ChatGPT

Developing a Tabletop Hexapod Walking Robot: Software & Theory Guide

1. Kinematics and Math

Leg Configuration: A tabletop hexapod typically uses six legs with 3 degrees of freedom (3-DOF) each. Each leg consists of: - **Coxa (hip yaw joint):** rotates the leg horizontally about a vertical axis (yaw) at the body attachment [1]. This joint moves the leg forward/backward and sideways in the horizontal plane. - **Femur (hip pitch joint):** raises or lowers the leg by pitching in a vertical plane (attached to the coxa) [1]. - **Tibia (knee pitch joint):** the lower leg segment that also pitches in the vertical plane [1].

All three joints work together to position the foot. By convention, the **body coordinate frame** is a Cartesian frame fixed to the robot's body (e.g. origin at the robot's center). A common choice is a right-handed system where z is vertical (upwards), x lateral (right), and y forward [2]. Each leg has a local coordinate frame at its hip (coxa) joint. The hip's position relative to the body center is known (from the robot's design). Transforming a foot position from **body frame** to a **hip-local frame** involves translating by the hip's offset and rotating into the leg's plane. For example, a front-right leg might require rotating the coordinates by 45° if the legs are mounted at angles around the body. Clearly defining these frames is crucial for math consistency.

Forward Kinematics (FK): Forward kinematics computes the foot's position given the joint angles. One can derive this by applying transformations for each joint. For example, starting at the hip: yaw the coordinate by the coxa angle θ_1 in the horizontal plane, then translate along the coxa length, then pitch down by femur angle θ_2 and translate along the femur, then pitch by tibia angle θ_3 and translate along the tibia. The end result is the foot coordinates (p_x, p_y, p_z) in body frame. In practice, you can derive FK equations or use Denavit-Hartenberg parameters [3], but often inverse kinematics is of primary importance for leg control.

Inverse Kinematics (IK): Inverse kinematics finds the joint angles (θ_1, θ_2, θ_3) needed to place the foot at a desired position. For a 3-DOF leg, there is a closed-form IK solution [4]. We break the problem into two sub-problems due to the coxa yaw symmetry [5]:

1. **Coxa Yaw (Horizontal Plane):** Project the target foot position onto the horizontal plane. Let p_x, p_y be the coordinates of the foot relative to the hip joint (after transforming into the leg's local frame). The radial distance in this plane is $h = \sqrt{p_x^2 + p_y^2}$ [6]. The required yaw angle aligns the leg toward the foot in the horizontal plane. We can compute:

$$\theta_1 = \arctan 2(p_y,\ p_x)$$

(using appropriate quadrant-handling). Depending on your axis convention, you may need a negative sign or swapped arguments. For instance, one reference gives $\theta_1 = -\arctan 2(p_x, p_y)$ in a coordinate

frame where +Y is forward [7]. If the foot is almost exactly under the hip (very small p_x, p_y), the yaw angle is undefined; in practice one can set $\theta_1 = 0$ in that singular case [8].

1. **Planar Leg IK (Femur & Tibia):** After yaw rotation, we solve the leg as a planar 2-link mechanism in the sagittal plane. First, account for the coxa link offset: define $d_y = h - \ell_1$ (horizontal distance from the hip joint after the coxa) and $d_z = p_z$ (vertical distance) [9]. Now $r = \sqrt{d_y^2 + d_z^2}$ is the distance from the hip joint (pitch axis) to the foot [10]. The femur and tibia form a triangle with the line to the foot (length r). We use the **Law of Cosines** to find the knee (tibia) angle:

$$\cos\theta_3 = \frac{\ell_2^2 + \ell_3^2 - r^2}{2\,\ell_2 \ell_3}$$

where ℓ_2, ℓ_3 are femur and tibia lengths. Then $\theta_3 = \arccos(\mathrm{clamp}(\cos\theta_3, -1, 1))$ [11]. (Clamping is done to avoid domain errors if due to rounding $\cos\theta_3$ is slightly outside [-1,1].) This θ_3 is the **knee pitch** angle. Note that by our convention, $\theta_3 = \pi$ (180°) would mean the leg is fully stretched straight [12]. Often, controllers prefer the "elbow-down" solution where the knee bends downward; this corresponds to the positive (or default) arccos result [13].

Next, to find the femur angle θ_2, we find two auxiliary angles: ϕ and δ. ϕ is the angle of the hip-to-foot vector in the vertical plane, i.e. $\varphi = \arctan 2(d_z,\ d_y)$ [14]. δ is the angle between the femur link and the line to the foot (inside the triangle at the hip). We get δ from the Law of Cosines as well (or using the sine law):

$$\cos\delta = \frac{\ell_2^2 + r^2 - \ell_3^2}{2\,\ell_2\, r}, \quad \sin\delta = \frac{\sqrt{4\ell_2^2 r^2 - (\ell_2^2 + r^2 - \ell_3^2)^2}}{2\,\ell_2\, r}\,.$$

Then the femur angle is simply the difference: $\theta_2 = \varphi - \delta$ [15]. This yields the hip pitch angle that elevates the leg to the correct height. (In practice, one can compute θ_2 directly via an `atan2` using the formulas for $\sin\theta_2$ and $\cos\theta_2$ to avoid numerical cancellation [16] [17], but the above concept is easier to understand.)

