Lab 5: Forces and torques in Simbody

There are a variety of ***models for forces*** in nature and the study of forces is a major differentiating factor in fields such as molecular dynamics, biomechanics, automotive, aerospace, robotics, and machine design. For example, there are force models for: uniform gravity, universal gravity, springs (elastic and inelastic), dampers, viscosity, adhesion, surface tension, osmosis, buoyancy, hydrostatics, hydraulics, pneumatics, tires, aerodynamics (lift, drag, and thrust), muscles, tendons, ligaments, friction (e.g., Coulomb, kinetic, static, and Dahl), contact (e.g., Hertzian), compression, tension, pressure, traction, shear, electrostatics, electromagnetism (e.g., Lorentz and Faraday), pizoelectric, motors, nuclear (strong and weak), van der Waals, etc. It is worth emphasizing that ***"all models are approximate - some are useful"***.[1]

Forces are a key ingredient in Newton's law $\mathbf{F} = m\mathbf{a}$ and accurate models of forces are **essential** for physics-based simulation of biological structures. SimTK Simbody simulates a wide range of biological structures and physical systems with user-specified and built-in forces.

Replacement of forces

It is worth noting that a set of forces acting on a ***rigid body*** B can be replaced with a single force $\mathbf{F}$ applied to point B_p of B, together with a couple of torque $\mathbf{T}$. To reiterate, when a biological structure is ***rigid***, it can be helpful to replace a set of forces with a single force and a torque.

[0]Last updated May 29, 2007 by Paul Mitiguy.

[1]In 1950, the statistician John Tukey (who invented the fast-Fourier transform algorithm, coined the terms "***software***" and "***bit***", and did a statistical critique of the Kinsey report on the sexual behavior of males) provided an insightful comment about common practices in sciences "The combination of some data and an aching desire for an answer does not ensure that a reasonable answer can be extracted from a given body of data.".

5.1 Simulating Newton's apple with air-resistance

The figure to the right shows an apple falling from a tree. The point of this simulation is to determine the affect of air-resistance on the falling apple using Simbody's `GeneralForceElement::UserForce class`.

The resultant of all air-resistance forces on the apple is ***modeled***[a] as a single force $\mathbf{F}$ applied to the apple's mass center, equal to

$$\mathbf{F} \;=\; -b\,\mathbf{v}^2\,\frac{\mathbf{v}}{|\mathbf{v}|} \;\;=\;\; -b\,|\mathbf{v}|\,\mathbf{v}$$

where b is a constant associated with the force (***drag***) due to the air-resistance and $\mathbf{v}$ is the velocity of the apple's mass center.

Quantity	Symbol	Type	Value
Earth's gravity	g	constant	9.8 m/sec
Mass of apple	m	constant	0.142 kg (5 ozm)
Air-resistance damping constant	b	constant	0.005 kg/m
Apple's vertical position from N_o	y	variable	varies

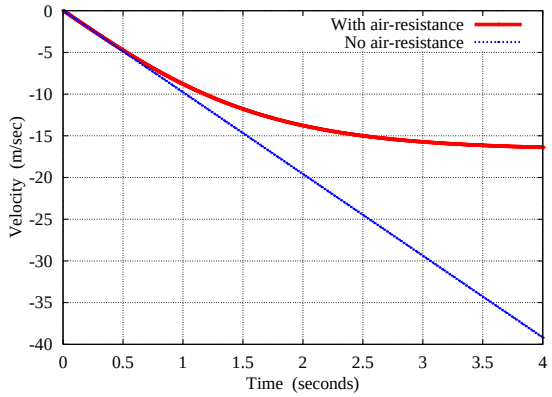
[a] A ***model*** is a simplified representation of a complex system.

