Autonomous and Mobile Robotics

Prof. Giuseppe Oriolo

Humanoid Locomotion: a Demonstration

Filippo Smaldone

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti

Sapienza Università di Roma

information

- for any question: smaldone at diag.uniroma1.it
- the code of this demonstration is available at: https://github.com/FilippoSmald1/Robotis-OP3-MPC-walking
- ROS based gazebo simulation of OP3 walking
- the same implementation on the DART dynamic simulator is available upon request

the OP3 robot

- the available platform is Robotis OP3

- open source robot
https://github.com/ROBOTIS-GIT/ROBOTIS-OP3

- hardware:
  - -20 dof, position controlled
  - -encoders, imu
  - -camera
  - -main controller: INTEL NUC i3, 8 GB RAM

- the hardware necessarily constrains our solution to the problem

the OP3 robot

- software:
  - -Linux Mint 16
  - -ROS Kinetic
  - -custom real-time control manager
  - -arbitrary sampling time for motor commands
  - -C++ (convenient for real-time control), python

- the software framework gives us enough versatility for our solution in spite of the hardware

the OP3 robot

- pros:
  - -open source
  - -ROS based
  - -modular (easy to upgrade)
  - -easy maintenance
  - -easy set up
  - -GitHub issues responsiveness
  - -low cost, < 20k € in 2021
- cons:
  - -position controlled actuators
  - -comes without F/T sensors
  - -comes without any range sensors
  - -large and slippery feet

problem statement

- high level description: “make a humanoid navigate to reach a goal”

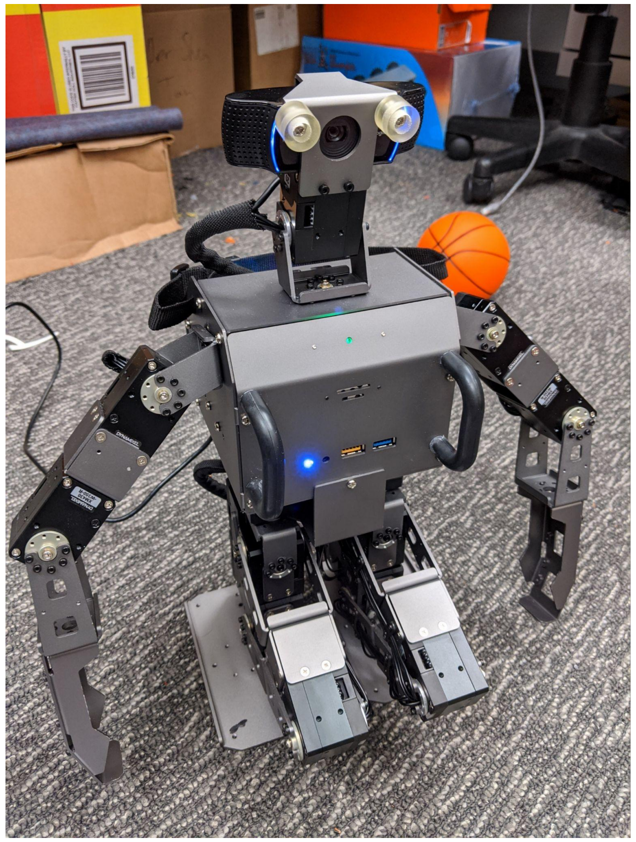
- applications: environment exploration, data acquisition, object transportation

- what does navigation require?

  -motion planning
  -trajectory generation
  -control
  -localization and mapping

addressing the problem

- decompose the big problem into small problems and identify the solution for each one of them



- the robot has a standard initial configuration

- the task will be realized by composing three different motions:
  - -reach a configuration to start walking
  - -walk and reach the goal
  - -come back to the initial configuration

- let's keep it simple: use time pre-programmed motion modes (stand up, walk, sit down)