Computing Servo Angles: The IK procedure provides the angles (θ_1, θ_2, θ_3) required to place the foot at target (p_x, p_y, p_z) relative to the body. These angles must be translated to servo commands. That involves adding any zero-offset (e.g. many servos are centered at 90° or 1500 µs pulse) and ensuring the direction aligns with servo rotation direction. For example, if $\theta_2 = 0$ was defined as femur horizontal, but physically that might correspond to the servo at mid pulse. In Oscar Liang's hexapod, he notes "to convert the angle into usable servo value, simply add 90° before writing to servos" [18] (this assumes 0° in math corresponds to servo at -90° from its neutral). You will calibrate these offsets per robot during initial setup.

Numerical Considerations: The IK should be computed frequently (e.g. every control cycle) for all six legs, so efficiency matters on limited hardware. A 16 MHz microcontroller may need to solve all legs' IK at 50–200 Hz [19]. The closed-form above is efficient (only a couple of `atan2` and `acos`). Avoiding unnecessary square roots or heavy trig in loops helps. Also handle unreachable targets gracefully: if the target foot position is beyond the leg's maximum reach (e.g. $r > \ell_2 + \ell_3$), the usual method would give $\cos\theta_3 > 1$. Rather than erroring out, you can clamp and set $\theta_3 = 0$ (or the physical limit) meaning the leg is fully extended toward the target. Some IK solutions do exactly that: if beyond reach, **orient the leg as far as possible toward the target** [20]. This "best effort" approach is preferable to failing completely, and it serves as a **fall-back motion** in case your gait planner asks for an unattainable foot position.

2. Gait Planning and Control

A **gait** is the pattern of leg movements a robot uses to walk. Hexapods are capable of many gaits (tripod, wave, ripple, etc.), but a simple and effective one for a six-legged robot is the **tripod gait** [21]. We will focus on tripod gait and general principles of gait generation.

Swing and Stance Phases: Each leg alternates between two phases during walking [22]: - **Stance phase:** The foot is on the ground, pushing against it. In this phase, the leg supports the robot's weight and "drives" the robot forward (or backward/turning as commanded). The foot moves **opposite** to the robot's travel direction relative to the body (since the foot is planted on ground) [22]. For example, if the robot is walking forward (body moving +Y direction), the foot in stance will move backward relative to the body. - **Swing phase:** The foot is lifted off the ground and "swings" forward through the air to a new foot placement for the next step [22]. During swing, the leg is not supporting weight.

In a **tripod gait**, the hexapod uses two groups of three legs. At any time, three legs (forming a tripod shape) are in stance supporting the body, while the other three are in swing moving to the next position [23]. Specifically, one common grouping is: **Group A:** front-left, rear-left, and middle-right legs; **Group B:** front-right, rear-right, and middle-left legs. In Phase 1 of the cycle, Group A is in swing (legs lifted and moving forward) while Group B is on the ground propelling the robot. In Phase 2, the roles swap: Group A is down in stance, and Group B swings forward. This alternating pattern repeats every two steps, resulting in a fluid tripod gait where at all times the robot has a stable tripod of support [23].

Tripod Gait Step Sequence: To illustrate, here's a breakdown for walking forward: 1. **Phase 1:** Legs in Group A (left front, left rear, right middle) lift up and swing forward to the next foothold. Simultaneously, the three supporting legs in Group B push the body forward. The body shifts weight onto Group B's triangle while Group A legs are airborne [23]. The middle legs often play a role in weight shifting – for example, the middle leg on the ground can angle to "lean" the body toward that side before the opposite legs lift, ensuring stability [24]. 2. **Phase 2:** Now Group A legs plant down in their new forward positions. The robot's weight shifts onto the tripod formed by Group A. Then Group B legs (right front, right rear, left middle) are lifted and enter swing, moving forward while Group A's legs are in stance, pushing the body forward.

By continuously alternating, the hexapod achieves a smooth forward walk [25]. At any given time, three feet contact the ground, giving excellent stability. (Tripod gait is statically stable since the center of mass can be kept within the triangle of support at all times.) In fact, hexapod robots **"always have three limbs in contact with the ground, so their gaits are [statically] stable, making walking control much simpler"** [26] compared to bipeds or quadrupeds. This is one reason tripod gait is popular – it allows the robot to move quickly without complex balance calculations, as long as the shifts between tripods are done carefully.

Timing and Coordination: Key gait parameters include step length and cycle timing. One gait cycle (for tripod, a two-phase cycle) has a certain duration or frequency. All legs cycle in phase with their group. For example, if the cycle period is T (time for Group A swing + Group B swing), then each swing or stance phase lasts $T/2$. Within each half-cycle, we typically implement **smooth interpolation** for leg motion rather than immediate movements. That is, instead of instantly raising a leg and teleporting it forward, we use a smooth trajectory (e.g. a spline or trapezoidal velocity profile) so the leg lifts, moves, and settles down gently. This avoids shocks and ensures stability.

