INTRODUCTION TO MATLAB

PRISMA Lab
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione
Università degli Studi di Napoli Federico II
www.prisma.unina.it

Introduction / 1

- The MATLAB program originated primarily as a program intended for the management of matrices.
- Subsequent versions have been completed with a series of functions that allow the most complex numerical analyzes, suitable, for example, for the analysis and solution of control problems

- Installation instruction: https://ch.mathworks.com/help/install/ug/install-products-with-internet-connection.html

Introduction / 2

- The MATLAB command line is indicated by a prompt as in DOS: >>.
- It accepts declarations of variables, expressions, and calls to all functions available in the program.
- All MATLAB functions are nothing more than text files, similar to those that the user can generate with a text editor, and are performed simply by typing their name on the command line.
- MATLAB also allows you to recall the last lines of commands entered using the up and down arrows

- MATLAB Help
  - MATLAB presents an online help with information on the syntax of all available functions.
  - To access this information, just type :
    - *help function_name*
  - It is also possible to have a help of all the functions of a certain category; for example, to find out what are the specific functions for robotics systems, just type:
    - `help robotics`
  - To find out what are the various categories of functions available (the so-called toolboxes), just type:
    - *help*
  - To access the help home page, type :
    - *doc*

Introduction / 4

- The files interpreted by the program are ASCII text files with the extension .m; they are generated with a text editor and are executed in MATLAB simply by typing their name on the command line (without extension!)
- You can insert comments in them by preceding each comment line with a percent `%`
- *Attention! It can be very useful to go to the directories where the program is located and analyze how the various functions have been implemented. This is possible because each MATLAB function and command calls a file* `.m`

Punctuation and variables/ 1

- The instructions (whether contained in a `.m` file launched by MATLAB or typed directly from the command line) must always be terminated with a semicolon, otherwise, the result of the instruction application is displayed
  - Ex: `var1=6;`
  - Ex: `var2=linspace(-10,10,10000);`

Punctuation and variables/ 2

- To view the contents of a variable, simply type its name without a semicolon on the command line
- All calculations performed in MATLAB are performed in double precision but can be viewed in a different format using commands:
  - `format short` Fixed point with 4 decimals
  - `format long` Fixed point with 15 decimals
  - `format short e` Scientific notation with 4 decimals
  - `format long e` Scientific notation with 15 decimals

  The result of the last operation is stored in the variable `ans`.

Scalar operators

- The available operators are:
  - `+, -, *, /, ^`
  - `sin, cos, tan` (rad)
  - `sind, cosd, tand` (deg)
  - `asin, acos, atan`
  - `exp, log` (natural), `log10` (base 10)
  - `abs, sqrt, sign`

Complex numbers

- The default complex units are `i` or `j`
  - Do not use `i` or `j` as variables or loop indices
  - A complex number is written as `a+j*b`
  - Ex: `z=2+j*3`
- Operators applicable to complex numbers are:
  - `abs` : absolute value, ex. `abs(z)`
  - `angle` : phase, ex. `angle(z)`
  - `real` : real part, ex. `real(z)`
  - `imag` : imaginary part, ex. `imag(z)`

Matrices and related operators / 1

- The insertion of a vector or a matrix, in general, is carried out in square brackets, separating the elements of the lines with spaces or commas, and the different lines with semicolons (or by going to the head to each new line)
  - Ex. of raw vector: `x = [1, 2, 3];`
  - Ex. of column vector: `y = [1; 4; 7];`
  - Ex. of matrix: `A = [1 2 3; 4 5 6; 7 8 9];`
    or:
```
A = [1 2 3
     4 5 6
     7 8 9 ];
```

Matrices and related operators / 2

- To refer to the elements of a matrix A
  - a_{mn} element is addressed as `A(m,n);`
    - ex. `A(2,3)` is 6
  - the m-th row is addressed as A (m, :), where all columns are indicated with a colon;
    - ex. `A(2,:)` is [4 5 6]
  - the n-th column is addressed as A (:, n), where all the rows are indicated with a colon;
    - ex. `A(:,3)` is [3; 6; 9]
  - the submatrix having elements a_{mn}, with $m_1<m<m_2$ and $n_1<n<n_2$, is addressed as A (m1: m2, n1: n2);
    - ex. `A(1:2,2:3)` is [2, 3; 5, 6]

Matrices and related operators / 3

- The operators applicable to matrices are
  - `+ - * ^ / \ '`
  - Left division: `A\B=inv(A)*B`
  - Right division: `B/A=B*inv(A)`

Matrices and related operators / 4

- Other functions essentially operating on vectors (row or column) are
  - `max, min`
  - `sort`
  - `sum, prod`
  - `median`
- There are also particular operators (`.*, ./, .^`) that allow you to perform operations on vectors element by element, without resorting to cycles. For example, if x is a vector, to multiply element by element the two vectors sin (x) and cos (x) just do
  - `y = sin(x) .* cos(x);`

