# Hexapod Kinematics with Geometric Algebra: A Comprehensive Tutorial and Implementation Guide

Prepared for Bjørn Remseth

September 15, 2025

## Contents

# 1 Introduction

This document serves as both a comprehensive tutorial on geometric algebra applications in robotics and a practical implementation guide for the `goik-ga` library. We focus specifically on hexapod leg kinematics as a compelling case study that demonstrates the power and elegance of geometric algebra (GA) methods compared to traditional matrix-based approaches.

**Acknowledgment:** This work is strongly inspired by and builds upon the excellent foundation provided by Hans Jørgen Grimstad's `goik` package [1], which pioneered practical hexapod kinematics implementations in Go. Our contribution refactors this work to use geometric algebra primitives for improved mathematical clarity and computational efficiency.

## 1.1 Why This Tutorial Matters

Traditional robotics often relies on rotation matrices, Euler angles, and homogeneous transformations. While functional, these approaches suffer from:

- **Gimbal lock:** Euler angles become singular at certain configurations
- **Computational overhead:** Matrix multiplications are expensive and accumulate numerical errors
- **Frame bookkeeping complexity:** Managing coordinate frame transformations becomes unwieldy
- **Non-intuitive composition:** Multiple transformations require careful ordering and mental overhead

Geometric algebra, particularly through *motors* (the GA equivalent of dual quaternions), provides a unified, efficient, and mathematically elegant alternative that addresses all these issues.

# 2 Geometric Algebra Foundations

Before diving into hexapod kinematics, we establish the mathematical foundations that make our approach both powerful and intuitive.

## 2.1 What is Geometric Algebra?

Geometric algebra extends vector algebra to include oriented areas (bivectors), volumes (trivectors), and higher-dimensional entities. The key insight is that geometric products naturally encode both the magnitude and orientation of geometric transformations.

For 3D space, we work in Clifford algebra $Cl(3, 0, 1)$ with basis elements:

$$\text{Scalars: } 1 \tag{1}$$

$$\text{Vectors: } \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \tag{2}$$

$$\text{Bivectors: } \mathbf{e}_{12}, \mathbf{e}_{23}, \mathbf{e}_{31} \tag{3}$$

$$\text{Trivector: } \mathbf{e}_{123} \tag{4}$$

$$\text{Pseudoscalar: } \mathbf{e}_0 \tag{5}$$

[1] https://github.com/hansj66/goik/tree/main/goik

### 2.2 Projective vs. Conformal Geometric Algebra

Our implementation supports two complementary approaches:

**Projective Geometric Algebra (PGA).** PGA represents rigid motions (rotations + translations) as *motors* [2], which you can view as the geometric-algebraic analog of dual quaternions. The key advantages:

- Unified representation of rotations and translations
- Associative composition: $(M_1 \cdot M_2) \cdot M_3 = M_1 \cdot (M_2 \cdot M_3)$
- Natural interpolation between poses
- Efficient sandwich product for point transformation: $X' = MX\tilde{M}$

**Conformal Geometric Algebra (CGA).** CGA encodes points as null vectors and treats planes, lines, circles, and spheres as native geometric objects [1]. Benefits include:

- Translations become rotations in higher-dimensional space
- Natural meet/join operations for intersections and constraints
- Direct computation of distances and projections
- Elegant handling of degenerate cases

Our current implementation focuses on PGA for production use, with CGA as a parallel API for future enhancement.

## 3 Mathematical Representation Details

### 3.1 Motors as Dual Quaternions

In our PGA implementation, motors are represented as dual quaternions $(r + \varepsilon d)$ where:

- $r$ is a unit quaternion representing rotation
- $d$ is the dual part encoding translation
- $\varepsilon$ is the dual unit with $\varepsilon^2 = 0$

Given axis-angle rotation $(\hat{\mathbf{u}}, \theta)$ and translation $\mathbf{t}$:

$$r = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}\,(\hat{u}_x\mathbf{i} + \hat{u}_y\mathbf{j} + \hat{u}_z\mathbf{k}) \tag{6}$$

$$d = \frac{1}{2}\,\mathbf{t}\,r \tag{7}$$

Composition follows dual quaternion multiplication:

$$(r_1, d_1)(r_2, d_2) = (r_1 r_2,\ r_1 d_2 + d_1 r_2)$$

Point transformation uses the sandwich product:

$$\mathbf{p}' = r\,\mathbf{p}\,r^* + 2\,\mathrm{vec}(d\,r^*)$$

This matches the PGA motor action for SE(3) rigid body motions.