[27] In practice, gait planning defines a **trajectory** for each foot during swing – for instance, a common approach is to use a half-ellipse or similar curve for the foot: up and forward then down. During stance,

the foot moves relative to the body in the opposite direction of travel. You can use interpolation techniques or mathematical functions (sine waves, Bezier curves, etc.) to generate these smooth trajectories [27]. The goal is to prevent abrupt changes in velocity that could shake the robot or cause feet to slip.

Tripod Gait Example (Pseudocode): Below is a simplified pseudocode illustrating a tripod gait controller for forward walking:

```
initialize phase = 0
loop:
    if phase == 0:  # Phase 1: Group A swings, Group B stance
        for each leg in Group A:
            foot_target = current_foot_position + forward_step  # move foot
forward
            foot_target.z = lift_height                    # lift foot up
        for each leg in Group B:
            foot_target = current_foot_position  # ideally keep it at ground
contact
            # (During stance, foot moves relative to body as body moves
forward)
        # (Compute IK for all legs to get servo angles for these foot_target
positions)
        # (Command servos to move smoothly to new positions over half-cycle
time)
    else:  # Phase 2: Group B swings, Group A stance
        (similar structure but swap roles of Group A and B)
    wait for half_cycle_duration (allowing the move to complete)
    phase = 1 - phase  # toggle phase between 0 and 1
```

In a real implementation, the **foot_target** during stance is adjusted opposite to the body motion. Often instead of explicitly moving stance feet, you move the body relative to the fixed feet. For example, if the robot should move forward 10 mm in a half-cycle, you can consider the body moving forward 10 mm while stance feet stay in place (which relative to the body, looks like feet moved backward 10 mm). After each half-cycle, the roles swap and the previously moving feet are now on the ground ready to push.

Turning and Other Motions: The tripod gait can be adapted for turning or side-stepping. Turning in place, for instance, can be done by moving the legs in a mirror pattern (all left-side legs swing forward while right-side swing backward, or vice versa) so the body rotates [28]. To turn gradually, you can have foot placement curves that advance in a curved path. Input commands (like "turn left" or "move diagonally") are translated into appropriate modifications of each foot's target trajectory during swing. The gait sequencing (which legs lift when) remains the same; only the vector of movement for each foot changes with the desired motion.

Other Gaits: Aside from tripod, hexapods can use **wave gait** (one leg moves at a time) or **ripple gait** (two legs move at a time, e.g. front-left and rear-right, etc., in a wave sequence). These gaits have more of the legs on the ground (for wave gait, 5 on ground at a time) and thus are **extremely stable but slower** [21]. Tripod gait is the fastest (three legs move per step), but it requires the robot's center of gravity to be well within the tripod support at the moment of swap for stability. Wave gait is slowest but can keep the

center of gravity always well inside a large support polygon (5 legs down). **Ripple gait** is intermediate (typically 4 legs down at any time). For a tabletop robot on slightly uneven terrain, tripod gait is a good compromise of speed and stability, but if the robot needs to carefully navigate very rough terrain, a ripple or wave gait (or even adaptive gait) might be used for better stability at the cost of speed.

Phase Coordination: Ensure that the leg groups move 180° out of phase. In practice, you initialize the gait so that at time 0, one set of legs is about to lift while the others are mid-stance. It's crucial that **all three legs of a tripod lift (and land) simultaneously** and likewise for the opposite tripod. If one leg lags, the weight might not transfer properly, leading to a stumble. Using synchronous servo commands (see Servo Management) or very precise timing in your loop will help keep the legs in lockstep.

Interpolation and Smoothness: To avoid jerky motion, interpolate joint angles or positions at a high update rate. For example, if the gait cycle is 1 second, you might compute intermediate joint angles at 50 Hz or more and update the servos in small increments. This makes the foot trajectory smooth and ensures the center of mass transitions smoothly. The smoother the motion, the less likely the robot is to slip, and the lower the forces on each leg at lift-off and touch-down. Some frameworks use **Central Pattern Generators (CPGs)** or splines to continuously generate these smooth cyclic motions, but you can also achieve smooth gait via simple interpolation between key frames of the step.

3. Terrain Awareness

Even on moderately uneven terrain, a hexapod's advantage is its ability to keep stability by moving legs independently. **Terrain awareness** in software refers to adjusting the gait or leg motions based on the ground conditions.