Matrices and related operators / 5

- Other functions that operate essentially on matrices are
  - `inv`
  - `det`
  - `size`
  - `rank`
  - `eig`
- Attention: all functions that operate on matrices have constraints on the introduced operands. For example, a non-square matrix cannot be inverted. For further explanation on the syntax of the function use the command `help`

Matrices and related operators / 6

- There are also various predefined functions for creating matrices
  - `eye(n)` : `nxn` identity matrix
  - `zeros(m,n)` : `mxn` zeroes matrix
  - `ones(m,n)` : `mxn` ones matrix
  - `rand(m,n)` : `mxn` random values (between 0 and 1) matrix
  - `diag(X)`
    - if X is a vector with n elements, produces a diagonal square matrix of size `nxn` with the elements of X on the diagonal
    - if instead X is a square matrix of dimension `nxn`, it produces a vector of n elements equal to those on the diagonal of X

Matrices and related operators / 7

- The command : can be used to generate vectors
  - without specifying increment
    - ex. `t=1:5` => `t=[1 2 3 4 5]`
  - with specified positive increment
    - ex. `t=0:0.2:1` => `t=[0 0.2 0.4 0.6 0.8 1]`
  - with specified negative increment
    - ex. `t=2:-0.2:1` => `t=[2 1.8 1.6 1.4 1.2 1]`

Polynomials / 1

- Some functions that will be listed below require polynomials as input parameters
- MATLAB treats polynomials as particular row vectors, whose elements are the coefficients of the monomials of the polynomial in order of decreasing power
  - Ex. the polynomial $s^4+111s^3+1100s^2+1000s+4$ is represented as: `num=[1 111 1100 1000 4];`

Polynomials / 2

- The `conv` function multiplies two vectors and thus two polynomials
  - Ex. the polynomials product $(s^2+s+1)*(s^2+111s+1000)$ can be obtained via: `prod = conv([1 1 1],[1 111 1000])` which result is the vector `[1 112 1112 1111 1000]`

Polynomials / 3

- The `roots` function evaluates the roots of a given polynomial
  - Ex: `p = [1 3 5 2]; r = roots(p);`
    The roots of `p` are memorized within `r`
- The inverse function is `poly`
  - `pp = poly(r);`
  - The original polynomial `p` is restored in `pp`

Istruction IF – THEN - ELSE

- The general form of the IF-THEN-ELSE construct is the same as in any programming language
- ```
  if        condition1,
            operations1;
  elseif    conditition2,
            operations2;
  else
            operations3;
  end;
  ```
  ```

Istruction IF – THEN – ELSE / 2

- `Condition 1,2` must be conditions that return TRUE or FALSE as a result. The operators available for such comparisons are

  `<` , `>`

  `<=` , `>=`

  `==` ← equal

  `~=` ← different

  `&` ← logic and

  `|` ← logic or

  `~` ← logic not

Instructions for loops / 1

- Loops can be done with two different constructs :
  - `for k = 1:step:n,`
    `operations;`
    `end;`
- The loop performs its operations by incrementing the variable `k` from `1` to `n` with an increment equal to `step`

Instructions for loops / 2

- Or
  - ```
    while condition
     operations;
    end;
    ```
- The loop performs the operations until the condition is met. The condition is constructed with the same rules (constraints and operators) as that of the IF-THEN-ELSE.
- Warning: provide an initialization before the cycle that verifies the condition to allow the program to enter the cycle, and also insert in the operations something that can interact and therefore modify the condition, otherwise the cycle will be repeated indefinitely
    ```

Graphs / 1

- The `plot` function creates two-dimensional graphics: it receives two vectors of the same length as input and prints the points corresponding to the coordinates provided by the two vectors. For example, if you have two vectors `x` and `y`, the corresponding graph is obtained as
  - `plot(x,y);`