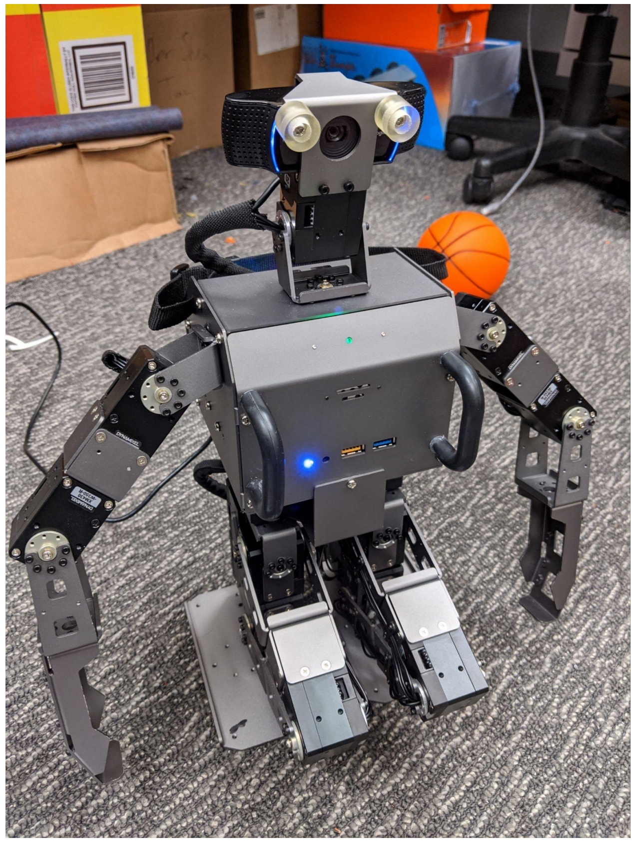
addressing the problem

- use a hierarchical approach

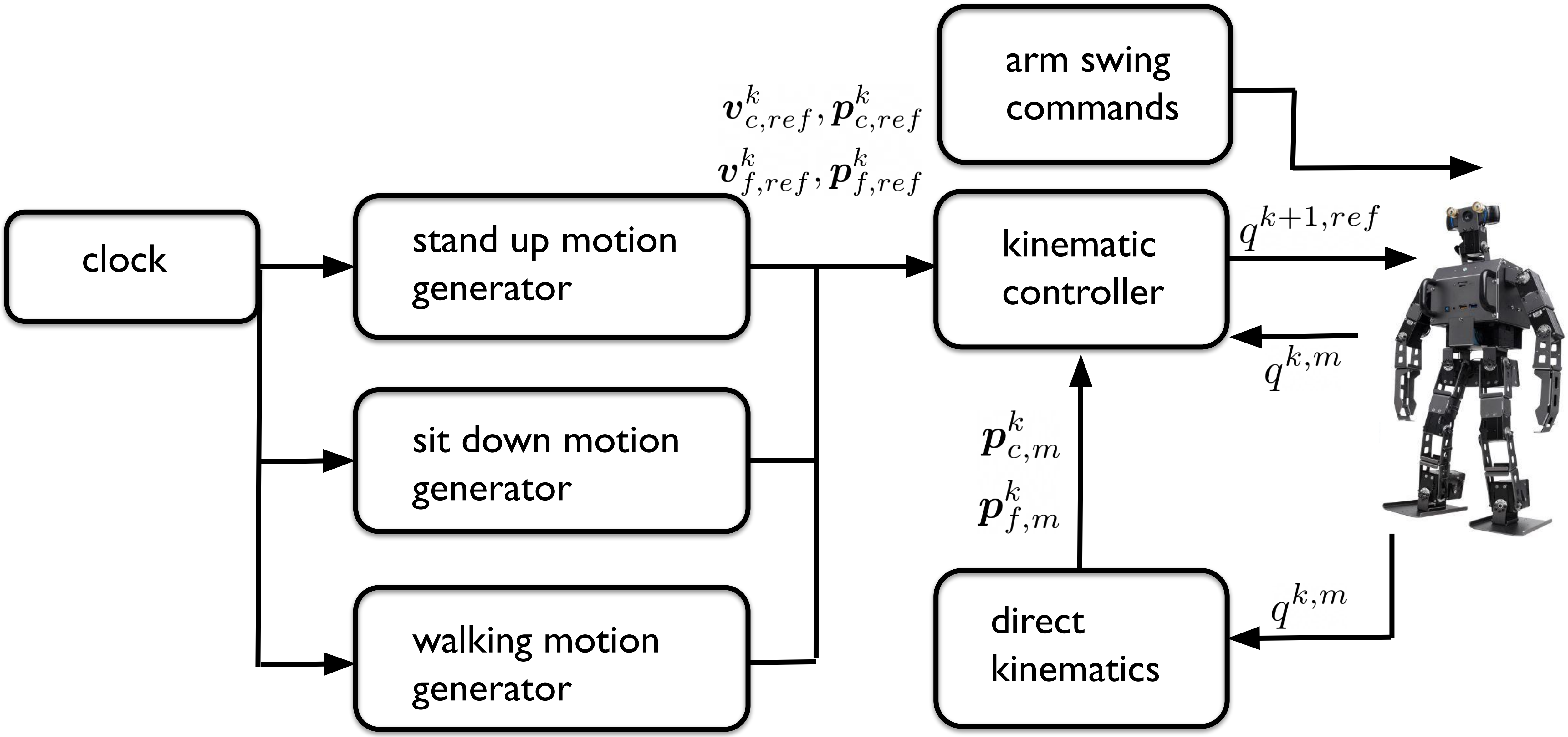
- for each motion mode generate proper body cartesian trajectories (e.g. CoM, feet, arms)

- track them with a kinematic controller

- note that:
  - -kinematic control is the most practical choice with position controlled actuators
  - -we assume that we do not need localization nor mapping
  - -there exist different solutions to this problem

