | (1<<WGM12);
	TCCR3A |= (1<<WGM31);
	TCCR3B |= (1<<WGM33) | (1<<WGM32);
	TCCR4A |= (1<<WGM41);
	TCCR4B |= (1<<WGM43) | (1<<WGM42);

	// Set clock at 2MHz counting by 0.5 us every tick
	TCCR1B |= (1<<CS11);
	TCCR3B |= (1<<CS31);
	TCCR4B |= (1<<CS41);

	// Set period at 20ms ICP = 20000us = 40000
	ICR1 = 40000;
	ICR3 = 40000;
	ICR4 = 40000;

	// Output on OC1A, OC3A, OC4A
	TCCR1A |= (1<<COM1A1);
	TCCR3A |= (1<<COM3A1);
	TCCR4A |= (1<<COM4A1);

	DDR_MOTOR  |= (1<<MOTOR5) | (1<<MOTOR6);
	DDR_MOTOR3 |= (1<<MOTOR3);

	// Init 20ms Timer to interrupt
	TCCR0B |= (1<<CS00) | (1<<CS02);
	TIMSK0 |= (1<<TOIE0);

	DDR_MOTOR  |= (1<<MOTOR1) | (1<<MOTOR2) | (1<<MOTOR4);

}
```

Прекъсването задава периода на ШИМ сигнала а коефициента на запълване се задава в обслужващата прекъсването функция `ISR(TIMER0_OVF_vect)` , с помощта на функцията `_delay_us(double val)`. Тук коефициента на запълване се задава като число между 500 и 2500 което съответства на микросекунди между 500 и 2500 съответно. Тъй като функция `_delay_us` е обикновенно закъснение при което процесора не прави нищо, за да намалим това време задаваме коефициентите на запълване така, че първо задаваме високо ниво на изходите който ще управляват моторите 1, 2 и 4 `PORT_MOTOR |= (1<<MOTOR1) | (1<<MOTOR2) | (1<<MOTOR4)`. След като изтече минималното закъснение(това е времето в μs съответно на коефициента на запълване на ШИМ с минимален коефициент на запълване от мотори 1,2 и 4)

```
_delay_us(servo_delay.min_delay)
```

от тези три времена задаваме ниско ниво на изхода отговарящ на мотора чийто ШИМ сигнал е с коефициент на запълване с минимална продължителност

```
PORT_MOTOR&= ~(1<<servo_delay.min_delay_motor) .
```

Изчакваме да изтече разликата от времената между коефициента на запълване със средна продължителност и коефициента на запълване със минимална продължителност

```
_delay_us( servo_delay.mid_delay - servo_delay.min_delay )
```

и задаваме ниско ниво на изхода съответстващ на мотора чийто ШИМ сигнал е с коефициент на запълване със средна продължителност

```
PORT_MOTOR &= ~(1<<servo_delay.mid_delay_motor);.
```

Изчаква се да изтече разликата между коефициента на запълване с максимална продължителност и коефициента на запълване със средна продължителност

```
_delay_us( servo_delay.max_delay - servo_delay.mid_delay  )
```

и задаваме ниско ниво на изхода отговарящ на мотора чийто ШИМ сигнал е с коефициент на запълване с максимална продължителност измежду ШИМ сигналите на мотори 1,2 и 4

```
PORT_MOTOR &= ~(1<<servo_delay.max_delay_motor).
```

Цялата функция обслужваща прекъсването е:

```
ISR(TIMER0_OVF_vect){
      sei();
      PORT_MOTOR |= (1<<MOTOR1) | (1<<MOTOR2) | (1<<MOTOR4);
      _delay_us(servo_delay.min_delay);
      PORT_MOTOR &= ~(1<<servo_delay.min_delay_motor);
      _delay_us( servo_delay.mid_delay - servo_delay.min_delay );
      PORT_MOTOR &= ~(1<<servo_delay.mid_delay_motor);
      _delay_us( servo_delay.max_delay - servo_delay.mid_delay  );
      PORT_MOTOR &= ~(1<<servo_delay.max_delay_motor);
      TCNT0 = 0x00;
}
```

Функцията определяща ШИМ сигнала на кой от моторите е с минимално закъснение е:

```
void delay_determine(){
      // Find minimum, maximum and middle delay

      if (servo1_us > servo2_us){
            if(servo1_us > servo4_us){
                  servo_delay.max_delay = servo1_us;
                  servo_delay.max_delay_motor = MOTOR1;
                  if(servo4_us > servo2_us){
                        servo_delay.mid_delay = servo4_us;
                        servo_delay.min_delay = servo2_us;
                        servo_delay.mid_delay_motor = MOTOR4;
                        servo_delay.min_delay_motor = MOTOR2;
                  }
                  else{
                        servo_delay.mid_delay = servo2_us;
                        servo_delay.min_delay = servo4_us;
                        servo_delay.mid_delay_motor = MOTOR2;
                        servo_delay.min_delay_motor = MOTOR4;
                  }
            }
            else{
                  servo_delay.max_delay = servo4_us;
                  servo_delay.mid_delay = servo1_us;
                  servo_delay.min_delay = servo2_us;

                  servo_delay.max_delay_motor = MOTOR4;
                  servo_delay.mid_delay_motor = MOTOR1;
                  servo_delay.min_delay_motor = MOTOR2;
            }
      }
      else if(servo2_us > servo4_us){
            servo_delay.max_delay = servo2_us;
            servo_delay.max_delay_motor = MOTOR2;
            if(servo1_us > servo4_us){
                  servo_delay.mid_delay = servo1_us;
                  servo_delay.min_delay = servo4_us;
                  servo_delay.mid_delay_motor = MOTOR1;
                  servo_delay.min_delay_motor = MOTOR4;
            }
            else{
                  servo_delay.mid_delay = servo4_us;
                  servo_delay.min_delay = servo1_us;
                  servo_delay.mid_delay_motor = MOTOR4;
                  servo_delay.min_delay_motor = MOTOR1;
            }
      }
      else{
            servo_delay.max_delay = servo4_us;
            servo_delay.mid_delay = servo2_us;
            servo_delay.min_delay = servo1_us;
            servo_delay.max_delay_motor = MOTOR4;
            servo_delay.mid_delay_motor = MOTOR2;
            servo_delay.min_delay_motor = MOTOR1;
      }
}
```

След като вече видяхме как се задават управляващите сигнали на моторите можем да разгледаме функцията `setServoAngles(joints)` която се изпълнява след като се получат и шестте ъгъла на завъртане от функцията `getJointsFromSPI(joints)`.

```
void setServoAngles(uint8_t* angles){

	uint16_t angles_in_ms[NUM_JOINTS];

	// Map joint angles from [0; 180] to [500; 2500]
	// except for the gripper he is mapped to [600; 1200]
	anglesToMiliSec(angles, angles_in_ms);

	for(int i=0; i<NUM_JOINTS-1; i++){
		if(angles_in_ms[i] >2500){
			angles_in_ms[i] = 2500;
		}
	}

	delay_determine();

	servo1_us = (double) (angles_in_ms[0]);
	servo2_us = (double) angles_in_ms[1];
	servo3_us = angles_in_ms[2]*2;
	servo4_us = (double) angles_in_ms[3];
	servo5_us = angles_in_ms[4]*2;
	servo6_us = angles_in_ms[5]*2;

}
```

Тази функция първо преобразува ъглите от интервала [0°; 180°] в интервала [500ms; 2450ms] с помощта на функцията `anglesToMiliSec(angles, angles_in_ms)` само изпълнителното устройство се преобразува в интервала [600ms; 1200ms]. След което се извиква функцията която определя закъсненията за моторите които се управляват със софтуерна ШИМ модулация `delay_determine()` , тази функция беше показана по-горе. След приключване на тази функция ъглите се задават с помощта на макросите `servoX_us (X=1:6)`.

```
	servo1_us = (double) (angles_in_ms[0]);
	servo2_us = (double) angles_in_ms[1];
	servo3_us = angles_in_ms[2]*2;
	servo4_us = (double) angles_in_ms[3];
	servo5_us = angles_in_ms[4]*2;
	servo6_us = angles_in_ms[5]*2;
```

За да преобразуваме стойност x от интервала $[a,b]$ към стойност y в интервала $[c,d]$ използваме формулата.

$$y = (x - a)\frac{d - c}{b - a} + c$$

Функцията за преобразуване на ъглите е показана на следващата страница.

