ASSIGNMENT PROJECT

Assignment Project on 3d Arm Modeling and Programming using Double Pendulum Equation

Name	: Jeremia Christ Immanuel Manalu
NRP	: 5023231017
Course	: Biomodelling (A)
Class	: A
Lecturer	: Dr. Achmad Arifin S.T., M.Eng.
Department	: Biomedical Engineering

FACULTY OF INTELLIGENT ELECTRICAL AND INFORMATICS TECHNOLOGY

INSTITUT TEKNOLOGI SEPULUH NOPEMBER

2025

CHAPTER I. FUNDAMENTAL THEORY

1.1. Kinetic Energy

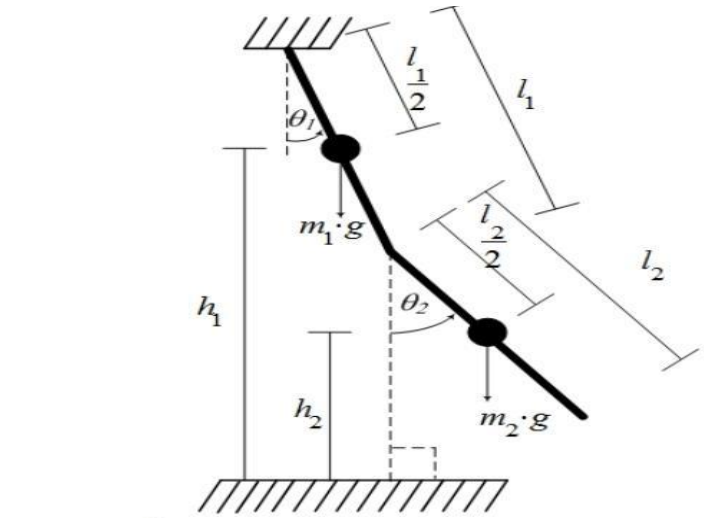

Figure 1.1. *An Example of Double Pendulum Properties*

The total kinetic energy of the system contains the translational contribution of the center of mass (COM) of each segment and the rotation of each segment. For a double-pendulum consisting of two segments (index 1 and 2) with masses m_1, m_2, COM distances r_1, r_2 from the joint axis, moments of inertia I_1, I_2, bone/rod lengths l_1, l_2, and general angles θ_1, θ_2 (for example θ_1= shoulder angle, θ_2 = relative angle at the elbow). The COM positions and velocities can be written down and then inserted into the kinetic energy.

Position (writing example) for COM:

$$x_1 = r_1 \sin\theta_1, \qquad y_1 = -r_1 \cos\theta_1,$$
$$x_2 = l_1 \sin\theta_1 + r_2 \sin(\theta_1 + \theta_2), \qquad y_2 = -l_1 \cos\theta_1 - r_2 \cos(\theta_1 + \theta_2).$$

The squared velocity of each COM (with time derivative $\dot{\theta}_i$) yields the expression v_1^2, v_2^2. The full kinetic energy:

$$T = \frac{1}{2} m_1 v_1^2 + \frac{1}{2} I_1 \dot{\theta}_1^2 + \frac{1}{2} m_2 v_2^2 + \frac{1}{2} I_2 (\dot{\theta}_1 + \dot{\theta}_2)^2.$$

By plugging in v_1^2 and v_2^2 (their derivatives) we get a form that is usually written as the square of $\dot{\theta}_1$, $\dot{\theta}_2$ plus the cross term $\dot{\theta}_1 \dot{\theta}_2$. One frequently used form, emphasizing the structure of the inertia matrix, is:

$$T = \frac{1}{2} [\dot{\theta}_1 \quad \dot{\theta}_2] M(\theta_1, \theta_2) \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix},$$

where $M(\theta)$is a symmetric configuration-dependent mass/inertia matrix containing $I_i, m_i, l_i, r_i$and trigonometric functions like $\cos(\theta_2)$. The full equation for Kinetic Energy is as follows:

$$E_K = \tfrac{1}{2} m_1 v_1^2 + \tfrac{1}{2} I_1 \dot{\theta}_1^2 + \tfrac{1}{2} m_2 v_2^2 + \tfrac{1}{2} I_2 \dot{\theta}_2^2$$

$$E_K = \tfrac{1}{2} m_1 \left(\tfrac{l_1}{2}\right)^2 \dot{\theta}_1^2 + \tfrac{1}{2} m_2 \left[l_1^2 \dot{\theta}_1^2 + \left(\tfrac{l_2}{2}\right)^2 \dot{\theta}_2^2 + 2 l_1 \left(\tfrac{l_2}{2}\right) \cos(\theta_1 - \theta_2) \dot{\theta}_1 \dot{\theta}_2 \right] + \tfrac{1}{2} I_1 \dot{\theta}_1^2 + \tfrac{1}{2} I_2 \dot{\theta}_2^2$$

$$E_K = \tfrac{1}{2}\left(m_1\frac{l_1^2}{4} + m_2 l_1^2 + I_1\right)\dot{\theta}_1^2 + \tfrac{1}{2}\left(m_2\frac{l_2^2}{4} + I_2\right)\dot{\theta}_2^2 + m_2\frac{l_1 l_2}{2}\cos(\theta_1 - \theta_2)\,\dot{\theta}_1\dot{\theta}_2$$

Where:

- $m1$, $m2$: mass of the upper arm and lower arm
- $l1$, $l2$: length of the upper arm and lower arm
- $I1$, $I2$: moment of inertia of each segment about its center of mass
- $\theta 1$, $\theta 2$: angle of rotation of the upper arm and lower arm about the reference axis
- $\dot{\theta}1$, $\dot{\theta}2$: angular velocity of the upper arm and lower arm

In simulations, kinetic energy can be used to determine the speed and impact of collisions between objects.

1.2. Potential Energy

Gravitational potential energy is computed from the vertical heights of each segment's center of mass relative to a datum (e.g. $y = 0$). For the example configuration:

$$V = m_1 g y_1 + m_2 g y_2.$$

Substituting y_1, y_2yields a form such as:

$$V = -m_1 g r_1 \cos\,\theta_1 - m_2 g(l_1 \cos\,\theta_1 + r_2 \cos\,(\theta_1 + \theta_2)),$$

Also noted that the sign convention depends on the chosen positive axis, the important point is that $\partial V / \partial\theta_i$produces the gravitational torques. This expression supplies the conservative-force terms that appear later in the Lagrangian equations. From **Figure 1.1**, the height of the center of mass of the upper arm and lower arm (m1 and m2) is given in the following equation:

$$h_1 = l_1 + l_2 - \tfrac{l_1}{2}\cos\theta_1,$$

$$h_2 = l_1 + l_2 - \left(l_1\cos\theta_1 + \tfrac{l_2}{2}\cos\theta_2\right).$$

The potential energy when substituted will be as in the given equation:

$$E_P = m_1 g h_1 + m_2 g h_2$$

$$E_P = m_1 g\left(l_1 + l_2 - \tfrac{l_1}{2}\cos\theta_1\right) + m_2 g\left[l_1 + l_2 - \left(l_1\cos\theta_1 + \tfrac{l_2}{2}\cos\theta_2\right)\right].$$

1.3. Torque

Torque (generalized torque τ_i) on generalized coordinate $\theta_i$is defined as virtual work per unit angular displacement. In the Lagrangian framework the relation between kinetic energy T, potential energy V, and external torques $\tau_i$is given by the Euler–Lagrange equations (see the Lagrangian section). Briefly:

For generalized coordinates q_i,

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = Q_i,$$

with $L = T - V$and Q_i the generalized (non-conservative) forces (e.g. applied torques). Thus, applied torques τchange the rate of generalized momentum $\partial L / \partial \dot{q}$, while gradients of the potential $\partial L / \partial q$produce gravitational torques (components of $G(\theta)$).

Common matrix form for multibody systems:

$$M(\theta)\,\ddot{\theta} + C(\theta, \dot{\theta})\,\dot{\theta} + G(\theta) = \tau,$$

where:

- $M(\theta)$ derives from T(mass/inertia matrix),
- $C(\theta, \dot{\theta})\dot{\theta}$ are Coriolis/centripetal terms coming from cross derivatives in T,

- $G(\theta) = \partial V / \partial \theta$ is the gravity torque vector,
- τ is the external/actuator torque vector.

1.4. Degree of Freedom

DOF is the number of independent coordinates required to describe the configuration of a mechanical system. For example, an ideal planar double pendulum without constraints has 2 DOF: θ_1, θ_2. For rigid bodies in 3D, DOF is based on $6N$minus constraints.

- General formulas for DOF:
  - A rigid body in 3D: 6 DOF per body; in 2D: 3 DOF per body.
  - If there are $k$holonomic constraints, total DOF = $6N - k$ (for 3D rigid bodies).
- For generalized coordinates, choose $q = [q_1, \ldots, q_n]$ with $n =$ DOF. Equations of motion are expressed in terms of $q, \dot{q}, \ddot{q}$. For a planar double pendulum $q = [\theta_1, \theta_2]$.

Note that the choice of coordinates (absolute vs relative angles, etc.) changes the algebraic form of M, C, Gbut not the DOF count.

1.5. Lagrangian Mechanics

- Lagrangian: $L(q, \dot{q}, t) = T(q, \dot{q}) - V(q)$.
- Euler–Lagrange equations for each generalized coordinate q_i:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = Q_i,$$

where $Q_i$are generalized non-conservative forces/torques. If only gravity acts and there is no non-conservative forcing, $Q_i = \tau_i$represents input torques; for passive systems $Q_i = 0$.

- Matrix form commonly used:

$$M(q)\,\ddot{q} + C(q, \dot{q})\,\dot{q} + G(q) = \tau.$$

Terms used:

- $M(q)$: symmetric positive-definite mass/inertia matrix,
- $C(q, \dot{q})\dot{q}$: Coriolis and centripetal contributions (contains products $\dot{q}_i \dot{q}_j$),
- $G(q)$: conservative gravity vector,
- τ: input torques/damping/friction if present.
- Energy relation: $\partial L / \partial \dot{q}$is generalized momentum; its time derivative equals applied generalized forces minus conservative gradients. Lecture materials show the step-by-step derivation from positions → velocities → $T, V \rightarrow L \rightarrow$ equations of motion.

1.6. Fourth Order Runge-Kutta Method

RK4 is an explicit numerical integrator for first-order ODE systems below:

$$\dot{y} = f(t, y), y(t_0) = y_0.$$

With timestep $h$and $y_n \approx y(t_n)$:

$$\begin{aligned} k_1 &= f(t_n, y_n), \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right), \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right), \\ k_4 &= f(t_n + h, y_n + h\,k_3), \\ y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4). \end{aligned}$$

For second-order systems (like $\ddot{q} = g(q, \dot{q}, t)$) convert to first-order form with the state vector:

$$x = \begin{bmatrix} q \\ \dot{q} \end{bmatrix}, \dot{x} = \begin{bmatrix} \dot{q} \\ g(q, \dot{q}, t) \end{bmatrix},$$

and apply RK4 to $\dot{x} = F(t, x)$. There are also Runge–Kutta–Nyström variants that directly integrate second-order equations; the RK4 principle. combining four slope estimates for fourth-order accuracy, remains the same.

1.7. Other Important Parameters

a. Inertia/mass matrix $M(q)$

Derived from the quadratic form of kinetic energy. If the COM positions change with configuration, the inertia matrix depends on q. $M(q)$must be symmetric and positive-definite for physical consistency.

b. Coriolis and centripetal terms $C(q, \dot{q})$

Arise from time derivatives of $\partial T / \partial \dot{q}$; typically contain terms proportional to $\dot{q}_i \dot{q}_j$and trig functions. Practically represented as a matrix multiplying $\dot{q}$or as a vector $C(q, \dot{q})\dot{q}$.

c. Gravity vector $G(q)$

Obtained from $\partial V / \partial q$; produces restoring torques toward equilibrium. Accurate G is important for torque compensation.

d. Damping/friction

Realistic models include damping (e.g. linear $-D\dot{q}$) or nonlinear friction; damping removes energy and aids numerical stability.