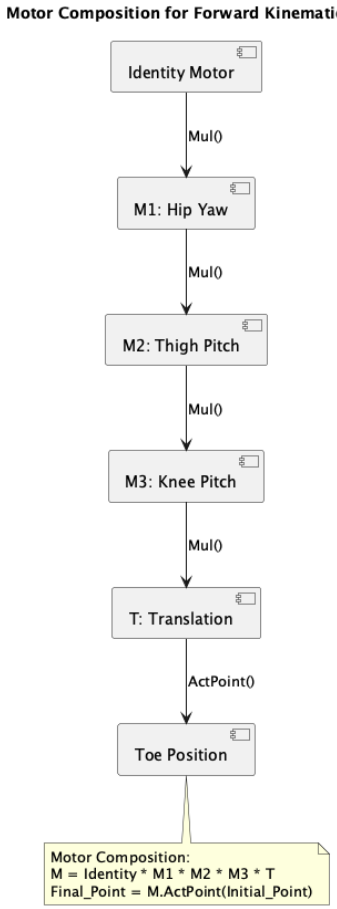


***Figure 1.*** *Motor composition workflow showing the associative nature of geometric transformations. Each motor represents a screw motion (combined rotation and translation), and their composition naturally represents the complete kinematic chain.*

# 4 Hexapod Leg Kinematics: Case Study

Now we apply these concepts to a concrete robotics problem: computing the forward and inverse kinematics of a 3-degree-of-freedom hexapod leg.

## 4.1 Leg Structure and Configuration

Our hexapod leg consists of three joints in series:

1. **Hip Joint (J1):** Yaw rotation about Z-axis, range typically $\pm 45°$
2. **Thigh Joint (J2):** Pitch rotation about Y-axis, forward/backward swing
3. **Knee Joint (J3):** Pitch rotation about Y-axis, leg extension/flexion

The leg geometry parameters are:

- $l_1 = 0.05$ m: Hip offset (lateral displacement to thigh joint)