Basic Terrain-Aware Movement (No Specialized Sensors): With no extra sensors, you should still plan for the possibility that the ground is not perfectly flat. This can be done by: - Using a higher foot trajectory: Lift the feet higher during swing to avoid tripping over small obstacles. For instance, on flat ground you might lift the foot 10 mm; on rough ground you might lift 30 mm. - Slowing down the gait: This gives the robot more time with three legs on the ground, reducing dynamic shocks when the terrain is uneven. - **Body articulation:** If your software allows body rotation, you can roll or pitch the body slightly to keep legs within reach. For example, if the robot's right side is going over a bump, you could roll the body a bit to the left to shift weight to the left tripod, preventing a tip. (This requires IMU feedback or predefined sequences.) - Statically probing with legs: In absence of sensors, one strategy is to use the legs themselves to feel the ground. For example, when lowering a foot, do it slowly and monitor the servo or motor feedback (some servos provide a crude load or position error estimate). If the leg encounters an obstacle sooner than expected (motor stalls or current increases), the software can infer contact. This is effectively using the leg as a **contact sensor**. Many modern smart servos or controllers allow reading the motor load or impedance, which can serve as a touch indicator [29].

Using Sensors for Terrain Adaptation: Sensors can significantly improve terrain handling: - **Foot Contact Sensors:** Simple switches or pressure sensors on the feet can directly signal when a foot touches ground. If a foot doesn't hit ground when expected (e.g., it moved to the commanded xyz but no contact was sensed), the terrain might be lower (a hole) – the controller could then lower the foot further or adjust the body down until contact. Conversely, if contact is sensed early (higher ground than expected), the controller can stop lowering that leg to avoid pushing the body up too high. - **IMU (Inertial Measurement Unit):** An IMU gives the robot's body orientation (pitch, roll) and acceleration. The software can use IMU data to detect if the body is tilting unexpectedly (e.g., one side rising or dropping). If so, it can react by adjusting leg positions. For instance, if the robot pitches forward (front dipping), perhaps the front legs stepped in a hole or are too low – the controller could compensate by extending

those legs or by shifting the body backward a bit. The IMU can also be used to actively keep the body level on uneven ground (a form of stabilization where leg heights are continuously adjusted to keep the roll and pitch near zero). - **Terrain Mapping:** Advanced implementations may use sensors like stereo vision or lidar to map the terrain ahead, but for a small tabletop hexapod, a simpler approach is to create an **elevation map** on the fly using foot contacts. For example, if a foot lands much lower than other feet, you could mark that area as a depression. You can also program the robot to probe with a foot before committing weight (much like an insect might). However, these go into advanced territory; a basic terrain-aware gait might not explicitly map the ground, but rather react to immediate feedback.

A concrete example of sensor use: The hexapod project by Arrigoni et al. implemented a **terrain-adapting control algorithm** which "recognizes leg–terrain touch and reacts accordingly to ensure movement stability" [30]. In their setup, they relied on detecting when a leg touches the ground (using feedback from Dynamixel servos' load/position data) and then rapidly adjusting the control – e.g., stopping the leg descent and transitioning to stance when contact is detected, even if the planned trajectory hadn't reached the ground yet. This prevents driving a leg into a rock or leaving it dangling in a hole. They also note that **tactile perception** can use IMUs, pressure or torque sensors at the feet to sense the ground [31]. Even without dedicated foot sensors, their robot's servos provided enough feedback to infer contact.

Obstacle Avoidance vs. Obstacle Negotiation: If the robot has a sensor (like an ultrasonic distance sensor or a small camera), it can detect larger obstacles and decide to stop or turn before hitting them. However, if an obstacle is low (like a rock that's within step height), a hexapod can sometimes just **step over it**. The software could incorporate a behavior where if a foot encounters an obstacle (contact detected early), instead of stopping, the robot tries to lift the foot higher and forward to climb over, and maybe raises the body a bit. This requires careful control to avoid tipping. For a tabletop robot, this might be as simple as: if front legs hit something early, treat it as an obstacle and either switch gait to a careful climb (maybe a wave gait) or back up and try a different path.

In summary, **terrain-aware gait** means: adjust foot heights and placements based on ground feedback, use the IMU to keep the body stable, and possibly modify gait timing (e.g., pause the gait if a leg is searching for ground). These concepts can be implemented gradually – you might start with a conservative fixed high-step gait for safety, then add IMU leveling, then add foot contact sensing for refinement.

4. Input and Output Handling

This section covers how the robot's movement commands are given (inputs) and how the software drives the servos (outputs). A well-designed software architecture separates **what the robot should do** (e.g. "move forward") from **how the servos must move** to achieve it.

High-Level Input Abstraction: Rather than manually specifying each leg's motions, it's useful to control the hexapod via high-level commands: - **Velocity vector:** e.g., (v_x, v_y) in the robot's frame for forward/sideways velocity, plus a rotational velocity ω for turning (yaw rate). - **Gait mode or step parameters:** e.g., command it to walk, trot, turn-in-place, or change gait speed.

The gait engine will translate these inputs into coordinated leg motions. For instance, if the user pushes a joystick forward, the software sets a forward velocity command. The controller then computes appropriate foot trajectories: feet should cycle at a rate proportional to the forward speed, and stance phase foot motion should correspond to that speed. If v_y is forward speed and the cycle frequency is f cycles per second, the stride length per leg might be $d = v_y / f$. The foot will move d meters backward

relative to body during each stance (since the body is moving forward) and be lifted and placed d meters ahead during swing. Similarly, a yaw rate ω can be translated into asymmetric foot motions (e.g., for turning left, the right side legs might take larger strides or move faster than the left side legs).