CHAPTER II. RESULT AND ANALYSIS

2.1. Problems Statement

Use anthropometric data of a subject with bw=60 kg and bh=160 cm to define segment length, mass, position of center of mass, and moment of inertia of upper arm and lower arm as double pendulum model. Derive motion equation of double pendulum with shoulder and elbow joint dynamics. Solve the motion equation using RK IV integration method and perform passive movement test (active torques=0.0). Write your report of all steps of this assignment. By learning how to develop a 3d rendering using OpenGL, visualize your model of 3d movements.

By referring to 3d hand robot program that I shared to you, develop 3d visualization of upper limb consists of joints and segments as closed as possible to nature of human upper limb. Do render the model of 3d movement by generate trajectory using sinusoidal function with specified range of motion in step of development of your visualization program.

Motion of the model should be generated from integration of motion equations. Your 3d visualization program shows the trajectories of elbow joint and shoulder joint angles in passive test released from initial angles.

2.2. Code Explanation

The following program is developed in Delphi 12 (RAD Studio Athens) and structured around a component named TPageControl component, with each of the eight tabs corresponding to a distinct step in the linear prediction process. The logic is event-driven, with each step being initiated by a button click.

a. Top-level/Interface area

```
1.  unit Unit3DArmModel;
2.
3.  interface
4.
5.  uses
6.    Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
    System.Classes, Vcl.Graphics,
7.    Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls, Vcl.ExtCtrls,
8.    VclTee.TeeGDIPlus, VCLTee.TeEngine, VCLTee.TeeProcs,
    VCLTee.Chart,
9.    VCLTee.Series, Vcl.Buttons, Vcl.ComCtrls,
10.   opengl, math;
11.
12. type
13.   TSimMode = (smKinematic, smPhysics);
14.   TPhysicsStateVector = array[0..5] of Double;
15.
16.   TForm1 = class(TForm)
17.     Timer1: TTimer;
```

```
18.    btnStartStop: TButton;
19.    btnReset: TButton;
20.    Panel2: TPanel;
21.    pnlOpenGL: TPanel;
22.    GroupBox1: TGroupBox;
23.    rgMode: TRadioGroup;
24.    GroupBox2: TGroupBox;
25.    ChartTrajectories: TChart;
26.    Series1: TLineSeries;
27.    Series2: TLineSeries;
28.    Series3: TLineSeries;
29.    ListBoxLog: TListBox;
30.    BitBtnExit: TBitBtn;
31.    ScrollBox1: TScrollBox;
32.    GroupBox3: TGroupBox;
33.    TrackBarPitch: TTrackBar;
34.    TrackBarYaw: TTrackBar;
35.    TrackBarRoll: TTrackBar;
36.    Label1: TLabel;
37.    Label2: TLabel;
38.    Label3: TLabel;
39.    lblPitchValue: TLabel;
40.    lblYawValue: TLabel;
41.    lblRollValue: TLabel;
42.    cbShowAxes: TCheckBox;
43.    procedure FormCreate(Sender: TObject);
44.    procedure FormDestroy(Sender: TObject);
45.    procedure FormResize(Sender: TObject);
46.    procedure Timer1Timer(Sender: TObject);
47.    procedure btnStartStopClick(Sender: TObject);
48.    procedure btnResetClick(Sender: TObject);
49.    procedure rgModeClick(Sender: TObject);
50.    procedure BitBtnExitClick(Sender: TObject);
51.    procedure TrackBarPitchChange(Sender: TObject);
52.    procedure TrackBarYawChange(Sender: TObject);
53.    procedure TrackBarRollChange(Sender: TObject);
54.    procedure cbShowAxesChange(Sender: TObject);
55.  private
56.    { Private Declarations }
57.    myDC : HDC;
58.    myRC : HGLRC;
59.    CurrentMode: TSimMode;
60.    PhysicsState: TPhysicsStateVector;
61.    time: Double;
62.    shoulder_yaw_deg, shoulder_pitch_deg, shoulder_roll_deg: Double;
63.    elbow_pitch_deg: Double;
```

```
64.    wrist_roll_deg, wrist_yaw_deg, wrist_pitch_deg: Double;
65.    camera_pitch_deg, camera_yaw_deg, camera_roll_deg: Double;
66.    TrajectorySeries: array of TLineSeries;
67.    FontListBase: GLuint;
68.
69.    procedure SetupPixelFormat(ADc: HDC);
70.    procedure BuildFont;
71.    procedure RenderText(x, y, z: GLfloat; const text: string);
72.    procedure DrawScene;
73.    procedure DrawAxes;
74.    procedure DrawTrajectoryArcs;
75.    procedure DrawHandAndFingers;
76.    procedure DrawCube(size: GLfloat);
77.    procedure InitializeState;
78.    procedure GetAccelerations(const AState: TPhysicsStateVector; var
    ADerivs: TPhysicsStateVector);
79.    procedure RungeKutta4_Step(var AState: TPhysicsStateVector;
    ATimeStep: Double);
80.    procedure UpdateCameraLabels;
81.  public
82.    { Public Declarations }
83.  end;
```

- **unit Unit3DArmModel; & interface**
  Declares the module that implements the 3D arm demo and physics. The rest of the file is the interface (public form class and signatures) and implementation.
- **uses clause**
  Imports Windows, VCL GUI, charting, OpenGL and math helpers. Key libraries:
  - opengl, OpenGL bindings used for rendering (gluSphere, glRotatef, etc.)
  - math, trig and numeric helpers (Sin, Cos, Sqr, DegToRad, RadToDeg).

  These libraries explain why later code calls gluNewQuadric, glRotatef, Sin, Cos, etc.

- **Types**
  - TSimMode = (smKinematic, smPhysics); mode switch between a kinematic demo (predefined sinusoids) and the physics-integrated simulation.
  - TPhysicsStateVector = array[0..5] of Double; the 6-element state vector used by the integrator. The code maps it as:

$$AState[0] = q_1 \quad \text{(shoulder yaw)}, AState[1] = \dot{q}_1$$
$$AState[2] = q_2 \quad \text{(shoulder pitch)}, AState[3] = \dot{q}_2$$
$$AState[4] = q_3 \quad \text{(elbow pitch)}, AState[5] = \dot{q}_3$$

    This ordering is used throughout GetAccelerations and RK4.

- **TForm1 class**
  Declares GUI controls (timer, buttons, chart, trackbars) and private fields:
  - myDC, myRC, device/context handles for OpenGL.

- CurrentMode, which demo mode is active.
- PhysicsState: TPhysicsStateVector, the simulation state.
- time, simulation time.
- many *_deg fields, cached joint angles in degrees for drawing.
- TrajectorySeries array, references to chart series used to plot angles over time.
- FontListBase, OpenGL display-list base for font rendering.

Procedure declarations appear here (later implementations follow).

b. Constant and Global parameter block

```
1.  const
2.    // Anthropometric Data (bw=60kg, bh=1.6m)
3.    grav = 9.81;
4.    // Upper Arm (Segment 1)
5.    l1 = 0.2976; // Length
6.    m1 = 1.68;   // Mass
7.    r1 = 0.1298; // COM distance
8.    I1 = 0.0154; // Moment of Inertia
9.    // Lower Arm (Segment 2)
10.   l2 = 0.2336; // Length
11.   m2 = 0.96;   // Mass
12.   r2 = 0.1004; // COM distance
13.   I2 = 0.0048; // Moment of Inertia
14.   // Hand (Segment 3 - simplified)
15.   l3 = 0.087;  // Hand length
16.
17.   // Simulation Constants
18.   DAMPING_SHOULDER = 0.1; DAMPING_ELBOW = 0.05;
19.   DT = 0.02;
20.
21. var
22.   Form1: TForm1;
23.   Sphere, Cylinder: GLUquadricObj;s
```

- **Anthropometry and simulation constants** (all declared as const):
  - grav = 9.81, gravity.
  - Link geometry & mass: l1, m1, r1, I1 for upper arm; l2, m2, r2, I2 for lower arm; l3 for hand length. These feed mass matrix, kinetic & potential energy calculations.
  - DAMPING_SHOULDER, DAMPING_ELBOW, simple viscous damping coefficients applied in force vector.
  - DT = 0.02, default integrator timestep used by the timer loop. These constants determine $M(q)$, $G(q)$, damping, and numerical resolution.

- Sphere, Cylinder: GLUquadricObj; OpenGL quadric objects used for rendering spheres and cylinders.

c. *SetupPixelFormat(ADc: HDC)*

```
1.  procedure TForm1.SetupPixelFormat(ADc: HDC);
2.  var pfd: TPixelFormatDescriptor; nPixelFormat: Integer;
3.  begin
4.    FillChar(pfd, SizeOf(pfd), 0); pfd.nSize := SizeOf(pfd); pfd.nVersion := 1;
5.    pfd.dwFlags := PFD_DRAW_TO_WINDOW or
    PFD_SUPPORT_OPENGL or PFD_DOUBLEBUFFER;
6.    pfd.iPixelType := PFD_TYPE_RGBA; pfd.cColorBits := 32;
    pfd.cDepthBits := 32;
7.    pfd.iLayerType := PFD_MAIN_PLANE;
8.    nPixelFormat := ChoosePixelFormat(ADc, @pfd);
9.    SetPixelFormat(ADc, nPixelFormat, @pfd);
10. end;
```

The purpose of this procedure is to set an appropriate Windows pixel format for the OpenGL drawing surface (RGBA, double-buffer, 32-bit depth). There are no physics, just OpenGL/Win32 boilerplate. It calls ChoosePixelFormat and SetPixelFormat.

d. *GetAccelerations(const AState: TPhysicsStateVector; var ADerivs: TPhysicsStateVector)*

```
1.  procedure TForm1.GetAccelerations(const AState: TPhysicsStateVector;
    var ADerivs: TPhysicsStateVector);
2.  var
3.    p_d, t1, t1_d, t2, t2_d: Double; c1, s1, c2, s2: Double;
4.    // M(q) or The Mass/Inertia Matrix of the system
5.    M11, M22, M33, M23, M32: Double;
6.    // F or The Vector of forces (Coriolis, Gravity, Damping)
7.    F1, F2, F3: Double;
8.    // C(q, q_dot): Coriolis and Centripetal terms
9.    C1_force, C2_force, C3_force: Double;
10.   // G(q) or The Gravitational torques
11.   G1_force, G2_force, G3_force: Double;
12.   det: Double;
13. begin
14.   p_d  := AState[1]; t1  := AState[2]; t1_d := AState[3];
15.   t2  := AState[4]; t2_d := AState[5];
16.   c1 := Cos(t1); s1 := Sin(t1); c2 := Cos(t2); s2 := Sin(t2);
17.
18.   // 1. Build the Mass Matrix M(q) from the Kinetic Energy terms (Ek)
19.   M11 := m1*Sqr(r1*s1) + m2*Sqr(l1*s1 + r2*Sin(t1+t2)) + I1*Sqr(s1) +
    I2*Sqr(Sin(t1+t2));
20.   M22 := m1*Sqr(r1) + m2*(Sqr(l1) + Sqr(r2) + 2*l1*r2*c2) + I1 + I2;
21.   M33 := m2*Sqr(r2) + I2;
22.   M23 := m2*(Sqr(r2) + l1*r2*c2) + I2;
23.   M32 := M23;
24.
```

```
25.  // 2. Calculate Coriolis/Centripetal forces C(q, q_dot)
26.  C1_force := (I1+m1*Sqr(r1))*p_d*t1_d*Sin(2*t1) +
     (I2+m2*Sqr(r2))*p_d*(t1_d+t2_d)*Sin(2*(t1+t2));
27.  C2_force := -0.5*(I1+m1*Sqr(r1))*Sqr(p_d)*Sin(2*t1) -
     0.5*(I2+m2*Sqr(r2))*Sqr(p_d)*Sin(2*(t1+t2)) - m2*l1*r2*s2*(Sqr(t2_d) +
     2*t1_d*t2_d);
28.  C3_force := 0.5*m2*l1*r2*Sqr(t1_d)*s2;
29.
30.  // 3. Calculate Gravitational torques G(q) from the Potential Energy terms
     (Ep)
31.  G1_force := 0;
32.  G2_force := (m1*r1 + m2*l1)*grav*s1 + m2*r2*grav*Sin(t1+t2);
33.  G3_force := m2*r2*grav*Sin(t1+t2);
34.
35.  // 4. Build the final Force Vector F = -C - G - D*q_dot
36.  F1 := -C1_force - G1_force - DAMPING_SHOULDER * p_d;
37.  F2 := -C2_force - G2_force - DAMPING_SHOULDER * t1_d;
38.  F3 := -C3_force - G3_force - DAMPING_ELBOW    * t2_d;
39.
40.  // 5. Solve the system M*q_ddot = F for q_ddot
41.  det := M11*(M22*M33 - M23*M32);
42.  if Abs(det) < 1e-9 then begin FillChar(ADerivs, SizeOf(ADerivs), 0); Exit;
     end;
43.  ADerivs[1] := (F1*(M22*M33-M23*M32)) / det;       // phi_ddot
44.  ADerivs[3] := (F2*M33 - F3*M23) / (M22*M33 - M23*M32); //
     theta1_ddot
45.  ADerivs[5] := (F3*M22 - F2*M32) / (M22*M33 - M23*M32); //
     theta2_ddot
46.  ADerivs[0] := p_d; ADerivs[2] := t1_d; ADerivs[4] := t2_d;
47. end;
```