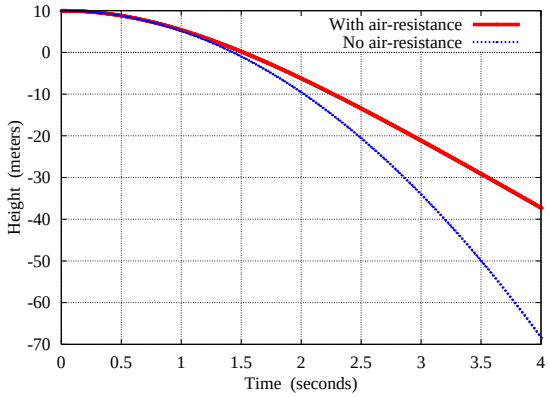
Modify the file `NewtonsAppleForVisualization.cpp` from Lab 2.2.2 to add an air-resistance force subsystem to the multibody system as specified on the next page.

- The apple is dropped from **rest** from a height of 10 m. Using the values for g and b in the previous table, **attempt** to find analytical solutions for $\dot{y}(t)$ and $y(t)$.

$$\dot{y}(t) = \qquad\qquad y(t) =$$

- Using the data in `NewtonsAppleResults.txt` plot the time-history of $\dot{y}$ and y for $0 \le t \le 4$ sec. Compare the graph of Simbody's results for $y(t)$ with the graph of the exact analytical results with **no air-resistance** of $\dot{y}(t) = -4.9\,t$ and $y(t) = 10 - 4.9\,t^2$.





- Using free-body diagrams, calculate the ***terminal velocity*** of the falling apple to 5 significant digits.

$$\text{Terminal velocity} \;=\; \qquad \text{m/sec}$$

- In view of the data in `NewtonsAppleResults.txt`, write down the value for $\dot{y}$(t) at $t = 4$ sec. Determine the difference (in percent) $\dot{y}(4)$ is to terminal velocity (to 2 significant digits).

$$\dot{y}(4) \;=\; \qquad \text{m/sec} \qquad\qquad \text{Difference:} \qquad \%$$

To modify `NewtonsAppleForVisualization.cpp` to add air-resistance:

1. Add the following line of code to near the top of the file:

```
#include "UserForceAirResistanceVertical.h"
```

2. Add the following code before realizing the state for the first time, i.e., before `mbs.realize(s);`

```
// Add an air-resistance force sub-system to this multi-body system.
GeneralForceElements userForceElements;
mbs.addForceSubsystem( userForceElements );

// Although "new" was used to allocate this UserForce, do not "delete" it.
// This bug will be fixed in the next version of Simbody so it can take an object from the stack or heap.
// For now, addUserForce takes ownership of the allocated item and takes care of deleting it at the end.
UserForceAirResistanceVertical *airResistanceOnApple = new UserForceAirResistanceVertical( appleBodyId, 0.005 );
userForceElements.addUserForce( airResistanceOnApple );
```

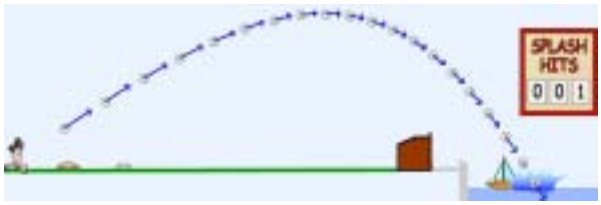
3. Ensure that the file `UserForceAirResistanceVertical.h` (contents shown on the next page) is in the same folder (directory) as `NewtonsAppleForVisualization.cpp`.

