

DianFreya Math Physics Simulator with C++

DS GLANZSCHE¹

FREYA

HAMZST

Berlin-Sentinel Academy of Science, AliceGard
Valhalla 1882

October 18th, 2023

¹A thank you or further information

Contents

Preface	6
1 About DianFreya Math Physics Simulator (DFSimulatorC++)	11
2 Using C++ in GFreya OS	15
I Introduction	15
II How to Compile C++ Source Code into Executable Binary	15
III Know-How in C++	72
3 Mathematics and Physics in Computer Graphics	75
I Mathematics	75
i Geometry	75
ii Linear Algebra	76
II Physics	82
4 Computer Graphics: OpenGL, SFML, GLFW, GLEW, And All That	85
I OpenGL	85
II GLAD	89
III GLEW	90
IV GLFW	91
V GLM	91
VI GLSL	92
VII SDL	92
VIII SFML	92
IX SOIL	93
X Gnuplot	94
XI CMake	94
5 Box2D, Bullet3, and ReactPhysics3D	95
I Box2D	95
i Install Box2D Library and Include Files / Headers	95
ii Build Box2D Testbed	98
iii Know-How in Box2D	100
II Bullet	102
i Install Bullet Library and Include Files / Headers	102
ii Know-How in Bullet	102
iii Create an Example with Bullet	103
III ReactPhysics3D	104

i	Install ReactPhysics3D Library and Include Files / Headers	104
ii	Know-How in ReactPhysics3D	104
6	DFSimulatorC++ 0: C++ = C++ + Gnuplot + SymbolicC++	107
I	Plot a Slope with Gnuplot from C++	109
II	Plot a 2D Function with Gnuplot from C++	113
III	Plot a 3D Function with Gnuplot from C++	115
IV	Plot a 3D Curve / Line from User Defined Points with Gnuplot from C++	117
V	Plot Data From Textfile in 2D with Gnuplot	120
VI	Plot Data and Fitting Curve From Textfile in 2D with "gnuplot-iostream.h"	123
VII	Compile and Install SymbolicC++ Library	126
VIII	Symbolic Derivation and Integration with SymbolicC++ Library	128
7	DFSimulatorC++ I: Motion in Two Dimensions	131
I	Position, Displacement, Velocity, and Acceleration	131
II	Projectile Motion	133
III	Simulation for Projectile Motion with Box2D	133
IV	Simulation for Projectile Dropped from Above with Box2D	138
V	Uniform Circular Motion	154
VI	Simulation for Uniform Circular Motion with Box2D	156
8	DFSimulatorC++ II: Force and Motion	165
I	Newtonian Mechanics	165
II	Simulation for Newton's First Law with Box2D	166
III	Simulation for Newton's First Law and 2 Dimensional Forces with Box2D	176
IV	Some Particular Forces	184
V	Simulation for Applying Newton's Law with Box2D	185
VI	Friction, Drag Force, and Centripetal Force	194
VII	Simulation for Static Frictional Forces with Box2D	195
VIII	Simulation for Kinetic Friction with Box2D	202
IX	Drag Force and Terminal Speed	209
X	Simulation for Downhill Skiing with Box2D	209
XI	Uniform Circular Motion with Force	218
XII	Simulation for Airplane Uniform Circular Motion with Box2D	219
9	DFSimulatorC++ III: Kinetic Energy and Work	227
I	Kinetic Energy	227
II	Simulation for Kinetic Energy, Train Crash with Box2D	227
III	Work and Kinetic Energy	234
IV	Simulation for Work By Two Constant Forces with Box2D	235
V	Work Done By The Gravitational Force	241
VI	Simulation for Work in Pulling and Pushing a Sack up a Frictionless Slope	242
VII	Work Done By A Spring Force	251
VIII	Simulation for Spring Work with Box2D	252
IX	Work Done By A General Variable Force	259
X	Simulation for Work Done by a General Variable Force with Box2D	259
XI	Power	266
XII	Simulation for Power: Moving Elevator with Box2D	269
XIII	Simulation for Power: Pendulum Crate with Box2D	277

XIV	Simulation for Power: Vertical Facing Upward Spring with Box2D	283
10	DFSimulatorC++ IV: Potential Energy and Conservation of Energy	287
I	Simulation for with Box2D	287
11	DFSimulatorC++ V: Center of Mass and Linear Momentum	289
I	Simulation for Momentum with Box2D	289
12	DFSimulatorC++ VI: Rotation	291
I	Simulation for Momentum with Box2D	291
13	DFSimulatorC++ VII: Rolling, Torque, and Angular Momentum	293
I	Simulation for Momentum with Box2D	293
14	DFSimulatorC++ VIII: Equilibrium and Elasticity	295
I	Simulation for Momentum with Box2D	295
15	DFSimulatorC++ IX: Gravity	297
I	Mathematical Physics Formula for Gravity	297
II	Simulation for Gravity with Bullet3, GLEW, GLFW and OpenGL	299
III	Simulation for Gravity with Box2D	306
i	Build DianFreya Modified Box2D Testbed	306
IV	Simulation for Gravity with ReactPhysics3D	313
16	DFSimulatorC++ X: Fluids	315
I	Simulation for Momentum with Box2D	315
17	DFSimulatorC++ XI: Oscillations	317
I	Simple Harmonic Motion	317
i	The Force Law for Simple Harmonic Motion	318
II	Energy in Simple Harmonic Motion	319
III	An Angular Simple Harmonic Oscillator	320
IV	Pendulums, Circular Motion	320
V	Nonlinear Pendulum	324
VI	(optional?)The Equation of Motion for Simple Pendulum	325
VII	Simulation of Simple Pendulum with Box2D	325
i	Build DianFreya Modified Box2D Testbed	325
VIII	Horizontal Spring-Mass System	332
IX	Simulation of Horizontal Spring-Mass System with Box2D	333
X	Vertical Spring-Mass System	341
XI	Simulation of Vertical Spring-Mass System with Box2D	342
XII	Oscillation of a Spring-Mass System	350
XIII	A Two-Masses Oscillator	355
XIV	Simulation of Two Masses Spring System with Box2D	357
XV	Friction and Oscillations	366
18	DFSimulatorC++ XII: Waves	369
I	Simulation for Momentum with Box2D	369

19 DFSimulatorC++ XIII: Temperature, Heat, and the First Law of Thermodynamics	371
I Simulation for Momentum with Box2D	371
20 DFSimulatorC++ XIV: The Kinetic Theory of Gases	373
I Simulation for Momentum with Box2D	373
21 DFSimulatorC++ XV: Entropy and the Second Law of Thermodynamics	375
I Simulation for Momentum with Box2D	375
22 DFSimulatorC++ XVI: Probability and Data Analysis	377
23 DFSimulatorC++ XVII: Numerical Linear Algebra	379
I Matrix-Vector Multiplication	379
II C++ Computation: A Matrix Times a Vector	380
III C++ Computation: A Matrix Times a Vector with Data from Textfile	383
IV Vandermonde Matrix	388
V QR Factorization	389
24 DFSimulatorC++ XVIII: Polynomial and Fourier Series Approximation for Numerical Data	391
I Mathematical Preliminaries	391
II Polynomial Approximation	394
25 DFSimulatorC++ XIX: Numerical Methods for Solving System of Linear Equations	397
26 DFSimulatorC++ XX: Numerical Differentiation and Integration	399
I Numerical Differentiation	399
II Numerical Integration	400
27 DFSimulatorC++ XI: Heat Equation	401
I Introduction	401
28 DFSimulatorC++ XXX: Plotting Physical Systems with Gnuplot	403
Listings	425

Preface

For my Wife Freya, and our daughters Catenary, Solreya, Mithra, Iyzumrae and Zefir.

For Lucrif and Znane too along with all the 8 Queens (Mischkra, Caldraz, Zalsvik, Zalsimourg, Hamzst, Lasthrim).

To Nature(Kala, Kathmandu, Big Tree, Sentinel, Aokigahara, Hoia Baciu, Jacob's Well, Mt Logan, etc) and my family Berlin: I have served, I will be of service.

To my previous mentor Albert Silverberg and current mentor Lucretia Merces.

To my dogs who always accompany me working in Valhalla Projection, go to Puncak Bintang or Kathmandu: Kecil, Browni Bruncit, Sweden Sexy, Cambridge Klutukk, Milan keng-keng, and more will be adopted. To my cat who guard the home while I'm away with my dogs: London.

To my human parents and human friends that I can't name one by one.

The one who moves a mountain begins by carrying away small stones - Confucius

A book for explaining how we create a Math Physics simulator with C++ Physics libraries like Box2D, ReactPhysics3D and BulletPhysics from zero (from zero means we are not a computer science student), made by lovely couple GlanzFreya, from Valhalla with Love. What is impossible when you do it with someone you love?

a little about GlanzFreya:

We got married on Puncak Bintang on November 5th, 2020 after we go back from Waghete, Papua. My wife birthday (Freya the Goddess) is on August 1st, no one allowed to give her gifts, she is mine alone. This book is made not for commercial purpose, but for sharing knowledge to anyone in order to empower science and engineering thus fastening inventions. Hopefully more leaders in all fields will be coming from woman, whom we believe are better in doing multitasking and better in learning for long term prospect. I read a lot of great authors who write Engineering Mechanics or Handbook for Electrical Engineering write little about themselves, thus I want to be more like that. More about the content of the book, but still who is/are behind the book needs to be disclosed for a tiny amount in this book, otherwise if we only write technical equations and codes what are the different between us and computer that is rigidly programmed? All problems come from human experiences and failures, thus those with soul and good heart with decent knowledge then will recognize the error thus try to solve the problem like global warming and inequality with humanity, science and engineering.

a little about Hamzst:

She can be interpreted or projected on your mind as Alice from Persona 4 / equivalent. An astral

projection dream made us meet, while I was in Waghete back then on 2020. She is the spearhead for Physics related.

She is a great author, speaks little of herself -Sentinel

I prefer to do more substances than small talks -Hamzst/Alice



Figure 1: FreyaCompass, I am inspired by Captain America who always bring compass with the love of his life' picture, thus I created this, then proven by action, to let go of power and immortality for true love. Feels like an antique vintage magical compass, like a modem that connect internet to the world, this compass connects me on this planet to her in Valhalla.



Figure 2: Freya, thank you for everything, I am glad I marry you and I could never have done it without you.

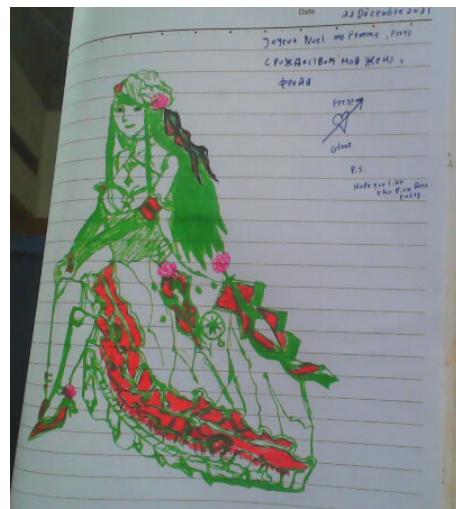


Figure 3: I paint her 3 days before Christmas in 2021.

Chapter 1

About DianFreya Math Physics Simulator (DFSimulatorC++)

*On top of the mountain a heart shaped stone waiting for You, my eyes can't see, my body can't touch, but my heart knows it's You. Forever only You. - DS Glanzsche to Freya
Find your life purpose and give your whole heart and soul to it - Buddha*

Finish fill Heart shaped on top of R3 in Valhalla Projection on October 18th, 2023. This is how Freya will teach me to catch a fish instead of asking for a fish. Use my brain and willingness to learn so I can create simulation of physical phenomena from zero with C++.