This is the physics core, it builds the mass matrix $M(q)$, computes Coriolis/centripetal terms $C(q,\dot{q})$, gravity torques $G(q)$, adds damping, then solves $M\ddot{q} = F$where $F = -C - G - D\dot{q} + \tau$(here $\tau = 0$for a passive test). The code maps directly to Lagrangian/Euler–Lagrange derived equations. The steps implemented and their equations are as follows:

1. Extract state and trig helpers that prepares notation. Here p is yaw (angle) and p_d its rate, t1 shoulder pitch, t2 elbow pitch.
2. Mass / inertia matrix $M(q)$, where code computes scalar entries that are coefficients derived from kinetic energy $T = \frac{1}{2}\dot{q}^{\top}M(q)\dot{q}$. If COM positions are $x_i(q), y_i(q)$, velocity terms $v_i(q,\dot{q})$produce $T = \frac{1}{2}\sum m_i v_i^2 + \frac{1}{2}\sum I_i\omega_i^2$. Collecting coefficients of $\dot{q}_i\dot{q}_j$gives the symmetric mass matrix entries
3. Coriolis and centripetal forces $C(q,\dot{q})$. The code forms scalar C1_force, C2_force, C3_force that are nonlinear in velocities. In the Euler–Lagrange derivation, terms of the form $\partial/\partial q$and time derivatives of $\partial T/\partial\dot{q}$yield Christoffel-like terms

$\Gamma_{ijk}\dot{q}_j\dot{q}_k$. The code has grouped those resulting scalar combinations into C*_force. These terms are of form $c_{ijk}(q)\ \dot{q}_j\dot{q}_k$.

4. Gravity vector $G(q)$, in:

$$V = -m_1 g r_1 \cos\ t_1 - m_2 g(l_1 \cos\ t_1 + r_2 \cos\ (t_1 + t_2)).$$

Then $G_i = \partial V / \partial q_i$. The code matches:

$$\frac{\partial V}{\partial t_1} = (m_1 r_1 + m_2 l_1) g \sin\ t_1 + m_2 r_2 g \sin\ (t_1 + t_2), \frac{\partial V}{\partial t_2} = m_2 r_2 g \sin\ (t_1 + t_2).$$

G1_force = 0 because yaw (axis 1) has no gravity torque in this simplified model (symmetric about yaw).

5. Damping + assemble force vector F, Which corresponds to the right-hand side of $M\ddot{q} = F$. If actuators existed, Fwould contain $+\tau$. The damping terms are simple viscous forces $-D\dot{q}_i$.
6. Solve linear system $M\ddot{q} = F$. The code computes a determinant det := M11*(M22*M33 - M23*M32) and then solves for accelerations with closed-form expressions. This computes $\ddot{q} = M^{-1}F$. The code uses an algebraic simplification assuming structure of M(block triangular simplification) to compute compact inverses. There is also a numerical safety check if Abs(det) < 1e-9 then zero out to avoid division by (near) zero. The summary is as follows:

$$M(q)\ \ddot{q} + C(q, \dot{q}) + G(q) + D\dot{q} = 0 \Rightarrow \ddot{q} = M^{-1}(q)(-C - G - D\dot{q}).$$

e. *RungeKutta4_Step(var AState: TPhysicsStateVector; ATimeStep: Double)*

```
1.  procedure TForm1.RungeKutta4_Step(var AState: TPhysicsStateVector;
    ATimeStep: Double);
2.  var
3.    k1, k2, k3, k4, tempState: TPhysicsStateVector; i: Integer;
4.  begin
5.    GetAccelerations(AState, k1);
6.    for i := 0 to 5 do tempState[i] := AState[i] + 0.5 * ATimeStep * k1[i];
7.    GetAccelerations(tempState, k2);
8.    for i := 0 to 5 do tempState[i] := AState[i] + 0.5 * ATimeStep * k2[i];
9.    GetAccelerations(tempState, k3);
10.   for i := 0 to 5 do tempState[i] := AState[i] + ATimeStep * k3[i];
11.   GetAccelerations(tempState, k4);
12.   for i := 0 to 5 do
13.     AState[i] := AState[i] + (ATimeStep / 6.0) * (k1[i] + 2*k2[i] + 2*k3[i] +
    k4[i]);
14. end;
```

Implements standard RK4 for the first-order system $\dot{x} = f(x)$where $x = [q, \dot{q}]$and GetAccelerations computes $f(x)$. Code steps:

1. GetAccelerations(AState, k1)
2. tempState := AState + 0.5*h*k1; GetAccelerations(tempState,k2)
3. tempState := AState + 0.5*h*k2; GetAccelerations(tempState,k3)
4. tempState := AState + h*k3; GetAccelerations(tempState,k4)
5. AState := AState + (h/6)*(k1 + 2*k2 + 2*k3 + k4)

In Equation form:

$$k_1 = f(x_n),$$
$$k_2 = f(x_n + \frac{h}{2}k_1), k_3 = f(x_n + \frac{h}{2}k_2), k_4 = f(x_n + hk_3),$$
$$x_{n+1} = x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4).$$

Uses DT from constants in the timer loop. RK4 integrates both angles and angular velocities because state contains both.

f. BuildFont and RenderText

```
1.  procedure TForm1.BuildFont;
2.  var hFont: Winapi.Windows.HFONT;
3.  begin
4.    FontListBase := glGenLists(256);
5.    hFont := CreateFont(-12, 0, 0, 0, FW_BOLD, 0, 0, 0, ANSI_CHARSET,
6.      OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,
    ANTIALIASED_QUALITY,
7.      FF_DONTCARE or DEFAULT_PITCH, 'Arial');
8.    SelectObject(myDC, hFont);
9.    wglUseFontBitmaps(myDC, 0, 255, FontListBase);
10.   DeleteObject(hFont);
11. end;
12.
13. procedure TForm1.RenderText(x, y, z: GLfloat; const text: string);
14. begin
15.   glRasterPos3f(x, y, z);
16.   glPushAttrib(GL_LIST_BIT);
17.   glListBase(FontListBase);
18.   glCallLists(Length(text), GL_UNSIGNED_BYTE,
    PAnsiChar(AnsiString(text)));
19.   glPopAttrib();
20. end;
```

BuildFont creates an OpenGL display-list base from a Windows font (CreateFont, wglUseFontBitmaps) and stores it in FontListBase. RenderText places raster position and calls the lists to draw text. These are OpenGL/Win32 text helpers for on-screen labels. No physics.

g. DrawCube(size: GLfloat)

```
1.  procedure TForm1.DrawCube(size: GLfloat);
2.  var s: GLfloat;
3.  begin
4.    s := size / 2.0; glBegin(GL_QUADS);
5.    glNormal3f(0,0,1); glVertex3f(-s,-s,s); glVertex3f(s,-s,s); glVertex3f(s,s,s);
    glVertex3f(-s,s,s);
6.    glNormal3f(0,0,-1); glVertex3f(-s,-s,-s); glVertex3f(-s,s,-s);
    glVertex3f(s,s,-s); glVertex3f(s,-s,-s);
7.    glNormal3f(0,1,0); glVertex3f(-s,s,-s); glVertex3f(-s,s,s); glVertex3f(s,s,s);
    glVertex3f(s,s,-s);
```

```
8.   glNormal3f(0,-1,0); glVertex3f(-s,-s,-s); glVertex3f(s,-s,-s); glVertex3f(s,-
     s,s); glVertex3f(-s,-s,s);
9.   glNormal3f(1,0,0); glVertex3f(s,-s,-s); glVertex3f(s,s,-s); glVertex3f(s,s,s);
     glVertex3f(s,-s,s);
10.  glNormal3f(-1,0,0); glVertex3f(-s,-s,-s); glVertex3f(-s,-s,s); glVertex3f(-
     s,s,s); glVertex3f(-s,s,-s);
11.  glEnd();
12. end;
```

This procedure will draws a cube with normals for basic lighting (six quads). Used by DrawHandAndFingers to render the palm. Graphical primitive; no physics.

h. DrawHandAndFingers

```
1.  procedure TForm1.DrawHandAndFingers;
2.  const
3.    // Phalanx lengths for each finger
4.    thumb_len: array[0..1] of double = (0.04, 0.035);
5.    index_len: array[0..2] of double = (0.045, 0.025, 0.02);
6.    middle_len: array[0..2] of double = (0.05, 0.03, 0.025);
7.    ring_len: array[0..2] of double = (0.045, 0.028, 0.022);
8.    pinky_len: array[0..2] of double = (0.035, 0.022, 0.018);
9.    // Passive flexion angles for each joint
10.   mcp_bend = 20.0; pip_bend = 30.0; dip_bend = 10.0;
11. var
12.   i: integer;
13. begin
14.   glPushMatrix();
15.   glColor3f(1.0, 0.85, 0.7); // Skin color
16.   // Palm
17.   glPushMatrix(); glScalef(0.09, 0.025, 0.09); DrawCube(1.0);
      glPopMatrix();
18.
19.   // Draw 4 Fingers
20.   for i := 0 to 3 do begin
21.     glPushMatrix();
22.       case i of // Position each finger at its base
23.         0: glTranslatef(-0.035, 0.0, 0.045); // Index
24.         1: glTranslatef(-0.01, 0.0, 0.05);  // Middle
25.         2: glTranslatef(0.015, 0.0, 0.045); // Ring
26.         3: glTranslatef(0.04, 0.0, 0.04);   // Pinky
27.       end;
28.       // Metacarpophalangeal (MCP) Joint
29.       glRotatef(mcp_bend, 1, 0, 0);
30.       case i of
31.         0: gluCylinder(Cylinder, 0.008, 0.007, index_len[0], 16, 1);
32.         1: gluCylinder(Cylinder, 0.009, 0.008, middle_len[0], 16, 1);
33.         2: gluCylinder(Cylinder, 0.008, 0.007, ring_len[0], 16, 1);
34.         3: gluCylinder(Cylinder, 0.007, 0.006, pinky_len[0], 16, 1);
```

```
35.       end;
36.       // Proximal Interphalangeal (PIP) Joint
37.       case i of
38.         0: glTranslatef(0,0,index_len[0]); 1: glTranslatef(0,0,middle_len[0]);
39.         2: glTranslatef(0,0,ring_len[0]); 3: glTranslatef(0,0,pinky_len[0]);
40.       end;
41.       glRotatef(pip_bend, 1, 0, 0);
42.       case i of
43.         0: gluCylinder(Cylinder, 0.007, 0.006, index_len[1], 16, 1);
44.         1: gluCylinder(Cylinder, 0.008, 0.007, middle_len[1], 16, 1);
45.         2: gluCylinder(Cylinder, 0.007, 0.006, ring_len[1], 16, 1);
46.         3: gluCylinder(Cylinder, 0.006, 0.005, pinky_len[1], 16, 1);
47.       end;
48.       // Distal Interphalangeal (DIP) Joint
49.       case i of
50.         0: glTranslatef(0,0,index_len[1]); 1: glTranslatef(0,0,middle_len[1]);
51.         2: glTranslatef(0,0,ring_len[1]); 3: glTranslatef(0,0,pinky_len[1]);
52.       end;
53.       glRotatef(dip_bend, 1, 0, 0);
54.       case i of
55.         0: gluCylinder(Cylinder, 0.006, 0.005, index_len[2], 16, 1);
56.         1: gluCylinder(Cylinder, 0.007, 0.006, middle_len[2], 16, 1);
57.         2: gluCylinder(Cylinder, 0.006, 0.005, ring_len[2], 16, 1);
58.         3: gluCylinder(Cylinder, 0.005, 0.004, pinky_len[2], 16, 1);
59.       end;
60.     glPopMatrix();
61.   end;
62.
63.   // Draw Thumb
64.   glPushMatrix();
65.     glTranslatef(-0.05, 0, 0.01); // Position thumb base
66.     glRotatef(45, 0, 1, 0); // Rotate thumb out
67.     glRotatef(mcp_bend, 1, 0, 0);
68.     gluCylinder(Cylinder, 0.01, 0.009, thumb_len[0], 16, 1);
69.     glTranslatef(0,0,thumb_len[0]);
70.     glRotatef(pip_bend, 1, 0, 0);
71.     gluCylinder(Cylinder, 0.009, 0.008, thumb_len[1], 16, 1);
72.   glPopMatrix();
73.
74.   glPopMatrix();
75. end;
```