Graphs / 2

- To plot the graph of any function, it is, therefore, necessary to create a suitable vector to be used as abscissa, pass it to the function to obtain a vector containing the ordinates, and use the plot function on the two vectors thus obtained. For example to trace the function `sin(x)` between `-4` e `4` the following commands can be exploited
  - ```
    x=-4:0.01:4;
    y=sin(x);
    plot(x,y);
    ```
    ```

Graphs / 3

- To create graphs of different colors or using characters other than the point, you can specify a string of 2 elements after the coordinates. The first is the color of the graph, the second the symbol used to mark the points. Eg.
  - `plot(x,y,'g+');`
  - Create a graph in green using + instead of dots. This option can be used in cases of overlapping graphics to be printed (if the printer available is not in color and if you do not change the type of symbol, you no longer understand anything ...)
  - To print several graphs simultaneously on the same axes, then use the `hold on` command and continue with the various plots, or
  - `plot(x,y,'g+',x,z,'r-')`

Graphs / 4

- Possible choices

r	red	.	point
g	green	o	circle
b	blue	x	x-mark
w	white	+	plus
m	magenta	*	star
c	cyan	-	solid
y	yellow	:	dotted
k	black	--	dashed
		-.	dash-dot

- Other commands are
  - `grid` : overlaps a grid
  - `title` : adds a title
  - `xlabel` : adds a legend on the x axis
  - `ylabel` : adds a legend on the y axis
  - `axis` : set the scale for the axis
  - `clf` : cancel current graph