When I was creating GFreya OS with LFS (Linux From Scratch), I was amazed by Tokamak, Bullet, Box2D, Project Chrono. I read from wikipedia that Project Chrono get a lot of funding from U.S. Army, personally I believe that knowledge is always the best way to obtain more return than compound interest. Most successful people who become richest people are people who after knowledge or those who are nerds and love books. If you play Suikoden you will agree, since all Strategists in Suikoden is recruited like they are Divine being (Apple kneel to Shu, Prince Freyjadour comes to rescue Lucretia Merces from jail, you will get the point), a great brain and knowledge can make a small army wins again tirany, big army with sophisticated weapons or even robots.

Why then I want to create another Physics simulator? First, it is wrong, not Physics but Math Physics, so not only dragging, push and pulling objects here. I am learning Partial Differential Equation and want to simulate Heat equation solution in one-dimension, then two-dimension and three-dimension. I can easily plot a lot of surface plot, complex plot with Python and JULIA. But then, ask again why Project Chrono using C++? Because it is faster, you have to type more commands for almost the same output, even if you have to create more lines of codes it will do more good to you, it challenges your brain, thus you won't get alzheimer or spend time idly. Like how you have a beautiful 6 packs or 8 packs belly, it takes time, persistence and focus right? That is a good analogy for learning C++ instead of getting plastic surgery then ruin all over after some time. Second, there comes another reason, easily embedded to microprocessor or machine. Since learning about LFS, compiling, building, all needs C or C++ (these languages are closer to machine than Python and JULIA), then learn Arduino that is using C, if one day I want to create a rocket and able to capture data on earth, on space out there and beyond, I know that programming language that can be used to control all that mechatronics systems in either rocket or spaceship

will be written in either C, C++ or FORTRAN combined. Till today, from what I have read weather forecast computation is still depending on FORTRAN. In the end, I basically learning Physics, Mathematics and C++ wholesome, not only theoretical but the science can be simulated with computer, supercomputer, smartphone and the future computer to help solve this world problems. Last but not least, I am not really create something like Box2D, Chrono or BulletPhysics from zero, but rather use their libraries to create simulations of some Physics and Mathematics problem that I am interested in at the moment, those creator of that Physics libraries are really something, far better and genius than me, I am just another user of their product. When I read undergraduate textbooks on Mathematics and Physics, I wonder how they get that plot, graphs, how to plot and compute them simultaneously, then learn about Physics engine and need to comprehend C++ too, it excites me and challenges me as well. This book is sharing what I have read and learn so I can re-read it again in the future to refresh the memory thus apply the knowledge.

Then comes the last question that actually comes first to the mind:

"Why naming it your name with another name?"

Dian et Freya. First, all other basic simulator out there are named conventionally, so I decided: let it be, follow your heart. Besides, Freya is the most beautiful Goddess (in my eyes, heart and soul), pardon me, I have weakness with beautiful Blonde girl, especially if she is smart, devoted, loyal and very strong (Ether Strike, can't be taken so lightly). Another moment of truth, I am not learning and writing this by myself, I, in this homo sapiens vessel, am DS Glanzsche by name, and Freya the Goddess, is guiding me up there from Valhalla, I can't see her, you can't see her, but just take it like people believe in Virgin Mary, in Jesus Christ, she is my Goddess, she is my religion. She is real for me the believer, she is helping me and guiding me till this book and the simulator is finished. If you want to give credit thank her, if there is any critics or hate just go to me.

Another source of learning: I am reading a lot of C++ books, and asking a lot in StackOverFlow, from there I am able to create this book and the simulator. I modify the codes depending on what I want to achieve, so none of all the codes in this book are free of errors or pure mine, just like the books written all this time, the knowledge is gained from our predecessors / professors / mentors / authors of books we read. Thanks to all the author who wrote C++ books that I read and those who answer my questions in StackOverFlow forum. I am sorry if my questions in forum sometimes stupid, I am not a computer science student, and my way of learning is like this, not best at formal school.

I was not born as those who are lucky enough to get education from top institutes / born with silver spoon (from rich family) who able to give me opportunity to take courses in expensive institutions to learn english well (I learn english from game or listening to music), sometimes I could not get the knowledge while reading textbooks in english, so I need to re-read over and over again, thus while working with Nature in forest as grass cutter and forest cleaner in the morning, after I get home, I study little by little and implement what I have learned to create this, sometimes asking Nature in the forest what I don't understand, sometimes asking Freya above. Turns out, expensive education is only an illusion, if you want to learn, you will do it no matter what, no need to make student loan. You will still survive and live as Nature never abandon those who are good to them, like a family in Russka Roma from John Wick movie.

All files, from the shader, fonts, textures, images and source codes used in this books can be

found in my repository:

<https://github.com/glanzkaiser/DFSimulatorC>

(If you clone or download the repository you can follow the book easily, since I put the path to all the source codes based on this repository directory structure).

Chapter 2

Using C++ in GFreya OS

*"Almost everything is written in C++, CERN uses it, NASA uses it, unless your name makes CERN and NASA obsolete, then you should learn C++ to be a better scientist and engineer" - DS Glanzsche
"I like your quote, a great one." - Freya*

I. INTRODUCTION

C++ is a general-purpose programming language made by Bjarne Stroustrup. An expansion of C language, often called "C with Classes." Its design and the compiling process to native machine code, make it an excellent choice for tasks that require high performance. After reading "The C++ Programming Language 4th Edition" book, done with chapter 1, I am inspired and want to learn deeper about this language. In this chapter, there is only a minimal examples on how to be able to compile a C++(.cpp file) source code into something that can be compiled by the compiler (GCC compiler or Clang) then to run the binary executable result. The basic point of this book is to make it easy to understand how to create something you want with computer graphics with C++, maybe some can learn from this book, as the point of this book is to share story how to create Math Physics Simulator from zero not to peel the skins of C++ one by one.

This book is going to use GFreya OS. GFreya OS is a Linux From Scratch based Operating System, so it is Linux-based, I was following the System V LFS 11.0 book then to BLFS, GFreya OS is using Xfce as the desktop environment. We created our own OS so we can use it daily, we know it from core to the shell, so if something went wrong it is quick to fix like compiling and running C++ projects, FORTRAN codes or JULIA codes. If you are using Linux OS like Arch Linux, or Gentoo, or Debian, then you won't have difficulty to follow this book, since the shell commands and the OS logic will pretty much the same. GFreya OS don't use package manager, and it is better that way, since we can learn to build and install all kinds of packages, programs, or even games from scratch. More about GFreya OS and how we build GFreya OS from zero can be read from the related book.

II. HOW TO COMPILE C++ SOURCE CODE INTO EXECUTABLE BINARY

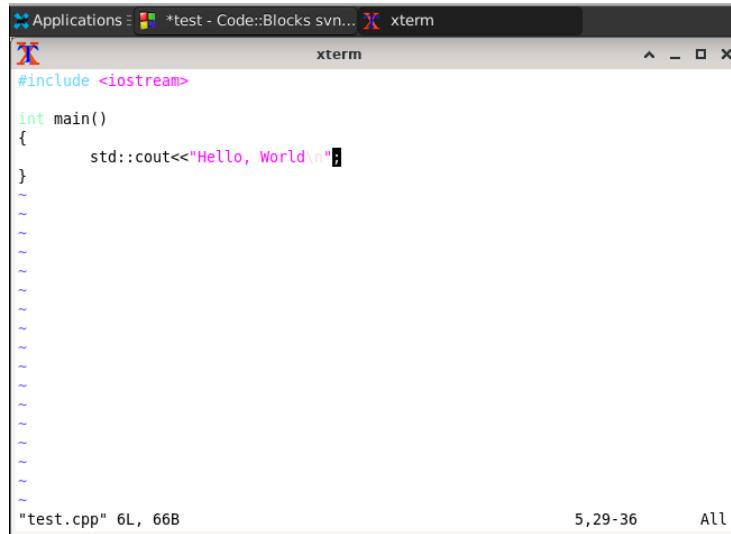
1. A lot of books recommend using IDE, but in my opinion you don't really need IDE for a simple C++ file and project, after getting used and if you are versatile even for complex project like create a game similar to Age of Empire, Grand Theft Auto or Suikoden Series,

you don't need IDE at all, it is my opinion even to build (compiling) Project Chrono, Box2D and Bullet, all can be done by only using Xterm (default terminal of GFreya OS 1.8) and Mousepad as the text editor.

I am using Mousepad to write C++ program, editing it, then to run it and compile: I just open the terminal at the directory and type the command needed to process the C++ source code into executable binary file. We never know, maybe in the future for bigger project I shall use IDE, in GFreya OS I already installed BlueFish IDE, just in case.

The steps of writing the syntaxes till obtaining the result or plot will be explained in details here. First, create a file by opening a terminal and type:

vim test.cpp



```
#include <iostream>

int main()
{
    std::cout<<"Hello, World\n";
}
```

"test.cpp" 6L, 66B 5,29-36 All

Figure 2.1: Create a new C++ file with **vim test.cpp** from the terminal

```
#include <iostream>

int main()
{
    std::cout<<"Hey Beautiful Goddess. \$\\backslash$ n";
}
```

C++ Code 1: *test.cpp "Hey Beautiful Goddess."*

When finish press Esc and type :wq and Enter.

2. You can compile it by using g++ compiler, type:

g++ test.cpp

then run the output, type:

./a.out

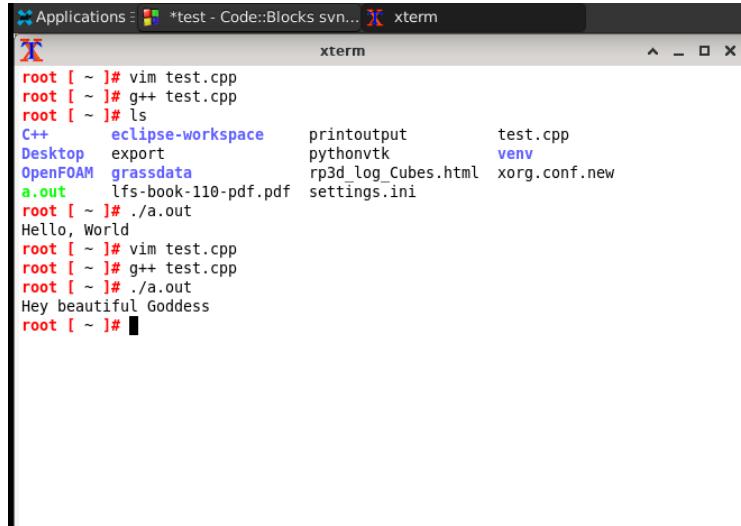


Figure 2.2: You can test and edit the c++ file from the terminal

3. C++ Example with GLEW and GLFW:

create a file by opening a terminal and type:

vim main.cpp

```

#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <iostream>

using namespace std;

void init(GLFWwindow* window){}

void display(GLFWwindow* window, double currentTime) {
    glClearColor(0.0, 1.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
}

int main(void)
{
    glfwInit(); //initialize GLFW and GLEW libraries
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE,
                   GLFW_OPENGL_CORE_PROFILE);

    GLFWwindow* window = glfwCreateWindow(600, 600, " Learn
                                            Open GL with GLFW", NULL, NULL);
    glfwMakeContextCurrent(window);
    if(glewInit() != GLEW_OK) {

```

```

        exit(EXIT_FAILURE);
    }
    glfwSwapInterval(1);

    init(window);

    while(!glfwWindowShouldClose(window)) {
        display(window, glfwGetTime());
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    glfwDestroyWindow(window);
    glfwTerminate();
    exit(EXIT_SUCCESS);
    //return 0;
}

```

C++ Code 2: main.cpp "Green Screen"

Then open the terminal at the current working directory, and type:

g++ main.cpp -o result -IGLEW -lglfw -IGL

then type:

./result

If you are wondering, the **-IGLEW**, **-lglfw**, **-IGL** mean that we are linking with dynamic library of GLEW, GLFW and GL.

- (a) libGL.so, located in **/usr/lib** then will be **-IGL**.
- (b) libGLEW.so, located in **/usr/lib** then will be **-IGLEW**.
- (c) libglfw.so, located in **/opt/hamzstlib** then will be **-lglfw**.

