CPRE 1850 – DROP KINEMATICS

LAB 5
SECTION CN

SUBMITTED BY:

MAXWELL MILLER

10/17/2025

Problem

This lab project involves basic kinematic mechanical physics. Essentially a controller is to be dropped from an undetermined height and using kinematics this program is supposed to determine the total height that the controller has fallen. This lab falls in two different parts, each highlighting a different equation. The first part uses $x_{\mathrm{f}} = x_{\mathrm{i}} + vt_f + \frac{1}{2}gt_f{}^2$, while the second part uses $v_i = v_{\mathrm{i\text{-}1}} + g\left(1 - \left|\overrightarrow{a_g}\right|\right)(t_i - t_{\mathrm{i\text{-}1}})$.

Analysis

Given these two equations, and the problem at hand, it became clear, after some basic testing, that in order to be considered "free falling" the magnitude of relative gravitational acceleration would need to approach zero - $\lim_{t \to t_{\mathrm{ff}}} \left|\overrightarrow{a_g}\,(t)\right| = 0$. Additionally, for the first part the program will need to record the time when the fall starts and stops. For the second part, it must record the instantaneous change in velocity and then calculate the change in height.

Design

The first step was to build key features in order to handle all the vectors this program would require. To simplify this, the program implements a Vector3D struct that stores an x, y, z, and t (time). Then a version of mag was added that took in a Vector3D and returned the magnitude of the vector. Additionally, the float equality approximation function close_to was reused from lab4.

Since the program would need to take many identical inputs from ds4rd.exe, a "getInput()" function was created as to take in and construct a Vector3D for each input from ds4rd.exe. This helped simplify the code, and reduced repetition.

Then, an enumeration was created to store the four parts of the fall, called FallState. These states are "unknown", "still", "falling", and "stopped". These are described below

- Unknown: describes when the controller might be falling (when accelerations are decreasing)
- Still: describes when there is still an outside force that is acting on it in addition to gravity or is stationary within a tolerance (MOTION_TOL). Better described as "hasn't fallen yet" rather than its shorthand "still"
- Falling: when the g-acceleration magnitude becomes around zero within FALL_TOL
- Stopped: when the g-acceleration magnitude increases between samples by a magnitude greater than JERK_TOL, or the magnitude reaches one (aka the still acceleration magnitude).

- Note: For the original version of this program, we were ignoring drag, which means that there was no reason we had to consider the increase in acceleration as the controller continued to fall. This meant the program only recorded until the magnitude was greater than 0.2 (where the ideal was 0.0, and the tolerance was 0.2)

Now that the key infrastructure was created, the focused was moved to building the main section. First all variables were declared, and the required outputs were made. The wait for data feature was implemented simply by waiting for the getInput function to finish – as long as there wasn't an input from ds4rd.exe that means the controller is not yet outputting data. While both part A and part B are similar at the top, the main loop implementation was quite different between the two parts. For the first part I tried using a single loop, using the FallState to determine what phase of the program we were in. What I realized is that this decreased readability and increased complexity. So, while it technically worked, I eventually moved to a much cleaner solution for part b. Both perform nearly identical functions, apart from the additional distance calculation made for part b. In this simplified implementation in part b, the FallState struct is more so used for readability.

Another difference between the two implementations is that the way they determine if the controller has started and stopped falling. The part A implementation waits for the acceleration magnitude to reach zero within the constant tolerance FALL_TOL; while the part B implementation waits for a decreasing acceleration that eventually reaches zero within the same tolerance as part B. For the fall ending, part A waits for the acceleration magnitude to leave the FALL_TOL, while the part B version waits for the magnitude to increase between samples by a magnitude greater than JERK_TOL, or the magnitude reaches one (aka the still acceleration magnitude). Essentially, part A relies on the assumption that there is no drag, while part B relies on the Newton's first law of motion, that there must be some kind of reduction in acceleration to counteract the current motion of the controller.

In order to make sure there wasn't spam from the "I'm Waiting……", and "Help me! I'm Falling!!!!!!" the program uses a counter, since each time doesn't follow the same exact same interval, it's just close. Since there's more time that it takes to wait than to fall, the interval between '.' is set to output every five time-intervals, while the '!' are set to output every four time-intervals.