- The command `figure` create a new graphic window in which to display the drawing; to move to the n-th graphic window, just type
  - `figure(n)`

Graphs / 6

- To display multiple graphs on the same screen, but with different axes, you can use the function `subplot`. This function needs 3 parameters:
  - the first indicates how many vertical parts to divide the screen into,
  - the second in how many horizontal parts,
  - the third in which part to execute the next plot
    - ```
      subplot(2,1,1), plot(fun1);
      subplot(2,1,2), plot(fun2);
      ```
    - creates two windows divided by a horizontal line and displays the graph in the upper one `fun1` and in the lower one `fun2`
      ```

- The two functions that can be used to create equally spaced vectors or in any case typical for the abscissa axes are
  - `x = linspace(0.01,100,1000);`
  - `x = logspace(-2,2,1000);`
- `linspace` creates a vector `x` of `1000` linearly separated elements comprised between `0.01` and `100`
- `logspace` creates the same vector, with logaritmically separated elements. Note that the first two parameters are the exponents of the extremes of the interval expressed in base `10`

Definition of new functions / 1

- In MATLAB you can create new functions. Just create a file with the `.m` extension and file name equal to that of the desired function
- The first line of the file must contain the function name and the input and output arguments
  - `function z = fun1(a,b)`
  - `function [x,y] = fun2(a,b)`
  - It turns out that `fun1` and `fun2` are function names; `a` and `b` are entry points; `x`, `y` and `z` are output arguments