Builds a simplified hand: palm (scaled cube) and 4 fingers + thumb modeled as stacked cylinders with passive bend angles (mcp_bend, pip_bend, dip_bend). Each finger is constructed by sequence of glRotatef and gluCylinder, similar to forward kinematics but static, there is no dynamics for finger joints. This is purely visual realism.

i. DrawAxes

```
1. procedure TForm1.DrawAxes;
2. const AXIS_LENGTH = 100.0;
3. begin
4.   glLineWidth(2.0); glDisable(GL_LIGHTING); glBegin(GL_LINES);
5.   glColor3f(1,0,0); glVertex3f(-AXIS_LENGTH,0,0);
   glVertex3f(AXIS_LENGTH,0,0);
6.   glColor3f(0,1,0); glVertex3f(0,-AXIS_LENGTH,0);
   glVertex3f(0,AXIS_LENGTH,0);
7.   glColor3f(0,0,1); glVertex3f(0,0,-AXIS_LENGTH);
   glVertex3f(0,0,AXIS_LENGTH);
8.   glEnd();
9.   glColor3f(1,0,0); RenderText(0.6,0,0, 'X');
10.  glColor3f(0,1,0); RenderText(0,0.6,0, 'Y');
11.  glColor3f(0,0,1); RenderText(0,0,0.6, 'Z');
12.  glEnable(GL_LIGHTING); glLineWidth(1.0);
13. end;
```

Draws X/Y/Z axes lines and labels them using RenderText. Visual helper used when cbShowAxes.Checked is true.

j. DrawTrajectoryArcs

```
1. procedure TForm1.DrawTrajectoryArcs;
2. const ARC_RADIUS = 0.1;
3. var i: Integer; angle: Double;
4. begin
5.   glLineWidth(1.5); glDisable(GL_LIGHTING);
6.   glLineStipple(1, $AAAA); glEnable(GL_LINE_STIPPLE);
7.
8.   glPushMatrix();
9.     glColor3f(0.5,0.5,1); glBegin(GL_LINE_STRIP);
10.    for i := 0 to abs(Round(shoulder_yaw_deg)) do begin angle := DegToRad(i
   * Sign(shoulder_yaw_deg));
11.      glVertex3f(ARC_RADIUS*Sin(angle),0,ARC_RADIUS*Cos(angle)); end;
   glEnd();
12.
   RenderText(ARC_RADIUS*1.1*Sin(DegToRad(shoulder_yaw_deg)),0,ARC_
   RADIUS*1.1*Cos(DegToRad(shoulder_yaw_deg)),
   Format('%.1f',[shoulder_yaw_deg]));
13.    glRotatef(shoulder_yaw_deg,0,1,0);
14.    glColor3f(1,0.5,0.5); glBegin(GL_LINE_STRIP);
15.    for i := 0 to abs(Round(shoulder_pitch_deg)) do begin angle :=
   DegToRad(i*Sign(shoulder_pitch_deg));
16.      glVertex3f(0,ARC_RADIUS*Sin(angle),ARC_RADIUS*Cos(angle)); end;
   glEnd();
17.
   RenderText(0,ARC_RADIUS*1.1*Sin(DegToRad(shoulder_pitch_deg)),ARC
```

```
    _RADIUS*1.1*Cos(DegToRad(shoulder_pitch_deg)),
    Format('%.1f',[shoulder_pitch_deg]));
18.    if CurrentMode = smKinematic then begin
19.      glRotatef(shoulder_pitch_deg,1,0,0);
20.      glColor3f(1,1,0.5); glBegin(GL_LINE_STRIP);
21.      for i := 0 to abs(Round(shoulder_roll_deg)) do begin angle :=
    DegToRad(i*Sign(shoulder_roll_deg));
22.        glVertex3f(ARC_RADIUS*Sin(angle),ARC_RADIUS*Cos(angle),0);
    end; glEnd();
23.
    RenderText(ARC_RADIUS*1.1*Sin(DegToRad(shoulder_roll_deg)),ARC_R
    ADIUS*1.1*Cos(DegToRad(shoulder_roll_deg)),0,
    Format('%.1f',[shoulder_roll_deg]));
24.    end;
25.  glPopMatrix();
26.
27.  glPushMatrix();
28.    glRotatef(shoulder_yaw_deg,0,1,0); glRotatef(shoulder_pitch_deg,1,0,0);
29.    glRotatef(shoulder_roll_deg,0,0,1); glTranslatef(0,0,l1);
30.    glColor3f(1,0.5,0.5); glBegin(GL_LINE_STRIP);
31.    for i := 0 to abs(Round(elbow_pitch_deg)) do begin angle :=
    DegToRad(i*Sign(elbow_pitch_deg));
32.      glVertex3f(0,ARC_RADIUS*Sin(angle),ARC_RADIUS*Cos(angle)); end;
    glEnd();
33.
    RenderText(0,ARC_RADIUS*1.1*Sin(DegToRad(elbow_pitch_deg)),ARC_R
    ADIUS*1.1*Cos(DegToRad(elbow_pitch_deg)),
    Format('%.1f',[elbow_pitch_deg]));
34.    if CurrentMode = smKinematic then begin
35.      glRotatef(elbow_pitch_deg,1,0,0); glTranslatef(0,0,l2);
36.      glColor3f(1,1,0.5); glBegin(GL_LINE_STRIP);
37.      for i := 0 to abs(Round(wrist_roll_deg)) do begin angle :=
    DegToRad(i*Sign(wrist_roll_deg));
38.        glVertex3f(ARC_RADIUS*Sin(angle),ARC_RADIUS*Cos(angle),0);
    end; glEnd();
39.
    RenderText(ARC_RADIUS*1.1*Sin(DegToRad(wrist_roll_deg)),ARC_RADI
    US*1.1*Cos(DegToRad(wrist_roll_deg)),0, Format('%.1f',[wrist_roll_deg]));
40.      glRotatef(wrist_roll_deg,0,0,1);
41.      glColor3f(0.5,1,0.5); glBegin(GL_LINE_STRIP);
42.      for i := 0 to abs(Round(wrist_yaw_deg)) do begin angle :=
    DegToRad(i*Sign(wrist_yaw_deg));
43.        glVertex3f(ARC_RADIUS*Sin(angle),0,ARC_RADIUS*Cos(angle));
    end; glEnd();
44.
    RenderText(ARC_RADIUS*1.1*Sin(DegToRad(wrist_yaw_deg)),0,ARC_RA
```

```
    DIUS*1.1*Cos(DegToRad(wrist_yaw_deg)),
    Format('%.1f',[wrist_yaw_deg]));
45.      glRotatef(wrist_yaw_deg,0,1,0);
46.      glColor3f(1,0.5,0.5); glBegin(GL_LINE_STRIP);
47.      for i := 0 to abs(Round(wrist_pitch_deg)) do begin angle :=
    DegToRad(i*Sign(wrist_pitch_deg));
48.        glVertex3f(0,ARC_RADIUS*Sin(angle),ARC_RADIUS*Cos(angle));
    end; glEnd();
49.
    RenderText(0,ARC_RADIUS*1.1*Sin(DegToRad(wrist_pitch_deg)),ARC_R
    ADIUS*1.1*Cos(DegToRad(wrist_pitch_deg)),
    Format('%.1f',[wrist_pitch_deg]));
50.    end;
51.  glPopMatrix();
52.  glDisable(GL_LINE_STIPPLE); glEnable(GL_LIGHTING);
    glLineWidth(1.0);
53. end;
```

Visual helper that draws arcs indicating current joint angles (shoulder yaw/pitch/roll, elbow, wrist angles) around joints. Uses glBegin(GL_LINE_STRIP) and samples points on a circle: x = R*sin(angle), z = R*cos(angle) (or variants depending on rotation axis). Also renders numeric text of the angle value. If in kinematic mode it displays more joints (wrist arcs). This is visualization of joint angles (not an FK calculation per se, but it uses current *_deg angles to draw arcs).

k. DrawScene

```
1.  procedure TForm1.DrawScene;
2.  begin
3.    glClearColor(0.1, 0.1, 0.2, 1.0); glClear(GL_COLOR_BUFFER_BIT or
    GL_DEPTH_BUFFER_BIT);
4.    glLoadIdentity();
5.    glTranslatef(0.0, 0.4, -2.0);
6.    glRotatef(camera_pitch_deg, 1.0, 0.0, 0.0);
7.    glRotatef(camera_yaw_deg,   0.0, 1.0, 0.0);
8.    glRotatef(camera_roll_deg,  0.0, 0.0, 1.0);
9.    if cbShowAxes.Checked then DrawAxes;
10.   if cbShowAxes.Checked then DrawTrajectoryArcs;
11.   //glRotatef(90, 1, 0, 0); // Orient base coordinate system to hang down Y-
    axis
12.
13.   glPushMatrix();
14.     glColor3f(0.8, 0.8, 0.8); gluSphere(Sphere, 0.05, 32, 32);
15.     glRotatef(shoulder_yaw_deg, 0.0, 1.0, 0.0);
16.     glRotatef(shoulder_pitch_deg, 1.0, 0.0, 0.0);
17.     glRotatef(shoulder_roll_deg, 0.0, 0.0, 1.0);
18.     glColor3f(1.0, 0.8, 0.6); gluCylinder(Cylinder, 0.04, 0.03, l1, 32, 10);
19.     glTranslatef(0.0, 0.0, l1);
20.     glColor3f(0.8, 0.8, 0.8); gluSphere(Sphere, 0.04, 32, 32);
```

```
21.    glRotatef(elbow_pitch_deg, 1.0, 0.0, 0.0);
22.    glColor3f(1.0, 0.8, 0.6); gluCylinder(Cylinder, 0.03, 0.025, l2, 32, 10);
23.    glTranslatef(0.0, 0.0, l2);
24.    gluSphere(Sphere, 0.03, 32, 32);
25.    glRotatef(wrist_roll_deg, 0.0, 0.0, 1.0);
26.    glRotatef(wrist_yaw_deg, 0.0, 1.0, 0.0);
27.    glRotatef(wrist_pitch_deg, 1.0, 0.0, 0.0);
28.    glTranslatef(0.0, 0.0, 0.03);
29.    DrawHandAndFingers;
30.  glPopMatrix();
31.  SwapBuffers(myDC);
32. end;
```

The main rendering routine procedure. Steps:

1. Clear buffers, reset modelview.
2. Apply camera transform: glTranslatef(0.0,0.4,-2.0) then glRotatef(camera_pitch_deg,...) etc.
3. Optionally draw axes & angle arcs.
4. Push matrix, draw base sphere, apply shoulder rotations (yaw → pitch → roll) with glRotatef in that order.
5. Draw upper arm cylinder (gluCylinder) of length l1, glTranslatef(0,0,l1) to elbow.
6. Draw elbow sphere, apply elbow pitch rotation, draw forearm cylinder length l2, translate to wrist.
7. Apply wrist rotations and draw hand (DrawHandAndFingers).