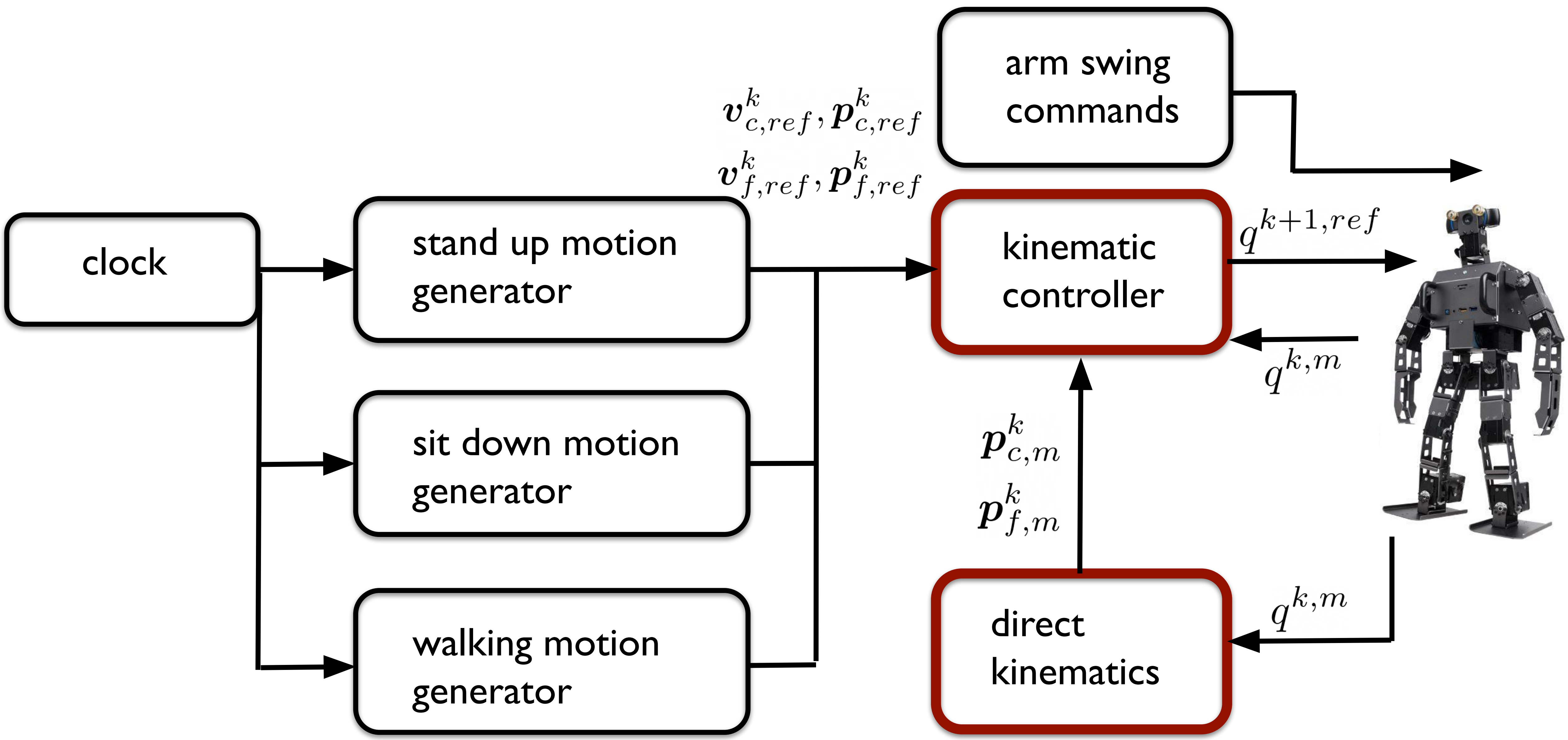


block scheme



$\boldsymbol{p}, \boldsymbol{v}$ denote the pose and its time derivative

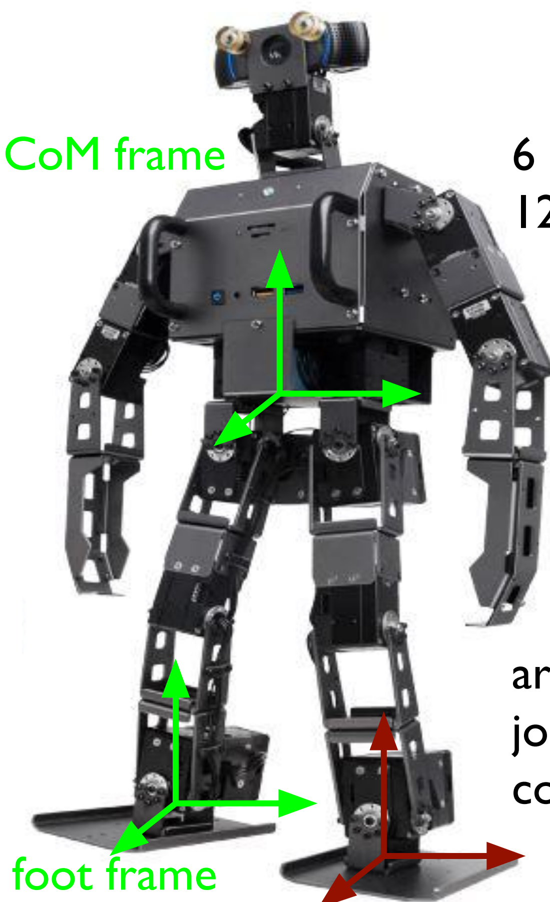
kinematic controller and direct kinematics



kinematic controller and direct kinematics

- input: reference pose and velocity of CoM and a foot (left or right), denoted as swing foot

  why?
  -humanoid as fixed base manipulator where the base frame coincides with a supporting foot
  -CoM and swing foot are regarded as End-Effector frames
  -regulation via multi-task kinematic control law



- output: joint position commands

kinematic controller and direct kinematics

the two blocks work in this way:

- get joint positions $q^{k,m}$ from encoder readings
- compute direct kinematics from $q^{k,m}$ to get $\boldsymbol{p}^k_{c,m}, \boldsymbol{p}^k_{f,m}$
- compute Jacobians from $q^{k,m}$ (support-CoM, support-swing)
- stack the Jacobians
- compute the joint velocities as

$$\dot{q}^k = \begin{bmatrix} J^k_c \\ J^k_f \end{bmatrix}^{\#} \left[\begin{pmatrix} \boldsymbol{v}^k_{c,ref} \\ \boldsymbol{v}^k_{f,ref} \end{pmatrix} + \boldsymbol{K} \begin{pmatrix} \boldsymbol{p}^k_{c,ref} - \boldsymbol{p}^k_{c,m} \\ \boldsymbol{p}^k_{f,ref} - \boldsymbol{p}^k_{f,m} \end{pmatrix} \right]$$

damped pseudoinverse of stacked jacobians

reference velocities

position error gains

position error

- integrate to get the joint position commands

$$q^{k+1,ref} = q^{k,m} + \delta \dot{q}^k$$

sampling time

kinematic controller and direct kinematics

- damped least squares to prevent singularity issues
- the direct kinematics and the Jacobians are computed with efficient recursive algorithms which use the robot URDF (Unified Robot Description Format), provided by the manufacturer
- state of the art C++ libraries for these computations: kdl, rbdl, pinocchio
- the choice of the gain matrix is crucial
- the choice of the sampling time is also crucial

kinematic controller and direct kinematics

a quick look at the code - left support foot

```
...
left_leg_fk_solver->JntToCart(q0_left_leg, x_left_leg_fk);
for (int i = 0; i < 12; i++) {
  if (i<6) q0_sf_to_swg(i) = q0_left_leg(i);
  else q0_sf_to_swg(i) = q0_right_leg(11-i);
}
left_foot_to_right_foot_fk_solver->JntToCart(q0_sf_to_swg, x_sf_to_swg);
CoM_pose_meas.segment(0,3) = Eigen::Vector3d(x_left_leg_fk.p(0),x_left_leg_fk.p(1),x_left_leg_fk.p(2));
swg_pose_meas.segment(0,3) = Eigen::Vector3d(x_sf_to_swg.p(0),x_sf_to_swg.p(1),x_sf_to_swg.p(2));
x_left_leg_fk.M.GetRPY(CoM_pose_meas(3),CoM_pose_meas(4),CoM_pose_meas(5));
x_sf_to_swg.M.GetRPY(swg_pose_meas(3),swg_pose_meas(4),swg_pose_meas(5));
```

use kdl forward kinematic routine to compute the current CoM and swing foot pose

```
sf_pose << desired.leftFootPos, desired.leftFootOrient;
CoM_pose_des << desired.comPos, desired.torsoOrient;
CoM_pose_des(2) = CoM_pose_des(2);
swg_pose_des << desired.rightFootPos, desired.rightFootOrient;
CoM_pose_des = vvRel(CoM_pose_des, sf_pose);
swg_pose_des = vvRel(swg_pose_des, sf_pose);
CoM_pose_des(0) = CoM_pose_des(0);

Eigen::VectorXd v_des, pos_des, pos_meas;
v_des = Eigen::VectorXd::Zero(12);
v_des.segment(0,3) = desired.comVel;
v_des.segment(6,3) = desired.rightFootVel;
pos_des = Eigen::VectorXd::Zero(12);
pos_meas = Eigen::VectorXd::Zero(12);
pos_des << CoM_pose_des, swg_pose_des;
pos_meas << CoM_pose_meas, swg_pose_meas;
```

stack the measurements, the reference poses (in the current support foot frame) and velocities