- $l_2 = 0.20$ m: Thigh length
- $l_3 = 0.20$ m: Shank length

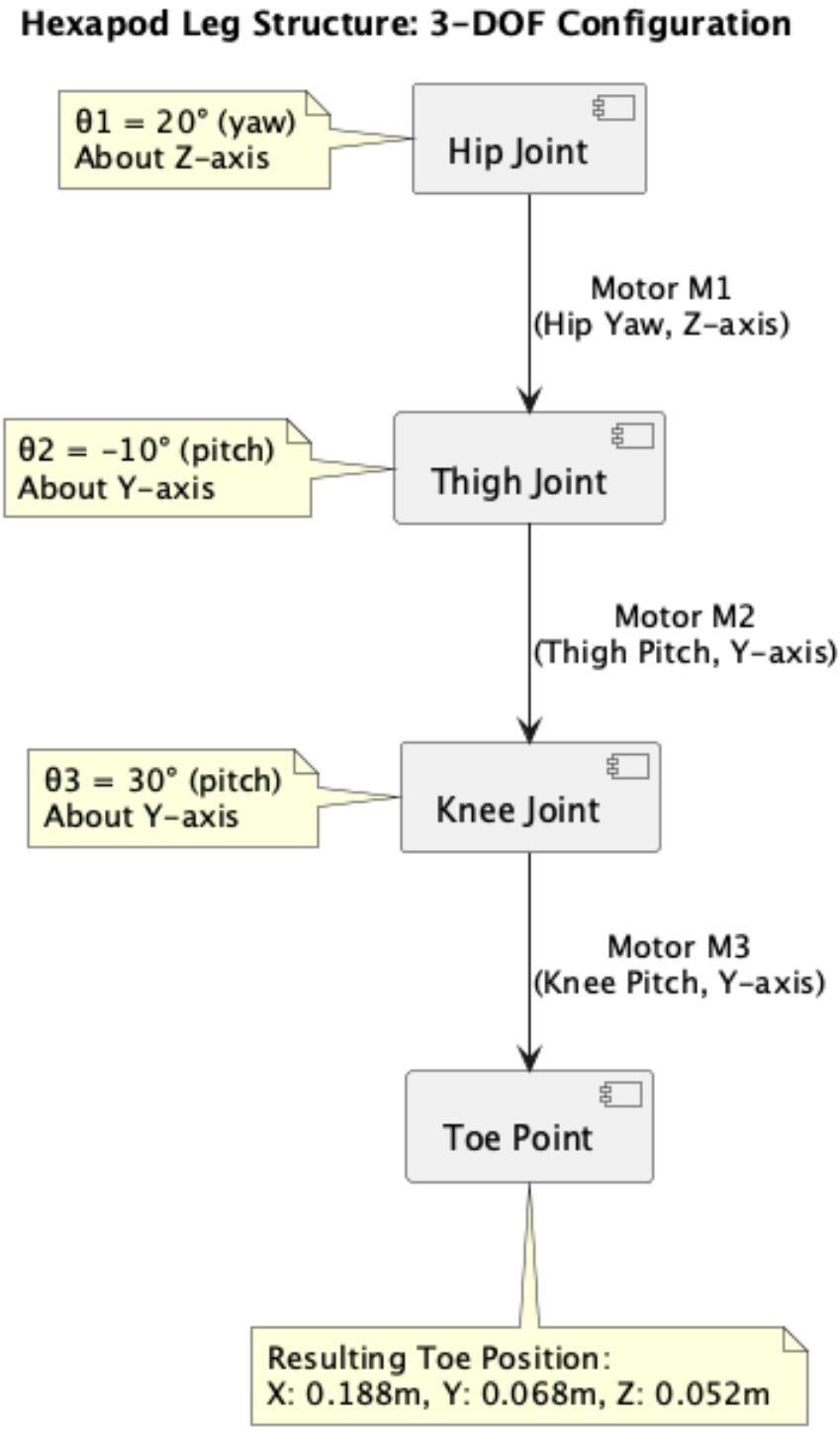


***Figure 2.*** *Hexapod leg structure showing the three joints and their motion axes. Each joint contributes a motor to the overall kinematic chain.*

## 4.2 Coordinate Frame Relationships

The kinematic chain establishes a sequence of coordinate frame transformations from the robot base to the toe. Each transformation is represented by a motor that encodes both the joint rotation and any necessary translation.