By using such abstraction, the operator or higher-level code can simply specify the desired motion and let the IK and gait planner handle leg coordination. Indeed, recent hexapod control frameworks allow "complete control over a robot's speed, body orientation, and gait type" via high-level commands [32]. You might design your input handler to accept commands like `set_velocity(vx, vy, omega)` or even commands like `move_to(x,y)` which your planner then achieves by appropriate gait steps.

Command Examples: - **Manual control:** If using a remote control or keyboard, map inputs to velocity commands. E.g., Up arrow = $v_y = +0.1$ m/s forward, Left arrow = $\omega = 15^\circ/s$ turn. - **Autonomous modes:** If the hexapod should follow a path, your software could generate a sequence of such velocity commands or foot positions.

Output to Servos: Once the desired leg joint angles or positions are computed by the gait planner and IK, these must be sent to the servo motors. There are typically two ways to control hobby servos from a microcontroller or single-board computer: 1. **Direct PWM via GPIO:** Hobby servos use Pulse-Width Modulation signals. The control signal is a periodic pulse (period ~20 ms or 50 Hz) where the **pulse width** encodes the angle [33]. Typically ~1.5 ms pulse = center, ~1 ms = one extreme, ~2 ms = the other extreme [34] (some servos accept a slightly wider range, e.g. 0.5 ms to 2.5 ms for the full mechanical range). In software, you can use timers to generate these pulses. For example, using MicroPython on an MCU, you would set up a PWM object at 50 Hz and set the duty cycle corresponding to the desired pulse width. (On MicroPython for ESP8266/ESP32, a duty of 77/1023 was ~1.5 ms if 50 Hz, while ~40 corresponds to ~1 ms, ~115 to ~2 ms [35].) Many microcontrollers have PWM peripherals to handle this precisely. 2. **Dedicated Servo Controller or Serial/I2C Commands:** If using a board like Raspberry Pi (which doesn't have many PWM outputs by default), a common choice is to use a servo driver (e.g., PCA9685 16-channel PWM board) or a dedicated servo controller (like Pololu Maestro, Lynxmotion SSC-32U, etc.). These devices receive high-level commands (often over UART, I2C, or USB) and generate the PWM signals for each servo. For instance, the SSC-32 controller listens on a serial port – you send a command string specifying servo angles or pulse widths. Example: `#5 P1600 T1000` could mean move servo #5 to pulse 1600 µs over 1.0 s [36]. By offloading the PWM generation, you ensure precise timing and can often control many servos simultaneously with ease.

Implementing Outputs in Code: If using MicroPython or a microcontroller:

```
# Pseudocode for direct PWM control (e.g., on MicroPython)
servos = [PWM(Pin(i), freq=50) for i in servo_pins]  # initialize PWM on each pin
def angle_to_duty(angle, min_us=500, max_us=2500):
    # Convert an angle in degrees to a duty 0-1023 (for ESP32/ESP8266 MicroPython)
    pulse_us = min_us + (angle/180.0) * (max_us - min_us)  # assuming 0-180° range
    duty_val = int(pulse_us/20000 * 1023)  # 20ms period -> duty fraction of that
    return max(0, min(1023, duty_val))
# To move a servo i to angle θ:
servos[i].duty(angle_to_duty(theta_degrees[i]))
```

(This example assumes you have a mapping from IK angles to servo angles in degrees; the conversion to pulse microseconds may need calibration.)

If using a serial servo controller (like Pololu Maestro via UART/USB), you might do:

```
# Pseudocode for sending group move to a servo controller (e.g., SSC-32 format)
cmd = ""
for servo_id, pulse_us in targets.items():
    cmd += f"#{servo_id} P{pulse_us} "
cmd += "T100\r"   # move all servos within 100 ms
uart.write(cmd.encode())
```

This would command all listed servos to move to their target pulse width in 100 ms, and the controller will synchronize the movement. The benefit of group commands is that **"the servos will both start and stop moving at the same time… easy to synchronize complex gaits."** [37] Many controllers support a **time parameter** or a **synchronous update** to achieve this.

Microcontroller vs. SBC: A Raspberry Pi can run the IK and gait code (likely in Python/C++), and then either bit-bang GPIO outputs (not very precise for many servos, not recommended without a driver) or send commands to a microcontroller/hat. An alternative is to use a microcontroller (Arduino, etc.) to do both the computing and PWM. Since hexapod IK and gait logic can be intensive but still feasible on microcontrollers (especially modern ones or using fixed-point tricks), some projects use an Arduino or similar for all logic. Others use a Pi for high-level planning and an Arduino just for servo output. **MicroPython** on an embedded board can potentially handle both as well, given its performance is enough for ~50 Hz updates on 18 servos with IK math (which is borderline on slower chips but doable on something like an ESP32 or PyBoard with optimized math).