Additionally, for part B, as soon as the FallState becomes unknown, the program starts adding velocity based on the change in time and the current acceleration, as to take air resistance (drag) into consideration. It then uses that to calculate the change in position and add it to a total distance fallen variable 'x'. If the FallState ends up as Still again, then the velocity and position counters are reset. Once it's considered definitely falling (when FallState = FALLING), the

program continues to calculate change in velocity and distance and adds to their respective variables. Once the controller has stopped, the 'x' variable is the total distance fallen when considering air resistance.

Finally, the program calculates the fall distance in two ways. The first way is using the function fallDist() which takes in a change in time and returns the equivalent of $\Delta d = \frac{1}{2} g(\Delta t^2)$. Second (for part B only) the program uses the 'x' variable – which is the total distance fallen when considering air resistance – and calculates the precent difference between the two. The program then prints both of these in accordance with the format given in the lab 5 outline.

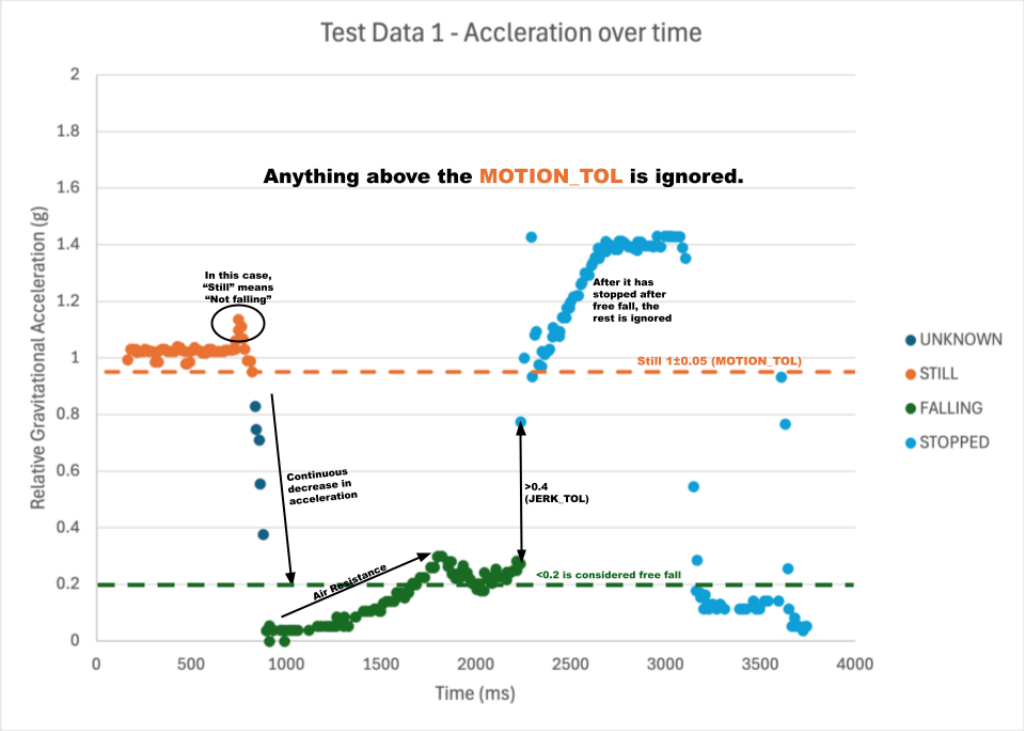
Testing

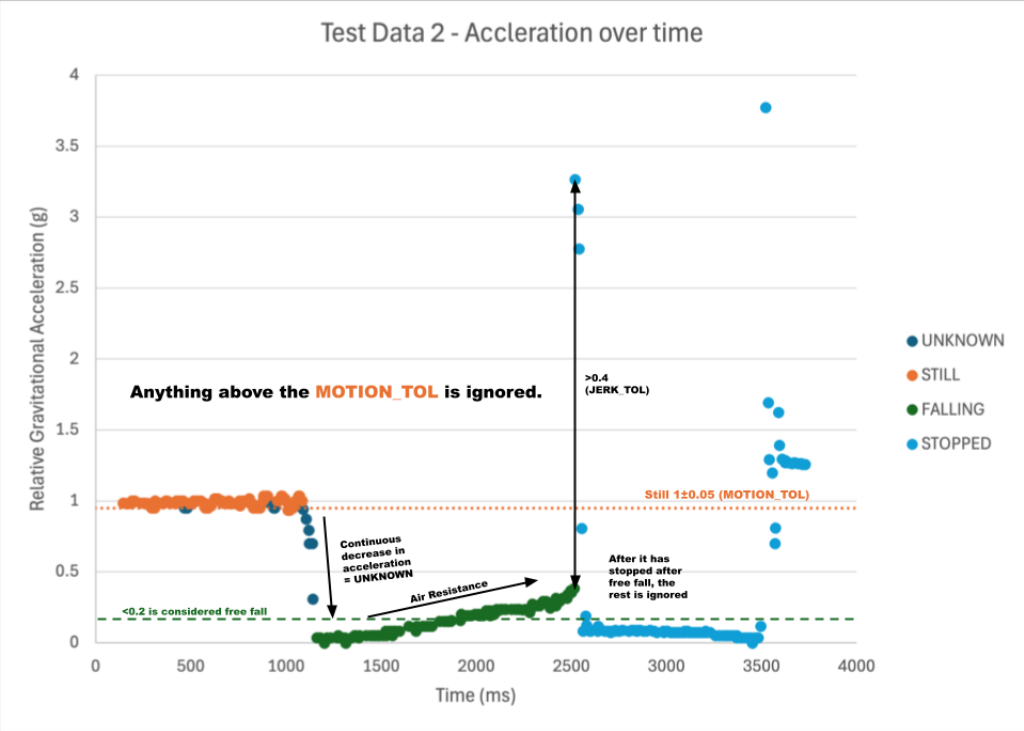
When conducting the same drop 5 times, using the top of desk as a reference point, then trying to catch it as close to the floor as safely possible, I got the following results: 0.544m, 0.602m, 0.562m, 0.589m, and 0.529m. When calculating the average value found, it gives: $\frac{0.544+0.602+0.562+0.589+0.529}{5}$=0.565m, this means that the results were consistent up to ±3.7cm. This variation is likely to have stemmed from inconsistencies in where the controller was dropped, where it was caught, potential acceleration induced during the drop, and even a bad accelerometer. In order to get better data, this data would need to be collected in a more scientific manner. When testing for part 2 with a similar drop as before (around 0.5m), the difference was 8% and 12% between the linear calculation and the one accounting for air-resistance.

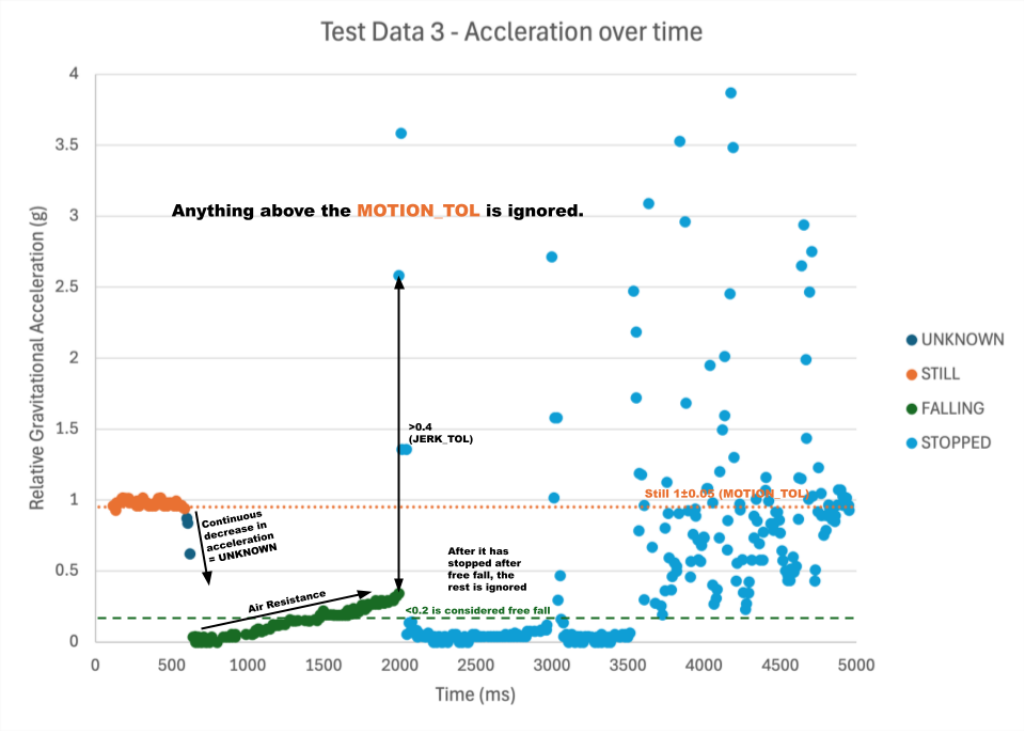
When using the test data, the program outputted the following results:

- Sample Data 1: 9.563m, 8.388m after accounting for air resistance (12% difference)
- Sample Data 2: 9.936m, 8.503m after accounting for air resistance (14% difference)
- Sample Data 3: 9.495m, 8.599m after accounting for air resistance (9% difference)

This means that, at least according to my program, the average distance from the balcony to the floor is around 8.50 meters, or around 28 feet.


Test Data 1 - Accleration over time
Anything above the MOTION_TOL is ignored.
In this case, "Still" means "Not falling"
After it has stopped after free fall, the rest is ignored
Still 1±0.05 (MOTION_TOL)
Continuous decrease in acceleration
>0.4 (JERK_TOL)
Air Resistance
<0.2 is considered free fall
Relative Gravitational Acceleration (g)
Time (ms)
UNKNOWN
STILL
FALLING
STOPPED


Test Data 2 - Accleration over time
Anything above the MOTION_TOL is ignored.
>0.4 (JERK_TOL)
Still 1±0.05 (MOTION_TOL)
Continuous decrease in acceleration = UNKNOWN
After it has stopped after free fall, the rest is ignored
Air Resistance
<0.2 is considered free fall
Relative Gravitational Acceleration (g)
Time (ms)
UNKNOWN
STILL
FALLING
STOPPED



Comments

The biggest issue that arose when implementing part 2 had to do with my original method of handling the loops. My original program used a single loop and an enumeration state variable to keep track of where in the loop it was. The problem was, however, that it wasn't capturing the original falling of the controller, and it was much harder to increment the position and velocity variables only where it was supposed to. So, my new version of the program used several different sections to do a similar task in a much more readable way, but also making the implementation of part 2 much easier.

Code implementation – PART 1

```c
/*-----------------------------------------------------------------------------
-                            CprE 1850 Lab 05
-----------------------------------------------------------------------------*/

#include <stdio.h>
#include <math.h>


typedef struct {
    double x;
    double y;
    double z;
    int t;
} Vector3D;

enum FallState {
    UNKNOWN,
    STILL,
    FALLING,
    DECELERATING,
    STOPPED
};


/*-----------------------------------------------------------------------------
-                               Prototypes
-----------------------------------------------------------------------------*/
double mag(Vector3D accel);
int close_to(double tolerance, double point, double value);
int in_motion(Vector3D accel);

Vector3D getInput(void);
double fallDist(double t);
```

Continued on the next page.

```c
/*------------------------------------------------------------------------------
-                            Implementation
------------------------------------------------------------------------------*/
int main(void) {
    int ts;
    int tf;
    Vector3D accel;
    enum FallState fs = UNKNOWN;

    fflush(stdout);
    printf("Maxwell Miller\n");
    printf("mill06\n");

    accel = getInput();

    printf("Ok, I'm now receiving data.\n");
    printf("I'm Waiting");
    fflush(stdout);


    while (fs != STOPPED) {
        // Get values from ds4rd.exe
        accel = getInput();

        if (in_motion(accel) && fs != UNKNOWN) {
            double a_mag = mag(accel);

            // If the acceleration is approaching zero, then it just started falling
            if (close_to(0.2, a_mag, 0.0) && fs != FALLING) {
                fs = FALLING;

                ts = accel.t;
                printf("\n\tHelp me! I'm Falling");
            }
            // If it is falling
            else if (fs == FALLING) {
                // If we're in motion but no longer approaching zero,
                // then we have hit the ground
                if (!close_to(0.2, a_mag, 0.0)) {
                    fs = STOPPED;
                    break;
                }

                if (accel.t % 10 == 0) {
                    printf("!");
                }
            }
```