Definition of new functions / 2

- The block of consecutive comment lines that eventually follows the first line of the file is displayed by typing the help command followed by the name of the function created
- The variables used in a function are local and therefore independent from those of the calling environment
- You can also use global variables, as long as they are defined both in the calling environment and the function, using the `global` command followed by the names of the variables, separated by spaces
  - `global F G H`

Getting Started with Matlab

- A lot of tutorial videos are available on the Mathworks website
  - https://www.mathworks.com/videos/getting-started-with-matlab-68985.html

INTRODUCTION TO MATLAB: Robotics Toolbox

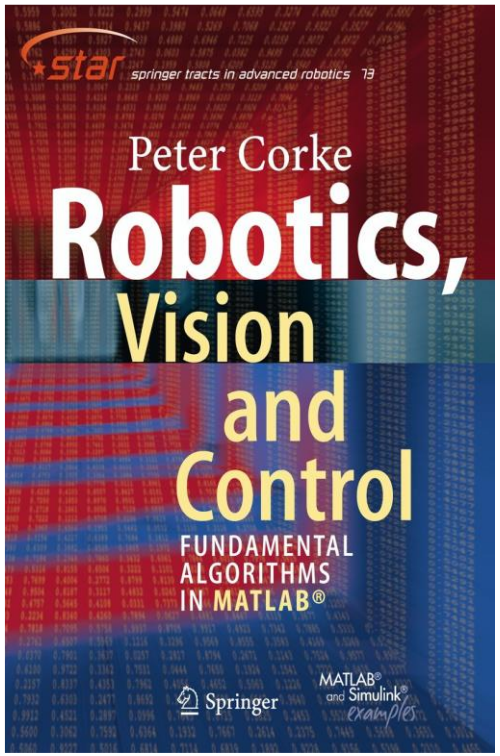
PRISMA Lab

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Università degli Studi di Napoli Federico II

www.prisma.unina.it

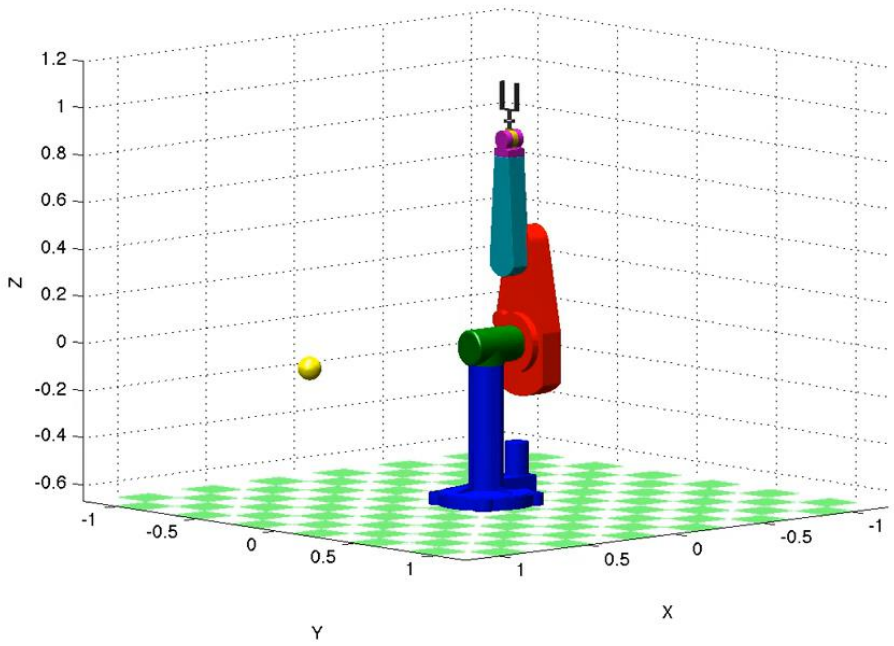
Introduction / 1

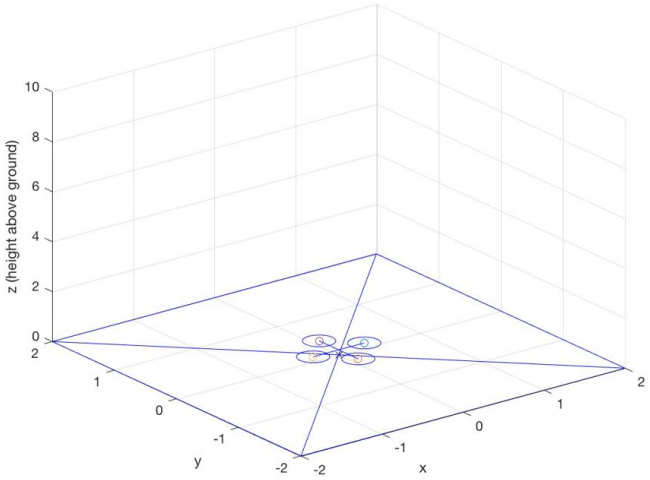


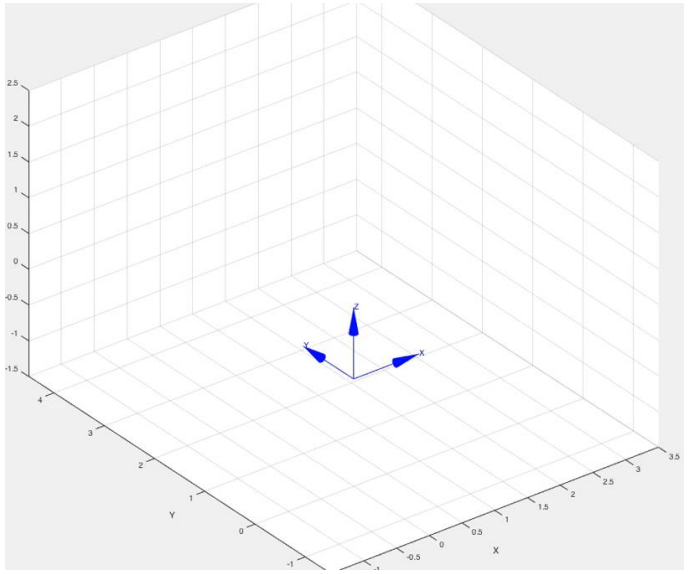
- An open-source MATLAB toolbox for robotics and machine vision.
- A collection of useful functions to study arm-type serial-link robot manipulators
  - Rotations, Translations, Transformations
    - matrices, quaternions, twists, triple angles, and matrix exponentials
  - Kinematics, Dynamics, Trajectory generation
  - 2D-3D Visualization and Simulation
- Mobile robots support
  - Unicycle, bicycle, drones, etc..