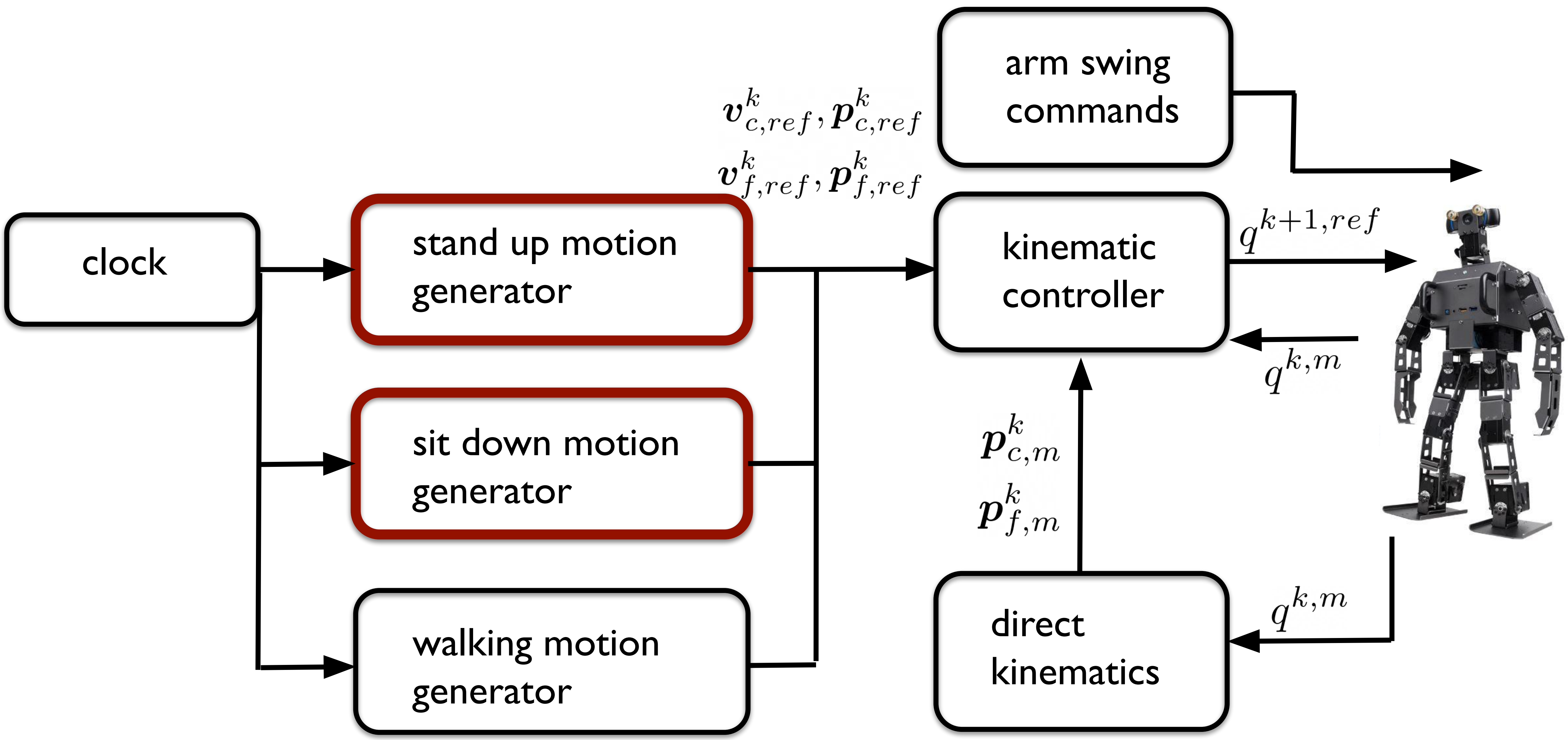
```
if (left_foot_to_right_foot_jacobian_solver->JntToJac(q0_sf_to_swg, J_leftf_to_rightf_leg) < 0) {
  ROS_ERROR( "jacobian error");
}
J_left_leg_to_right = J_leftf_to_rightf_leg.data;
if (left_leg_jacobian_solver->JntToJac(q0_left_leg, J_left_leg) < 0) {
  ROS_ERROR( "jacobian error");
}
J_left_leg_ = J_left_leg.data;
J_stacked << J_left_leg_, Eigen::MatrixXd::Zero(6,6), J_left_leg_to_right;
```

use kdl jacobian solver to compute the Jacobians and then stack them

```
Eigen::VectorXd q_dot = J_stacked.transpose() * (J_stacked*J_stacked.transpose() + Id*sigma).inverse() * (v_des + gains*(pos_des-pos_meas));
for (int i = 0; i < 6; i++) q_left_leg(i) = q0_left_leg(i) + (1.0/rate)*q_dot(i);
for (int i = 0; i < 6; i++) q_right_leg(5-i) = q0_right_leg(5-i) + (1.0/rate)*q_dot(i+6);
...
```