Output Rate and Synchronization: Servos typically are updated at ~50 Hz. You don't want to update much faster (standard analog servos won't respond faster and you might just saturate communication), and much slower could make motion jaggy. So a loop of 50–100 Hz sending new positions is typical. If using a servo controller, it might handle interpolation for you (e.g., you send a target and it slews the servo smoothly). If not, your software should incrementally change servo commands to ensure smooth movement. For example, if a servo needs to move from 1000 µs to 2000 µs in 1 second, you could step it by ~10 µs every 5 ms tick.

5. Servo Management

Controlling 18 servos reliably and safely is a significant part of the software design. Key considerations include servo signal ranges, timing, power management (though we focus on software), and coordinated motion.

PWM Range and Calibration: Standard hobby servos interpret a 50 Hz PWM signal, with pulse widths roughly between **1 ms (1000 µs)** and **2 ms (2000 µs)** for about 90° in each direction from center. Many servos can extend a bit beyond this, for example **0.5 ms to 2.5 ms** often corresponds to the absolute mechanical limits [34]. It's important to **calibrate and clamp** these values. You should determine the safe minimum and maximum pulse for each joint on your robot so that the servo never tries to push the leg beyond its physical limit (which could cause straining or damage) [38]. For instance, if your femur servo at 0° (horizontal) is 1500 µs, and it can move down to -45° at 1000 µs and up to +75° at 2000 µs before hitting a mechanical stop, use those as your limits. Always clamp your computed pulse or angle commands to within safe ranges to avoid binding the servo [38]. Some controllers (or libraries) let you set these limits so it never goes out of bounds.

It’s also good practice to **zero** or center the servos at startup (move them to a known “home” position slowly) so the robot begins in a stable stance.

Servo Speed Control: Not all hobby servos support speed control internally (that’s a feature of high-end servos or via external controller logic), but you can emulate speed by sending small increments as mentioned. Dedicated controllers like the SSC-32 have a speed parameter per servo (in µs/s) or a group move time parameter [39] [36]. If using such a controller, take advantage of those features to simplify your software. For example, you can command a 90° move to happen over 0.5 s rather than instantly [40], and the controller will generate intermediate pulses. If coding it yourself, implement a simple ramp: e.g., if current angle = 30°, target = 60°, and you want it done in 0.2 s at 50 Hz updates, increase by 1.5° each tick (or corresponding pulse µs step).

Synchronous Movements: A major point in hexapod motion is that multiple servos (ideally all the servos in one tripod group) should move together. If you send commands one by one in sequence without synchronization, you risk slight delays causing uneven weight distribution. For example, if the front leg lifts a fraction of a second before the rear leg, the robot’s weight might shift unpredictably. Thus, either broadcast the commands nearly simultaneously or use a grouped command. Many controllers treat any command with a common timestamp as a group move: “Any move that involves more than one servo and uses either the speed or time modifier is considered a Group Move, and all servos will start and stop moving at the same time.” [41] Using the **T (time) parameter** in SSC-32 or the Maestro’s group move achieves this [37]. If doing it in software, ensure your loop updates all servos within a very short time (ideally within the same 20 ms frame). Some advanced users generate a single pulse train for all servos offset in the 20 ms period, but libraries/controllers abstract that.

Power and Safety: (Briefly, though hardware, it impacts software) – avoid commanding all servos to move huge angles at maximum speed simultaneously from rest, as this draws a large current spike. It’s better to stagger motions by a few milliseconds or ramp them. Also, if a servo isn’t achieving its commanded position (maybe due to an obstacle or overload), your software might detect that (if feedback available) and stop pushing it further – this prevents burning out a stalled servo. Many smart servos will cut off or reset if stalled, but hobby ones can burn if held against a stop for too long. As a safety measure, some code will not hold a foot in a bad position for long – instead, it might relax or step back if something goes wrong.

Thermal/Current Monitoring: If your servos or driver provide feedback for load or temperature (e.g., Dynamixel servos give load %, or an external current sensor on supply), you can incorporate that in software to avoid overheating. For example, if a particular leg is consistently drawing high current, maybe your gait is causing excessive load – you might adjust the gait to lighten load on that leg (e.g., shorter stance duty or slower speed).