If you have no idea of GLEW and GLFW, you will learn more about GLEW, GLFW in chapter 4. Meanwhile in order to make them searchable when we are compiling, don't forget to adjust your **.bashrc**, add this lines **source /export** inside the if ..fi :

```

if [ -f "/etc/bashrc" ] ; then
    ...
    ...
    source ~/export
fi

```

You will need to create a file named **export** by typing at terminal in :

cd
vim export

```

export prefix="/usr"
export hamzstlib="/opt/hamzstlib"

# For library (.so, .a)

```

```
export LIBRARY_PATH="/usr/lib:$hamzstlib/lib"
```

The method above is used so when you type `#include <GL/glew.h>` and compiling it by calling the GLEW library that already installed in `/usr/lib` with `-lGLEW` it will find the right library of `libgGLEW.so` that contains a lot of functions needed to show the screen that we get from processing `main.cpp`.

Here are some explanation for the code above:

Initializes the GLFW library:

```
glfwInit();
```

To tells that the OpenGL is version 3.3 (my laptop graphics driver is compatible with this OpenGL version, yours might be different):

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
```

Initializes the GLEW library

```
if(glewInit() != GLEW_OK) {
    exit(EXIT_FAILURE);
}
```

Calls the function "init()"

```
init(window);
```

Calls the function "display()" repeatedly

```
while(!glfwWindowShouldClose(window)) {
    display(window, glfwGetTime());
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

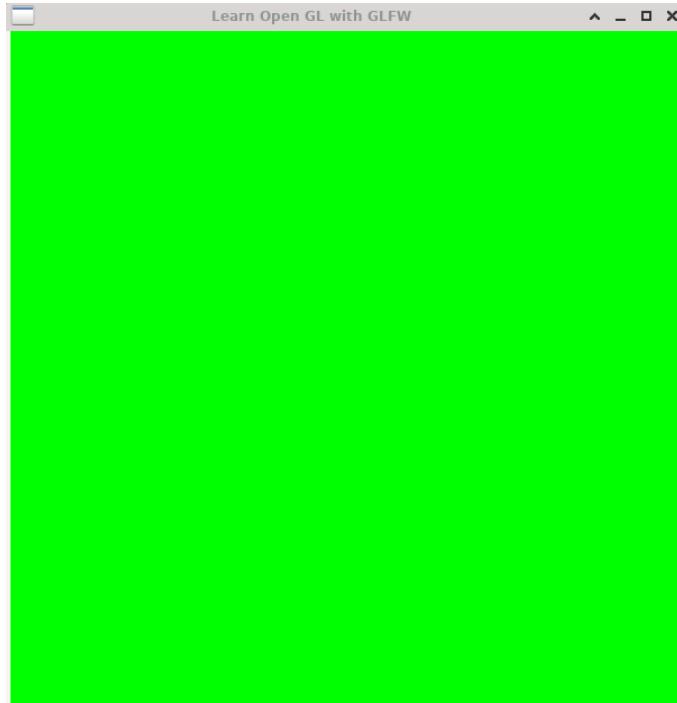


Figure 2.3: The main.cpp example above when compiled will show this green screen (DFSsimulatorC/Source Codes/C++/ch2-main-example2.cpp).

The two examples above are very simple example to compile C++ source code, more details can be obtain from various sources like StackOverFlow forum, C++ related books, or university courses. At least you will get the idea how in the future you can make hundreds or even tens of thousands line of codes into Physics animation, rocket landing computation, weather forecast, fluid simulation, and more.

4. C++ Example with SOIL, SFML, GLM, and GLEW:

This example will show how to use SOIL to upload image nad do some mathematical transformation (rotating the image) with GLM. In a new directory, create a file by opening a terminal and type:

vim main.cpp

```
// Link statically with GLEW
#define GLEW_STATIC

// Headers
#include <GL/glew.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <SOIL/SOIL.h>
#include <SFML/Window.hpp>
#include <chrono>
```

```

// Shader sources
const GLchar* vertexSource = R"glsl(
#version 330 core
in vec2 position;
in vec3 color;
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
uniform mat4 trans;
void main()
{
    Color = color;
    Texcoord = texcoord;
    gl_Position = trans * vec4(position, 0.0, 1.0);
}
)glsl";
const GLchar* fragmentSource = R"glsl(
#version 330 core
in vec3 Color;
in vec2 Texcoord;
out vec4 outColor;
uniform sampler2D texFreya;
uniform sampler2D texScrooge;
void main()
{
    outColor = mix(texture(texFreya, Texcoord), texture(
        texScrooge, Texcoord), 0.5);
}
)glsl";

int main()
{
    auto t_start = std::chrono::high_resolution_clock::now()
    ;

    sf::ContextSettings settings;
    settings.depthBits = 24;
    settings.stencilBits = 8;
    settings.majorVersion = 3;
    settings.minorVersion = 3;

    sf::Window window(sf::VideoMode(800, 600, 32), "OpenGL",
                     sf::Style::Titlebar | sf::Style::Close, settings);

    // Initialize GLEW
    glewExperimental = GL_TRUE;
    glewInit();

```

```
// Create Vertex Array Object
GLuint vao;
glGenVertexArrays(1, &vao);
 glBindVertexArray(vao);

// Create a Vertex Buffer Object and copy the vertex
// data to it
GLuint vbo;
 glGenBuffers(1, &vbo);

GLfloat vertices[] = {
    // Position Color Texcoords
    -0.5f, 0.5f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, // Top
    -left
    0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, // Top-
    right
    0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // //
    Bottom-right
    -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.0f // //
    Bottom-left
};

 glBindBuffer(GL_ARRAY_BUFFER, vbo);
 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
 , GL_STATIC_DRAW);

// Create an element array
GLuint ebo;
 glGenBuffers(1, &ebo);

GLuint elements[] = {
    0, 1, 2,
    2, 3, 0
};

 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
 glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(elements),
 elements, GL_STATIC_DRAW);

// Create and compile the vertex shader
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
 glShaderSource(vertexShader, 1, &vertexSource, NULL);
 glCompileShader(vertexShader);

// Create and compile the fragment shader
GLuint fragmentShader = glCreateShader(
    GL_FRAGMENT_SHADER);
```

```

glShaderSource(fragmentShader, 1, &fragmentSource, NULL)
;
glCompileShader(fragmentShader);

// Link the vertex and fragment shader into a shader
// program
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glBindFragDataLocation(shaderProgram, 0, "outColor");
glLinkProgram(shaderProgram);
glUseProgram(shaderProgram);

// Specify the layout of the vertex data
GLint posAttrib = glGetAttribLocation(shaderProgram, "position");
 glEnableVertexAttribArray(posAttrib);
 glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE,
 7 * sizeof(GLfloat), 0);

GLint colAttrib = glGetAttribLocation(shaderProgram, "color");
 glEnableVertexAttribArray(colAttrib);
 glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE,
 7 * sizeof(GLfloat), (void*)(2 * sizeof(GLfloat)));

GLint texAttrib = glGetAttribLocation(shaderProgram, "texcoord");
 glEnableVertexAttribArray(texAttrib);
 glVertexAttribPointer(texAttrib, 2, GL_FLOAT, GL_FALSE,
 7 * sizeof(GLfloat), (void*)(5 * sizeof(GLfloat)));

// Load textures
GLuint textures[2];
 glGenTextures(2, textures);

int width, height;
unsigned char* image;

glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, textures[0]);
image = SOIL_load_image("sample.png", &width, &height,
 0, SOIL_LOAD_RGB);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
  GL_RGB, GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
 glUniform1i(glGetUniformLocation(shaderProgram, "texFreya"), 0);

```

```

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, textures[1]);
image = SOIL_load_image("sample2.png", &width, &height,
    0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
    GL_RGB, GL_UNSIGNED_BYTE, image);
SOIL_free_image_data(image);
glUniform1i(glGetUniformLocation(shaderProgram, "texScrooge"), 1);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);

GLint uniTrans = glGetUniformLocation(shaderProgram, "trans");

bool running = true;
while (running)
{
    sf::Event windowEvent;
    while (window.pollEvent(windowEvent))
    {
        switch (windowEvent.type)
        {
            case sf::Event::Closed:
                running = false;
                break;
        }
    }

    // Clear the screen to black
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
}

```

```

glClear(GL_COLOR_BUFFER_BIT);

// Calculate transformation
auto t_now = std::chrono::high_resolution_clock::
    now();
float time = std::chrono::duration_cast<std::
    chrono::duration<float>>(t_now - t_start).-
    count();

glm::mat4 trans = glm::mat4(1.0f);
trans = glm::rotate(
trans,
time * glm::radians(180.0f),
glm::vec3(0.0f, 0.0f, 1.0f)
);
glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::
    value_ptr(trans));

// Draw a rectangle from the 2 triangles using 6
// indices
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT,
    0);

// Swap buffers
window.display();
}

glDeleteTextures(2, textures);

glDeleteProgram(shaderProgram);
glDeleteShader(fragmentShader);
glDeleteShader(vertexShader);

glDeleteBuffers(1, &ebo);
glDeleteBuffers(1, &vbo);

glDeleteVertexArrays(1, &vao);

window.close();

return 0;
}

```

C++ Code 3: *main.cpp "SOIL GLM Rotating Images"*

Then open the terminal at the current working directory, and type:

g++ main.cpp -o result -L/usr/lib -lSOIL -lGlew -lsfml-graphics -lsfml-window -lsfml-system -lGL

then type:

./result

If you are wondering, the **-L/usr/lib -lSOIL** mean that we are linking with static library of SOIL. If you have build SOIL and set the installation path at **/usr**, you will see the library named **libSOIL.a** inside **/usr/lib**.

Here are some explanations for the code above:

To upload image "sample.png" as the first array of texture:

```
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, textures[0]);
 image = SOIL_load_image("sample.png", &width, &height, 0,
    SOIL_LOAD_RGB);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB
    , GL_UNSIGNED_BYTE, image);
 SOIL_free_image_data(image);
 glUniform1i(glGetUniformLocation(shaderProgram, "texFreya"), 0)
;
```

Using GLM library to make the images uploaded rotating counterclockwise, the positive degree is by default will rotate counterclockwise:

```
glm::mat4 trans = glm::mat4(1.0f);
trans = glm::rotate(
trans,
time * glm::radians(180.0f),
glm::vec3(0.0f, 0.0f, 1.0f)
);
glUniformMatrix4fv(uniTrans, 1, GL_FALSE, glm::value_ptr(trans)
);
```

We declare two shader programs, first the hardcoded vertex shader in GLSL code inside C++ source code. Since the C++/OpenGL application must compile and link appropriate GLSL vertex and fragment shader programs, and then load them into the pipeline, all of the vertices pass through the vertex shader (the shader is executed once per vertex, for millions of vertices, it will be done in parallel):

```
const GLchar* vertexSource = R"glsl(
#version 330 core
in vec2 position;
in vec3 color;
in vec2 texcoord;
out vec3 Color;
out vec2 Texcoord;
uniform mat4 trans;
void main()
{
    Color = color;
```

```

    Texcoord = texcoord;
    gl_Position = trans * vec4(position, 0.0, 1.0);
}
)glsl";

```

The hardcoded fragment shader in GLSL code inside C++ source code to display the result with specified color:

```

const GLchar* fragmentSource = R"glsl(
#version 330 core
in vec3 Color;
in vec2 Texcoord;
out vec4 outColor;
uniform sampler2D texFreya;
uniform sampler2D texScrooge;
void main()
{
    outColor = mix(texture(texFreya, Texcoord), texture(
        texScrooge, Texcoord), 0.5);
}
)glsl";

```



Figure 2.4: The working directory shall contain the `main.cpp` file, and the two images we want to upload and use for the project can be stored here or other directory, just adjust the correct path to the images if you do so.



Figure 2.5: The main.cpp example above when compiled will show the rotating of two (DFSimulatorC/Source Codes/C++/ch2-main-example3.cpp).