The ordered glRotatef + glTranslatef calls implement the product of transforms:

$$p_{\text{elbow}} = T_{base}\, R_{\text{yaw}}\, R_{\text{pitch}}\, R_{\text{roll}} \begin{bmatrix} 0 \\ 0 \\ l_1 \\ 1 \end{bmatrix},$$

and similarly for wrist. Each glRotatef multiplies the current matrix so transforms are composed in the usual FK way. This yields the endpoint/world positions used for drawing and for optional logging/plots (via TrajectorySeries).

l. InitializeState

```
1.  procedure TForm1.InitializeState;
2.  begin
3.    PhysicsState[0] := DegToRad(45.0); PhysicsState[1] := 0.0;
4.    PhysicsState[2] := DegToRad(45.0); PhysicsState[3] := 0.0;
5.    PhysicsState[4] := DegToRad(30.0); PhysicsState[5] := 0.0;
6.    shoulder_yaw_deg   := 0; shoulder_pitch_deg := 0; shoulder_roll_deg := 0;
7.    elbow_pitch_deg    := 0; wrist_roll_deg     := 0; wrist_yaw_deg     := 0;
8.    wrist_pitch_deg    := 0;
9.    time := 0;
10.   camera_pitch_deg := -15; camera_yaw_deg := 20; camera_roll_deg := 0;
11.   TrackBarPitch.Position := -15; TrackBarYaw.Position := 20;
      TrackBarRoll.Position := 0;
12.   UpdateCameraLabels;
13. end;
```

Sets initial physics state (angles in radians and zero velocities). Also Resets on-screen degree variables and camera to defaults and resets time := 0. This prepares both modes (kinematic and physics) to restart from known initial conditions.

m. *FormCreate*

```
1.  procedure TForm1.FormCreate(Sender: TObject);
2.  var i: Integer; aSeries: TLineSeries;
3.  begin
4.    myDC := GetDC(pnlOpenGL.Handle); SetupPixelFormat(myDC);
5.    myRC := wglCreateContext(myDC); wglMakeCurrent(myDC, myRC);
6.    glEnable(GL_DEPTH_TEST); glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
7.    glEnable(GL_COLOR_MATERIAL); glShadeModel(GL_SMOOTH);
8.    Sphere := gluNewQuadric(); Cylinder := gluNewQuadric();
9.    gluQuadricNormals(Sphere, GLU_SMOOTH);
    gluQuadricNormals(Cylinder, GLU_SMOOTH);
10.   BuildFont;
11.
12.   SetLength(TrajectorySeries, 7);
13.   TrajectorySeries[0] := Series1; TrajectorySeries[1] := Series2;
    TrajectorySeries[2] := Series3;
14.   for i := 3 to 6 do begin
15.     aSeries := TLineSeries.Create(Self);
16.     aSeries.ParentChart := ChartTrajectories;
17.     TrajectorySeries[i] := aSeries;
18.   end;
19.   rgMode.ItemIndex := 0;
20.   rgModeClick(nil);
21. end;
```

Called at form creation. Key actions:

- Acquire myDC and set pixel format, create GL rendering context myRC, enable depth test & lighting, create quadric objects for sphere/cylinder, call BuildFont.
- Prepare TrajectorySeries array (first three series are created in the form, extra series created dynamically).
- Set initial radio mode rgMode.ItemIndex := 0 and call rgModeClick(nil) to initialize mode/UI and state.

n. *FormDestroy*

```
1.  procedure TForm1.FormDestroy(Sender: TObject);
2.  var i: Integer;
3.  begin
4.    glDeleteLists(FontListBase, 256);
5.    for i := 3 to 6 do TrajectorySeries[i].Free;
6.    gluDeleteQuadric(Sphere); gluDeleteQuadric(Cylinder);
7.    wglMakeCurrent(0, 0); wglDeleteContext(myRC);
8.    ReleaseDC(pnlOpenGL.Handle, myDC);
9.  end;
```

This procedure will frees OpenGL display lists, dynamic series, quadrics, deletes GL context and releases DC. Cleanup.

o. *FormResize*

```
1.  procedure TForm1.FormResize(Sender: TObject);
2.  begin
3.    if pnlOpenGL.Height = 0 then Exit;
4.    glViewport(0, 0, pnlOpenGL.Width, pnlOpenGL.Height);
5.    glMatrixMode(GL_PROJECTION); glLoadIdentity();
6.    gluPerspective(45.0, pnlOpenGL.Width / pnlOpenGL.Height, 0.1, 100.0);
7.    glMatrixMode(GL_MODELVIEW); glLoadIdentity();
8.  end;
```

This procedure will adjusts OpenGL viewport and projection (gluPerspective) when panel size changes. No physics.

p. *Timer1Timer*

```
1.  procedure TForm1.Timer1Timer(Sender: TObject);
2.  var logStr: String; i: integer;
3.  begin
4.    case CurrentMode of
5.      smKinematic:
6.      begin
7.        shoulder_yaw_deg   := 45 * sin(time * 0.8); shoulder_pitch_deg := 30 *
    sin(time * 1.2);
8.        shoulder_roll_deg  := 40 * sin(time * 0.5); elbow_pitch_deg    := 60 *
    (0.5 * sin(time * 2.0) + 0.5);
9.        wrist_roll_deg     := 50 * sin(time * 1.5); wrist_yaw_deg      := 20 *
    sin(time * 2.5);
10.       wrist_pitch_deg    := 25 * sin(time * 3.0);
11.       for i := 0 to 6 do case i of
12.           0: TrajectorySeries[i].AddXY(time, shoulder_yaw_deg); 1:
    TrajectorySeries[i].AddXY(time, shoulder_pitch_deg);
13.           2: TrajectorySeries[i].AddXY(time, shoulder_roll_deg); 3:
    TrajectorySeries[i].AddXY(time, elbow_pitch_deg);
14.           4: TrajectorySeries[i].AddXY(time, wrist_roll_deg); 5:
    TrajectorySeries[i].AddXY(time, wrist_yaw_deg);
15.           6: TrajectorySeries[i].AddXY(time, wrist_pitch_deg);
16.         end;
17.       logStr := Format('%.2f | Kinematic | Y: %-6.1f| P: %-6.1f| R: %-6.1f| E:
    %-6.1f| WR: %-6.1f| WY: %-6.1f| WP: %-6.1f',
18.         [time, shoulder_yaw_deg, shoulder_pitch_deg, shoulder_roll_deg,
    elbow_pitch_deg, wrist_roll_deg, wrist_yaw_deg, wrist_pitch_deg]);
19.     end;
20.     smPhysics:
21.     begin
22.       {RungeKutta4_Step(PhysicsState, DT);
23.       shoulder_yaw_deg   := RadToDeg(PhysicsState[0]); shoulder_pitch_deg
    := RadToDeg(PhysicsState[2]);
```

```
24.      elbow_pitch_deg    := RadToDeg(PhysicsState[4]);
25.      shoulder_roll_deg := 0; wrist_roll_deg := 0; wrist_yaw_deg := 0;
    wrist_pitch_deg := 0;
26.      TrajectorySeries[0].AddXY(time, shoulder_yaw_deg);
27.      TrajectorySeries[1].AddXY(time, shoulder_pitch_deg);
28.      TrajectorySeries[2].AddXY(time, elbow_pitch_deg);
29.      logStr := Format('%.2f | Physics   | Shoulder Yaw: %-8.2f| Shoulder
    Pitch: %-8.2f| Elbow Pitch: %-8.2f',
30.        [time, shoulder_yaw_deg, shoulder_pitch_deg, elbow_pitch_deg]);}
31.      RungeKutta4_Step(PhysicsState, DT);
32.      // REALISM ENHANCEMENT: Enforce elbow joint limit (no
    hyperextension)
33.      if PhysicsState[4] < 0 then
34.      begin
35.        PhysicsState[4] := 0; // Clamp angle
36.        PhysicsState[5] := -PhysicsState[5] * 0.5; // Simulate bounce
37.      end;
38.      shoulder_yaw_deg   := RadToDeg(PhysicsState[0]);
39.      shoulder_pitch_deg := RadToDeg(PhysicsState[2]);
40.      elbow_pitch_deg    := RadToDeg(PhysicsState[4]);
41.      shoulder_roll_deg := 0; wrist_roll_deg := 0; wrist_yaw_deg := 0;
    wrist_pitch_deg := 0;
42.      TrajectorySeries[0].AddXY(time, shoulder_yaw_deg);
43.      TrajectorySeries[1].AddXY(time, shoulder_pitch_deg);
44.      TrajectorySeries[2].AddXY(time, elbow_pitch_deg);
45.      logStr := Format('%.2f | Physics   | Shoulder Yaw: %-8.2f| Shoulder
    Pitch: %-8.2f| Elbow Pitch: %-8.2f',
46.        [time, shoulder_yaw_deg, shoulder_pitch_deg, elbow_pitch_deg]);
47.    end;
48.  end;
49.  ListBoxLog.Items.Add(logStr);
50.  ListBoxLog.ItemIndex := ListBoxLog.Items.Count - 1;
51.  DrawScene;
52.  time := time + DT;
53. end;
```

This procedure runs every DT seconds (Timer interval likely set to DT*1000). The behaviour depends on CurrentMode:

- **Kinematic (smKinematic)**
  - Computes joint angles from sinusoids as time functions (different frequencies/amplitudes for each joint).
  - Adds angle values to TrajectorySeries for plotting and writes a log string.
  - No integrator call; positions are purely kinematic.
- **Physics (smPhysics)**
  - Calls RungeKutta4_Step(PhysicsState, DT) to advance state using the RK4 integrator.
  - Post-process:

- Enforces a simple elbow limit: if elbow angle PhysicsState[4] < 0 clamp to zero and reverse velocity (energy loss) PhysicsState[5] := -PhysicsState[5] * 0.5.
- Converts radians → degrees for rendering: shoulder_yaw_deg := RadToDeg(PhysicsState[0]); ...
- Adds the main three joint plots to TrajectorySeries[0..2] and logs.

Finally it adds the log row, calls DrawScene, and increments time := time + DT.

q. btnStartStopClick and btnResetClick

```
1.  procedure TForm1.btnStartStopClick(Sender: TObject);
2.  begin
3.    Timer1.Enabled := not Timer1.Enabled;
4.    if Timer1.Enabled then btnStartStop.Caption := 'Stop' else
    btnStartStop.Caption := 'Start';
5.  end;
6.
7.  procedure TForm1.btnResetClick(Sender: TObject);
8.  begin
9.    Timer1.Enabled := False;
10.   btnStartStop.Caption := 'Start';
11.   rgModeClick(nil);
12. end;
```

btnStartStopClick toggles the Timer and updates button caption between "Start"/"Stop". While, btnResetClick stops timer, sets caption to "Start", and calls rgModeClick(nil) to reset mode and state. Simple UI controls.