4. Compile, link, and run your modified `NewtonsAppleForVisualization.cpp` code and complete the questions on the previous page. Submit your modified `NewtonsAppleForVisualization.cpp` code and your results to your SimTK project.

```
//------------------------------------------------------------------------------
// File:     UserForceAirResistanceVertical.h
// Class:    UserForceAirResistanceVertical
// Parent:   GeneralForceElements
// Children: None
// Purpose:  Applies an air-resistance force on an object falling vertically.
// Author:   Paul Mitiguy - May 20, 2007
//------------------------------------------------------------------------------
#ifndef __USERFORCEVERTICALAIRRESISTANCE_H__
#define __USERFORCEVERTICALAIRRESISTANCE_H__
//------------------------------------------------------------------------------
#include "StandardCppHeadersAndNamespace.h"
#include "SimTKsimbody.h"
using namespace SimTK;

//------------------------------------------------------------------------------
// User-defined classes for adding forces/torques are constructed as follows:
// 1. Create a constructor with whatever arguments make sense for the force or torque and copy the arguments into class d
// 2. Create a clone method (all clone methods are identical except for the class name appearing after "new")
// 3. Create a calc  method (the arguments and return type for all calc methods are identical).
//    The code in the calc method is specific to the calculation of force or torque.
// Note: The set of all forces is replaced by an equivalent set, consisting of a torque
//       that is equal to the moment of the forces about the body's origin together
//       with the resultant of the forces applied at the body's origin.
//------------------------------------------------------------------------------
class UserForceAirResistanceVertical : public GeneralForceElements::UserForce
{
public:
   // Constructor is explicit
   explicit UserForceAirResistanceVertical( BodyId bodyIdA, Real coefficientForAirResistance )
   {
      myBodyIdForApplyingForce = bodyIdA;
      myCoefficientMultiplyingVelocitySquared = coefficientForAirResistance;
   }

   // The clone method is used internally by Simbody (required by virtual parent class)
   UserForce*  clone() const  { return new UserForceAirResistanceVertical(*this); }

   // The calc method is where forces or torques are calculated (required by virtual parent class)
   void  calc( const MatterSubsystem& matter,              // Input information (matter)
               const State&           state,               // Input information (current state)
               Vector_<SpatialVec>&   bodyForces,          // Forces and torques on bodies
               Vector_<Vec3>&         particleForces,      // Forces on particles (currently unused)
               Vector&                mobilityForces,      // Generalized forces
               Real&                  pe ) const           // For forces with a potential energy
    {
        // Query the matter subsystem for the body's origin velocity in ground.
        // This vector is expressed in the ground's "x,y,z" unit vectors.
        const Vec3 bodyVelocity = matter.calcBodyOriginVelocityInBody( state, myBodyIdForApplyingForce, GroundId );
        Real yVelocity = bodyVelocity[1];

        // The force's magnitude is -b*|v|^2 and its direction is v/|v|, hence the force is  -b*|v|*v
        Real magVelocity = fabs( yVelocity );
        Real yForce = -myCoefficientMultiplyingVelocitySquared * magVelocity * yVelocity;

        // Get the proper memory location to increment the force and/or torque.
        // bodiesForces is a Vec6 whose elements are two Vec3.
        // The elements of the first  Vec3 are Tx, Ty, Tz  (expressed in the ground's "x,y,z").
        // The elements of the second Vec3 are Fx, Fy, Fz  (expressed in the ground's "x,y,z").
        SpatialVec& bodiesForces = bodyForces[ myBodyIdForApplyingForce ];
        Vec3& torqueSum = bodiesForces[0];
        Vec3& forceSum  = bodiesForces[1];

        // Increment the sum of all forces on this body (other force subsystems may also add forces/torque)
        // torqueSum[0] += 0;       // Increment torque in the ground's x-direction.
        // torqueSum[1] += 0;       // Increment torque in the ground's y-direction.
        // torqueSum[2] += 0;       // Increment torque in the ground's z-direction.
        // forceSum[0]  += 0;       // Increment force  in the ground's x-direction.
        forceSum[1]  += yForce;     // Increment force  in the ground's y-direction.
        // forceSum[0]  += 0;       // Increment force  in the ground's z-direction.
    }

private:
    BodyId  myBodyIdForApplyingForce;
    Real    myCoefficientMultiplyingVelocitySquared;
};
//------------------------------------------------------------------------------
#endif /* __USERFORCEVERTICALAIRRESISTANCE_H__ */
//------------------------------------------------------------------------------
```

5.2 Simulating projectile motion with air-resistance

The figure to the right shows a baseball being hit out of AT&T park. The point of this simulation is to **write a C^{++} class** that causes Simbody to apply a single air-resistance force $\mathbf{F}$ to the baseball's mass center with:



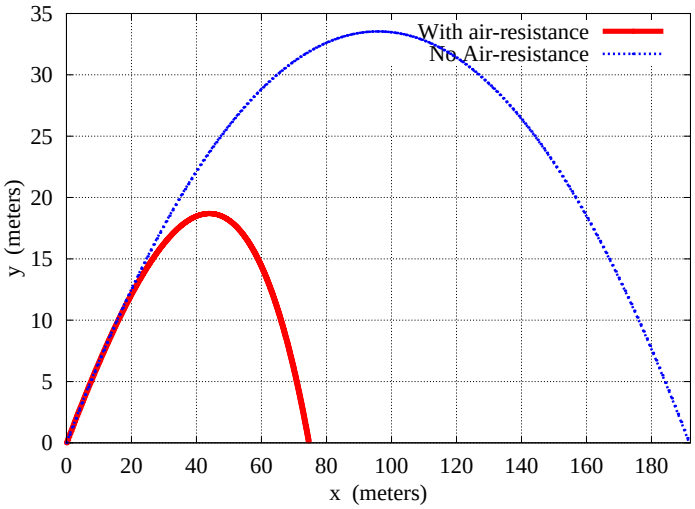
$$\mathbf{F} \;=\; \text{-}b\,\mathbf{v}^2\,\frac{\mathbf{v}}{|\mathbf{v}|} \quad = \quad \text{-}b\,|\mathbf{v}|\,\mathbf{v}$$

where b is an air-resistance damping constant and $\mathbf{v}$ is the velocity of the baseball's mass center.

Quantity	Symbol	Type	Value
Earth's gravity	g	constant	9.8 m/sec
Mass of baseball	m	constant	0.142 kg (5 ozm)
Radius of gyration of baseball (sphere)	r	constant	2.3114 cm (0.91 in)
Air-resistance damping constant	b	constant	0.002 kg/m
Baseball's horizontal position from N_o	x	variable	Initially $x = 0$
Baseball's vertical position from N_o	y	variable	Initially $y = 0$

- Modify the Simbody program `ProjectileMotion.cpp` to simulate the baseball and visualize the affect of air-resistance on 2D projectile motion. Submit your modified and **fully commented** C^{++} program `ProjectileMotion.cpp` to your SimTK project at www.simtk.org. Use a baseball that is launched at 35° from the horizontal with an initial speed of 44.7 m/sec (100 mph).
  Hint: Press the 'r' key on your keyboard to resize the VTK window while running the animation.
  Slow the animation by "sleeping" (suspend program execution) or using a smaller integration step (e.g., 0.005).

- Plot the Simbody results for $y(x)$ for 5.23 sec with an integration step of 0.01 sec. Compare the graph of Simbody's results for $y(x)$ with the graph of the exact analytical results with **no air-resistance** of $y(x) = \tan(35°)\,x \;-\; \frac{4.9}{[44.7\,\cos(35°)]^2}\,x^2$.
  Result:



- In view of these simulation results, it seems that accurate models of forces are critical to determining how an object moves. **True/False**.

- Accurate models of air-resistance forces on baseball exist. **True/False**.

- Optional**: Use the Simbody numerical integrator root finder to find the value of x at which the ball hits the ground.

To modify `ProjectileMotion.cpp` to add air-resistance:

1. Add the following line of code to near the top of the file:

   ```
   #include "UserForceAirResistanceProjectile.h"
   ```

2. Add the following code before realizing the state for the first time, i.e., before `mbs.realize(s);`

   ```
   // Add an air-resistance force sub-system to this multi-body system.
   GeneralForceElements userForceElements;
   mbs.addForceSubsystem( userForceElements );

   UserForceAirResistanceProjectile *airResistanceOnBaseball = new UserForceAirResistanceProjectile( baseballBodyId, 0.002 );
   userForceElements.addUserForce( airResistanceOnBaseball );
   ```

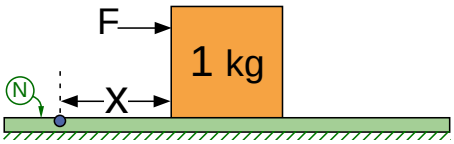
3. **Create** the file `UserForceAirResistanceProjectile.h` and place it in the same folder (directory) as `ProjectileMotion.cpp`.