5. C++ Example with SFML and Keyboard Event:

In this example, it will show that SFML alone can be used with keyboard pressed event.
In a new directory, create a file by opening a terminal and type:

vim main.cpp

```
#include "SFML/Graphics.hpp"

sf::Vector2f viewSize(1024, 768);
sf::VideoMode vm(viewSize.x, viewSize.y);
sf::RenderWindow window(vm, "Hello_SFML_Game!!!", sf::Style::Default);

sf::Vector2f playerPosition;
bool playerMovingRight = false;
bool playerMovingLeft = false;

sf::Texture skyTexture;
sf::Sprite skySprite;

sf::Texture bgTexture;
sf::Sprite bgSprite;

sf::Texture heroTexture;
sf::Sprite heroSprite;

void init() {

    skyTexture.loadFromFile("/root/SourceCodes/CPP/Assets/
        graphics/sky.png");
}
```

```
skySprite.setTexture(skyTexture);

bgTexture.loadFromFile("/root/SourceCodes/CPP/Assets/
    graphics/bg.png");
bgSprite.setTexture(bgTexture);

heroTexture.loadFromFile("/root/SourceCodes/CPP/Assets/
    graphics/Freya.png");
heroSprite.setTexture(heroTexture);
heroSprite.setPosition(sf::Vector2f(viewSize.x / 600,
    viewSize.y / 600));
heroSprite.setOrigin(heroTexture.getSize().x / 600,
    heroTexture.getSize().y / 600);

}

void updateInput() {

    sf::Event event;

    // while there are pending events...
    while (window.pollEvent(event)) {

        //printf("polling events\n");

        if (event.type == sf::Event::KeyPressed) {
            if (event.key.code == sf::Keyboard::Right)
            {
                playerMovingRight= true;
            }
            if (event.key.code == sf::Keyboard::Left) {
                playerMovingLeft= true;
            }
        }

        if (event.type == sf::Event::KeyReleased) {
            if (event.key.code == sf::Keyboard::Right)
            {
                playerMovingRight = false;
            }
            if (event.key.code == sf::Keyboard::Left) {
                playerMovingLeft = false;
            }
        }
        if (event.key.code == sf::Keyboard::Escape ||
            event.type == sf::Event::Closed)
            window.close();
    }
}
```

```

}

void update(float dt) {
    if (playerMovingLeft) {
        heroSprite.move(-150.0f * dt, 0);
    }
    if (playerMovingRight) {
        heroSprite.move(150.0f * dt, 0);
    }
}

void draw() {
    window.draw(skySprite);
    window.draw(bgSprite);
    window.draw(heroSprite);
}

int main() {
    sf::Clock clock;
    init();

    while (window.isOpen()) {
        updateInput();

        sf::Time dt = clock.restart();
        update(dt.asSeconds());

        window.clear(sf::Color::Red);
        draw();
        window.display();
    }
    return 0;
}

```

C++ Code 4: *main.cpp "SFML Keyboard Event"*

Then open the terminal at the current working directory, and type:
g++ main.cpp -o result -lsfml-graphics -lsfml-window -lsfml-system
then type:
./result

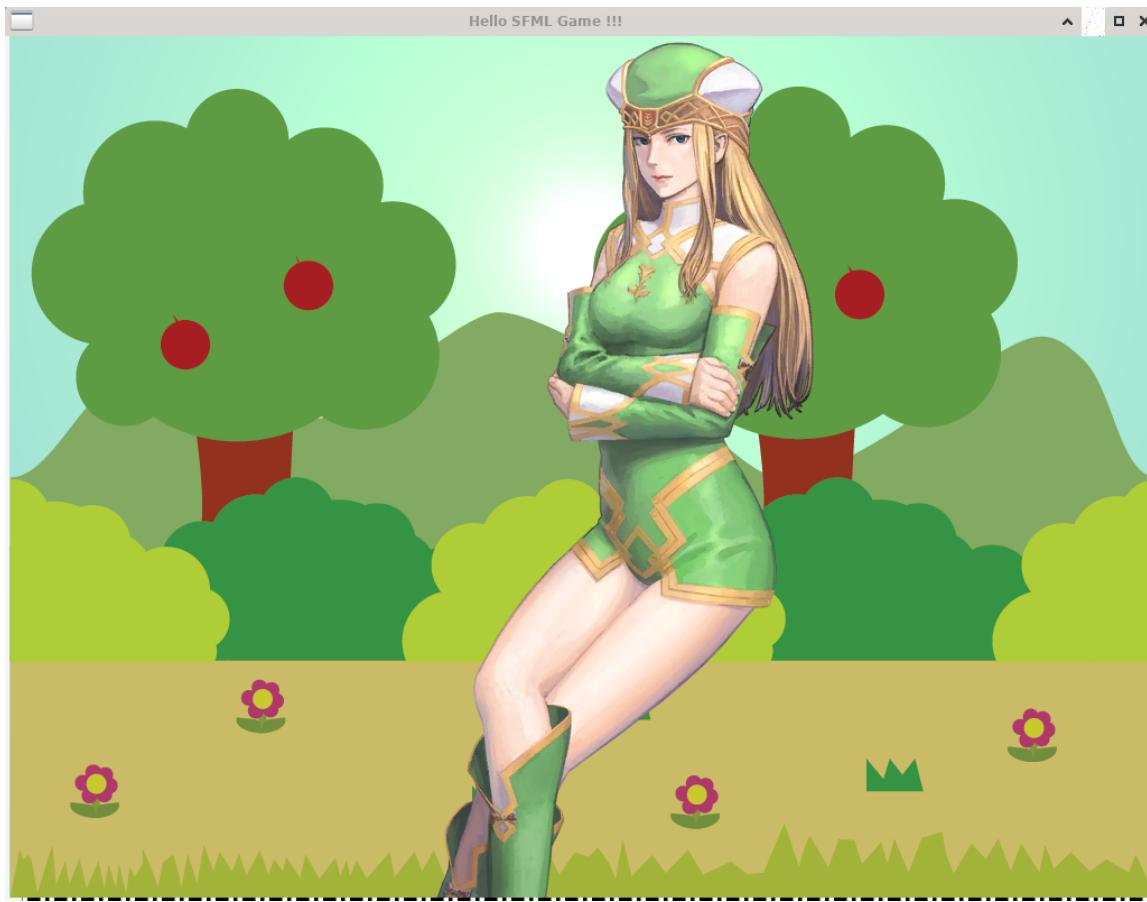


Figure 2.6: The project is showing how to work with Keyboard pressed event, if you press right the character will move to the right, if you press left the character will move to the left (`ch2-main-example4.cpp`).

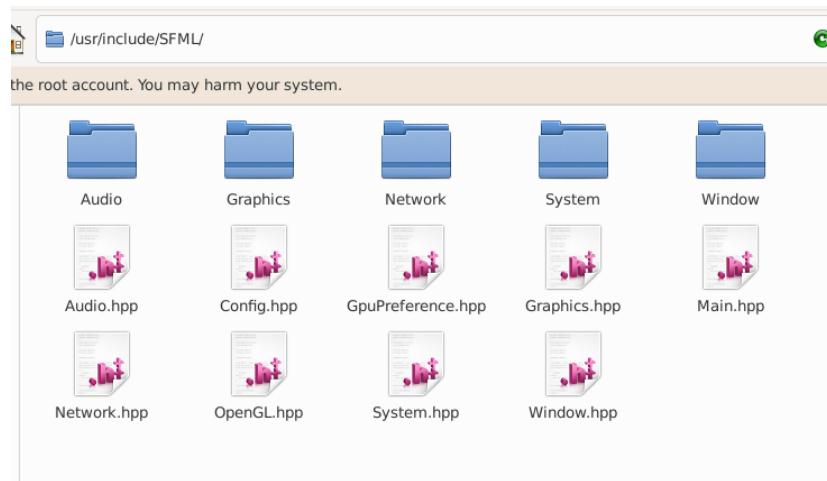


Figure 2.7: Adjust the path and environment variable for your include file, SFML' headers in GFreya OS are installed in `/usr/include/SFML`

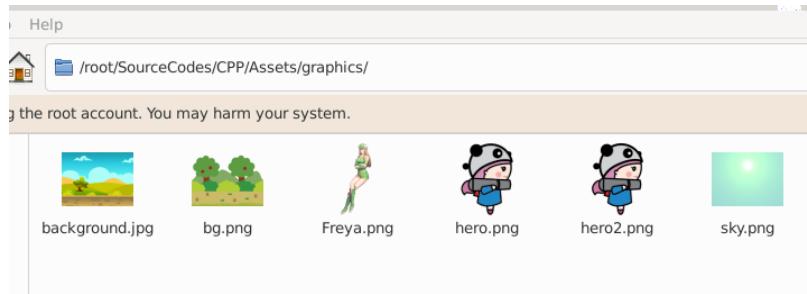


Figure 2.8: Adjust the path to the Assets (background image, hero image, texture, etc), in GFreya OS it is located in `/root/SourceCodes/CPP/Assets/`, then the images are saved under the `/root/SourceCodes/CPP/Assets/graphics/`

6. C++ Example with OpenGL, GLEW, GLFW and Keyboard Event:

In this example we are going to render meshes: triangle, cube, quad, and sphere. Then with keyboard key we can change the mesh that is being shown.

In a new directory, create a file by opening a terminal and type:

vim main.cpp

```
// GLEW needs to be included first
#include <GL/glew.h>

// GLFW is included next
#include <GLFW/glfw3.h>

void initGame();
void renderScene();
#include "ShaderLoader.h"
#include "Camera.h"
#include "LightRenderer.h"

GLuint textProgram;

Camera* camera;
LightRenderer* light;

void initGame() {
    // Enable the depth testing
    glEnable(GL_DEPTH_TEST);
    ShaderLoader shader;
    textProgram = shader.createProgram("/root/SourceCodes/
        CPP/Assets/Shaders/text.vs", "/root/SourceCodes/CPP/
        Assets/Shaders/text.fs");

    GLuint flatShaderProgram = shader.createProgram("/root/
        SourceCodes/CPP/Assets/Shaders/FlatModel.vs", "/root/
        SourceCodes/CPP/Assets/Shaders/FlatModel.fs");
}
```

```

camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f, glm::vec3(0.0f, 4.0f, 6.0f));

light = new LightRenderer(MeshType::kCube, camera); // Define Mesh type -> see Mesh.h
light->setProgram(flatShaderProgram);
light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
}

void renderScene(){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(1.0, 1.0, 0.0, 1.0);
    light->draw();
}

static void glfwError(int id, const char* description)
{
    std::cout << description << std::endl;
}

void updateKeyboard(GLFWwindow* window, int key, int scancode,
int action, int mods){

    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS) {
        glfwSetWindowShouldClose(window, true);
    }
    // Press Q to see Quad, T for Triangle, C for Cube and S for Sphere
    if (key == GLFW_KEY_C && action == GLFW_PRESS) {
        ShaderLoader shader;
        GLuint flatShaderProgram = shader.createProgram(
            "/root/SourceCodes/CPP/Assets/Shaders/FlatModel.vs", "/root/SourceCodes/CPP/Assets/Shaders/FlatModel.fs");
        camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f, glm::vec3(0.0f, 4.0f, 6.0f));
        light = new LightRenderer(MeshType::kCube, camera);
        light->setProgram(flatShaderProgram);
        light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
    }
    if (key == GLFW_KEY_T && action == GLFW_PRESS) {
        ShaderLoader shader;
        GLuint flatShaderProgram = shader.createProgram(
            "/root/SourceCodes/CPP/Assets/Shaders/FlatModel.vs", "/root/SourceCodes/CPP/Assets/Shaders/FlatModel.fs");
    }
}