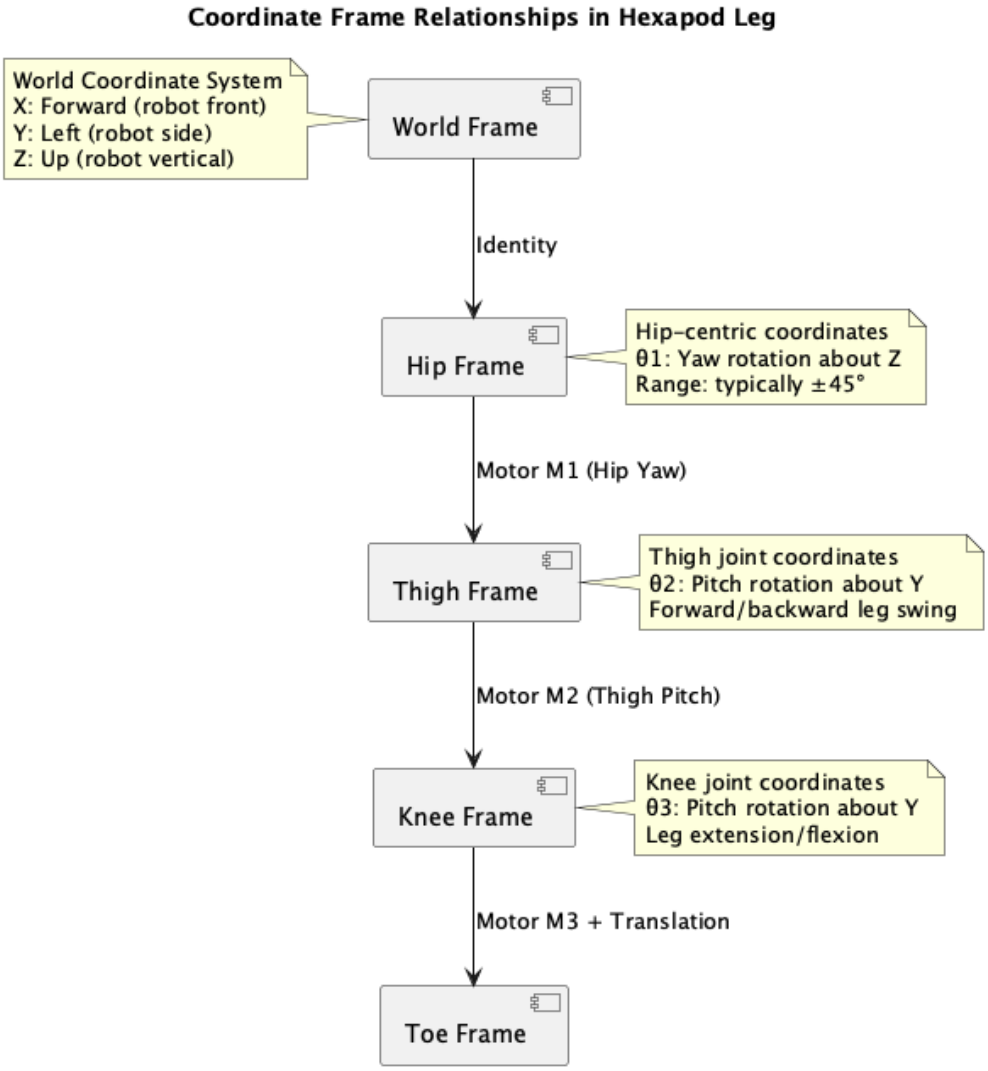


***Figure 3.*** *Coordinate frame relationships in the hexapod leg. Each frame transformation is represented by a geometric algebra motor, eliminating the need for separate rotation matrices and translation vectors.*

The coordinate system conventions are:

- **X-axis:** Forward direction (leg extension)
- **Y-axis:** Lateral direction (leg abduction)
- **Z-axis:** Vertical direction (leg lift)

# 5 Forward Kinematics Implementation

## 5.1 Motor Chain Construction

Forward kinematics involves constructing a chain of motors and applying them to compute the toe position. Each joint contributes a screw motor that combines rotation about the joint axis with any necessary translation.

***Listing 1.*** *Forward kinematics implementation in Go*

```go
func main() {
    // Physical parameters
    l1 := 0.05  // hip offset to thigh (m)
    l2 := 0.20  // thigh length
    l3 := 0.20  // shank length

    // Joint centers
    hip := pga.V(0, 0, 0)
    thighJ := hip.Add(pga.V(l1, 0, 0))
    kneeJ := thighJ.Add(pga.V(l2, 0, 0))

    // Joint axes
    z := pga.V(0, 0, 1)  // Hip yaw axis
    y := pga.V(0, 1, 0)  // Pitch axes

    // Joint angles (example configuration)
    theta1 := 20 * math.Pi / 180   // Hip yaw
    theta2 := -10 * math.Pi / 180  // Thigh pitch
    theta3 := 30 * math.Pi / 180   // Knee pitch

    // Build motor chain
    M := pga.Identity().
        Mul(pga.Screw(hip, z, theta1, 0)).       // Hip yaw
        Mul(pga.Screw(thighJ, y, theta2, 0)).    // Thigh pitch
        Mul(pga.Screw(kneeJ, y, theta3, 0)).     // Knee pitch
        Mul(pga.Translator(pga.V(l3, 0, 0)))     // Toe offset

    // Compute toe position
    toe0 := pga.V(0, 0, 0)  // Initial toe position
    toe := M.ActPoint(toe0)  // Final toe position
}
```

## 5.2 Geometric Interpretation

The beauty of the motor approach is that each transformation has a clear geometric interpretation:

$$\mathbf{P}_{\text{toe}} = M_{\text{total}} \cdot \mathbf{P}_0 \cdot \tilde{M}_{\text{total}} \tag{8}$$

$$\text{where } M_{\text{total}} = M_{\text{identity}} \cdot M_{\text{hip}} \cdot M_{\text{thigh}} \cdot M_{\text{knee}} \cdot M_{\text{translation}} \tag{9}$$

Each motor $M_i$ represents a screw motion:

$$M_i = \text{Screw}(\mathbf{center}, \mathbf{axis}, \theta_i, \text{pitch})$$

For pure rotations, the pitch is zero. The final translation motor moves from the knee joint to the toe along the shank.