r. rgModeClick

```
1.  procedure TForm1.rgModeClick(Sender: TObject);
2.  const
3.    KinematicTitles: array[0..6] of string = ('Shoulder Yaw', 'Shoulder Pitch',
    'Shoulder Roll', 'Elbow Pitch', 'Wrist Roll', 'Wrist Yaw', 'Wrist Pitch');
4.    KinematicColors: array[0..6] of TColor = (clRed, clGreen, clBlue,
    clYellow, clPurple, clAqua, clLime);
5.  var i: Integer;
6.  begin
7.    if rgMode.ItemIndex = 0 then CurrentMode := smKinematic else
    CurrentMode := smPhysics;
8.    InitializeState;
9.    for i := 0 to 6 do TrajectorySeries[i].Clear;
10.   ListBoxLog.Clear;
11.   ChartTrajectories.LeftAxis.Title.Caption := 'Angle (Degrees)';
12.   ChartTrajectories.BottomAxis.Title.Caption := 'Time (s)';
13.
14.   if CurrentMode = smPhysics then
15.   begin
16.     ChartTrajectories.Title.Text.Clear;
    ChartTrajectories.Title.Text.Add('Physics Trajectories (3-DOF)');
```

```
17.    ListBoxLog.Items.Add(Format('%-10s| %-11s | %-20s | %-20s | %-20s',
       ['Time (s)', 'Mode', 'Shoulder Yaw (Abd)', 'Shoulder Pitch (Flex)', 'Elbow
       Pitch (Flex)']));
18.    for i := 0 to 2 do TrajectorySeries[i].Active := True; for i := 3 to 6 do
       TrajectorySeries[i].Active := False;
19.    TrajectorySeries[0].Title := 'Shoulder Yaw (Abd)';
       TrajectorySeries[0].SeriesColor := clRed;
20.    TrajectorySeries[1].Title := 'Shoulder Pitch (Flex)';
       TrajectorySeries[1].SeriesColor := clGreen;
21.    TrajectorySeries[2].Title := 'Elbow Pitch (Flex)';
       TrajectorySeries[2].SeriesColor := clBlue;
22.    shoulder_yaw_deg   := RadToDeg(PhysicsState[0]);
23.    shoulder_pitch_deg := RadToDeg(PhysicsState[2]);
24.    elbow_pitch_deg    := RadToDeg(PhysicsState[4]);
25.  end
26.  else
27.  begin
28.    ChartTrajectories.Title.Text.Clear;
       ChartTrajectories.Title.Text.Add('Kinematic Demo (7-DOF)');
29.    ListBoxLog.Items.Add(Format('%-8s|%-11s|%-10s|%-10s|%-10s|%-
       10s|%-10s|%-10s|%-10s',
30.      ['Time (s)', 'Mode', 'Shld Yaw', 'Shld Pitch', 'Shld Roll', 'Elbow', 'Wrist
       Roll', 'Wrist Yaw', 'Wrist Pitch']));
31.    for i := 0 to 6 do begin
32.      TrajectorySeries[i].Active := True;
33.      TrajectorySeries[i].Title := KinematicTitles[i];
34.      TrajectorySeries[i].SeriesColor := KinematicColors[i];
35.    end;
36.  end;
37.  DrawScene;
38. end;
```

The procedure to switches CurrentMode between smKinematic and smPhysics according to the radio group value. Actions:

- Calls InitializeState.
- Clears and re-configures TrajectorySeries (which series are active, titles, colors).
- If physics mode: sets only series 0..2 active and maps their titles/colors; if kinematic: uses all 7 series with kinematic titles.
- Populates initial log header and draws the scene.

This bridges UI, plotting and simulation mode.

s. *BitBtnExitClick, UpdateCameraLabels and TrackBar handlers, and cbShowAxesChange*

```
1.  procedure TForm1.BitBtnExitClick(Sender: TObject);
2.  begin
3.    Close;
4.  end;
5.
```

```
6.  procedure TForm1.UpdateCameraLabels;
7.  begin
8.    lblPitchValue.Caption := IntToStr(TrackBarPitch.Position) + '°';
9.    lblYawValue.Caption   := IntToStr(TrackBarYaw.Position) + '°';
10.  lblRollValue.Caption  := IntToStr(TrackBarRoll.Position) + '°';
11. end;
12.
13. procedure TForm1.TrackBarPitchChange(Sender: TObject);
14. begin
15.  camera_pitch_deg := TrackBarPitch.Position;
16.  UpdateCameraLabels;
17.  if not Timer1.Enabled then DrawScene;
18. end;
19.
20. procedure TForm1.TrackBarYawChange(Sender: TObject);
21. begin
22.  camera_yaw_deg := TrackBarYaw.Position;
23.  UpdateCameraLabels;
24.  if not Timer1.Enabled then DrawScene;
25. end;
26.
27. procedure TForm1.TrackBarRollChange(Sender: TObject);
28. begin
29.  camera_roll_deg := TrackBarRoll.Position;
30.  UpdateCameraLabels;
31.  if not Timer1.Enabled then DrawScene;
32. end;
33.
34. procedure TForm1.cbShowAxesChange(Sender: TObject);
35. begin
36.  if not Timer1.Enabled then DrawScene;
37. end;
```

BitBtnExitClick will calls Close to exit app. UpdateCameraLabels and TrackBar handlers will keep the UI labels in-sync with TrackBar positions. If timer is stopped, calling a trackbar redraws the scene. No physics. And cbShowAxesChange will redraws scene if toggling the axes display while timer is not running.

2.3. Result of the Program and Analysis

*Initial GUI

Figure 2.1. *Initial GUI components (1)*

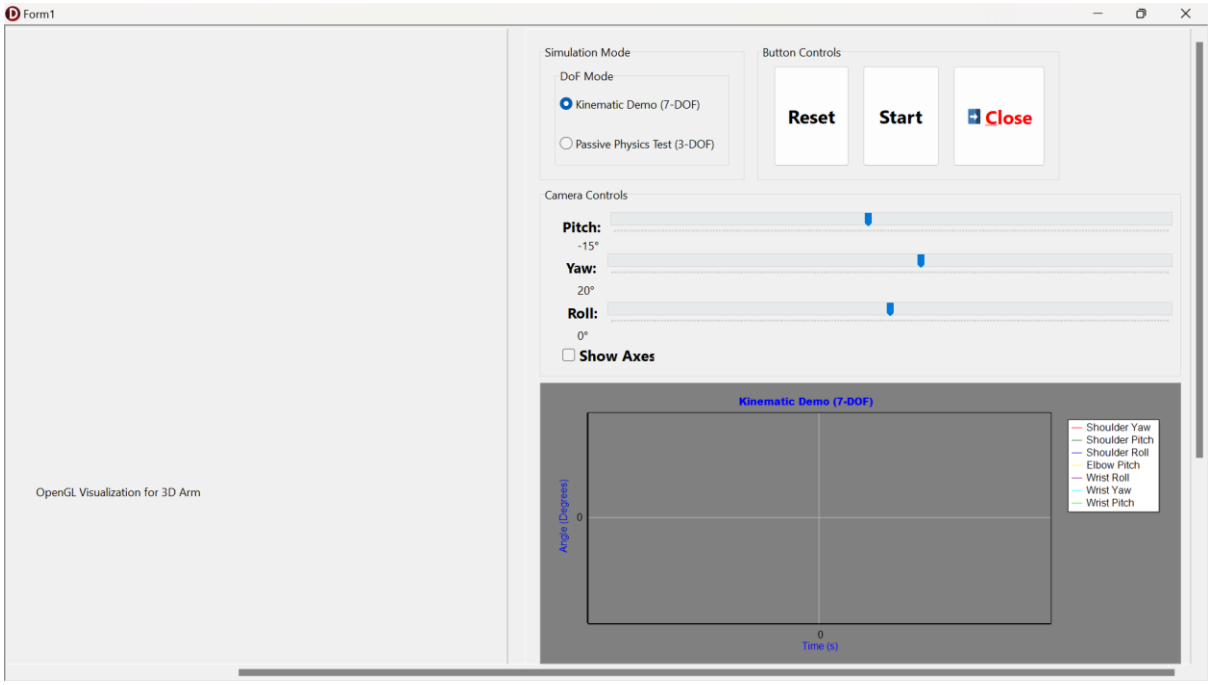

Figure 2.2. *Initial GUI components (2)*

Figure 2.3. *Initial GUI components (3)*

This program implements a compact 3-D upper-limb demonstrator that couples a physics-based dynamics simulator (3 degrees-of-freedom) with a separate kinematic demo (7 degrees-of-freedom) for visualization and pedagogical comparison. The demonstrator is organized into two main parts: (1) a Lagrangian dynamics engine that computes the equations of motion for a 3-DoF chain (shoulder yaw, shoulder pitch, elbow pitch) and integrates the resulting state in time, and (2) a forward-kinematics renderer that visualizes articulated poses for both the physics model and a higher-DOF kinematic animation driven by analytic commands. The code base, constants and routines referenced in the following analysis are implemented in the provided Unit Unit3DArmModel and the accompanying report figures.

The dynamic model is intentionally compact (3 DoF) to keep algebraic expressions tractable and to support real-time RK4 integration. The kinematic demo expands the number of visible joints to seven for richer visualization, but these extra joints are not dynamically simulated.

The yaw axis is modeled as gravity-neutral (no gravity torque on yaw), finger joints are purely geometric (no inertia), and joint limits are enforced with a simple clamp + velocity reversal rule for the elbow. These simplifications are reasonable for a teaching demonstrator but limit fidelity for precise biomechanical or control studies.

2.3.1. Passive Physics Test with 3-DoF System Model

The 3-DoF simulation implements the Lagrangian equations of motion in the standard matrix form

$$M(q)\,\ddot{q} + C(q,\dot{q}) + G(q) + D\dot{q} = \tau,$$

with $\tau = 0$for the passive tests. Kinetic energy T(translational COM terms + rotational inertias) is used to build the configuration-dependent mass matrix $M(q)$so that $T = \frac{1}{2}\dot{q}^{\top}M(q)\dot{q}$. The gravitational potential

$$V = -m_1 g r_1 \cos\ t_1 - m_2 g (l_1 \cos\ t_1 + r_2 \cos\ (t_1 + t_2))$$

produces the gravity vector $G(q) = \partial V / \partial q$. Coriolis/centripetal terms $C(q, \dot{q})$arise from the $\Gamma_{ijk}\dot{q}_j\dot{q}_k$terms in the Euler–Lagrange derivation. The code computes the mass entries (e.g. M11,M22,M33,M23), the scalar Coriolis forces (C1_force, C2_force, C3_force), and gravity torques (G2_force,G3_force) then solves $M(q)\ddot{q} = -C - G - D\dot{q}$for $\ddot{q}$and integrates the first-order state $[q, \dot{q}]$with classical RK4.

Figure 2.4 (initial released condition), Figure 2.5 (rest state) and the accompanying trajectory chart in Figure 2.6 together document the canonical passive response of the 3-DoF model. The initial snapshot (Fig. 2.4) shows the arm released from the prescribed starting angles (shoulder yaw/pitch and elbow pitch) and the rest snapshot (Fig. 2.5) shows the final gravitationally settled posture, a configuration consistent with minimization of the potential energy V. The time series (Fig. 2.6) makes this dynamic explicit: after release the shoulder and elbow angles undergo a transient oscillation whose amplitude decays and whose frequency matches the model's inertial parameters. The decay is produced primarily by the viscous damping terms (the DAMPING_SHOULDER and DAMPING_ELBOW coefficients) and by the elbow-limit energy loss applied in code (the simple clamp + velocity reversal), both of which are visible as reduced oscillation amplitude and a non-ideal (slightly underdamped) settling.