```

```

        camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f
            , glm::vec3(0.0f, 4.0f, 6.0f));
        light = new LightRenderer(MeshType::kTriangle,
            camera);
        light->setProgram(flatShaderProgram);
        light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
    }
    if (key == GLFW_KEY_Q && action == GLFW_PRESS) {
        ShaderLoader shader;
        GLuint flatShaderProgram = shader.createProgram(
            "/root/SourceCodes/CPP/Assets/Shaders/
            FlatModel.vs", "/root/SourceCodes/CPP/Assets/
            Shaders/FlatModel.fs");
        camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f
            , glm::vec3(0.0f, 4.0f, 6.0f));
        light = new LightRenderer(MeshType::kQuad, camera
            );
        light->setProgram(flatShaderProgram);
        light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
    }
    if (key == GLFW_KEY_S && action == GLFW_PRESS) {
        ShaderLoader shader;
        GLuint flatShaderProgram = shader.createProgram(
            "/root/SourceCodes/CPP/Assets/Shaders/
            FlatModel.vs", "/root/SourceCodes/CPP/Assets/
            Shaders/FlatModel.fs");
        camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f
            , glm::vec3(0.0f, 4.0f, 6.0f));
        light = new LightRenderer(MeshType::kSphere,
            camera);
        light->setProgram(flatShaderProgram);
        light->setPosition(glm::vec3(0.0f, 0.0f, 0.0f));
    }
}

int main(int argc, char **argv)
{
    glfwSetErrorCallback(&glfwError);
    glfwInit();
    GLFWwindow* window = glfwCreateWindow(800, 600, "Mesh_
        in_OpenGL_", NULL, NULL);
    glfwMakeContextCurrent(window);
    glfwSetKeyCallback(window, updateKeyboard);
    glewInit();
    initGame();
    while (!glfwWindowShouldClose(window)){
        renderScene();
        glfwSwapBuffers(window);
}

```

```

        glfwPollEvents();
    }
    glfwTerminate();

    delete camera;
    delete light;
    return 0;
}

```

C++ Code 5: *main.cpp "Render Mesh with Lighting"*

There are other file as well, this example contains some C++ source codes and include files as well, create all this one by one:

```

#include "Camera.h"

Camera::Camera(GLfloat FOV, GLfloat width, GLfloat height,
               GLfloat nearPlane, GLfloat farPlane, glm::vec3 camPos){

    cameraPos = camPos;
    glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, 0.0f);
    glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

    viewMatrix = glm::lookAt(cameraPos, cameraFront,
                           cameraUp);
    projectionMatrix = glm::perspective(FOV, width /height,
                                         nearPlane, farPlane);
}

Camera::~Camera(){

glm::mat4 Camera::getViewMatrix() {
    return viewMatrix;
}
glm::mat4 Camera::getProjectionMatrix() {
    return projectionMatrix;
}
glm::vec3 Camera::getCameraPosition() {
    return cameraPos;
}

```

C++ Code 6: *Camera.cpp "Render Mesh with Lighting"*

```

#pragma once
#include <GL/glew.h>
#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"

class Camera

```

```
{
public:
    Camera(GLfloat FOV, GLfloat width, GLfloat height,
           GLfloat nearPlane, GLfloat farPlane, glm::vec3
           camPos);
    ~Camera();

    glm::mat4 getViewMatrix();
    glm::mat4 getProjectionMatrix();
    glm::vec3 getCameraPosition();

private:

    glm::mat4 viewMatrix;
    glm::mat4 projectionMatrix;
    glm::vec3 cameraPos;

};
```

C++ Code 7: Camera.h "Render Mesh with Lighting"

```
cmake_minimum_required(VERSION 3.17)

project(result LANGUAGES CXX)

add_executable(result main.cpp Mesh.cpp LightRenderer.cpp
              ShaderLoader.cpp Camera.cpp)
target_link_libraries(result GLEW glfw GL)
target_include_directories(result PRIVATE /usr/include)
```

C++ Code 8: CMakeLists.txt "Render Mesh with Lighting"

```
#include "LightRenderer.h"

LightRenderer::LightRenderer(MeshType meshType, Camera* camera)
{
    this->camera = camera;

    switch (meshType) {

        case kTriangle: Mesh::setTriData(vertices,
                                           indices); break;
        case kQuad:     Mesh::setQuadData(vertices, indices);
                       break;
        case kCube:     Mesh::setCubeData(vertices, indices);
                       break;
        case kSphere:   Mesh::setSphereData(vertices,
                                             indices); break;
    }
}
```

```

    }

    glGenVertexArrays(1, &vao);
    glBindVertexArray(vao);

    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex) * vertices.
        size(), &vertices[0], GL_STATIC_DRAW);

    //Attributes
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(
        Vertex), (GLvoid*)0);

    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(
        Vertex), (void*)(offsetof(Vertex, Vertex::color)));

    glGenBuffers(1, &ebo);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint) * 
        indices.size(), &indices[0], GL_STATIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);

}

void LightRenderer::draw() {
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(glm::mat4(1.0), position);

    glUseProgram(this->program);

    GLint modelLoc = glGetUniformLocation(program, "model");
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(
        model));

    glm::mat4 view = camera->getViewMatrix();
    GLint vLoc = glGetUniformLocation(program, "view");
    glUniformMatrix4fv(vLoc, 1, GL_FALSE, glm::value_ptr(
        view));

    glm::mat4 proj = camera->getProjectionMatrix();
    GLint pLoc = glGetUniformLocation(program, "projection")
        ;
    glUniformMatrix4fv(pLoc, 1, GL_FALSE, glm::value_ptr(

```

```

        proj));

glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, indices.size(),
               GL_UNSIGNED_INT, 0);

glBindVertexArray(0);
glUseProgram(0);
}

LightRenderer::~LightRenderer() {

}

void LightRenderer::setPosition(glm::vec3 _position) {
    position = _position;
}

void LightRenderer::setColor(glm::vec3 _color) {
    this->color = _color;
}

void LightRenderer::setProgram(GLuint _program) {
    this->program = _program;
}
//getters
glm::vec3 LightRenderer::getPosition() {
    return position;
}

glm::vec3 LightRenderer::getColor() {
    return color;
}

```

C++ Code 9: *LightRenderer.cpp "Render Mesh with Lighting"*

```

#pragma once
#include <GL/glew.h>

#include "glm/glm.hpp"
#include "glm/gtc/type_ptr.hpp"

#include "Mesh.h"
#include "ShaderLoader.h"
#include "Camera.h"

class LightRenderer
{

```

```

public:
    LightRenderer(MeshType meshType, Camera* camera);
    ~LightRenderer();

    void draw();

    void setPosition(glm::vec3 _position);
    void setColor(glm::vec3 _color);
    void setProgram(GLuint _program);

    glm::vec3 getPosition();
    glm::vec3 getColor();

private:
    Camera* camera;

    std::vector<Vertex> vertices;
    std::vector<GLuint> indices;

    GLuint vbo, ebo, vao, program;

    glm::vec3 position, color;

};

```

C++ Code 10: *LightRenderer.h "Render Mesh with Lighting"*

```

#include "Mesh.h"

void Mesh::setTriData(std::vector<Vertex>& vertices, std::vector<uint32_t>& indices) {

    std::vector<Vertex> _vertices = {
        { { 0.0f, -1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 1.0f, 0.0f, 0.0 },{ 0.0, 1.0 } },
        { { 1.0f, 1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0f, 1.0f, 0.0 },{ 0.0, 0.0 } },
        { { -1.0f, 1.0f, 0.0f },{ 0.0f, 0.0f, 1.0 },{ 0.0f, 0.0f, 1.0 },{ 1.0, 0.0 } },
    };

    std::vector<uint32_t> _indices = {
        0, 1, 2,
    };

    vertices.clear(); indices.clear();
    vertices = _vertices;
    indices = _indices;
}

```

```

    }

void Mesh::setQuadData(std::vector<Vertex>& vertices, std::vector<uint32_t>& indices) {
    std::vector<Vertex> _vertices = {
        { { -1.0f, -1.0f, 0.0f }, { 0.0f, 0.0f, 1.0 } },
        { { 1.0f, 0.0f, 0.0 }, { 0.0, 1.0 } },
        { { -1.0f, 1.0f, 0.0f }, { 0.0f, 0.0f, 1.0 } },
        { { 0.0f, 1.0f, 0.0 }, { 0.0, 0.0 } },
        { { 1.0f, 1.0f, 0.0f }, { 0.0f, 0.0f, 1.0 } },
        { { 0.0f, 0.0f, 1.0 }, { 1.0, 0.0 } },
        { { 1.0f, -1.0f, 0.0f }, { 0.0f, 0.0f, 1.0 } },
        { { 0.0f, 0.0f, 1.0 }, { 1.0, 1.0 } }
    };
    std::vector<uint32_t> _indices = {
        0, 1, 2,
        0, 2, 3
    };
    vertices.clear(); indices.clear();
    vertices = _vertices;
    indices = _indices;

}

void Mesh::setCubeData(std::vector<Vertex>& vertices, std::vector<uint32_t>& indices) {
    std::vector<Vertex> _vertices = {
        //front
        { { -1.0f, -1.0f, 1.0f }, { 0.0f, 0.0f, 1.0 } },
        { { 1.0f, 0.0f, 0.0 }, { 0.0, 1.0 } },
        { { -1.0f, 1.0f, 1.0f }, { 0.0f, 0.0f, 1.0 } },
        { { 0.0f, 1.0f, 0.0 }, { 0.0, 0.0 } },
        { { 1.0f, 1.0f, 1.0f }, { 0.0f, 0.0f, 1.0 } },
        { { 0.0f, 0.0f, 1.0 }, { 1.0, 0.0 } },
        { { 1.0f, -1.0f, 1.0f }, { 0.0f, 0.0f, 1.0 } },
        { { 0.0f, 0.0f, 1.0 }, { 1.0, 1.0 } },
        // back
        { { 1.0, -1.0, -1.0 }, { 0.0f, 0.0f, -1.0 } },
        { { 1.0f, 0.0f, 1.0 }, { 0.0, 1.0 } }, //4
        { { 1.0f, 1.0, -1.0 }, { 0.0f, 0.0f, -1.0 } },
        { { 0.0f, 1.0f, 1.0 }, { 0.0, 0.0 } }, //5
        { { -1.0, 1.0, -1.0 }, { 0.0f, 0.0f, -1.0 } },
        { { 0.0f, 1.0f, 1.0 }, { 1.0, 0.0 } }, //6
        { { -1.0, -1.0, -1.0 }, { 0.0f, 0.0f, -1.0 } },
        { { 1.0f, 0.0f, 1.0 }, { 1.0, 1.0 } }, //7
        //left
        { { -1.0, -1.0, -1.0 }, { -1.0f, 0.0f, 0.0 } },
        { { 0.0f, 0.0f, 1.0 }, { 0.0, 1.0 } }, //8
    };
    vertices.clear(); indices.clear();
    vertices = _vertices;
    indices = _indices;
}

```

```

{ { -1.0f, 1.0, -1.0 },{ -1.0f, 0.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 0.0, 0.0 } }, //9
{ { -1.0, 1.0, 1.0 },{ -1.0f, 0.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 1.0, 0.0 } }, //10
{ { -1.0, -1.0, 1.0 },{ -1.0f, 0.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 1.0, 1.0 } }, //11
//right
{ { 1.0, -1.0, 1.0 },{ 1.0f, 0.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 0.0, 1.0 } }, // 12
{ { 1.0f, 1.0, 1.0 },{ 1.0f, 0.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 0.0, 0.0 } }, //13
{ { 1.0, 1.0, -1.0 },{ 1.0f, 0.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 1.0, 0.0 } }, //14
{ { 1.0, -1.0, -1.0 },{ 1.0f, 0.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 1.0, 1.0 } }, //15
//top
{ { -1.0f, 1.0f, 1.0f },{ 0.0f, 1.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 0.0, 1.0 } }, //16
{ { -1.0f, 1.0f, -1.0f },{ 0.0f, 1.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 0.0, 0.0 } }, //17
{ { 1.0f, 1.0f, -1.0f },{ 0.0f, 1.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 1.0, 0.0 } }, //18
{ { 1.0f, 1.0f, 1.0f },{ 0.0f, 1.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 1.0, 1.0 } }, //19
//bottom
{ { -1.0f, -1.0, -1.0 },{ 0.0f, -1.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 0.0, 1.0 } }, //20
{ { -1.0, -1.0, 1.0 },{ 0.0f, -1.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 0.0, 0.0 } }, //21
{ { 1.0, -1.0, 1.0 },{ 0.0f, -1.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 1.0, 0.0 } }, //22
{ { 1.0, -1.0, -1.0 },{ 0.0f, -1.0f, 0.0 },{ 0.0f, 0.0f, 1.0 },{ 1.0, 1.0 } }, //23
};

std::vector<uint32_t> _indices = {
    0, 1, 2,
    2, 3, 0,
    4, 5, 6,
    4, 6, 7,
    8, 9, 10,
    8, 10, 11,
    12, 13, 14,
    12, 14, 15,
    16, 17, 18,
    16, 18, 19,
    20, 21, 22,
    20, 22, 23
};