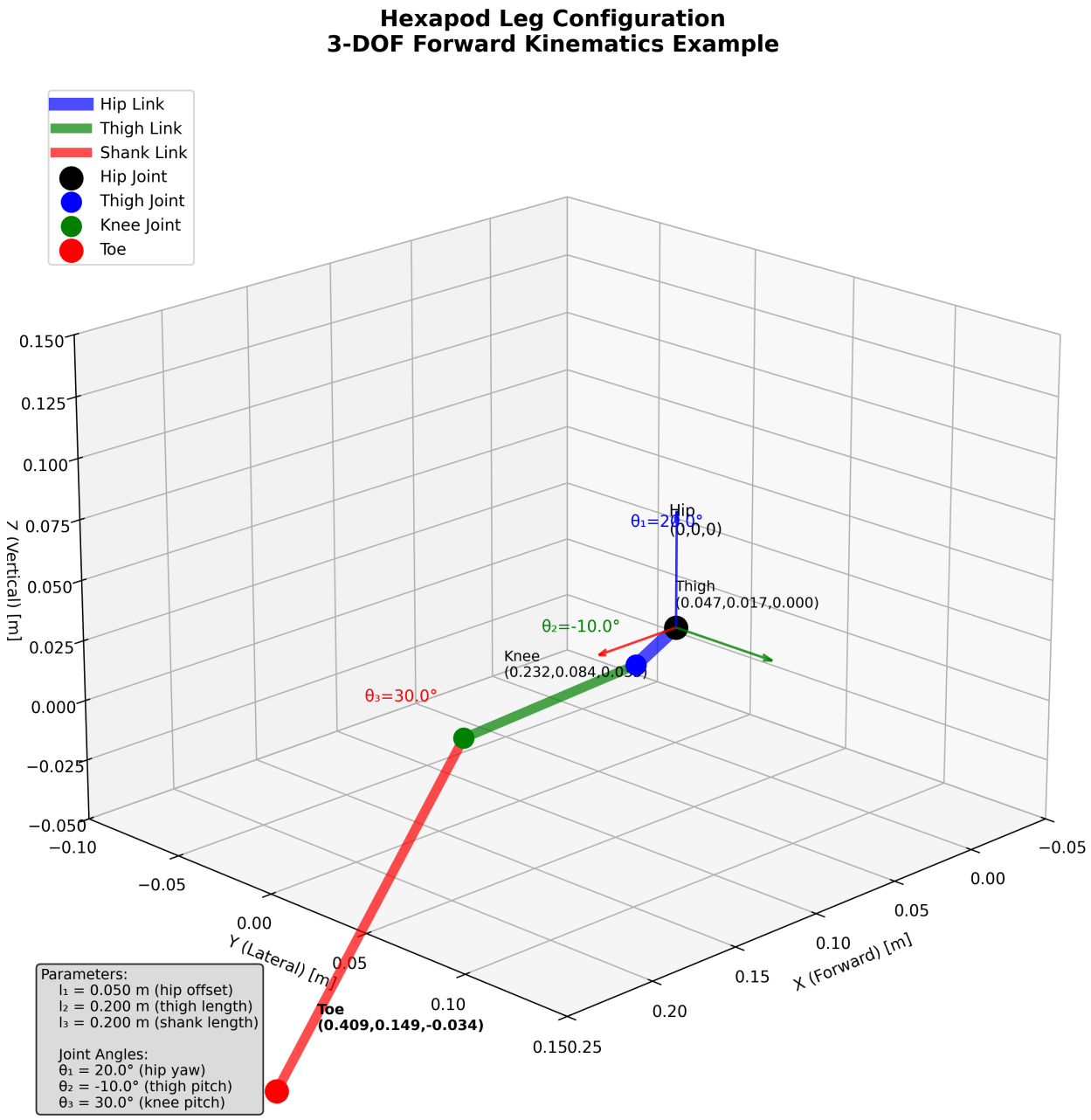


***Figure 4.*** *3D visualization of the hexapod leg configuration showing joint positions, link lengths, and the resulting toe position. The coordinate frames and joint angles are clearly visible, demonstrating the geometric relationships in the kinematic chain.*

## 6 Velocity Analysis and Jacobians

For motion control and trajectory planning, we need to understand how joint velocities map to end-effector velocities. This relationship is captured by the Jacobian matrix.