compute kinematic control law and integrate to get reference joint positions

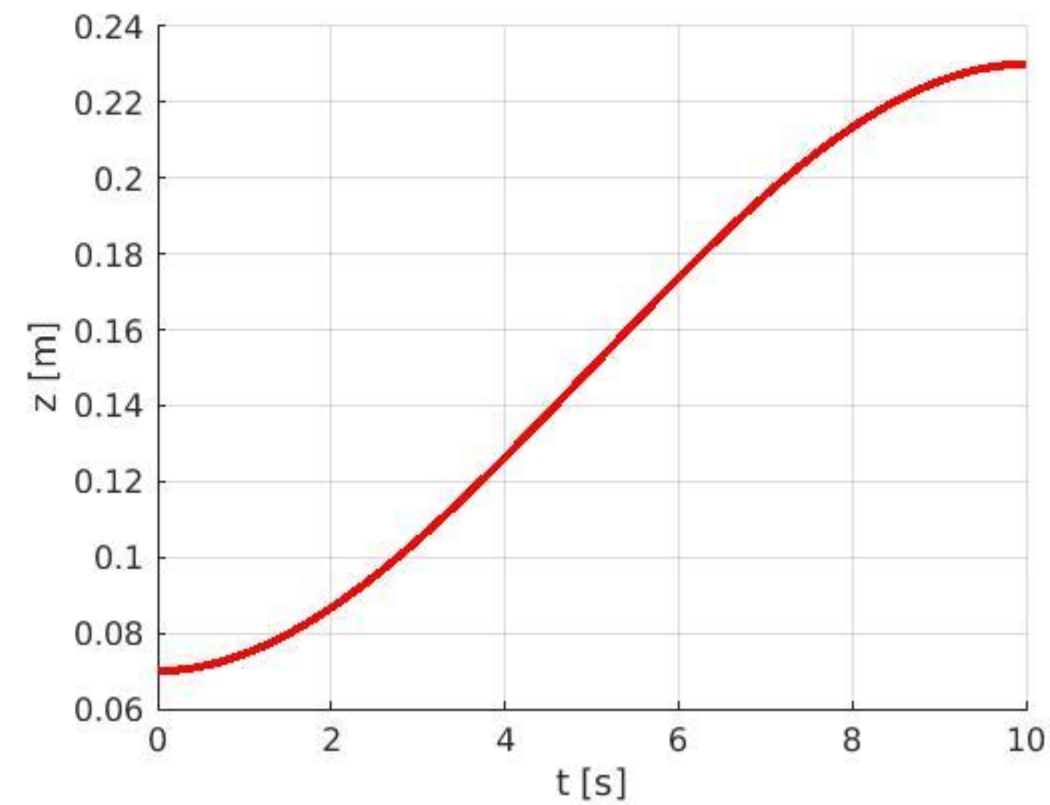
stand up and sit down motion



stand up and sit down motion

- start from a pose $\boldsymbol{p}^0$ and reach a target pose $\boldsymbol{p}^1$ in T seconds

- simply raise/lower the CoM while holding steady the swing foot

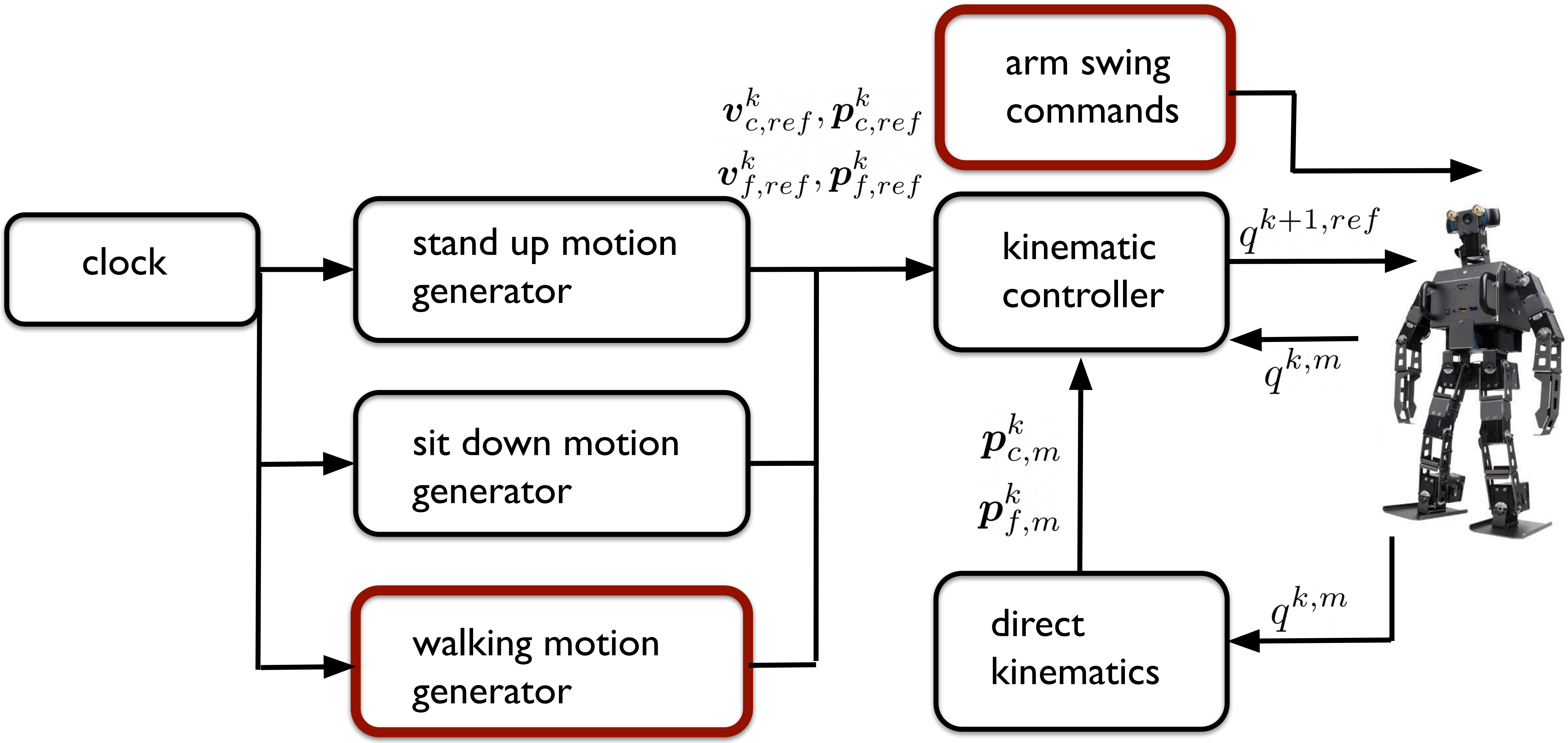


- in practice, it is only required a trajectory for the vertical CoM component

- use for instance a third order polynomial to reach a target pose with zero velocity in $t = T$

- at each time t_k the output of these blocks is $\boldsymbol{p}_{c,ref}(t_k), \boldsymbol{v}_{c,ref}(t_k)$

walking motion



walking motion

- objective: walk to reach the goal (planar ground)

- legged locomotion: exert forces towards the environment to move the robot

- forces are exerted through foot contact with the ground

- the robot must maintain dynamic balance at all times

- approach:
  - -plan suitable contacts, i.e. design a footstep plan
  - -generate CoM and ZMP trajectories to realize a dynamically balanced gait over the footstep plan
  - -generate also swing foot trajectories

walking motion - footstep plan

- footstep plan: cartesian positions and timings (step duration)