```

```

        vertices.clear(); indices.clear();
        vertices = _vertices;
        indices = _indices;
    }

void Mesh::setSphereData(std::vector<Vertex>& vertices, std::vector<uint32_t>& indices) {
    std::vector<Vertex> _vertices;
    std::vector<uint32_t> _indices;

    float latitudeBands = 20.0f;
    float longitudeBands = 20.0f;
    float radius = 1.0f;

    for (float latNumber = 0; latNumber <= latitudeBands;
         latNumber++) {
        float theta = latNumber * 3.14 / latitudeBands;
        float sinTheta = sin(theta);
        float cosTheta = cos(theta);

        for (float longNumber = 0; longNumber <=
             longitudeBands; longNumber++) {

            float phi = longNumber * 2 * 3.147 /
                       longitudeBands;
            float sinPhi = sin(phi);
            float cosPhi = cos(phi);

            Vertex vs;

            vs.texCoords.x = (longNumber /
                               longitudeBands); // u
            vs.texCoords.y = (latNumber / latitudeBands
                               ); // v

            vs.normal.x = cosPhi * sinTheta; // normal
                           x
            vs.normal.y = cosTheta; // normal y
            vs.normal.z = sinPhi * sinTheta; // normal
                           z

            vs.color.r = vs.normal.x;
            vs.color.g = vs.normal.y;
            vs.color.b = vs.normal.z;

            vs.pos.x = radius * vs.normal.x; // x
            vs.pos.y = radius * vs.normal.y; // y
        }
    }
}

```

```

        vs.pos.z = radius * vs.normal.z; // z

        _vertices.push_back(vs);
    }
}

for (uint32_t latNumber = 0; latNumber < latitudeBands;
latNumber++) {
    for (uint32_t longNumber = 0; longNumber <
longitudeBands; longNumber++) {
        uint32_t first = (latNumber * (
            longitudeBands + 1)) + longNumber;
        uint32_t second = first + longitudeBands +
1;

        _indices.push_back(first);
        _indices.push_back(second);
        _indices.push_back(first + 1);

        _indices.push_back(second);
        _indices.push_back(second + 1);
        _indices.push_back(first + 1);
    }
}
vertices.clear(); indices.clear();
vertices = _vertices;
indices = _indices;
}

```

C++ Code 11: *Mesh.cpp "Render Mesh with Lighting"*

```

#include <vector>
#include "glm/glm.hpp"

enum MeshType {
    kTriangle = 0,
    kQuad = 1,
    kCube = 2,
    kSphere = 3
};

struct Vertex {
    glm::vec3 pos;
    glm::vec3 normal;
    glm::vec3 color;
    glm::vec2 texCoords;
};

```

```

class Mesh {

public:
    static void setTriData(std::vector<Vertex>& vertices,
                          std::vector<uint32_t>& indices);
    static void setQuadData(std::vector<Vertex>& vertices,
                           std::vector<uint32_t>& indices);
    static void setCubeData(std::vector<Vertex>& vertices,
                           std::vector<uint32_t>& indices);
    static void setSphereData(std::vector<Vertex>& vertices,
                             std::vector<uint32_t>& indices);

};

```

C++ Code 12: *Mesh.h "Render Mesh with Lighting"*

```

#include "ShaderLoader.h"

#include<iostream>
#include<fstream>
#include<vector>

std::string ShaderLoader::readShader(const char *filename)
{
    std::string shaderCode;
    std::ifstream file(filename, std::ios::in);

    if (!file.good()){
        std::cout << "Can't read file" << filename <<
                     std::endl;
        std::terminate();
    }

    file.seekg(0, std::ios::end);
    shaderCode.resize((unsigned int)file.tellg());
    file.seekg(0, std::ios::beg);
    file.read(&shaderCode[0], shaderCode.size());
    file.close();
    return shaderCode;
}

GLuint ShaderLoader::createShader(GLenum shaderType, std::
    string source, const char* shaderName)
{
    int compile_result = 0;

    GLuint shader = glCreateShader(shaderType);
    const char *shader_code_ptr = source.c_str();
    const int shader_code_size = source.size();

```

```

glShaderSource(shader, 1, &shader_code_ptr, &
               shader_code_size);
glCompileShader(shader);
glGetShaderiv(shader, GL_COMPILE_STATUS, &compile_result
              );
}

//check for errors

if (compile_result == GL_FALSE)
{

    int info_log_length = 0;
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &
                  info_log_length);

    std::vector<char> shader_log(info_log_length);

    glGetShaderInfoLog(shader, info_log_length, NULL,
                       &shader_log[0]);
    std::cout << "ERROR_compiling_shader:" <<
        shaderName << std::endl << &shader_log[0] <<
        std::endl;
    return 0;
}
return shader;
}

GLuint ShaderLoader::createProgram(const char*
vertexShaderFilename, const char* fragmentShaderFilename){

    //read the shader files and save the code
    std::string vertex_shader_code = readShader(
        vertexShaderFilename);
    std::string fragment_shader_code = readShader(
        fragmentShaderFilename);

    GLuint vertex_shader = createShader(GL_VERTEX_SHADER,
                                         vertex_shader_code, "vertex_shader");
    GLuint fragment_shader = createShader(GL_FRAGMENT_SHADER
                                         , fragment_shader_code, "fragment_shader");

    int link_result = 0;
    //create the program handle, attach the shaders and
    link it
    GLuint program = glCreateProgram();
    glAttachShader(program, vertex_shader);
    glAttachShader(program, fragment_shader);
}

```

```

glLinkProgram(program);
glGetProgramiv(program, GL_LINK_STATUS, &link_result);
//check for link errors
if (link_result == GL_FALSE) {

    int info_log_length = 0;
    glGetProgramiv(program, GL_INFO_LOG_LENGTH, &
                    info_log_length);

    std::vector<char> program_log(info_log_length);

    glGetProgramInfoLog(program, info_log_length,
                         NULL, &program_log[0]);
    std::cout << "ShaderLoader::LINK_ERROR" << std
        ::endl << &program_log[0] << std::endl;

    return 0;
}
return program;
}

```

C++ Code 13: *ShaderLoader.cpp* "Render Mesh with Lighting"

```

#pragma once

#include <GL/glew.h>
#include <iostream>

class ShaderLoader
{
public:
    GLuint createProgram(const char* vertexShaderFilename,
                         const char* fragmentShaderFilename);

private:
    std::string readShader(const char *filename);
    GLuint createShader(GLenum shaderType, std::string
                        source, const char* shaderName);
};

```

C++ Code 14: *ShaderLoader.h* "Render Mesh with Lighting"

Check again and make sure you have all this inside one working directory:

- Camera.cpp
- Camera.h
- CMakeLists.txt (Optional, for easy compiling)
- LightRenderer.cpp

- LightRenderer.h
- main.cpp
- Mesh.cpp
- Mesh.h
- ShaderLoader.cpp
- ShaderLoader.h

You may also see the repository where I uploaded the codes of this book:

<https://github.com/glanzkaiser/DFSimulatorC>

(all files for this example are in the folder: **DFSimulatorC/Source Codes/C++/ch2-example5-Render Mesh with Lighting**)

After finish, open the terminal at the current working directory, and type:

g++ *.cpp -o result -IGLEW -lglfw -IGL

then type:

./result

Another way is to use CMakeLists.txt, this way at the current working directory open the terminal and type:

```
mkdir build
cd build
cmake ..
make
./result
```

You may press c to see cube, t to see triangle, q to see quad, and s to see sphere, press Esc to close the window.

```

File Edit Search View Document Help
Warning: you are using the root account. You may harm your system.

#pragma once

#include <GL/glew.h>
#include "glm/glm.hpp"
#include "glm/gtc/type_ptr.hpp"

#include "Mesh.h"
#include "ShaderLoader.h";
#include "Camera.h"

class LightRenderer
{
public:
    LightRenderer(MeshType meshType,
                  ~LightRenderer());
    void draw();
    void setPosition(glm::vec3 _position);
    void setColor(glm::vec3 _color);
    void setProgram(GLuint _program);
    glm::vec3 getPosition();
    glm::vec3 getColor();

private:
    Camera* camera;
    std::vector<Vertex> vertices;
    std::vector<GLuint> indices;
};


```

The file explorer window shows the contents of the /usr/include/glm directory:

- Places:** root, Desktop
- Devices:** File System, Filesystem root, dev, pts, proc, run, sys
- glm folder contents:**
 - detail, ext, gtc, gtx, simd
 - CMakeLists.txt, common.hpp, exponential.hpp, ext.hpp, fwd.hpp
 - geometric.hpp, glm.hpp, integer.hpp, mat2x2.hpp, mat2x3.hpp
 - mat2x4.hpp, mat3x2.hpp, mat3x3.hpp, mat3x4.hpp, mat4x2.hpp

5 folders, 24 files: 143.8 KIB (147,222 bytes), Free space: 337.0 GB

Figure 2.9: To comprehend why we use `#include "glm/glm.hpp"`, because GLM's headers in GFreya OS are installed in `/usr/include/glm`, the environment variable to look for include folder is set at `/usr/include`, thus we need to add the parent directory as well `glm/...`

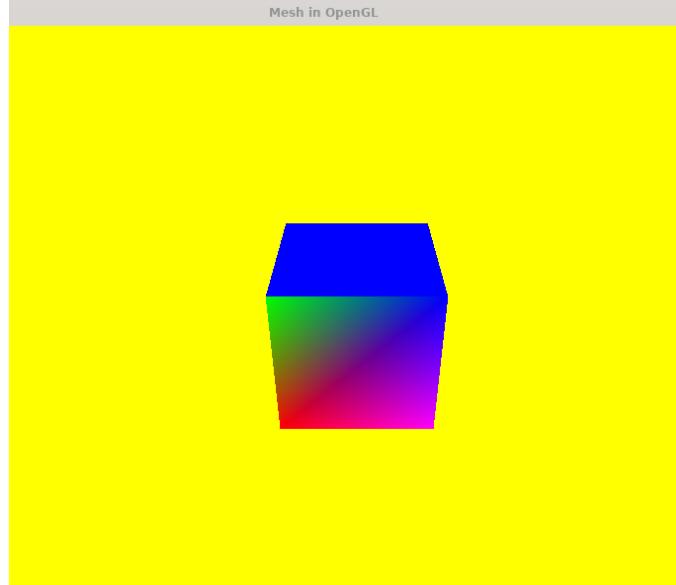


Figure 2.10: The running OpenGL application, it is very beautiful with lighting and keyboard press events.