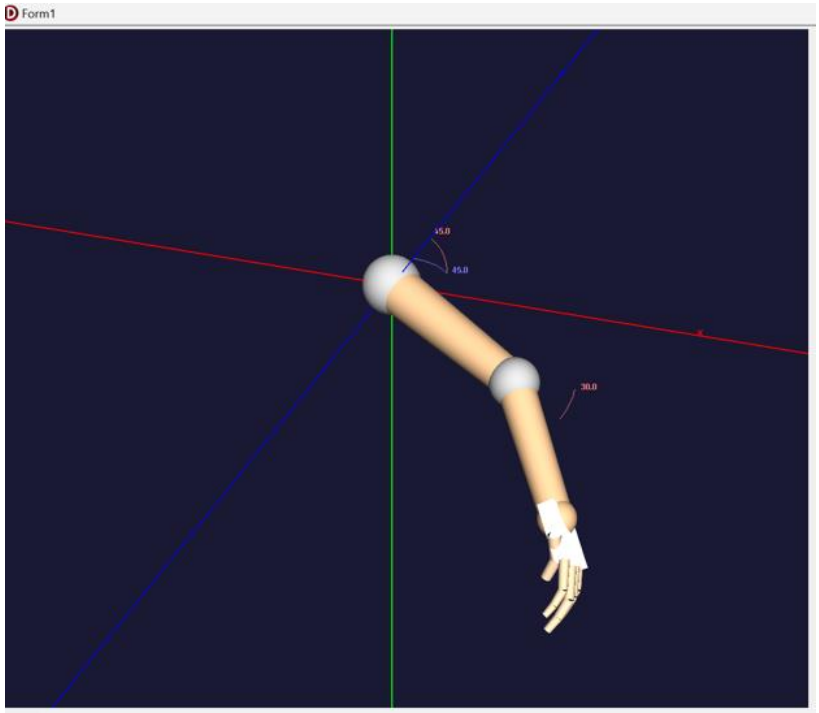

Figure 2.4. *The 3-DoF System Models at Initial Released Condition*

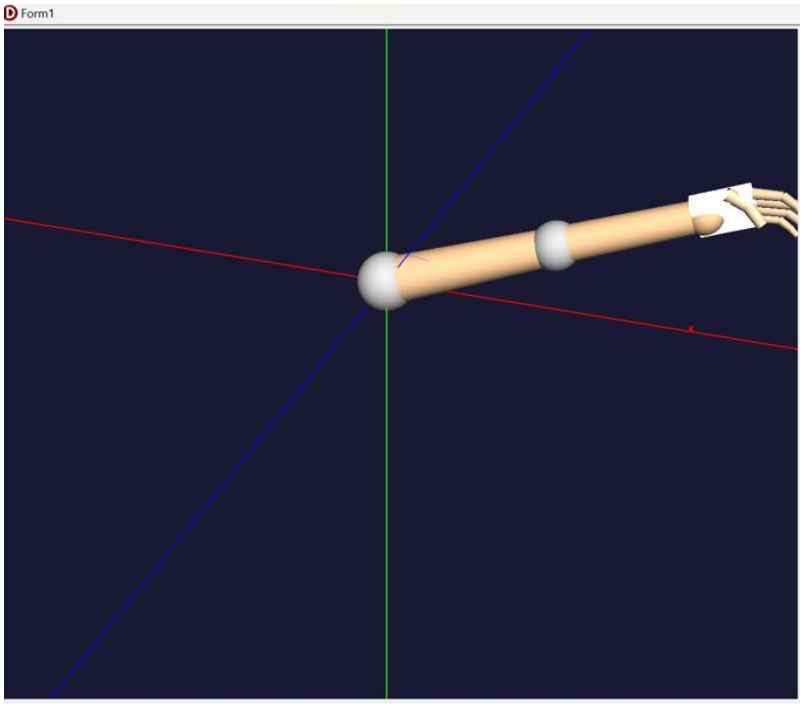

Figure 2.5. *The 3-DoF System Models at Rest State after Released*

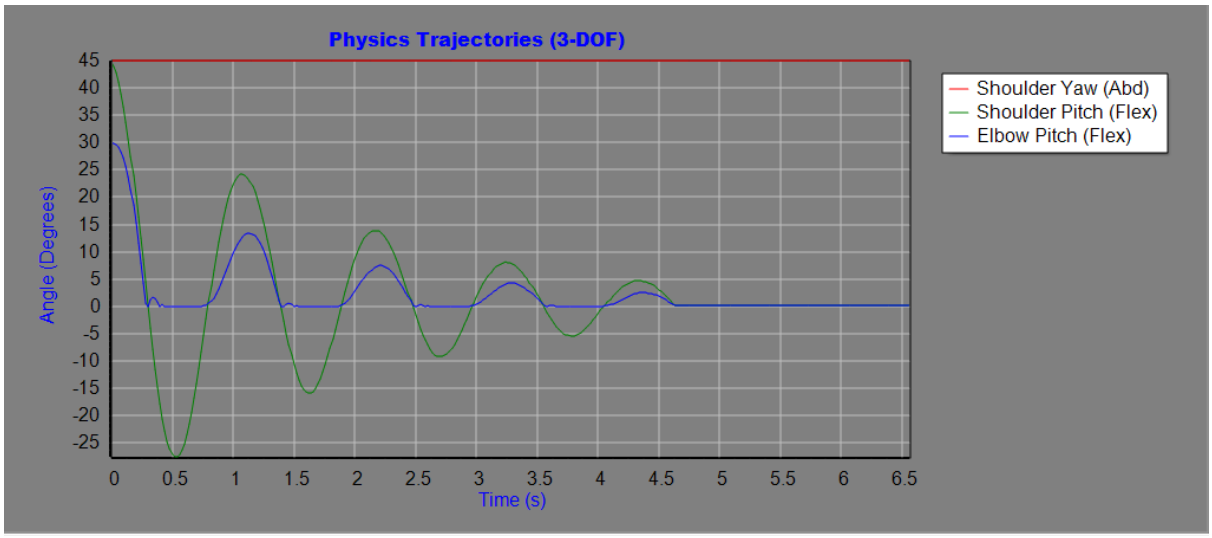

Figure 2.6. *The 3-DoF System Models Physics Trajectories Chart Results*

Table 2.1 below summarizes the numeric angle values before release and at rest (peak/transient values and the final steady values), these numbers align with the expected signs and magnitudes from the gravity torque expressions $\partial V / \partial q$implemented in the code. Overall, the images and chart confirm (1) correct mapping of potential-energy gradients into restoring torques, (2) presence of inertial coupling during transients (non-zero cross terms in $M(q)$produce coupled motion in shoulder/elbow), and (3) that the numerical scheme (RK4 with small DT plus damping) produces smooth, stable transients for the passive test shown.

Before Release	On Rest State

Time (s)	Mode	Shoulder Yaw (Abd)	Shoulder Pitch (Flex)	Elbow Pitch (Flex)
0.00	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 44.75	Elbow Pitch: 29.97
0.02	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 43.99	Elbow Pitch: 29.85
0.04	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 42.76	Elbow Pitch: 29.59
0.06	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 41.09	Elbow Pitch: 29.13
0.08	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 39.00	Elbow Pitch: 28.39
0.10	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 36.55	Elbow Pitch: 27.31
0.12	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 33.76	Elbow Pitch: 25.83
0.14	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 30.70	Elbow Pitch: 23.88
0.16	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 27.40	Elbow Pitch: 21.45
0.18	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 23.92	Elbow Pitch: 18.55
0.20	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 20.27	Elbow Pitch: 15.25
0.22	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 16.49	Elbow Pitch: 11.65
0.24	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 12.58	Elbow Pitch: 7.91
0.26	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 8.56	Elbow Pitch: 4.20
0.28	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 4.44	Elbow Pitch: 0.71
0.30	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.23	Elbow Pitch: 0.00
0.32	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: -3.91	Elbow Pitch: 1.15
6.24	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.26	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.28	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.30	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.32	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.34	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.36	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.38	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.40	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.42	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.44	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.46	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.48	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.50	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.52	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.54	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18
6.56	Physics	Shoulder Yaw: 45.00	Shoulder Pitch: 0.08	Elbow Pitch: 0.18

Table 2.1. *The 3-DoF System Models Physics Trajectories Results Before Release and on Rest State*

2.3.2. Kinematic Demo with 7-DoF System Model

The 7-DoF mode is a **kinematic** demonstration: joint angles are generated by explicit sinusoidal functions (time-dependent parametric commands) rather than integrating dynamics. Example signals from the code include:

$$\begin{aligned} \text{shoulder\_yaw}(t) &= 45^\circ \sin(0.8t), \\ \text{shoulder\_pitch}(t) &= 30^\circ \sin(1.2t), \\ \text{elbow\_pitch}(t) &= 60^\circ (0.5 \sin(2.0t) + 0.5), \end{aligned}$$

and similar formulas for wrist joints. These angles are then applied to the OpenGL transform stack in the order yaw → pitch → roll with glRotatef and glTranslatef to compute link poses, for example the forward kinematics via multiplication of elementary rotation/translation transforms (no mass matrix, no M, C, G terms are evaluated for these extra DoFs). The trajectories are sampled and plotted directly from these analytic angle functions.

Figures 2.7, 2.8, and 2.9 depict the 7-DoF model at different instants of the kinematic demonstration and Figure 2.10 shows the corresponding trajectory charts. Visually, the sequence highlights smooth, periodic articulation across shoulder, elbow and wrist axes: the sinusoidal command design produces visibly harmonic motion that is excellent for demonstrating joint ranges and joint-space coordination without dynamic side-effects.