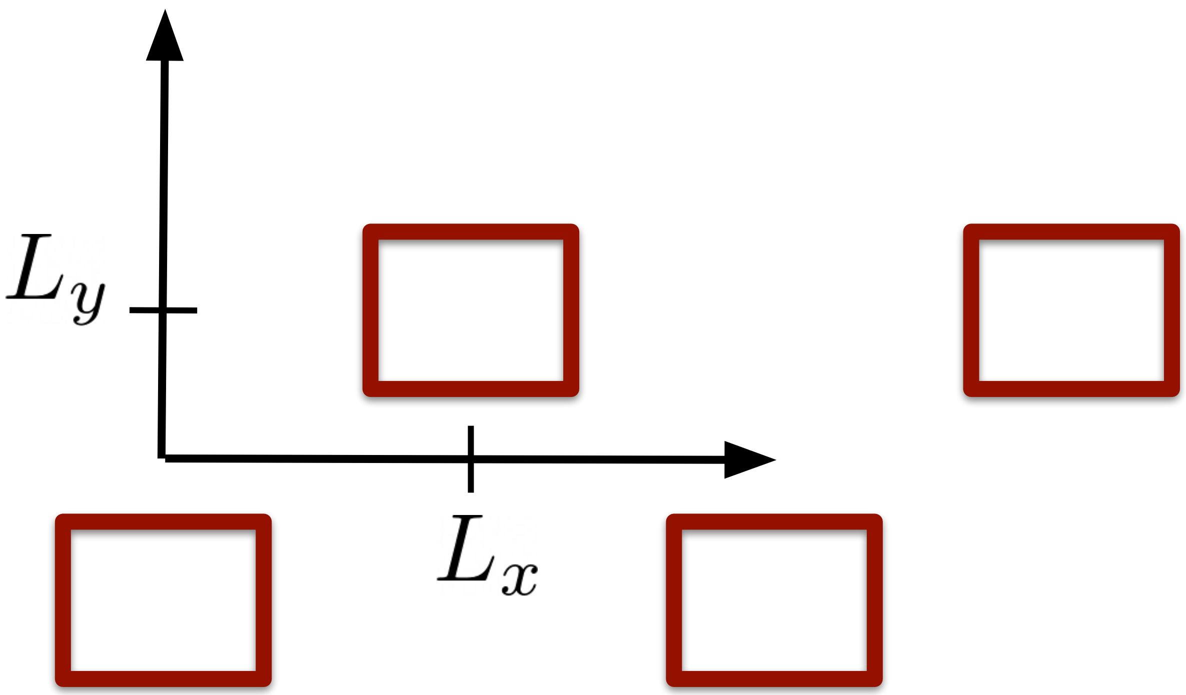
- left and right feet alternate during locomotion

- single and double support alternate during locomotion

- let's keep it simple:
  - -assign a step duration, e.g., $T_s = T_{ss} + T_{ds}$(single and double support duration)
  - -choose a sagittal reference velocity v_x
  - -the stride length on the x component is obtained as $L_x = v_x T_s$
  - -the y component of the footsteps, named as L_y, alternates (left and right support foot)

walking motion - footstep plan

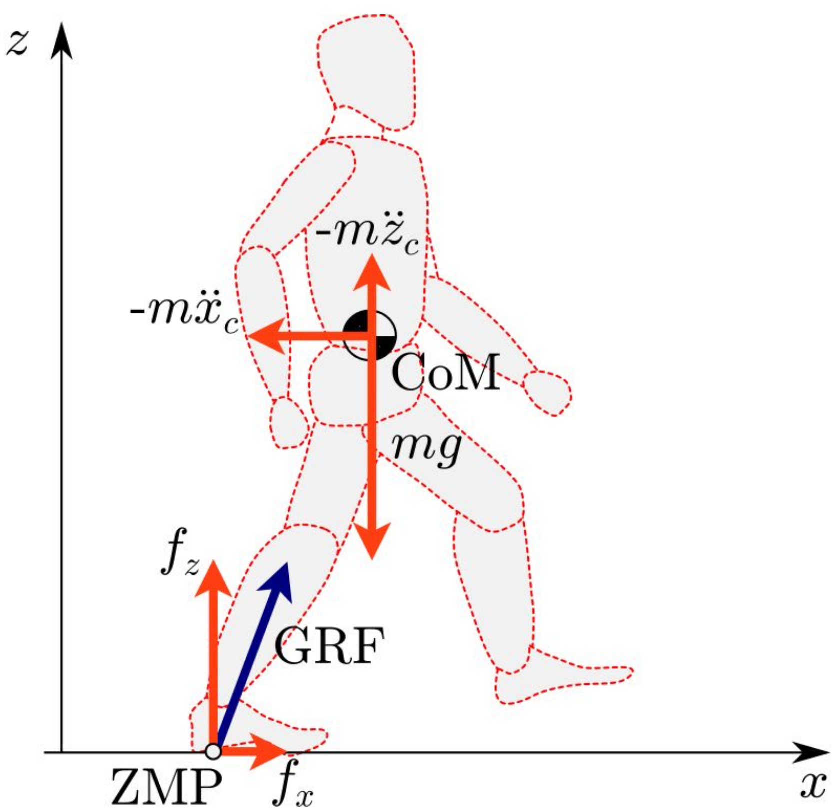
in world frame coordinates



x	y	t
0	-Ly	0
Lx	Ly	Ts
2Lx	-Ly	2Ts
3Lx	Ly	3Ts
...	...	...

walking motion - gait generation

- objective: realize the footstep plan
- relevant quantities: CoM and ZMP
- generate CoM/ZMP trajectories so that the robot is dynamically balanced
- use a simplified model: the Linear Inverted Pendulum (LIP)
- forward walking motion with constant footstep orientation: the sagittal and coronal components are decoupled



$$\ddot{x}_c = \eta^2 (x_c - x_z)$$
$$\ddot{y}_c = \eta^2 (y_c - y_z)$$

natural frequency   CoM   ZMP

$$\eta^2 = \frac{g}{h}$$

walking motion - gait generation

- linear MPC formulation: anticipation!

- ZMP as decision variable

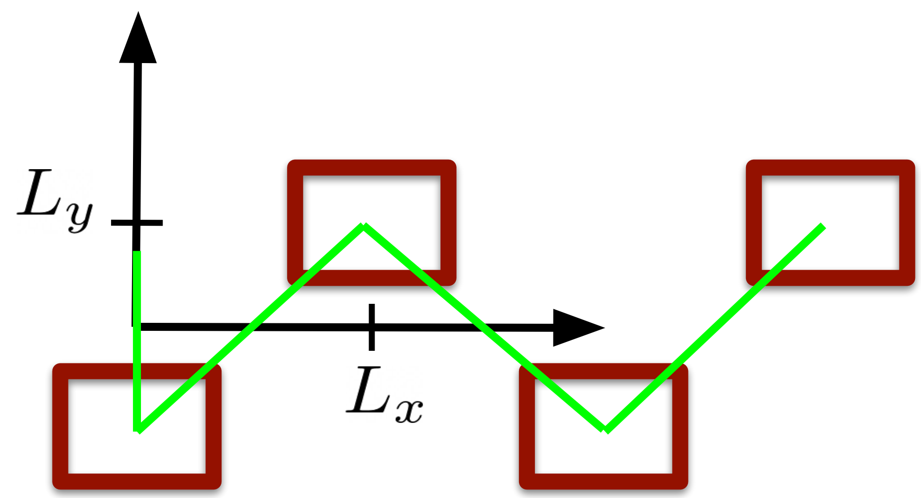
- formulation: track a reference ZMP trajectory, while maintaining dynamic balance and ensuring that the CoM is bounded with respect to the ZMP (the LIP is unstable!)