### 6.1 Jacobian Column Computation

Each column of the Jacobian represents the instantaneous linear velocity of the toe when the corresponding joint rotates at unit angular velocity. For a revolute joint, this is computed using the cross product:

$$\mathbf{J}_i = \hat{\mathbf{u}}_i \times (\mathbf{p}_{\text{toe}} - \mathbf{c}_i)$$

where:

- $\hat{\mathbf{u}}_i$ is the unit axis of joint $i$
- $\mathbf{c}_i$ is the center point of joint $i$
- $\mathbf{p}_{\text{toe}}$ is the current toe position

***Listing 2.*** *Jacobian computation for velocity analysis*

```
// Jacobian columns at current pose (linear velocity for each joint)
J1 := pga.RevoluteColumn(hip, z, toe)      // Hip yaw contribution
J2 := pga.RevoluteColumn(thighJ, y, toe)   // Thigh pitch contribution
J3 := pga.RevoluteColumn(kneeJ, y, toe)    // Knee pitch contribution

// Build complete Jacobian matrix: J = [J1 J2 J3]
// Joint velocities map to toe velocity: v_toe = J * [theta_dot1 theta_dot2
    theta_dot3]^T
```

### 6.2 Physical Interpretation of Jacobian Columns

Each Jacobian column has a clear physical meaning:

- **J1 (Hip):** Creates circular motion in the XY plane around the Z-axis
- **J2 (Thigh):** Affects primarily X and Z components, swinging the leg forward/backward
- **J3 (Knee):** Contributes to leg extension, affecting X and Z components