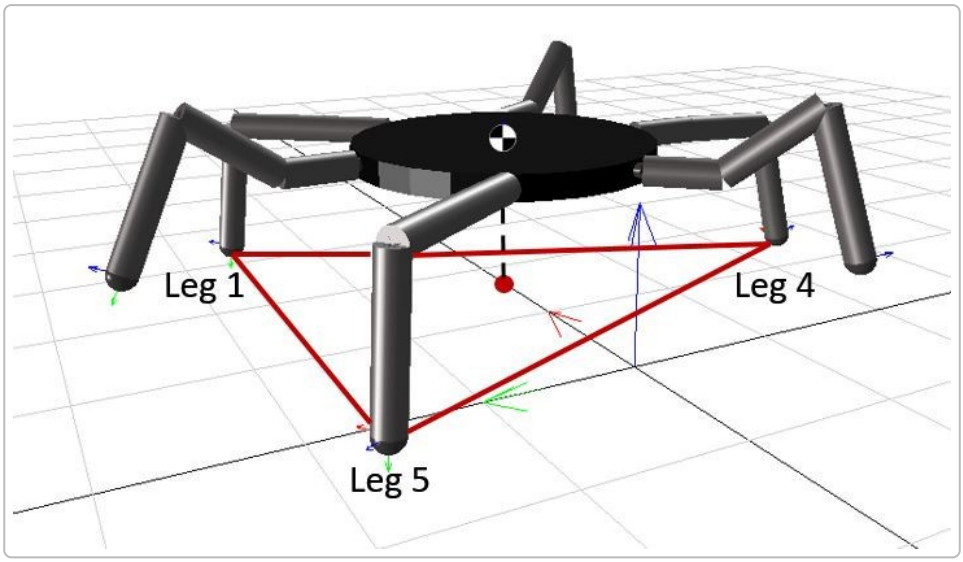
Grouped Packets vs. Individual Updates: Whenever possible, send servo updates in a batch. For instance, the Pololu Maestro has a command to set multiple targets in one USB packet. This not only syncs them but also is more efficient than many small commands. The difference might be millisecond-scale, but it could matter for high-frequency updates. If writing your own serial protocol (for example, you have a secondary MCU receiving commands), define a message that contains all servo positions at once.

Refreshing Idle Servos: Even if a leg is supposed to stay at a given angle (no motion), continue sending that command periodically. Many servos will hold position as long as the signal is present. If your loop is slow or hiccups and the signal stops updating, some servos might jitter or drift. So, maintain the 50 Hz signal to all channels continuously (this is usually handled by hardware PWM once set up, but with a controller you should periodically refresh or use the “hold last position” mode if it has one).

In summary, servo management in software is about **sending the right signals (pulse widths) at the right time**, not exceeding limits, and utilizing any features (speed, sync) to get coordinated motion. When done right, the legs should move **in unison** during a gait phase, reaching the target together and preventing slips.

6. Reminders and Design Considerations

Finally, here are some important design tips and caveats for your hexapod's software and theory:



A statically stable tripod support: the red triangle is the support polygon when three legs are on the ground, and the red dot is the robot's center of mass (COM). The COM must stay within this triangle for stability [42].

- **Center of Gravity and Stability:** Always ensure the robot's center of gravity (projection of COM onto the ground) lies within the support polygon formed by the legs in contact [42]. In tripod gait, the support polygon is the triangle connecting the three ground-contact points. If the COM is near the edge of this triangle, the robot is at risk of tipping if there's any disturbance [42]. Keeping a margin (stability margin) is wise, especially on uneven terrain. You can monitor this by approximating COM based on body and leg positions (some simulations calculate stability margin as in research [43] [44], but a simpler heuristic is to keep movements gentle enough that COM doesn't wildly swing). If you plan dynamic maneuvers (like fast acceleration or body motion), be aware that **dynamic stability** can allow brief outside-COM, but that's advanced; for now design assuming quasi-static balance.

- **Leg Workspace and Singularities:** Understand your leg's workspace (the volume of points the foot can reach). Do not command foot targets outside this space. If IK indicates no solution or a "singular" configuration, handle it gracefully (e.g., limit the command or choose an alternate action). A common singularity is when the leg is fully extended (θ_2 and θ_3 aligned straight) – the leg loses a degree of freedom in that line, and small errors can cause big swings in joint angles. Avoid moves that require the knee to be completely straight or completely folded; keep a slight bend as margin. If your IK solver fails or returns a weird result, have a fallback: for example, if a leg can't reach a point, you might keep it at maximum extension in that direction or move it to a default safe pose. In the MATLAB simulation code we saw, they explicitly check if the robot reached a singular configuration and stop or alert in that case [45]. It's better for your software to

detect these (e.g., by checking if the computed $\cos\theta_3$ is out of bounds or if the required femur angle is extremely high) than to send a bad command.

- **Joint Limits and Safety:** This ties to servo management – always respect joint angle limits (both mechanical and those due to leg design). If the gait planning asks for an extreme angle, clamp it and perhaps issue a warning. It's better for the robot to under-perform a step than to physically strain. Software can also enforce **speed limits** on servos – e.g., don't change a servo by more than X degrees per cycle to prevent sudden jerks.

- **Gait Transitions and Edge Cases:** If you implement multiple gaits (say tripod vs wave, or different speeds), ensure smooth transition between them. Also handle start/stop gracefully: when the robot goes from standing still to walking, ramp up the movements (don't jump straight into a full step), and when stopping, maybe finish a gait cycle completely so it ends with a stable stance. If the robot needs to **side-step or turn on the spot**, plan those motions carefully as they can have different stability considerations (turning in place might reduce the support polygon symmetry).