- solve at each iteration a quadratic program (QP) with linear constraints

- efficient state of the art solvers are available, e.g., hpipm https://github.com/giaf/hpipm

walking motion - gait generation

- reference ZMP trajectory:



- dynamic balance: ZMP inside the support polygon, formulated as a linear inequality constraint

- bounded CoM w.r.t. the ZMP through a stability constraint (Scianca et al, "MPC for Humanoid Gait Generation: Stability and Feasibility", T-RO, 2020), formulated as a linear equality constraint

walking motion - gait generation

- let X_z and Y_z be vectors collecting the decision variables over the prediction horizon
- let X_z^{ref} and Y_z^{ref} be vectors collecting the sampled reference ZMP trajectory over the prediction horizon
- let $\Delta X_z = [x_z^1 - x_z^0, x_z^2 - x_z^1, \dots]^T$

solve at each time step the following QP is solved:

$$min_{X_z,Y_z} \|X_z - X_z^{ref}\|^2 + \|Y_z - Y_z^{ref}\|^2 + \beta\|\Delta X_z\|^2 + \beta\|\Delta Y_z\|^2$$

subject to:
- ZMP constraints
- stability constraint

walking motion - gait generation

- integrate over a sampling interval the LIP dynamics using the first decision variable obtained from the QP and get the reference CoM position and velocity $\boldsymbol{v}_{c,ref}^{k}, \boldsymbol{p}_{c,ref}^{k}$
- generate a swing foot trajectory to reach the next target footstep during single support phases $\boldsymbol{v}_{f,ref}^{k}, \boldsymbol{p}_{f,ref}^{k}$
- use for instance a third order polynomial for the x and y components of the swing foot trajectory
- use a parabolic trajectory for the z component
- arm swing commands: sinusoidal trajectory for the shoulder joint

walking motion - gait generation

a quick look at the code

```
...
decisionVariables_x = solveQP_hpipm(costFunctionH_xy, costFunctionF_x, A_ineq_xy, Zmin_x, Zmax_x, Aeq_x, beq_x);
decisionVariables_y = solveQP_hpipm(costFunctionH_xy, costFunctionF_y, A_ineq_xy, Zmin_y, Zmax_y, Aeq_y, beq_y);
…
```

compute QP using hpipm

```
...
state_x = A_xy*state_x + B_xy*decisionVariables_x(0);
state_y = A_xy*state_y + B_xy*decisionVariables_y(0);
```

integrate LIP to get next CoM reference

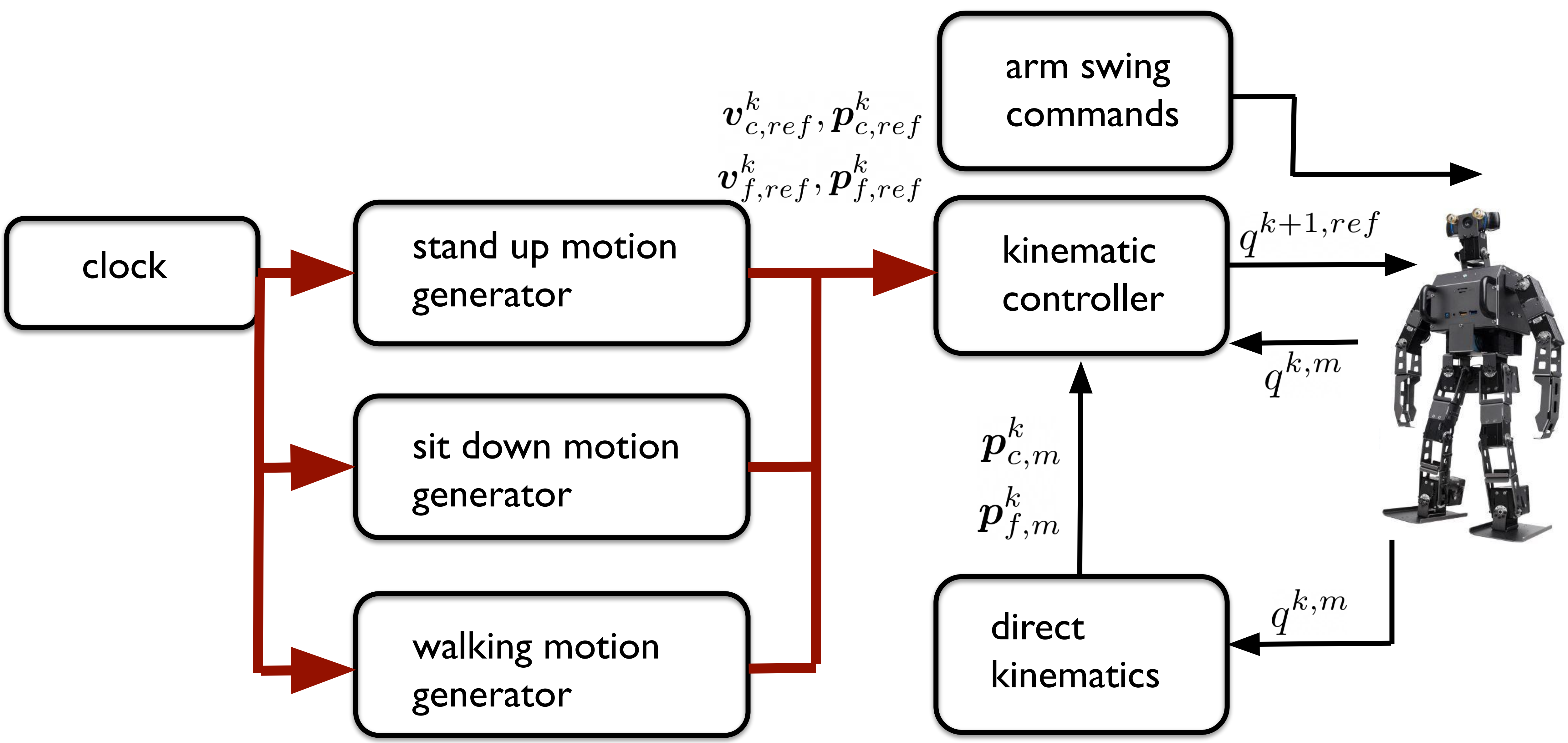
```
next.comPos << state_x(0), state_y(0), comTargetHeight;
next.comVel << state_x(1), state_y(1), 1.0;
next.comAcc << eta*eta * (state_x(0) - decisionVariables_x(0)), eta*eta * (state_y(0) - decisionVariables_y(0)), 1.0;
next.zmpPos << decisionVariables_x(0), decisionVariables_y(0), 1.0;
```