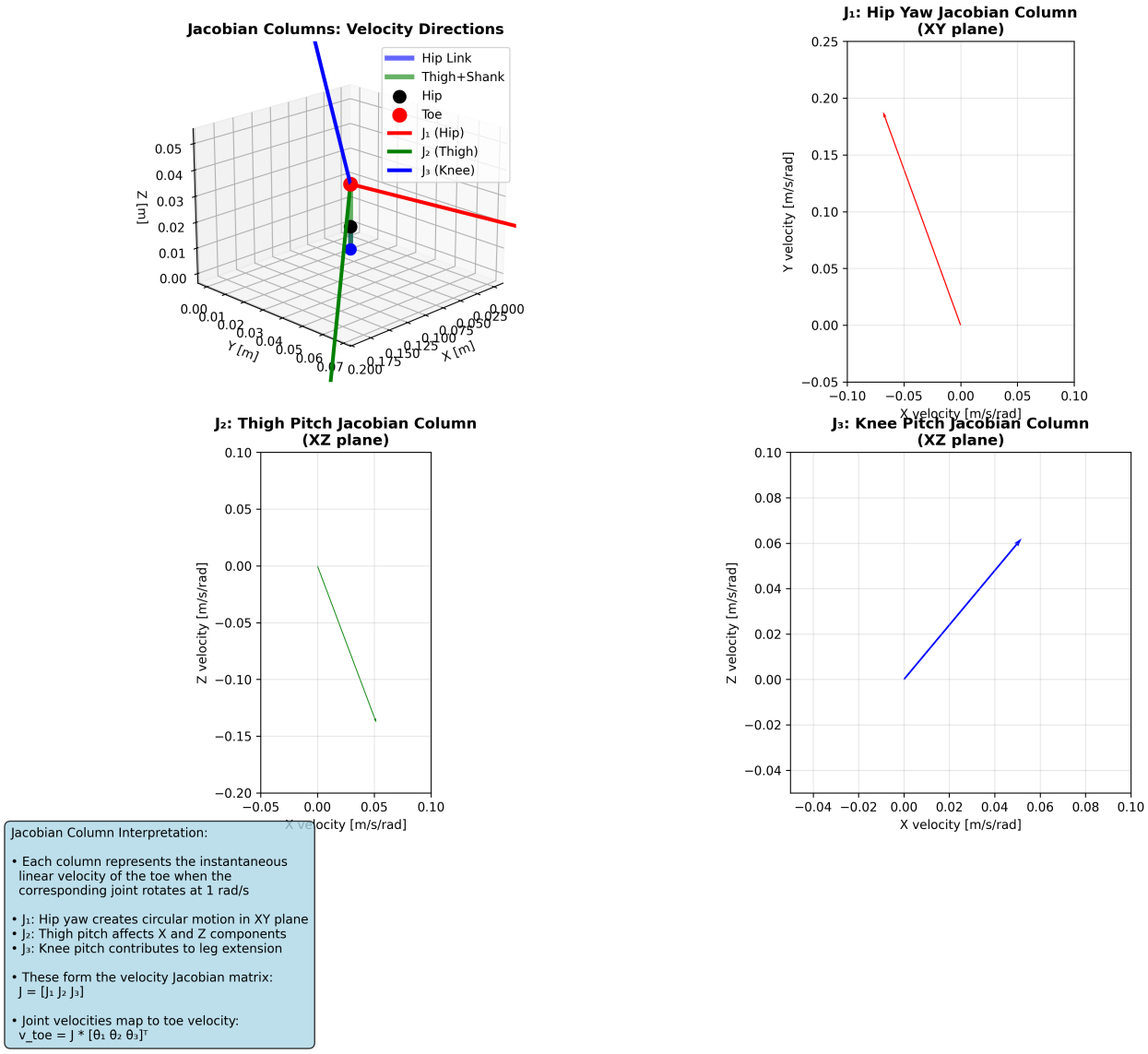


***Figure 5.*** *Visualization of Jacobian columns showing the instantaneous velocity directions for each joint. The 3D plot shows how each joint contributes to toe motion, while the 2D projections illustrate the velocity components in different planes.*

# 7 Inverse Kinematics Strategies

While forward kinematics is straightforward with motors, inverse kinematics requires more sophisticated approaches. We present both analytical and numerical methods.

## 7.1 Analytical Inverse Kinematics

For many hexapod leg configurations, analytical solutions exist by exploiting the geometric structure:

**Hip Yaw Angle.** The hip yaw angle can be computed from the desired toe position projection:

$$\theta_1 = \text{atan2}(y_{\text{toe}}, x_{\text{toe}})$$

**Planar Two-Link Problem.** After accounting for hip yaw, the thigh-knee pair forms a planar two-link mechanism. Using the cosine law on the triangle formed by hip, knee, and toe:

$$d = \sqrt{x_{\text{toe}}^2 + z_{\text{toe}}^2} \tag{10}$$

$$\cos(\theta_3) = \frac{d^2 - l_2^2 - l_3^2}{2l_2 l_3} \tag{11}$$

$$\theta_2 = \text{atan2}(z_{\text{toe}}, x_{\text{toe}}) - \text{atan2}(l_3 \sin(\theta_3), l_2 + l_3 \cos(\theta_3)) \tag{12}$$

### 7.2 Numerical Inverse Kinematics

For more complex configurations or constraints, numerical methods using the Jacobian provide a general solution:

$$\Delta\boldsymbol{\theta} = J^\dagger(\mathbf{x}^* - \mathbf{x})$$

where $J^\dagger$ is the pseudoinverse of the Jacobian, providing damped least-squares solutions that handle singularities gracefully.

## 8 Performance and Robustness Analysis

### 8.1 Computational Complexity

The motor-based approach offers significant computational advantages:

- **Forward kinematics:** $O(n)$ motor multiplications vs. $O(n^3)$ matrix operations
- **Memory efficiency:** 8 floats per motor vs. 16 floats per homogeneous matrix
- **Numerical stability:** Quaternion normalization is cheaper and more stable than matrix orthogonalization

### 8.2 Robustness Considerations

Geometric algebra methods are inherently more robust:

- **No gimbal lock:** Motors avoid the singularities of Euler angles
- **Smooth interpolation:** Motor SLERP provides natural pose interpolation
- **Constraint satisfaction:** CGA naturally handles geometric constraints
- **Automatic differentiation:** GA structure supports efficient gradient computation for optimization

## 9 Contact Constraints and Ground Interaction

Real hexapod locomotion requires handling ground contact constraints. Geometric algebra provides elegant tools for this.