4. Compile, link, and run your modified `ProjectileMotion.cpp` and complete the previous questions. Submit your modified `ProjectileMotion.cpp` code and your results to your SimTK project.

5.3 Harmonic forcing of a simple system

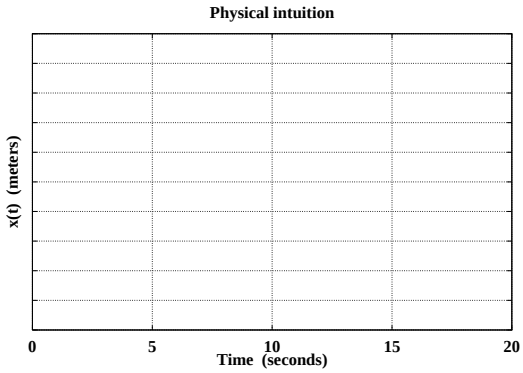
The point of this problem is to use physical intuition to **guess** at the time-response of a harmonically forced system and then validate your physical intuition against mathematical reality.

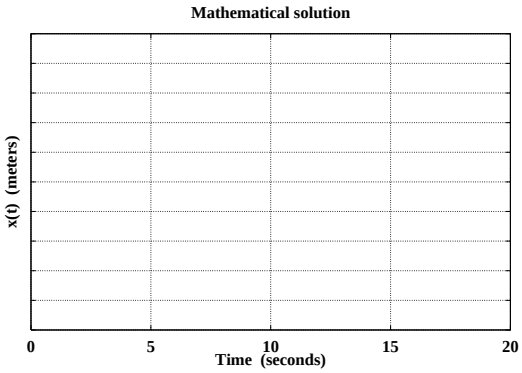
Consider a 1 kg particle on a frictionless horizontal surface N. The particle may slide right or left, depending on an applied horizontal force F.



1. Using Newton's law, form the equation of motion that governs $x(t)$ when $F = \sin(t)$. Write it in standard form (with all the inhomogeneous terms on the right-hand side).
   Result:
   $$\frac{d^2x}{dt^2} = \sin(t)$$

2. Knowing $x(0) = 0$ and $\dot{x}(0) = 0$, use your physical intuition to sketch the solution for $x(t)$ for $0 \le t \le 20$ on the left plot below.
   Result:



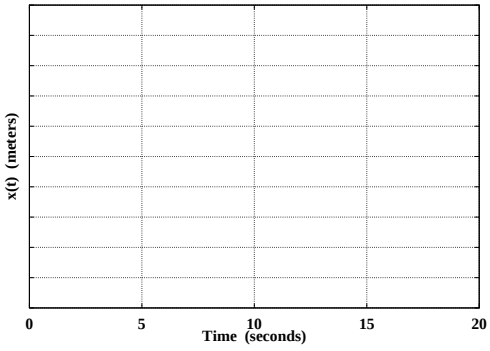


3. An easy way to solve the governing ODE is by separating variables and integrating twice. Calculate the solution for $x(t)$ corresponding to $x(0)$=0 and $\dot{x}(0)$=0 and sketch it for $0 \le t \le 20$ on the plot to the right of the one you created with physical intuition. $x(t) =$ _____
   Does your physical intuition match your mathematical solution **Yes/No** (circle one).

4. Copy the file `NewtonsAppleForVisualization.cpp` to `AppleWithHorizontalForce.cpp` and simulate the block's horizontal motion when:

   $$x(0) = 0 \qquad \dot{x}(0) = 0 \qquad f(t) = \cos(t)$$

   Plot the solution for $x(t)$ for $0 \le t \le 20$.
   Use an integration time step of 0.01 sec.