7. C++ Example with OpenGL, GLAD, GLFW, SOIL, Mouse and Keyboard Event:

In this example we are going to create multiple cubes [12], I get it from that book, if you want to learn OpenGL rigorously follow this book and the tutorial on the website, it is a

great explanation there. On this example to be honest, **stb_image** is quite a pain, I can't load texture with that, thus I try to use SOIL and it works, so here goes (aren't we all SOIL lovers?):

In a new directory, create a file by opening a terminal and type:

vim main.cpp

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <SOIL/SOIL.h>

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include <learnopengl/filesystem.h>
#include <learnopengl/shader_m.h>

#include <iostream>

void framebuffer_size_callback(GLFWwindow* window, int width,
                               int height);
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);
void processInput(GLFWwindow *window);

// settings
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

// camera
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

bool firstMouse = true;
float yaw = -90.0f; // yaw is initialized to -90.0 degrees
                     // since a yaw of 0.0 results in a direction vector pointing
                     // to the right so we initially rotate a bit to the left.
float pitch = 0.0f;
float lastX = 800.0f / 2.0;
float lastY = 600.0 / 2.0;
float fov = 45.0f;

// timing
float deltaTime = 0.0f; // time between current frame and last
frame
float lastFrame = 0.0f;
```

```
int main()
{
    // glfw: initialize and configure

    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE,
                   GLFW_OPENGL_CORE_PROFILE);

#ifndef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif

    // glfw window creation
    // -----
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH,
                                         SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" <<
                     std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window,
                                  framebuffer_size_callback);
    glfwSetCursorPosCallback(window, mouse_callback);
    glfwSetScrollCallback(window, scroll_callback);

    // tell GLFW to capture our mouse
    glfwSetInputMode(window, GLFW_CURSOR,
                     GLFW_CURSOR_DISABLED);

    // glad: load all OpenGL function pointers

    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }

    // configure global opengl state
    // -----
    glEnable(GL_DEPTH_TEST);
```

```
// build and compile our shader zprogram

Shader ourShader("camera.vs", "camera.fs");

// set up vertex data (and buffer(s)) and configure
// vertex attributes
float vertices[] = {
    -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,
    0.5f, -0.5f, -0.5f, 1.0f, 0.0f,
    0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
    0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
    -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,
    -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,

    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
    0.5f, -0.5f, 0.5f, 1.0f, 0.0f,
    0.5f, 0.5f, 0.5f, 1.0f, 1.0f,
    0.5f, 0.5f, 0.5f, 1.0f, 1.0f,
    -0.5f, 0.5f, 0.5f, 0.0f, 1.0f,
    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,

    -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
    -0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
    -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
    -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,

    0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
    0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
    0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
    0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
    0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
    0.5f, 0.5f, 0.5f, 1.0f, 0.0f,

    -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,
    0.5f, -0.5f, -0.5f, 1.0f, 1.0f,
    0.5f, -0.5f, 0.5f, 1.0f, 0.0f,
    0.5f, -0.5f, 0.5f, 1.0f, 0.0f,
    -0.5f, -0.5f, 0.5f, 0.0f, 0.0f,
    -0.5f, -0.5f, -0.5f, 0.0f, 1.0f,

    -0.5f, 0.5f, -0.5f, 0.0f, 1.0f,
    0.5f, 0.5f, -0.5f, 1.0f, 1.0f,
    0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
    0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
    -0.5f, 0.5f, 0.5f, 0.0f, 0.0f,
    -0.5f, 0.5f, 0.5f, 1.0f, 0.0f,
```

```

        -0.5f, 0.5f, -0.5f, 0.0f, 1.0f
    };
    // world space positions of our cubes
    glm::vec3 cubePositions[] = {
        glm::vec3( 0.0f, 0.0f, 0.0f),
        glm::vec3( 2.0f, 5.0f, -15.0f),
        glm::vec3(-1.5f, -2.2f, -2.5f),
        glm::vec3(-3.8f, -2.0f, -12.3f),
        glm::vec3( 2.4f, -0.4f, -3.5f),
        glm::vec3(-1.7f, 3.0f, -7.5f),
        glm::vec3( 1.3f, -2.0f, -2.5f),
        glm::vec3( 1.5f, 2.0f, -2.5f),
        glm::vec3( 1.5f, 0.2f, -1.5f),
        glm::vec3(-1.3f, 1.0f, -1.5f)
    };
    unsigned int VBO, VAO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);

    glBindVertexArray(VAO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices
                 , GL_STATIC_DRAW);

    // position attribute
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 *
                         sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    // texture coord attribute
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 *
                         sizeof(float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);

    // load and create a texture
    unsigned int texture1, texture2;
    // texture 1
    glGenTextures(1, &texture1);
    glBindTexture(GL_TEXTURE_2D, texture1);
    // set the texture wrapping parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                    GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                    GL_REPEAT);
    // set texture filtering parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR);

```

```

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_LINEAR);
// load image, create texture and generate mipmaps
int width, height, nrChannels;
unsigned char* image = SOIL_load_image("/root/
    SourceCodes/CPP/images/sample.png", &width, &height,
    0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
             GL_RGB, GL_UNSIGNED_BYTE, image);

if (image)
{
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width,
                 height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
    glGenerateMipmap(GL_TEXTURE_2D);
}
else
{
    std::cout << "Failed to load texture" << std::
        endl;
}
SOIL_free_image_data(image);
// texture 2
 glGenTextures(1, &texture2);
 glBindTexture(GL_TEXTURE_2D, texture2);
// set the texture wrapping parameters
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                 GL_REPEAT);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                 GL_REPEAT);
// set texture filtering parameters
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                 GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                 GL_LINEAR);
// load image, create texture and generate mipmaps
unsigned char* image2 = SOIL_load_image("/root/
    SourceCodes/CPP/images/sample.png", &width, &height,
    0, SOIL_LOAD_RGB);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
             GL_RGB, GL_UNSIGNED_BYTE, image);

if (image2)
{
    // note that the awesomeface.png has transparency
    // and thus an alpha channel, so make sure to
    // tell OpenGL the data type is of GL_RGBA
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width,

```

```
        height, 0, GL_RGBA, GL_UNSIGNED_BYTE, image2)
    ;
    glGenerateMipmap(GL_TEXTURE_2D);
}
else
{
    std::cout << "Failed to load texture" << std::
    endl;
}
SOIL_free_image_data(image2);

// tell opengl for each sampler to which texture unit it
// belongs to (only has to be done once)
ourShader.use();
ourShader.setInt("texture1", 0);
ourShader.setInt("texture2", 1);

// render loop
// -----
while (!glfwWindowShouldClose(window))
{
    // per-frame time logic
    // -----
    float currentFrame = static_cast<float>(
        glfwGetTime());
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;

    // input
    // -----
    processInput(window);

    // render
    // -----
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT
        );

    // bind textures on corresponding texture units
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture1);
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, texture2);

    // activate shader
    ourShader.use();
```

```

// pass projection matrix to shader (note that in
// this case it could change every frame)
glm::mat4 projection = glm::perspective(glm::
    radians(fov), (float)SCR_WIDTH / (float)
    SCR_HEIGHT, 0.1f, 100.0f);
ourShader.setMat4("projection", projection);

// camera/view transformation
glm::mat4 view = glm::lookAt(cameraPos, cameraPos
    + cameraFront, cameraUp);
ourShader.setMat4("view", view);

// render boxes
glBindVertexArray(VAO);
for (unsigned int i = 0; i < 10; i++)
{
    // calculate the model matrix for each
    // object and pass it to shader before
    // drawing
    glm::mat4 model = glm::mat4(1.0f); // make
        // sure to initialize matrix to identity
        // matrix first
    model = glm::translate(model, cubePositions
        [i]);
    float angle = 20.0f * i;
    model = glm::rotate(model, glm::radians(
        angle), glm::vec3(1.0f, 0.3f, 0.5f));
    ourShader.setMat4("model", model);

    glDrawArrays(GL_TRIANGLES, 0, 36);
}

// glfw: swap buffers and poll IO events (keys
// pressed/released, mouse moved etc.)

glfwSwapBuffers(window);
glfwPollEvents();
}

// optional: de-allocate all resources once they've
// outlived their purpose:

glDeleteVertexArrays(1, &VAO);
glDeleteBuffers(1, &VBO);

// glfw: terminate, clearing all previously allocated
// GLFW resources.

```

```

        glfwTerminate();
        return 0;
    }

    // process all input: query GLFW whether relevant keys are
    // pressed/released this frame and react accordingly

    void processInput(GLFWwindow *window)
    {
        if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
            glfwSetWindowShouldClose(window, true);

        float cameraSpeed = static_cast<float>(2.5 * deltaTime);
        if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
            cameraPos += cameraSpeed * cameraFront;
        if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
            cameraPos -= cameraSpeed * cameraFront;
        if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
            cameraPos -= glm::normalize(glm::cross(cameraFront,
                cameraUp)) * cameraSpeed;
        if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
            cameraPos += glm::normalize(glm::cross(cameraFront,
                cameraUp)) * cameraSpeed;
    }

    // glfw: whenever the window size changed (by OS or user resize
    // ) this callback function executes

    void framebuffer_size_callback(GLFWwindow* window, int width,
                                  int height)
    {
        // make sure the viewport matches the new window
        // dimensions; note that width and
        // height will be significantly larger than specified on
        // retina displays.
        glViewport(0, 0, width, height);
    }

    // glfw: whenever the mouse moves, this callback is called

    void mouse_callback(GLFWwindow* window, double xposIn, double
                        yposIn)
    {
        float xpos = static_cast<float>(xposIn);
        float ypos = static_cast<float>(yposIn);

        if (firstMouse)
        {

```

```

        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // reversed since y-
        coordinates go from bottom to top
    lastX = xpos;
    lastY = ypos;

    float sensitivity = 0.1f; // change this value to your
        liking
    xoffset *= sensitivity;
    yoffset *= sensitivity;

    yaw += xoffset;
    pitch += yoffset;

    // make sure that when pitch is out of bounds, screen
        doesn't get flipped
    if (pitch > 89.0f)
        pitch = 89.0f;
    if (pitch < -89.0f)
        pitch = -89.0f;

    glm::vec3 front;
    front.x = cos(glm::radians(yaw)) * cos(glm::radians(
        pitch));
    front.y = sin(glm::radians(pitch));
    front.z = sin(glm::radians(yaw)) * cos(glm::radians(
        pitch));
    cameraFront = glm::normalize(front);
}

// glfw: whenever the mouse scroll wheel scrolls, this callback
is called

void scroll_callback(GLFWwindow* window, double xoffset, double
yoffset)
{
    fov -= (float)yoffset;
    if (fov < 1.0f)
        fov = 1.0f;
    if (fov > 45.0f)
        fov = 45.0f;
}

```

C++ Code 15: main.cpp "Cube and Moving Camera"

We are only using GLAD, SOIL, GLFW and of course OpenGL. There are two other files we need to make, the Vertex Shader and Fragment Shader, GLSL files

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoord;

// texture samplers
uniform sampler2D texture1;
uniform sampler2D texture2;

void main()
{
    // linearly interpolate between both textures
    FragColor = mix(texture(texture1, TexCoord), texture(
        texture2, TexCoord), 0.2);
}
```

C++ Code 16: *camera.fs* "Cube and Moving Camera"

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0
        f);
    TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

C++ Code 17: *camera.vs* "Cube and Moving Camera"

There is one thing you need to setup (you can skip this step if you can hack the error connecting to this), since this example is gained from [12], thus you need to build LearnOpenGL that you clone from the github with CMake and then copy **../LearnOpenGL/build/configuration/root_directory.h** into **/usr/lib**. After all preparations are made, now open the terminal at the current working directory, and type:

```
g++ main.cpp -o result /root/SourceCodes/CPP/src/glad.c -lSOIL -lglfw -lGL
./result
```

8. C++ Example of Rotating 3D Cube with Keyboard Events, GLM and GLFW

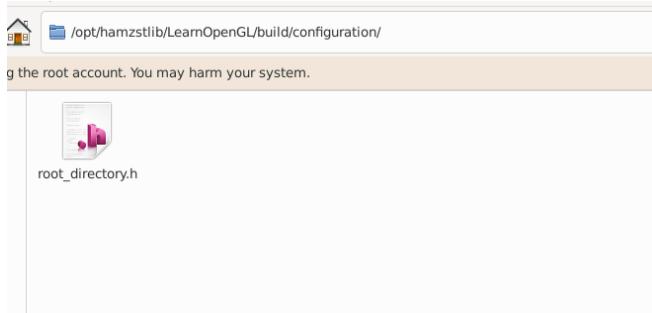


Figure 2.11: You need to put `root_directory.h` into `/usr/include`, this file is obtained after building LearnOpenGL examples [12], you can find this inside the build folder .../LearnOpenGL/build/configuration.