  - Path planning, kinodynamic planning (RRT)
  - Map building, SLAM
  - Non-holonomic vehicles
- Avaible source code

Introduction / 2

- **Some examples:**







Introduction / 3

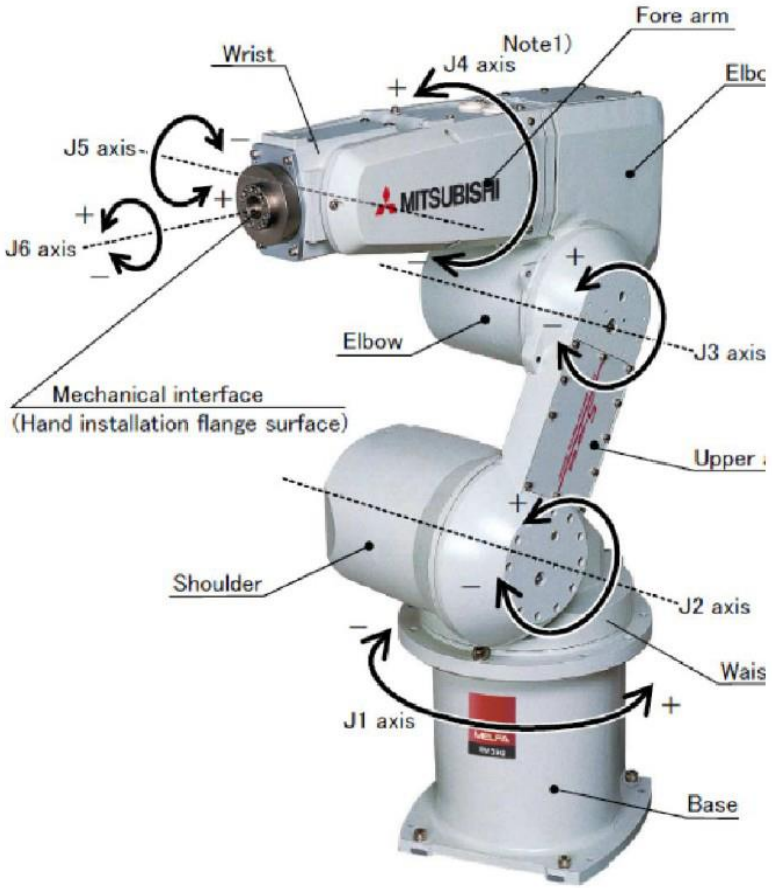
- **Where can I find it?**
  - Go to: https://petercorke.com/toolboxes/robotics-toolbox/
  - There are two versions of the Robotics Toolbox:
    - RTB9.10, the last in the 9th release is what is used in Robotics, Vision & Control (1st edition) and the Robot Academy.
    - RTB10.x is the current release and is used in Robotics, Vision & Control (2nd edition)
  - **Three** installation methods:
    - Direct access to a shared MATLAB Drive folder (for MATLAB19a onward)
    - Download a MATLAB Toolbox install file (.mltbx type), this is the latest version from GitHub
    - Clone the source files from GitHub

Robot modeling / 1

- RV3SD ROBOT

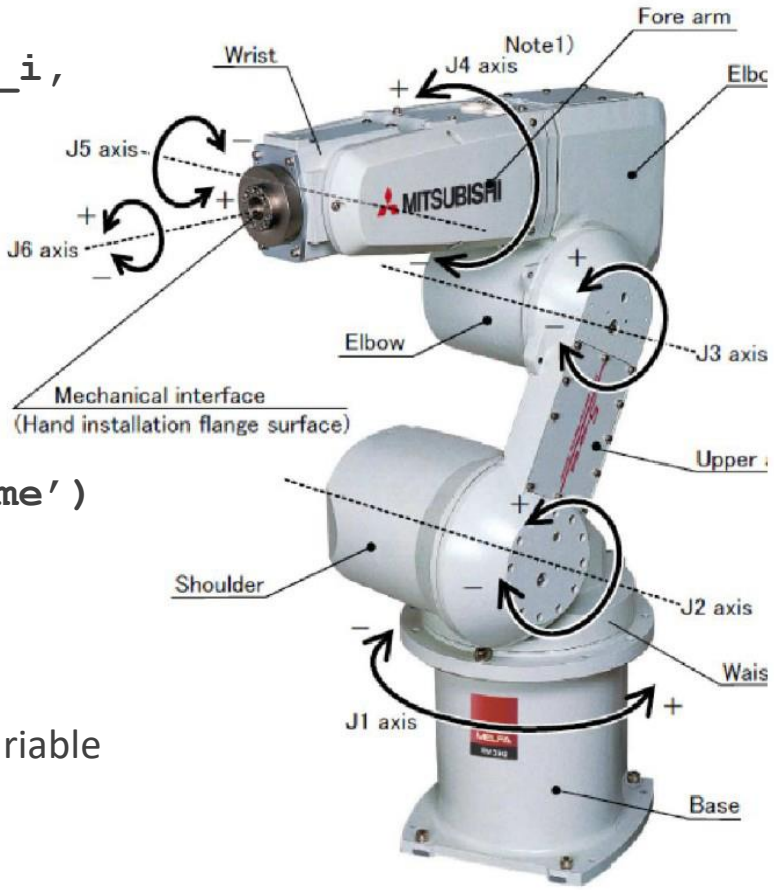
Joint no. *i*	Joint angle θ_i	Link offset d_i	Link length a_i	Link twist α_i	Joint
1	θ_1	d_1	a_1	-90°	Waist
2	θ_2	0	a_2	0°	Shoulder
3	θ_3	0	a_3	90°	Elbow
4	θ_4	d_4	0	-90°	Fore arm
5	θ_5	0	0	90°	Wrist
6	θ_6	d_6	0	0°	Tool

a1= 95, a2=245, a3=135, d1=350, d4= 270, d6=85 (all unit in mm)



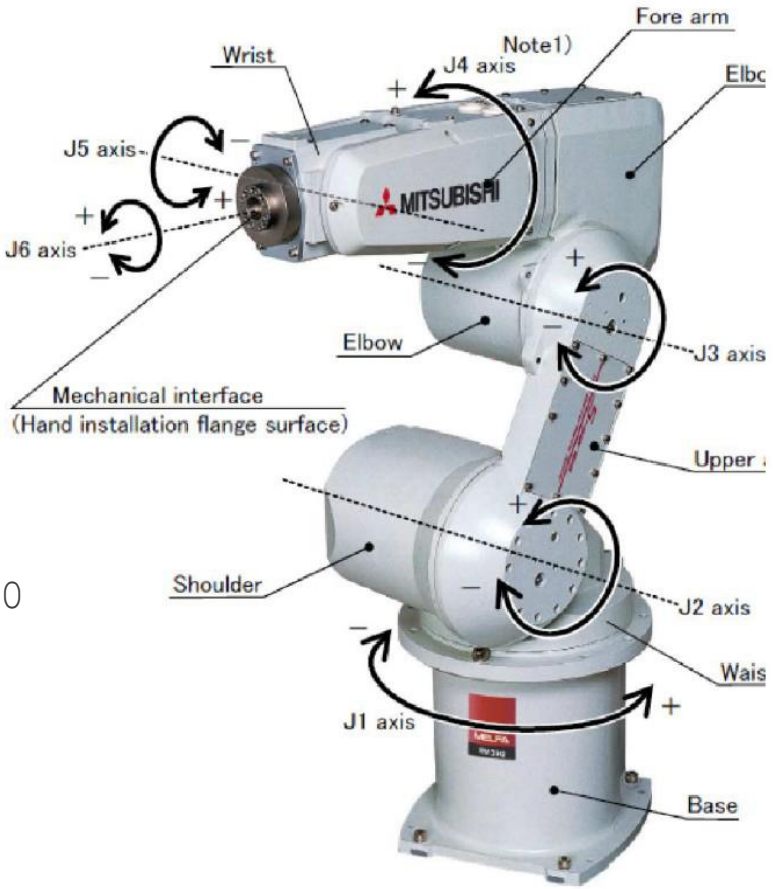
Robot modeling / 2

- Define DH params:
  - Create six links using link command
  - `L(i) = Link([theta_i, d_i, a_i, alpha_i, sigma_i])`
  - `theta_i, d_i, a_i, alpha_i`: DH params
  - `sigma_i`: joint type (`0` for revolute, `1` for prismatic)
- Define Robot:
  - Create robot using links defined.
  - `rv3sd = SerialLink([L1,..Ln],'name','robot_name')`