- **Weight Shifting:** Before lifting a leg, it can be useful to shift the body weight slightly toward the opposite side. For example, if lifting a front-left leg, nudging the body a few mm to the right (and maybe forward/back depending on which leg) can put the COM more over the other legs. This can be done by a small body IK adjustment (moving the body relative to feet) prior to swing. It's an advanced touch but increases stability. In our tripod gait description, the "middle leg angle determines which side is up" essentially references shifting weight using the middle legs [46].

- **Coordination & Phase Timing:** Within a phase, ensure all three legs of the tripod lift and land together – any desynchronization can cause slips. Similarly, the three stance legs should start pushing together. Using group moves or tight timing loops as discussed is crucial. If you find one leg consistently lagging (maybe due to heavier load or slower servo), you might need to tune that (either by giving it slightly advanced commands or using a faster servo for that position).

- **Fall-back Motions:** Think about what the robot should do if something goes wrong. For instance, if a leg doesn't reach its target (maybe an obstacle blocked it or a servo stalled), a simple strategy is to retry or put that leg down immediately to regain support, and perhaps stop the robot. Another scenario: if the robot does tip or a leg slips, detect it (via IMU sudden tilt or a foot not touching when expected) and have a recovery behavior (e.g., immediately drop all legs to a default stance to "catch" itself). While these are safety behaviors, they are important to consider in software design so the robot is robust.

- **Redundancy and Fault Tolerance:** One advantage of hexapods: even if one leg fails, the robot can possibly walk as a quadruped (using the remaining five or even four legs) [47]. While not a basic requirement, you could design your software to handle a disabled leg (imagine you detect a servo died or is not responding). The gait could switch to a alternate pattern using the other legs (there are gaits for 5-legged or 4-legged operation). This is complex but a great robustness feature. At minimum, your code could stop using a leg if it consistently fails to move (e.g., stop sending commands to a jammed servo to prevent further damage).

- **Testing in Simulation:** Before running on hardware, if possible, test your kinematics and gait logic in a simulation or at least a kinematic model. This can catch singularities, coordinate frame mistakes, etc. Many have used tools like MATLAB or V-REP or even a simple Python model to visualize foot paths. It helps ensure that when you run it on the real robot, there are no unexpected large motions that could flip it.

- **Continuous Improvement:** Once the basic walking works, you can gradually refine. For instance, add **body IK** to allow the robot to roll/pitch/yaw the body while keeping feet planted (useful for pointing a sensor or camera on the body without moving feet). Or implement **different gaits** (maybe a faster gait that is quasi-dynamic, or a very slow careful gait for challenging terrain). These improvements build on the same principles: kinematics, gait phase planning, and feedback control.

Remember, developing a walking robot's software is an iterative process. **Start simple:** get it to stand and maybe move one leg at a time. Then implement a basic tripod gait on flat ground. After that works, layer in more advanced features like smoothing, terrain response, and so on. With careful attention to kinematics, timing, and feedback, your tabletop hexapod will reliably scuttle across flat and moderately uneven surfaces, purely through the power of software and sound theoretical planning! Good luck and happy walking!

Sources: The information above is synthesized from hexapod robotics literature and practical guides, including kinematics derivations [5] [11], hobbyist implementations of tripod gait [23], and recent research on terrain-adaptive control [31] [30], among others. These provide a foundation for developing your hexapod's control software.

---

[1] [2] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [19] [20] Inverse Kinematics of a 3-DOF Spider Robot Leg • RAW
https://raw.org/book/robotics/inverse-kinematics-of-a-3-dof-spider-robot-leg/

[3] [PDF] Analysis Of Kinematic For Legs Of A Hexapod Using Denavit ...
https://scispace.com/pdf/analysis-of-kinematic-for-legs-of-a-hexapod-using-denavit-4ma8vze3rl.pdf

[18] Inverse Kinematics for Hexapod and Quadruped Robots - Oscar Liang
https://oscarliang.com/inverse-kinematics-implementation-hexapod-robots/

[21] [22] [42] [43] [44] 3. Fundamentals of Hexapod Robot | Details | Hackaday.io
https://hackaday.io/project/21904-hexapod-modelling-path-planning-and-control/log/62326-3-fundamentals-of-hexapod-robot

[23] [24] [25] [28] [38] [46] Pololu - 4. Sequencing the Hexapod Gait
https://www.pololu.com/docs/0J42/4

[26] [29] [30] [31] [32] [47] Control of a Hexapod Robot Considering Terrain Interaction
https://www.mdpi.com/2218-6581/13/10/142

[27] [45] MATLAB Gait Analysis for Hexapod Robot | WiredWhite
https://wiredwhite.com/matlab-gait-analysis-hexapod-robot/

[33] [34] DC Servo Motor Guide - With ESP32 & Arduino
https://dronebotworkshop.com/servoguide/

[35] 7. Pulse Width Modulation — MicroPython latest documentation
https://docs.micropython.org/en/latest/esp8266/tutorial/pwm.html

[36] [37] [39] [40] [41] SSC-32 Manual
https://wiki.lynxmotion.com/info/wiki/lynxmotion/view/servo-erector-set-system/ses-electronics/ses-modules/ssc-32/ssc-32-manual/