Continued on the next page.

```
            else {
                fs = UNKNOWN;
            }
        }

        // If it is not in motion but it was falling,
        // then say the fall has ended, breaking the while loop
        else if (!in_motion(accel) && fs == FALLING) {
            printf("\nStopping 2\n");
            fs = STOPPED;
        }

        // If it is not in motion, just keep waiting for it to be
        else if (!in_motion(accel)) {
            fs = STILL;

            if (accel.t % 10 == 0) {
                printf(".");
            }
        }

        // If it makes it down here then it must be IN MOTION but hasn't fallen yet.
        // Therefore we must keep waiting while it is in the UNKNOWN state
        else {
            if (accel.t % 100 == 0) {
                printf(".");
            }
        }
        fflush(stdout);
    }

    tf = accel.t;

    double delta_sec = (double)(tf - ts) / 1000.0;
    printf("\n");
    printf(
        "\t\tOuch! I fell %4.3lf meters in %4.3lf seconds.\n",
        fallDist(delta_sec), delta_sec
    );

    return 0;
}
```

Continued on the next page.

```
/**
 * Finds the magnitude of a 3D vector
 *
 * @param vec Vector3D
 * @returns The vector magnitude
 */
double mag(Vector3D vec) {

    // returns the vector acceleration magnitude
    // ie. sqrt(a^2 + b^2 + c^2)
    return sqrt( ( vec.x * vec.x ) + ( vec.y * vec.y ) + ( vec.z * vec.z ) );
}


/**
 * Finds if two doubles are close to each other within a given tolerance
 *
 * @param tolerance Tolerance for the difference between val1 and val2.
 * @param val1 Double 1
 * @param val2 Double 2
 * @return 0 if not within tolerance, 1 if it is within tolerance.
 */
int close_to(double tolerance, double val1, double val2) {
    return fabs(val1 - val2) <= tolerance;
}


/**
 * Finds if the controller is still or in motion
 *
 * @param accel Vector3D acceleration
 * @returns The vector magnitude
 */
int in_motion(Vector3D accel) {
    const double MOTION_TOL = 0.05;

    double aMag = mag(accel);

    return !close_to(MOTION_TOL, 1.0, aMag);
}
```

Continued on the next page.

```c
/**
 * Gets input from the user
 * @returns A Vector3D struct with the input values
 */
Vector3D getInput(void) {
    Vector3D v;

    scanf("%d, %lf, %lf, %lf", &v.t, &v.x, &v.y, &v.z);

    return v;
}

/**
 * Calculates the distance fallen in a given time
 *
 * @param t Time in seconds
 * @returns Distance fallen in meters
 */
double fallDist(double t) {
    const double g = 9.8; // m/s^2
    // Derived from the equation of motion for constant acceleration
    // d = v_i * t + 1/2 * a * t^2  -->  d = 1/2 * g * t^2
    return 0.5 * g * t * t; // meters
}
```

Code implementation – PART 2

```c
/*-------------------------------------------------------------------------------
-                        CprE 1850 Lab 05 - Part 2
-------------------------------------------------------------------------------*/

#include <stdio.h>
#include <math.h>


const double FALL_TOL = 0.2;
const double MOTION_TOL = 0.05;
const double JERK_TOL = 0.4;
const double FALL_ACCEL = 0.0;
const double STILL_ACCEL = 1.0;
const double G = 9.8; // m/s^2


typedef struct {
    double x;
    double y;
    double z;
    int t;
} Vector3D;

enum FallState {
    UNKNOWN,
    STILL,
    FALLING,
    STOPPED
};


/*-------------------------------------------------------------------------------
                                Prototypes
-------------------------------------------------------------------------------*/
double mag(Vector3D accel);
int close_to(double tolerance, double point, double value);

Vector3D getInput(void);
double fallDist(double t);
```