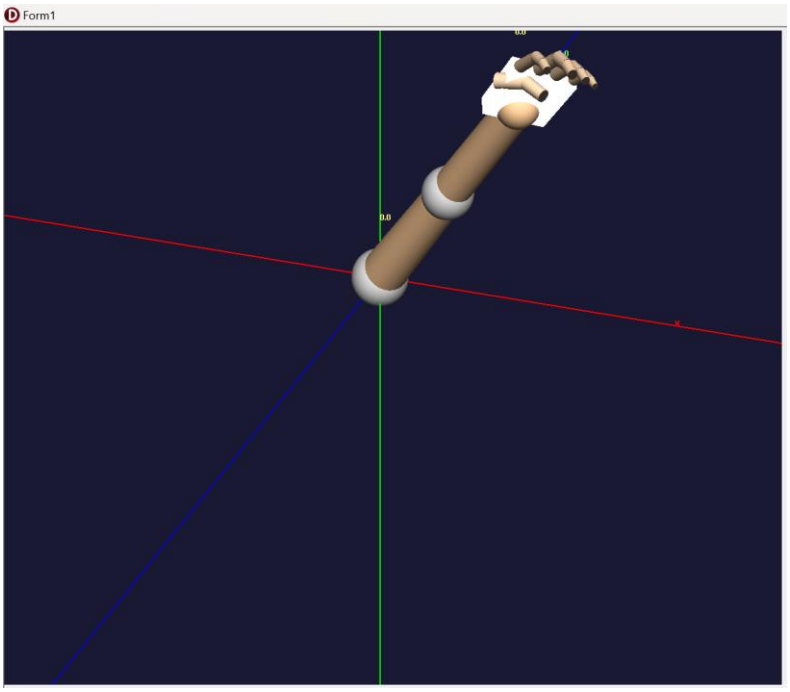

Figure 2.7. *The 7-DoF System Models at Initial Released Condition*

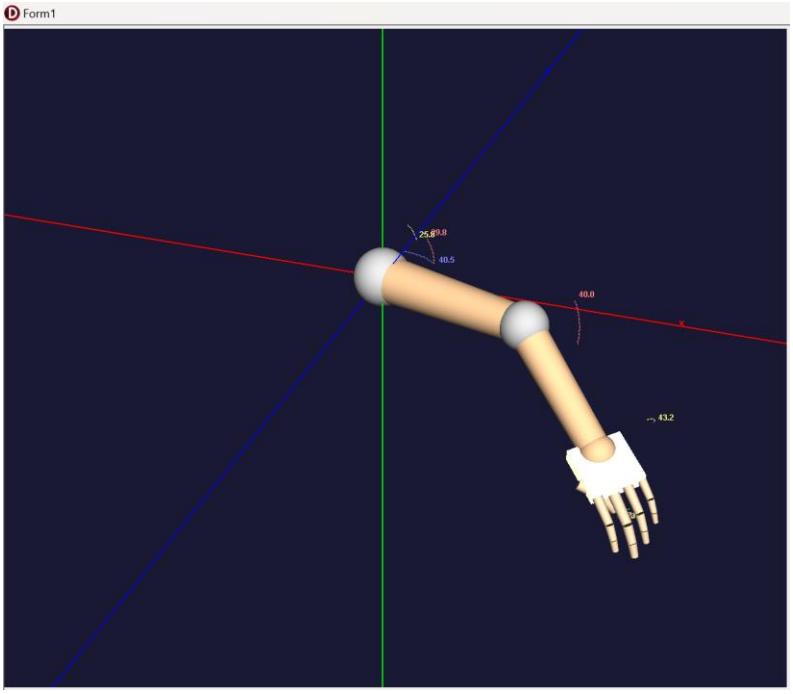

Figure 2.8. *The 7-DoF System Models Demonstration (1)*

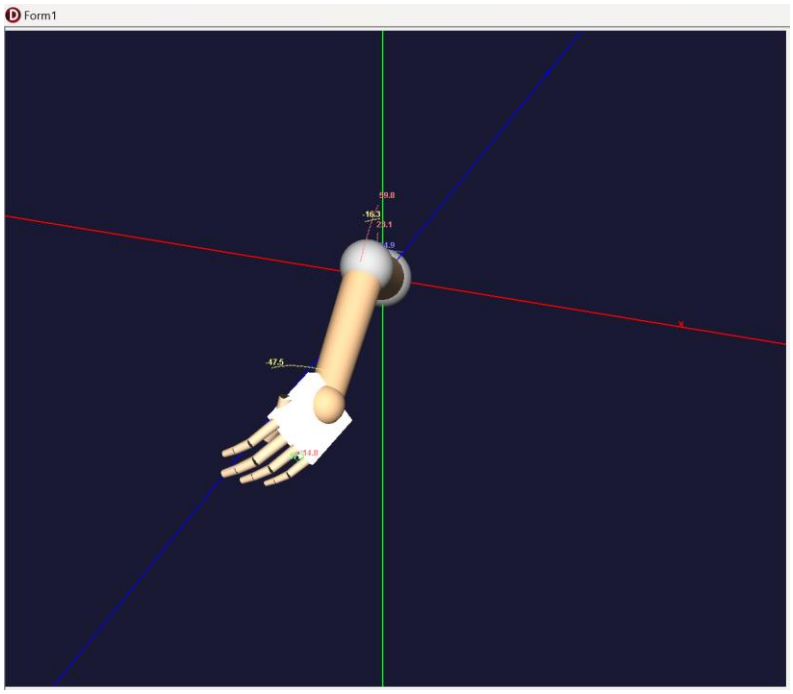

Figure 2.9. *The 7-DoF System Models Demonstration (2)*

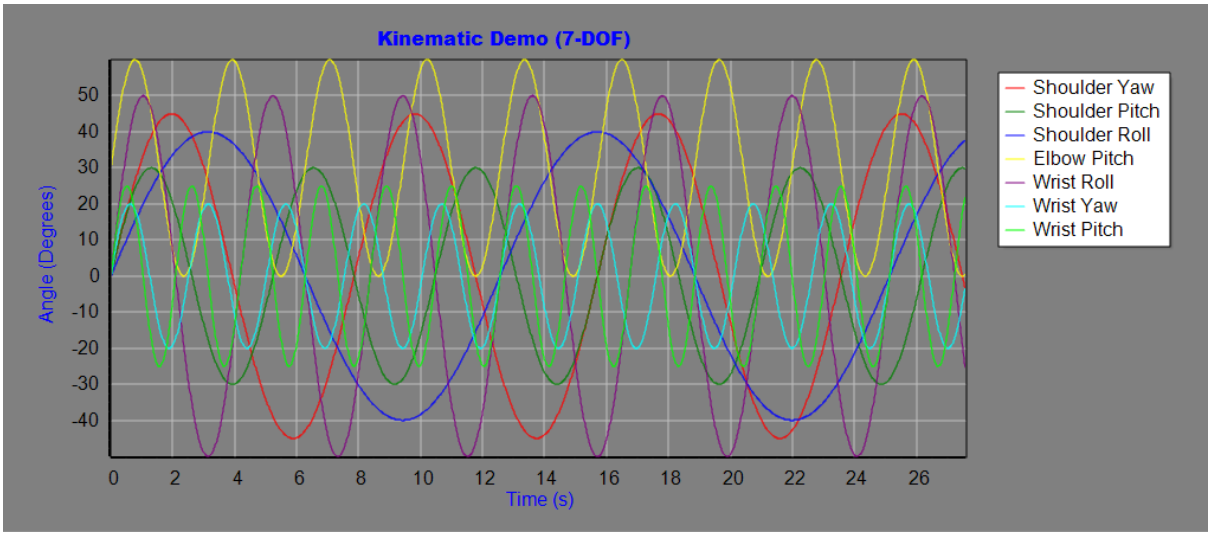

Figure 2.10. *The 7-DoF System Models Physics Trajectories Chart Results*

The trajectory chart above (Fig. 2.10) confirms near-perfect periodicity and phase differences between joints, which is expected because the signals are independent analytic sinusoids with different frequencies and amplitudes. Importantly, comparison with the 3-DoF physics charts shows a key pedagogical difference: the 7-DoF plots remain strictly periodic and do not show settling, coupling or energy decay because there are no inertial couplings, gravity torques or damping applied to those extra joints in kinematic mode. This visual contrast (periodic, repeatable kinematic traces vs. damped, coupled physics transients) is pedagogically useful, it illustrates forward kinematics and rendering separately from dynamics, and helps the reader appreciate why real motions (when dynamics matter) deviate from simple prescribed profiles. If the goal is to study realistic movement under inertia and torque control, the figures make clear that the 7-DoF demo is

only illustrative; dynamics would need to be added for quantitative predictions of forces or interaction with the environment.

Before Release	After Release
Time (s)|Mode |Shld Yaw |Shld Pitch|Shld Roll |Elbow |Wrist Roll|Wrist Yaw |Wrist Pitch 0.00 | Kinematic | Y: 0.0 | P: 0.0 | R: 0.0 | E: 30.0 | WR: 0.0 | WY: 0.0 | WP: 0.0 0.02 | Kinematic | Y: 0.7 | P: 0.7 | R: 0.4 | E: 31.2 | WR: 1.5 | WY: 1.0 | WP: 1.5 0.04 | Kinematic | Y: 1.4 | P: 1.4 | R: 0.8 | E: 32.4 | WR: 3.0 | WY: 2.0 | WP: 3.0 0.06 | Kinematic | Y: 2.2 | P: 2.2 | R: 1.2 | E: 33.6 | WR: 4.5 | WY: 3.0 | WP: 4.5 0.08 | Kinematic | Y: 2.9 | P: 2.9 | R: 1.6 | E: 34.8 | WR: 6.0 | WY: 4.0 | WP: 5.9 0.10 | Kinematic | Y: 3.6 | P: 3.6 | R: 2.0 | E: 36.0 | WR: 7.5 | WY: 4.9 | WP: 7.4 0.12 | Kinematic | Y: 4.3 | P: 4.3 | R: 2.4 | E: 37.1 | WR: 9.0 | WY: 5.9 | WP: 8.8 0.14 | Kinematic | Y: 5.0 | P: 5.0 | R: 2.8 | E: 38.3 | WR: 10.4 | WY: 6.9 | WP: 10.2 0.16 | Kinematic | Y: 5.7 | P: 5.7 | R: 3.2 | E: 39.4 | WR: 11.9 | WY: 7.8 | WP: 11.5 0.18 | Kinematic | Y: 6.5 | P: 6.4 | R: 3.6 | E: 40.6 | WR: 13.3 | WY: 8.7 | WP: 12.9 0.20 | Kinematic | Y: 7.2 | P: 7.1 | R: 4.0 | E: 41.7 | WR: 14.8 | WY: 9.6 | WP: 14.1 0.22 | Kinematic | Y: 7.9 | P: 7.8 | R: 4.4 | E: 42.8 | WR: 16.2 | WY: 10.5 | WP: 15.3 0.24 | Kinematic | Y: 8.6 | P: 8.5 | R: 4.8 | E: 43.9 | WR: 17.6 | WY: 11.3 | WP: 16.5 0.26 | Kinematic | Y: 9.3 | P: 9.2 | R: 5.2 | E: 44.9 | WR: 19.0 | WY: 12.1 | WP: 17.6 0.28 | Kinematic | Y: 10.0 | P: 9.9 | R: 5.6 | E: 45.9 | WR: 20.4 | WY: 12.9 | WP: 18.6 0.30 | Kinematic | Y: 10.7 | P: 10.6 | R: 6.0 | E: 46.9 | WR: 21.7 | WY: 13.6 | WP: 19.6 0.32 | Kinematic | Y: 11.4 | P: 11.2 | R: 6.4 | E: 47.9 | WR: 23.1 | WY: 14.3 | WP: 20.5	27.26 | Kinematic | Y: 8.2 | P: 28.9 | R: 35.0 | E: 3.1 | WR: -2.5 | WY: -16.4 | WP: 2.5 27.28 | Kinematic | Y: 7.5 | P: 29.1 | R: 35.2 | E: 2.6 | WR: -4.0 | WY: -15.9 | WP: 3.9 27.30 | Kinematic | Y: 6.8 | P: 29.2 | R: 35.3 | E: 2.1 | WR: -5.5 | WY: -15.2 | WP: 5.4 27.32 | Kinematic | Y: 6.1 | P: 29.4 | R: 35.5 | E: 1.7 | WR: -6.9 | WY: -14.6 | WP: 6.9 27.34 | Kinematic | Y: 5.3 | P: 29.5 | R: 35.7 | E: 1.3 | WR: -8.4 | WY: -13.9 | WP: 8.3 27.36 | Kinematic | Y: 4.6 | P: 29.6 | R: 35.9 | E: 1.0 | WR: -9.9 | WY: -13.1 | WP: 9.7 27.38 | Kinematic | Y: 3.9 | P: 29.7 | R: 36.1 | E: 0.7 | WR: -11.4 | WY: -12.3 | WP: 11.1 27.40 | Kinematic | Y: 3.2 | P: 29.8 | R: 36.2 | E: 0.5 | WR: -12.8 | WY: -11.5 | WP: 12.4 27.42 | Kinematic | Y: 2.5 | P: 29.9 | R: 36.4 | E: 0.3 | WR: -14.3 | WY: -10.7 | WP: 13.7 27.44 | Kinematic | Y: 1.8 | P: 29.9 | R: 36.6 | E: 0.1 | WR: -15.7 | WY: -9.9 | WP: 14.9 27.46 | Kinematic | Y: 1.0 | P: 30.0 | R: 36.7 | E: 0.1 | WR: -17.1 | WY: -9.0 | WP: 16.1 27.48 | Kinematic | Y: 0.3 | P: 30.0 | R: 36.9 | E: 0.0 | WR: -18.5 | WY: -8.1 | WP: 17.2 27.50 | Kinematic | Y: -0.4 | P: 30.0 | R: 37.0 | E: 0.0 | WR: -19.9 | WY: -7.1 | WP: 18.3 27.52 | Kinematic | Y: -1.1 | P: 30.0 | R: 37.2 | E: 0.1 | WR: -21.3 | WY: -6.2 | WP: 19.2 27.54 | Kinematic | Y: -1.8 | P: 29.9 | R: 37.3 | E: 0.2 | WR: -22.6 | WY: -5.2 | WP: 20.2 27.56 | Kinematic | Y: -2.6 | P: 29.9 | R: 37.5 | E: 0.3 | WR: -23.9 | WY: -4.3 | WP: 21.0 27.58 | Kinematic | Y: -3.3 | P: 29.8 | R: 37.6 | E: 0.5 | WR: -25.2 | WY: -3.3 | WP: 21.8

Table 2.2. *The 7-DoF System Models Physics Trajectories Results Before and After Release*

2.3.3. Evaluation

The program successfully implements the Lagrangian double-pendulum style dynamics for a simplified 3-DoF upper-limb model and a separate kinematic 7-DoF demonstration. The physics integrator (RK4) combined with the mass matrix, Coriolis and gravity terms produces physically plausible passive responses when the arm is released from the initial pose, and the rendered OpenGL scene correctly visualizes the kinematic chain and trajectory charts. Numerical safeguards (small viscous damping, determinant check) stabilize the simulation for the presented passive tests. Limitations include the algebraic inversion strategy for the mass matrix (fragile near singularities), a simple hard clamp for joint limits, and non-dynamic finger geometry. Recommended improvements are to add a robust linear solver for M^{-1}, to use compliant limit torques, and to validate energy conservation to quantify integrator accuracy.

CHAPTER III. CONCLUSION

This assignment implemented a 3-D upper-limb model and visualization program that couples Lagrangian dynamics with RK4 numerical integration and OpenGL forward kinematics. The program demonstrates two complementary modes: a physics-based 3-DoF passive simulation that shows realistic gravitational settling and inertial coupling, and a 7-DoF kinematic demo useful for generating visually rich trajectories. The implementation maps directly to the theoretical constructs (kinetic/potential energy → $M(q), C(q, \dot{q}), G(q)$; Lagrangian → Euler–Lagrange → state-space), and the visualization and trajectory plots allow straightforward qualitative validation. Remaining limitations (numerical robustness around near-singular configurations, simple joint-limit handling, and hand/finger dynamics missing) point to clear next steps: improve the linear solver for the equations of motion, adopt compliant or constrained limit handling, and extend the dynamic model if fine finger behavior is required. Overall, the project meets its learning objectives by combining derivation, numerical integration, and 3-D rendering in a single demonstrator and provides a solid base for further extension and quantitative validation.