```
void anglesToMiliSec(uint8_t* servo_angles, uint16_t* servo_angles_ms){
	float min_ms = 500;
	float max_ms = 2450;
	float min_ms_gripper = 600;
	float max_ms_gripper = 1200;

	float angles_min = 0;
	float angles_max = 180;

	uint8_t angle_num = 0;
	float temp_angle = 0.0;

	// Mapping arm
	for(angle_num=0; angle_num< ( NUM_JOINTS - 1); angle_num++){
		temp_angle = servo_angles[angle_num];
		// Map to 0-1
		temp_angle = (temp_angle - angles_min)/(angles_max - angles_min);
		// Map to 0-max_ms
		temp_angle = temp_angle*(max_ms  - min_ms);
		// map to min_ms-max_ms
		temp_angle = temp_angle + min_ms;
		servo_angles_ms[angle_num] = (uint16_t) temp_angle;
	}

	// Gripper mapping
	// Take the last angle from the array
	temp_angle = servo_angles[GRIPPER_JOINT-1];
	// Map to 0-1
	temp_angle = (temp_angle - angles_min)/(angles_max - angles_min);
	// Map to 0-max_ms_gripper
	temp_angle = temp_angle*(max_ms_gripper  - min_ms_gripper);
	// map to min_ms_gripper-max_ms_gripper
	temp_angle = temp_angle + min_ms_gripper;
	servo_angles_ms[GRIPPER_JOINT - 1] = (uint16_t) temp_angle;
}
```

Заключение

С помощта на дипломната работа разгледахме как можем да реализираме управление за 5 звенен робот манипулатор тип ръка. Чрез използване на ROS може да се реализира управление за много видове роботи. Видяхме каква е последователността при изграждане на управление за даден робот, в нашия случай робота с 5 степени на свобода показан в увода. Разбира се някой от разглежданията са тясно свързани с конструкцията и изпълнителните механизми на робота, но като идея последователността е същата.

Първо се изгражда модел на робота, което както вече видяхме не е сложна задача за роботи с проста конструкция, но за роботи с по-сложна конструкции може да се използва CAD софтуер за проектиране на робота, като например SolidWorks, след което модела да се експортира към URDF файл, но това не е въпрос на нашите разглеждания и няма да навлизаме в подробности за заинтересованите разгледайте http://wiki.ros.org/sw_urdf_exporter, където е обяснено как от SolidWorks модел експортираме URDF.

След като създадохме модела, го конфигурирахме с помощта на Moveit Setup Assistant което както видяхме ни предостави интерфейс с помощта на който можем да пишем програми за управление на робота. Видяхме как с помощта на предоставения python интерфейс се пише програма за задаване на позиция на робота и изпълнение на тази позиция. Разгледахме симулационната среда в помощта на която можем да тестване алгоритми преди да ги използваме в реалния робот.

За реалната реализация на робота използвахме Raspberry Pi, с помощта на който реализирахме управлението на реалния робот. За управлението на изпълнителните механизми (серво моторите) решихме да реализираме драйвер който да получава задание (ъгли на завъртане за всеки мотор) и да управлява моторите. Така при промяна на типа на моторите трябва да се смени само драйвера а алгоритмите за управление, и управляващия модул могат да останат непроменени.

Използвана Литература

1. Bruno Siciliano, Oussama Khatib, “Springer Handbook of Robotics”, ISBN: 978-3-319-32550-7
2. https://en.wikipedia.org/wiki/Euler_angles
3. https://en.wikipedia.org/wiki/Denavit%E2%80%93Hartenberg_parameters
4. https://moveit.ros.org/documentation/concepts/
5. https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all
6. https://learn.sparkfun.com/tutorials/raspberry-gpio/all
7. https://en.wikipedia.org/wiki/Servo_control
8. Elliot Williams, “Make: AVR Programming”, Published by Maker Media, Inc., ISBN: 978-1-449-35578-4
9. https://robotacademy.net.au/
10. Morgan Quigley, Brian Gerkey, and William D. Smart, “Programming Robots with ROS”, Published by O’Reilly Media, Inc, ISBN: 978-1-4493-2389-9
11. Peter Wilson, “The Circuit Designer’s Companion”, ISBN: 978-0-08-097138-4
12. https://ros-planning.github.io/moveit_tutorials/
13. Peter Corke, “Robotics Toolbox for Matlab”
14. http://petercorke.com/wordpress/toolboxes/robotics-toolbox

Приложение 1

Инсталиране на ROS

Документацията на ROS има подробно описание на това как се инсталира ROS http://wiki.ros.org/melodic/Installation/Ubuntu ние просто ще обобщим процеса на инсталиране . След което ще покажем инсталирането на нужните пакети и зависимости за нашето приложение.