### 9.1 Ground Plane Representation

In CGA, the ground plane $\Pi$ is represented as a bivector. Contact occurs when the toe point $P$ satisfies:

$$\Pi \wedge P = 0$$

This naturally handles:

- Distance to ground: $\Pi \cdot P$
- Projection onto ground: $P - (P \cdot \Pi)\Pi/|\Pi|^2$
- Contact normal forces: aligned with $\Pi$

### 9.2 Friction and Stability

The geometric approach extends naturally to:

- **Friction cones:** Represented as GA multivectors
- **Stability margins:** Computed using convex hull operations in GA
- **Force distribution:** Optimized using GA-based quadratic programming

## 10 Implementation Architecture

### 10.1 Package Structure

The `goik-ga` library is organized into focused packages:

- `pga/`: Production PGA implementation with motors, points, and Jacobians
- `cga/`: Parallel CGA API (currently routing to PGA backend)
- `examples/`: Demonstration code including the hexapod leg case study

### 10.2 Key Design Principles

1. **Zero external dependencies:** Production-lean implementation
2. **Consistent APIs:** PGA and CGA packages mirror each other
3. **Performance focus:** Optimized for real-time robotics applications
4. **Mathematical clarity:** Code structure reflects GA mathematical operations

## 11 Migration Guide from Traditional Methods

### 11.1 Replacing Rotation Matrices

Replace sequences of rotation matrices with motor composition:

***Listing 3.*** *Traditional vs. Motor approach*

```
// Traditional approach with matrices
R1 := RotationZ(theta1)
R2 := RotationY(theta2)
R3 := RotationY(theta3)
T := Translation(offset)
M_total := T.Mul(R3).Mul(R2).Mul(R1)
point_new := M_total.Mul(point_old)

// Motor approach
M := pga.Identity().
    Mul(pga.Screw(center1, z_axis, theta1, 0)).
    Mul(pga.Screw(center2, y_axis, theta2, 0)).
    Mul(pga.Screw(center3, y_axis, theta3, 0)).
    Mul(pga.Translator(offset))
point_new := M.ActPoint(point_old)
```

### 11.2 Benefits Realized

The migration provides immediate benefits:

- Reduced computational cost
- Elimination of matrix orthogonalization issues
- Natural interpolation between poses
- Unified representation of rotations and translations

## 12 Future Directions and Advanced Applications

### 12.1 Full CGA Implementation

The current CGA package is an API sketch. Full implementation would enable:

- Native circle and sphere operations for joint limits
- Elegant collision detection using meet/join operations
- Direct optimization of geometric constraints
- Natural handling of parallel and intersecting axes

### 12.2 Integration with Modern Control

Geometric algebra provides a foundation for advanced control methods:

- **Model Predictive Control (MPC):** GA gradients for efficient optimization
- **Adaptive control:** Motor-based parameter estimation
- **Learning-based methods:** GA structure for neural network layers
- **Distributed control:** Motor synchronization across multiple legs

## 13 Conclusion

This tutorial demonstrates that geometric algebra, through the use of motors, provides a superior approach to hexapod kinematics compared to traditional matrix methods. The key advantages include:

1. **Mathematical elegance:** Unified representation of rotations and translations
2. **Computational efficiency:** Reduced complexity and improved numerical stability
3. **Conceptual clarity:** Direct geometric interpretation of operations
4. **Extensibility:** Natural path to advanced geometric operations

The `goik-ga` implementation provides a practical, production-ready foundation for these methods. Whether you're developing hexapod robots, manipulator arms, or other kinematic systems, geometric algebra offers both theoretical insights and practical advantages that justify adoption.

As robotics continues to demand more sophisticated motion control and geometric reasoning, geometric algebra represents not just an alternative approach, but the natural mathematical language for spatial transformations and constraints.

## References


[1] Leo Dorst, Daniel Fontijne, and Stephen Mann. *Geometric Algebra for Computer Science: An Object-Oriented Approach to Geometry*. Morgan Kaufmann, 2009. Available on Amazon: https://www.amazon.com/Geometric-Algebra-Computer-Science-Object-Oriented/dp/0123749425.

[2] Charles Gunn. On the homogeneous model of euclidean geometry. arXiv preprint arXiv:1101.4542, 2011. PGA overview and applications. Available at: https://arxiv.org/abs/1101.4542.