- Define Offset for the link:
  - Create offset for links based on requirement `L(2).offset = pi/2`
  - The joint offset is a constant added to the joint angle variable before forward kinematics and subtracted after inverse kinematics.



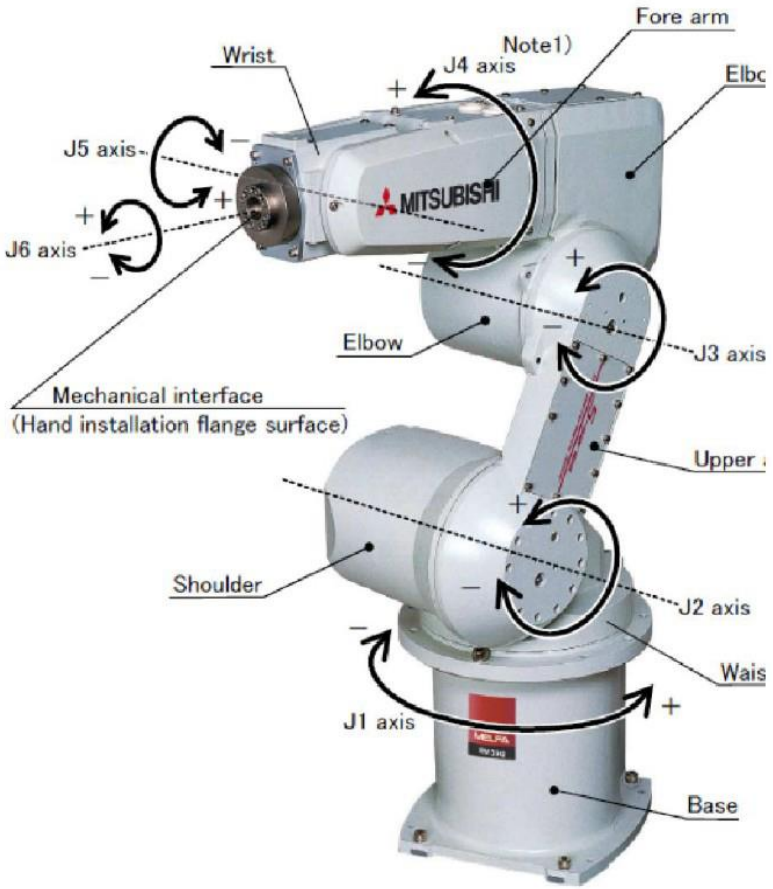
Robot modeling / 3

- Define end-effector position:
  - **`Ttool = transl([x y z])`**
  - **`rv3sd.tool = Tool_Position`**
- Define Base of the robot:
  - **`rv3sd.base = Tbase`**
- Define Robot limits:
  - To define limits of joint you use **`qlim`** command
  - **`L(2).qlim=[0 pi]`**
  - This will limit joint `L(2)` movement between `0` and `180` degrees
- Visualize robot:
  - **`rv3sd.plot([q1,q2,q3q4,q5,q6])`**



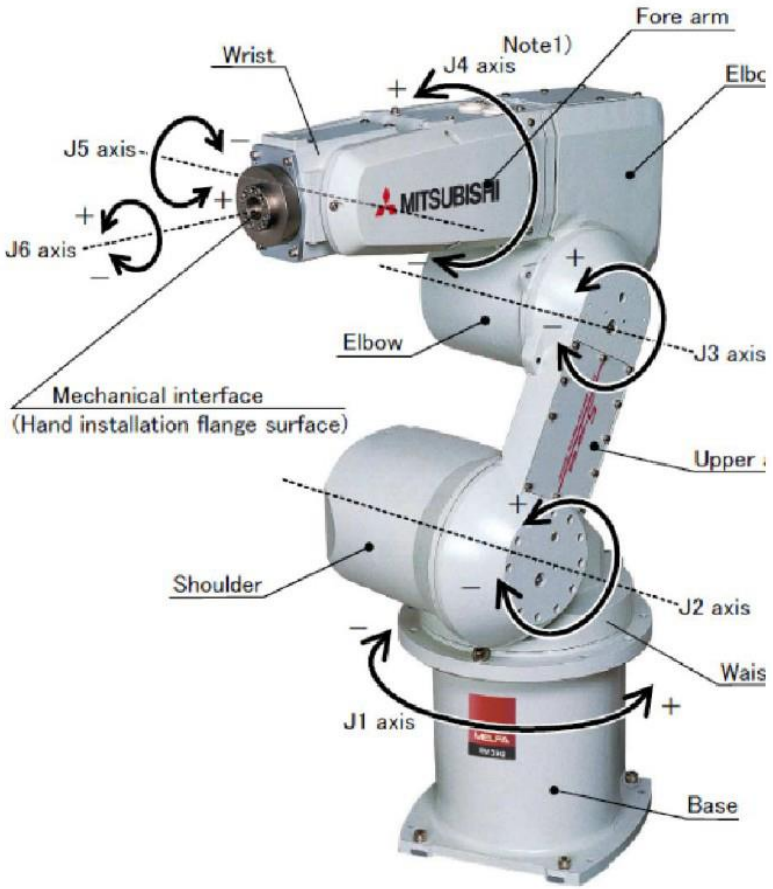
Robot modeling / 4

- **FORWARD KINEMATICS**
  - Joint-space coordinates are specified in a N-dimension vector
  - `fkine()` function
  - Joint space vector:
    ```
    Q = [0 -pi/6 -pi/6 0 pi/3 0]
    ```
  - Again, to visualize the robot:
    ```
    rv3sd.plot(Q)
    ```
  - Calculating transformation matrix:
    ```
    T = rv3sd.fkine(Q)
    ```



Functions/ 1

- **`transl()`**
  - This command converts the linear position vector in the relative transformation matrix
- **`rotx(), roty(), rotz()`**
  - Creates a rotation matrix (3x3) around certain axis
- **`trotx(), troty(), trotz()`**
  - Creates a transformation matrix (4x4) around certain axis
  - The last letter defines the axis
  - Radians: **`trotx(a)`**
  - Degrees: **`trotx(a, 'deg')`**



- **INVERSE KINEMATIC**