Първо стартираме тези команди:

sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116

След което ако има нужда актуализираме системата.

sudo apt-get update

sudo apt-get upgrade

След това стартираме инсталацията.

sudo apt install ros-melodic-desktop-full

След като инсталацията приключи инициализираме rosdep.

sudo rosdep init

rosdep update

Настройка на средата.

echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc

source ~/.bashrc

След което остава да инсталираме само зависимостите които са необходими за изграждане на пакети.

sudo apt install python-rosinstall python-rosinstall-generator python-wstool build-essential

Инсталиране на catkin.

sudo apt-get install ros-melodic-catkin python-catkin-tools

Инсталиране на Moveit

Официалната документация на ROS Moveit има добро описание на инсталацията на Moveit, https://ros-planning.github.io/moveit_tutorials/doc/getting_started/getting_started.html тук просто ще го повторим.

Актуализация на ROS и системата

rosdep update

sudo apt-get update

sudo apt-get dist-upgrade

Инсталиране на Moveit.

sudo apt install ros-melodic-moveit

След като се инсталира Moveit изграждаме примерните пакети за робот panda, което ще инсталира всички необходими зависимости.

mkdir -p ~/ws_moveit/src
cd ~/ws_moveit/src

Изтегляне на примерните пакети.

git clone https://github.com/ros-planning/moveit_tutorials.git -b melodic-devel
git clone https://github.com/ros-planning/panda_moveit_config.git -b melodic-devel

Инсталация на зависимостите.

rosdep install -y --from-paths . --ignore-src --rosdistro melodic

Изграждане на пакетите.

cd ~/ws_moveit
catkin config --extend /opt/ros/${ROS_DISTRO} --cmake-args -DCMAKE_BUILD_TYPE=Release
catkin build -j1

Приложение 2

Raspberry Pi Ubuntu 18.04 инсталиране и конфигурация.

След като изтеглим Ubuntu Mate 18.04 от тук https://ubuntu-mate.org/download/ (тук ще използваме 32 битовата версия) резархивираме архива и записваме img файла на карта с памет. Записването може да стане по много начини тук ще използваме **dd** командата. За целта отваряме терминал в директорията в която се намира img файла и стартираме командата

sudo dd bs=4M if=ubuntu-mate-18.04.2-beta1-desktop-armhf+raspi-ext4.img of=/dev/mmcblk0

При невнимание при използване на тази команда може да се изтрие погрешното съхраняващо устройство(например диска на който се намира операционната система).

След като командата приключи слагаме картата с памет в Raspberry Pi и следваме инсталационните инструкции. След като инсталацията приключи можем да премахнем стартирането на графичния интерфейс при стартиране на системата с командата

sudo systemctl disable lightdm.service.

След това трябва да конфигурираме SSH. За използване на SSH при тази конфигурация потребителското PC и Raspberry Pi трябва да се намират в една и съща локална мрежа. Разрешаването на SSH става след като стартираме командата **sudo raspi-config** избираме меню **Interfacing Options** и от там избираме **SSH**. След което стартиртираме командите

sudo rm -r /etc/ssh/ssh*key

sudo dpkg-reconfigure openssh-server

След това можем да използваме Raspberry Pi от друг компютър в същата локална мрежа с помощта на командата

ssh “username”@”hostname”.local

username – е името което избрахме при инсталация

hostname – по подразбиране при Ubuntu е username-desktop

hostname може да се види с командата **hostname** а username с командата **whoami**. Например ако сме задали username = robotuser командата ще изглежда по следния начин

ssh robotuser@robotuser-desktop.local

По подразбиране конфигурацията на SSH сървъра не е много сигурна и препоръчително да се променят някои настройка, но това не е предмет на нашите разглеждания, ако имате желание да промените тези настройки разгледайте https://linux-audit.com/audit-and-harden-your-ssh-configuration/ и https://www.ssh.com/ssh/sshd_config/.

Разрешаването на SPI интерфейса става като отново стартираме **sudo raspi-config** избираме меню **Interfacing Options** и от там избираме **SPI.**