store the reference states into a useful data structure

```
if (walkState.footstepCounter > 1 && walkState.footstepCounter <= n_steps+1)
  next = WalkingSwingFoot(current, next, walkState, ftsp_and_timings);
```

evaluate the swing foot trajectory at the current time instant

motion mode management



motion mode management

- motion modes change at fixed times
- wait some time t_{start} before starting the motion
- stand up motion is executed until time t_{stand} is reached
- walking motion is performed until time t_{walk} is reached (required time to physically execute the footstep sequence)
- the robot reaches its original configuration by executing a sit down motion, concluded at time t_{sit}

concluding remarks

- real-time computations on the robot computer: hard computational timing constraints
- test the algorithm in simulation first (gazebo, DART)
- Sim-To-Real gap: if it works in simulation, it is not 100% guaranteed that it works on the real robot
- robotics is mainly open source, but sometimes not well documented
- possible improvements:

  -footstep planner
  -3D ground
  -more sophisticated whole body controller
  -localization and mapping

experiment time

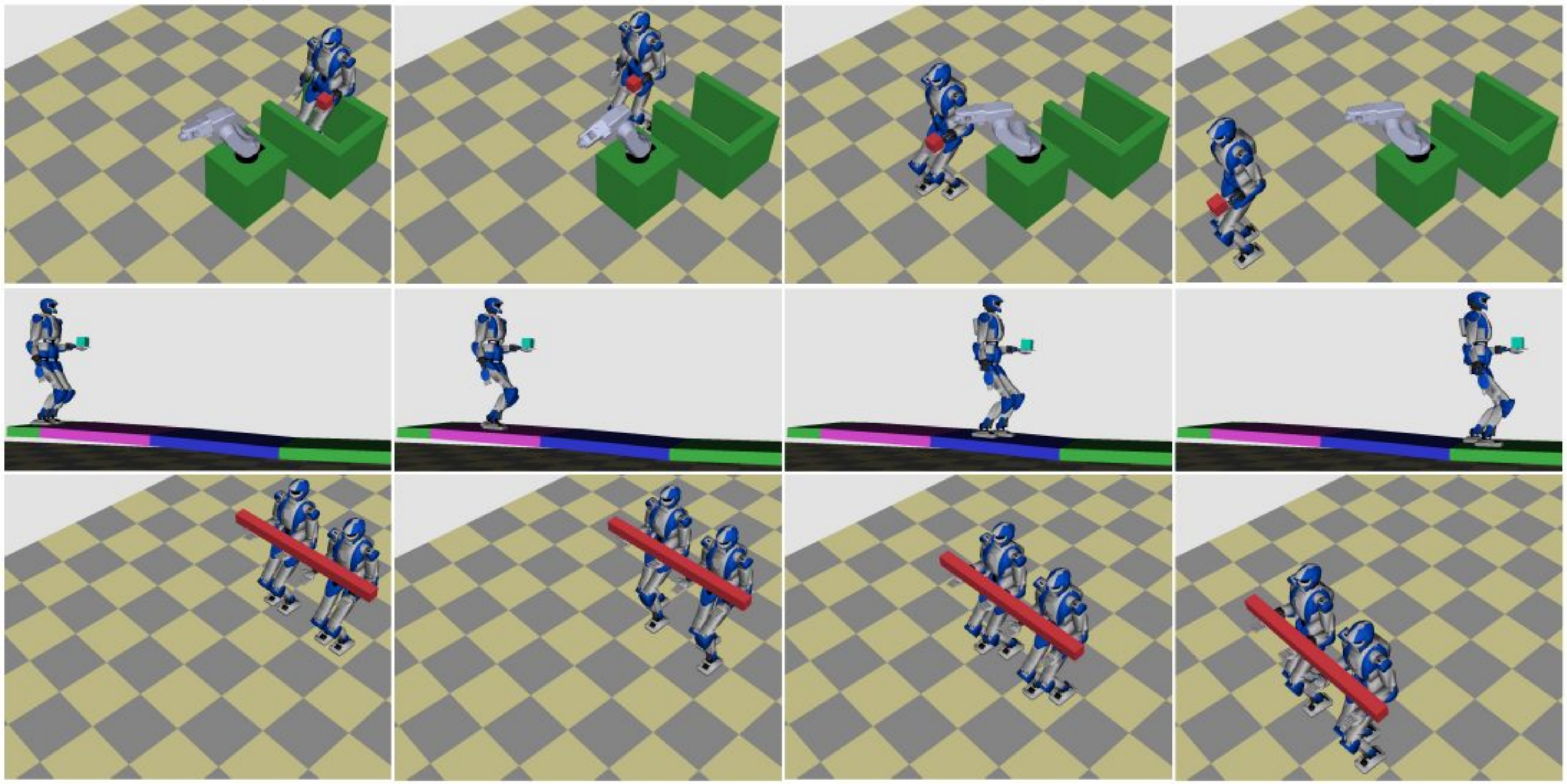
on going research - robust gait generation

- disturbances in MPC can cause constraint violation: in humanoid gait generation this can imply the loss of dynamic balance as well as instability

- different ways to address the problems: disturbance observers for persistent perturbations, constraint restriction for robustness to uncertainties, step position and timing adaptation for push recovery

- we published a contribution for each of the different methodologies and we are now working on a unified framework

on going research - robust gait generation



on going research - 3D walking and running

- LIP model assumes constant CoM height: for 3D motions such as stair climbing and running, this assumption must be removed

- use the Variable Height Inverted Pendulum (VH-IP)

- this model is nonlinear: a nonlinear MPC formulation is required

- we address the problem by computing the vertical motion first and then solving for the horizontal dynamics, considering them as a time-varying linear system

- simple but effective method (real-time implementation on OP3)

on going research - 3D walking and running