```
Point=[1, -0.3, 1.6]

Trans_Matrix= transl(Point)*troty(-90, 'deg') *trotz(180,
'deg')

Trans_Matrix= [-0,  -0,  -1,     1;
                0,  -1,   0, -0.03;
               -1,   0,   0,   1.6;
                0,   0,   0,     1];
Rv3sd.ikine(Trans_Matrix)
```

- Additional commands allow you to define starting point for robot (q0) and limit for iterations
- `rv3sd.ikine(Trans_Matrix,q0, 'ilimit', 2000)`

DiffKin/ 1

- **DIFFERENTIAL KINEMATIC**
- Matlab has two commands that create Jacobian matrix. The main difference between these commands regards the reference coordinate frame
  - ***Jacob0()*** uses base coordinate frame.
  - ***Jacobn()*** uses end-effector coordinate frame.
- Command syntax for both is the same
- Robot position is given in form of joint coordinates
- Ex:
  - `Q = [1.0969 -0.7986 -0.6465 1.1002 1.5136 -0.1120]`
  - `rv3sd.jacob0(Q)`
  - `rv3sd.jacobn(Q)`

Trajectory Planning/ 1

- Matlab has two main commands for the trajectory planning
  - ***ctraj()***, plotting a route in cartesian space
  - ***jtraj()***, plotting a route in joint space
  - Unlike Jtraj, Ctraj is not related to defined robot
- ***ctraj()***
- Command returns straight path in cartesian space
- Command syntax requires beginning and end points in form of translational matrix
- Additional options are number of points along the path.
- Ex with 50 points along the path:
  - `Pb = [0.75 1.26 0.16]`
  - `Pa = [0.55 1.79 0.26]`
  - `Tb = transl(Pb)`
  - `Ta = transl(Pa)`
  - `ctraj(Tb,Ta,50)`

Trajectory Planning/ 2

- **`jtraj()`**
  - Returns joint space path between two points
  - Command syntax requires starting (**`Qa`**) and ending points (**`Qb`**) as joint coordinate vectors
  - **`[Q,QD,QDD] = jtraj(Qa, Qb, time_interval);`**
  - Gives joint space position (**Q**) , joint velocity (**QD**) and joint acceleration (**QDD**)

- **`lspb()`**
  - Linear segment with parabolic blend
  - **`[S,SD,SDD] = lspb(S0, SF, M)`** is a scalar trajectory (Mx1)
  - **`S`** is the arc lenght
  - output varies **smoothly** from **`S0`** to **`SF`** in **M** steps using a constant velocity segment and parabolic blends (a trapezoidal velocity profile). Velocity and acceleration can be optionally returned as **`SD`** (Mx1) and **`SDD`** (Mx1) respectively.

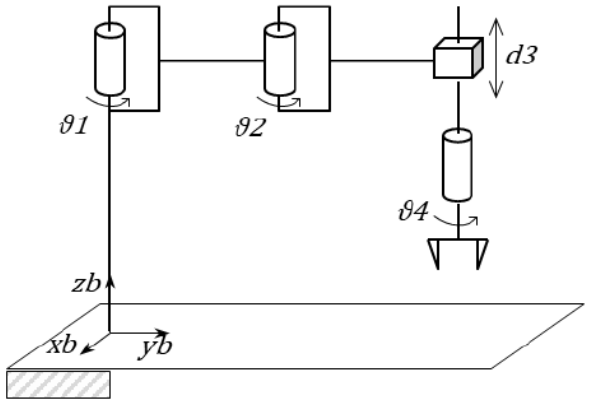
Co-funded by the Erasmus+ Programme of the European Union

Technical elaborate

SEAS 4.0
SUSTAINABLE SHIP AND SHIPPING 4.0
Erasmus Mundus Joint Master Degree
Excellence in education for a new generation of Naval Architects and Marine Engineers

Consider the **SCARA robot manipulator** in the figure:

1. Solve the direct and inverse kinematic problem.
2. Draw with the help of MATLAB or similar tools the shape of the accessible and dexterous working space (section in plan view from above, i.e., $(x\text{-}y)$-plane), indicating the coordinates of the points that characterize the boundary surface and the correspondence with points of the joint space.
3. Obtain the analytical and geometric Jacobian.
4. Plan the trajectory along a path characterized by 5 points within the workspace, in which there are at least one straight and one circular section and the passage for at least one waypoint.
5. Implement in MATLAB or similar tool the algorithms for kinematic inversion with inverse and transpose of the Jacobian along the planned trajectory (adopt the Euler integration rule with an integration time of 1ms).
6. Assuming to relax a component of operational space, implement the algorithm for the kinematic inversion with pseudo-inverse of the Jacobian along the trajectory in MATLAB or similar tool, in the hypothesis of having to maximize:
   1. the distance from the joint limits (relax the orientation component ϕ);
   2. the distance from an obstacle present along the planned trajectory (relax the z-component).



$d_0 = 1\text{m}$, $a_1 = a_2 = 0.5\text{m}$, $\vartheta_{1m} = -90°$, $\vartheta_{1M} = 90°$, $\vartheta_{2m} = -90°$, $\vartheta_{2M} = 45°$, $d_{3m} = 0.25\text{m}$, $d_{3M} = 1\text{m}$, $\vartheta_{4m} = -360°$, $\vartheta_{4M} = 360°$