Continued on the next page.

```c
/*-------------------------------------------------------------------------
                              Implementation
-------------------------------------------------------------------------*/
int main(void) {
    int ts;
    Vector3D tempAccel;
    Vector3D accel;

    double prev_mag;
    double curr_mag;

    double v = 0.0;
    double x = 0.0;
    double delta_sec;

    enum FallState fs = UNKNOWN;

    fflush(stdout);
    printf("Maxwell Miller\n");
    printf("mmill06\n");

    // Won't release until it has gotten the first input
    accel = getInput();

    printf("Ok, I'm now receiving data.\n");
    printf("I'm Waiting");
    fflush(stdout);

    int i = 0;
```

Continued on the next page.

```
while (fs == UNKNOWN || fs == STILL) {
        // Get values from ds4rd.exe
        tempAccel = getInput();
        prev_mag = mag(accel);
        curr_mag = mag(tempAccel);

        // Is falling if there's a significant change in
        // acceleration magnitude that has a negative trend
        if (close_to(FALL_TOL, curr_mag, FALL_ACCEL)) {
            fs = FALLING;
            break;
        }
        else if (fs == UNKNOWN && prev_mag - curr_mag > MOTION_TOL) {
            delta_sec = (double)(tempAccel.t - accel.t) / 1000.0;
            v += G * (1.0 - mag(tempAccel)) * delta_sec;
            x += v * delta_sec;
        }
        else if (fs != UNKNOWN && prev_mag - curr_mag > MOTION_TOL) {
            fs = UNKNOWN;
            ts = tempAccel.t;
            x = 0.0;
            v = 0.0;
        }
        else if (fs == UNKNOWN && prev_mag < curr_mag) {
            fs = STILL;
        }

        if (i % 5 == 0) {
            printf(".");
            fflush(stdout);
        }

        accel = tempAccel;
        i++;
    }

    printf("\n\n\tHelp me! I'm Falling");
```

Continued on the next page.

```
// While falling
    while (1) {
        tempAccel = getInput();

        if (close_to(MOTION_TOL, mag(accel), STILL_ACCEL) || mag(tempAccel) -
mag(accel) > JERK_TOL) {
            fs = STOPPED;
            break;
        }

        delta_sec = (double)(tempAccel.t - accel.t) / 1000.0;
        v += G * (1.0 - mag(tempAccel)) * delta_sec;
        x += v * delta_sec;

        // Only print every 4th time value to reduce spam
        if (i % 4 == 0) {
            printf("!");
        }

        accel = tempAccel;
        i++;
    }

    // Final Time in ms
    int tf = accel.t;

    // Delta Time in seconds
    delta_sec = (double)(tf - ts) / 1000.0;

    double dist = fallDist(delta_sec);
    int diff = (dist - x) / dist * 100;


    printf("\n");
    printf("\n\t\tOuch! I fell %4.3lf meters in %4.3lf seconds.\n",
fallDist(delta_sec), delta_sec);

    printf("\t\tCompensating for air resistance, the fall was %4.3lf meters.\n", x);
    printf("\t\tThis is %d%% less than computed before.\n", diff);


    return 0;
}
```

Continued on the next page.

```c
/**
 * Finds the magnitude of a 3D vector
 *
 * @param vec Vector3D
 * @returns The vector magnitude
 */
double mag(Vector3D vec) {

    // returns the vector acceleration magnitude
    // ie. sqrt(a^2 + b^2 + c^2)
    return sqrt( ( vec.x * vec.x ) + ( vec.y * vec.y ) + ( vec.z * vec.z ) );
}



/**
 * Finds if two doubles are close to each other within a given tolerance
 *
 * @param tolerance Tolerance for the difference between val1 and val2.
 * @param val1 Double 1
 * @param val2 Double 2
 * @return 0 if not within tolerance, 1 if it is within tolerance.
 */
int close_to(double tolerance, double val1, double val2) {
    return fabs(val1 - val2) <= tolerance;
}
/**
 * Gets input from the user
 * @returns A Vector3D struct with the input values
 */
Vector3D getInput(void) {
    Vector3D v;

    scanf("%d, %lf, %lf, %lf", &v.t, &v.x, &v.y, &v.z);

    return v;
}

/**
 * Calculates the distance fallen in a given time
 *
 * @param t Time in seconds
 * @returns Distance fallen in meters
 */
double fallDist(double t) {
    // Derived from the equation of motion for constant acceleration
    // d = v_i * t + 1/2 * a * t^2  -->  d = 1/2 * G * t^2
    return 0.5 * G * t * t; // meters
}
```