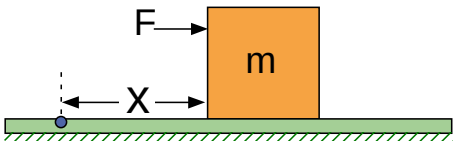
   (a) **Create** the file (class) `UserForceSintHorizontal.h`
   (b) Add `#include "UserForceSintHorizontal.h"` near the top of the file
   (c) Add the following code before `mbs.realize(s);`

```
GeneralForceElements userForceElements;
mbs.addForceSubsystem( userForceElements );

UserForceSintHorizontal *sinusoidalForceOnApple = new UserForceSintHorizontal( appleBodyId );
userForceElements.addUserForce( sinusoidalForceOnApple );
```

5.4 Computed torque control of a simple system

The figure to the right shows a particle of mass m that slides on a frictionless horizontal surface. The particle may slide right or left, depending on an applied horizontal force F.

$$F \;=\; m\,\ddot{x}$$



The point of this example is to use a ***computed torque control*** for F so x has a **desired** behavior (denoted x_{des}). In other words, a proper choice for F moves the block in a **desired** way, e.g., with $x_{\text{des}} = 0$ or $x_{\text{des}} = 5$ or $x_{\text{des}} = t\,\sin(3\,t)$ or ...

The computed torque control law for F that causes $x(t)$ to follow the ***desired trajectory*** $x_{\text{des}}(t)$ is

$$F \;=\; m\,[\ddot{x}_{\text{des}} \;+\; k_d\,(\dot{x}_{\text{des}} - \dot{x}) \;+\; k_p\,(x_{\text{des}} - x)]$$

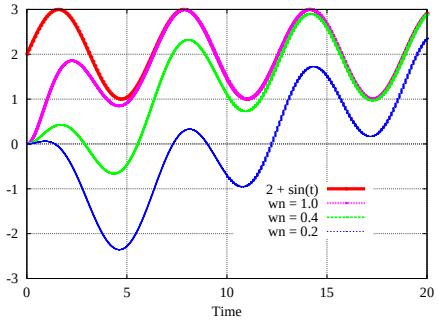
A good choice for k_d and k_p that causes the error to damp out quickly is[2]

$$k_d \;=\; 2\,\omega_n \qquad\qquad k_p \;=\; \omega_n^2$$

where ω_n is chosen to be as large as possible to provide appropriate response.[3]

Generate the figure to the right with:

- A desired trajectory of $x_{\text{des}} = 2 + \sin(t)$
- A value of $\omega_n = 0.4$ (control constants of $k_d = 0.8$ and $k_p = 1.6$)
- The block starting from **rest** ($\dot{x}$=0) at x=0
- Repeat with $\omega_n = 0.2$ and $\omega_n = 1.0$



1. **Create** the file `UserForceControlHorizontal.h`
2. Add `#include "UserForceControlHorizontal.h"` near the top of `AppleWithHorizontalForce.cpp`
3. In `AppleWithHorizontalForce.cpp`, **comment out** the following code before `mbs.realize(s);`

```
// UserForceSintHorizontal *sinusoidalForceOnApple = new UserForceSintHorizontal( appleBodyId );
// userForceElements.addUserForce( sinusoidalForceOnApple );
```

4. In `AppleWithHorizontalForce.cpp`, **insert** the lines:

```
UserForceControlHorizontal *computedTorque = new UserForceControlHorizontal( appleBodyId, massOfApple, 1.0, 0.4 );
userForceElements.addUserForce( computedTorque );
```

   Note: the 3^{rd} argument of the `UserForceControlHorizontal` constructor is ζ and the 4^{th} argument is ω_n.

5. Plot the Simbody solution for $x(t)$ for $0 \le t \le 20$.
   Use an integration time step of 0.01 sec. Repeat with $\omega_n = 0.2$ and again with $\omega_n = 1.0$.
6. Submit your modified and fully commented `AppleWithHorizontalForce.cpp` file and your results to your SimTK project.

[2] This choice for k_d and k_p is based on error that is ***critically damped***, i.e., $\zeta = 1$.
[3] Large values of ω_n correspond to large values of F - which may not be physically possible.