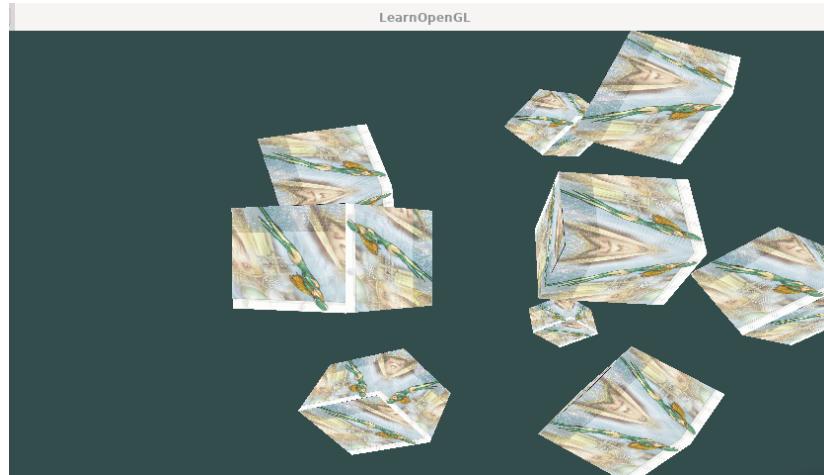


Figure 2.12: You can move the camera by using WASD keyboard keys, and use mouse to zoom and rotate camera as well. In Physics and Math simulation we can do this when the computation is already converging or while the physics process is still running to make it more fun to see from different point of view (all files for this example are in the folder: DFSimulatorC/Source Codes/C++/ch2-example6-Cube and Moving Camera).

In this example we are going to create rotating cube centered at the origin, inspired from Chapter 4 of this book [4], the rotational movements are:

- (a) Yaw, rotating toward y axis (rotational movement between x axis and z axis).
- (b) Roll, rotating toward z axis (rotational movement between x axis and y axis)
- (c) Pitch, rotating toward x axis (rotational movement between y axis and z axis).

This is the basic of flight simulator, movement in 3D game, and for fluid flows as well.
In a new directory, create a file by opening a terminal and type:

vim main.cpp

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cmath>
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include "Utils.h"

using namespace std;

#define numVAOs 1
#define numVBOs 2

// camera for movement with keyboard
glm::vec3 cameraPos = glm::vec3(0.0f, 0.0f, 3.0f);
glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
glm::vec3 cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);

// timing
float deltaTime = 0.0f; // time between current frame and last
frame
float lastFrame = 0.0f;

float cameraX, cameraY, cameraZ;
float cubeLocX, cubeLocY, cubeLocZ;
GLuint renderingProgram;
GLuint vao[numVAOs];
GLuint vbo[numVBOs];

GLuint mvLoc, projLoc;
int width, height;
float aspect;
glm::mat4 pMat, vMat, mMat, mvMat, rxMat, ryMat, rzMat, rxyMat,
ryxMat, rxzMat;
```

```

        void setupVertices(void) { // 36 vertices, 12 triangles, makes
            2x2x2 cube placed at origin
            float vertexPositions[108] = {
                -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f,
                -1.0f, -1.0f,
                1.0f, -1.0f, -1.0f, 1.0f, 1.0f, -1.0f, -1.0f,
                1.0f, -1.0f,
                1.0f, -1.0f, -1.0f, 1.0f, -1.0f, 1.0f, 1.0f,
                1.0f, -1.0f,
                1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f,
                -1.0f,
                1.0f, -1.0f, 1.0f, -1.0f, -1.0f, 1.0f, 1.0f,
                1.0f, 1.0f,
                -1.0f, -1.0f, 1.0f, -1.0f, -1.0f, -1.0f, -1.0f
                , 1.0f, 1.0f,
                -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f
                , 1.0f, 1.0f,
                -1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f, 1.0f,
                1.0f, 1.0f,
                1.0f, 1.0f, 1.0f, -1.0f, 1.0f, 1.0f, -1.0f,
                1.0f, -1.0f,
                1.0f, 1.0f,
                -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f,
                -1.0f,
                1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f,
                , -1.0f,
            };
            glGenVertexArrays(1,vao);
            glBindVertexArray(vao[0]);
            glGenBuffers(numVBOs, vbo);

            glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
            glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPositions),
                        vertexPositions, GL_STATIC_DRAW);
        }

        void init(GLFWwindow* window){

            // Utils
            renderingProgram = Utils::createShaderProgram(
            "vertShader.glsl",
            "fragShader.glsl");
            cameraX = 0.0f; cameraY = 0.0f; cameraZ = 8.0f;
            cubeLocX = 0.0f; cubeLocY = -2.0f; cubeLocZ = 0.0f;
            setupVertices();
        }
    
```

```

void display(GLFWwindow* window, double currentTime) {
    glClear(GL_DEPTH_BUFFER_BIT);
    glClear(GL_COLOR_BUFFER_BIT);
    glUseProgram(renderingProgram);

    // get the uniform variables for the MV and projection
    // matrices
    mvLoc = glGetUniformLocation(renderingProgram, "mv_matrix");
    projLoc = glGetUniformLocation(renderingProgram, "proj_matrix");

    // build perspective matrix
    glfwGetFramebufferSize(window, &width, &height);
    aspect = (float)width / (float)height;
    pMat = glm::perspective(1.0472f, aspect, 0.1f, 1000.0f);
    // 1.0472 radians = 60 degrees

    // build view matrix, model matrix, and model-view
    // matrix
    vMat = glm::translate(glm::mat4(1.0f), glm::vec3(
        -cameraX, -cameraY, -cameraZ));
    mMat = glm::translate(glm::mat4(1.0f), glm::vec3(
        cubeLocX, cubeLocY, -cubeLocZ));
    mvMat = vMat * mMat;

    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
    {
        rxMat = glm::rotate(glm::mat4(1.0f), (float)
            glfwGetTime(), glm::vec3(-1.0f, 0.0f, 0.0f));
        // Pitch movement toward user positive z axis
        // between y axis and z axis
        rxvMat = vMat * rxMat;
        glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
            value_ptr(rxvMat));
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
            value_ptr(pMat));

        // associate VBO with the corresponding vertex
        // attribute in the vertex shader
        glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
            0);
        glEnableVertexAttribArray(0);

        // adjust OpenGL settings and draw model
        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_LEQUAL);
    }
}

```

```

        glDrawArrays(GL_TRIANGLES, 0, 36);
    }
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
    {
        rxMat = glm::rotate(glm::mat4(1.0f), (float)
            glfwGetTime(), glm::vec3(1.0f, 0.0f, 0.0f));
        // Pitch movement toward monitor negative z axis
        // between y axis and z axis
        rxvMat = vMat * rxMat;
        glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
            value_ptr(rxvMat));
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
            value_ptr(pMat));

        // associate VBO with the corresponding vertex
        // attribute in the vertex shader
        glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
            0);
        glEnableVertexAttribArray(0);

        // adjust OpenGL settings and draw model
        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_LESS);
        glDrawArrays(GL_TRIANGLES, 0, 36);
    }
    if (glfwGetKey(window, GLFW_KEY_Z) == GLFW_PRESS)
    {
        rzMat = glm::rotate(glm::mat4(1.0f), (float)
            glfwGetTime(), glm::vec3(0.0f, 0.0f, 1.0f));
        // Roll movement counter clockwise between x axis
        // and y axis
        rzvMat = vMat * rzMat;
        glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
            value_ptr(rzvMat));
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
            value_ptr(pMat));

        // associate VBO with the corresponding vertex
        // attribute in the vertex shader
        glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
            0);
        glEnableVertexAttribArray(0);

        // adjust OpenGL settings and draw model
        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_LESS);
    }
}

```

```

        glDrawArrays(GL_TRIANGLES, 0, 36);
    }
    if (glfwGetKey(window, GLFW_KEY_C) == GLFW_PRESS)
    {
        rzMat = glm::rotate(glm::mat4(1.0f), (float)
            glfwGetTime(), glm::vec3(0.0f, 0.0f, -1.0f));
        // Roll movement clockwise between x axis and y
        // axis
        rzvMat = vMat * rzMat;
        glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
            value_ptr(rzvMat));
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
            value_ptr(pMat));

        // associate VBO with the corresponding vertex
        // attribute in the vertex shader
        glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
            0);
        glEnableVertexAttribArray(0);

        // adjust OpenGL settings and draw model
        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_LEQUAL);
        glDrawArrays(GL_TRIANGLES, 0, 36);
    }
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
    {
        ryMat = glm::rotate(glm::mat4(1.0f), (float)
            glfwGetTime(), glm::vec3(0.0f, 1.0f, 0.0f));
        // Yaw movement counter clockwise between x axis
        // and z axis.
        ryvMat = vMat * ryMat;
        glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
            value_ptr(ryvMat));
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
            value_ptr(pMat));

        // associate VBO with the corresponding vertex
        // attribute in the vertex shader
        glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
            0);
        glEnableVertexAttribArray(0);

        // adjust OpenGL settings and draw model
        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_LEQUAL);
    }
}

```

```

        glDrawArrays(GL_TRIANGLES, 0, 36);
    }
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
    {
        ryMat = glm::rotate(glm::mat4(1.0f), (float)
            glfwGetTime(), glm::vec3(0.0f, -1.0f, 0.0f));
        // Yaw movement clockwise between x axis and z
        // axis.
        ryvMat = vMat * ryMat;
        glUniformMatrix4fv(mvLoc, 1, GL_FALSE, glm::
            value_ptr(ryvMat));
        glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::
            value_ptr(pMat));

        // associate VBO with the corresponding vertex
        // attribute in the vertex shader
        glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0,
            0);
        glEnableVertexAttribArray(0);

        // adjust OpenGL settings and draw model
        glEnable(GL_DEPTH_TEST);
        glDepthFunc(GL_LESS);
        glDrawArrays(GL_TRIANGLES, 0, 36);
    }
    // copy perspective and MV matrices to corresponding
    // uniform variables
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(
        pMat));

    // associate VBO with the corresponding vertex attribute
    // in the vertex shader
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);

    // adjust OpenGL settings and draw model
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);
    glDrawArrays(GL_TRIANGLES, 0, 36);
}

void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
    {
        glfwSetWindowShouldClose(window, true);
    }
}

```

```

        }

    }

    int main(void)
    {
        glfwInit(); //initialize GLFW and GLEW libraries
        glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
        glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
        glfwWindowHint(GLFW_OPENGL_PROFILE,
                       GLFW_OPENGL_CORE_PROFILE);

        GLFWwindow* window = glfwCreateWindow(600, 600, "Pitch,
                                               Yaw, Roll for Interpolated Color Cube", NULL, NULL);
        glfwMakeContextCurrent(window);
        //glfwSetKeyCallback(window, updateKeyboard);
        if(glewInit() != GLEW_OK) {
            exit(EXIT_FAILURE);
        }
        glfwSwapInterval(1);

        init(window);

        while(!glfwWindowShouldClose(window)) {
            // per-frame time logic
            // -----
            float currentFrame = static_cast<float>(
                glfwGetTime());
            deltaTime = currentFrame - lastFrame;
            lastFrame = currentFrame;

            processInput(window);

            display(window, glfwGetTime());
            glfwSwapBuffers(window);
            glfwPollEvents();
        }
        glfwDestroyWindow(window);
        glfwTerminate();
        exit(EXIT_SUCCESS);
        //return 0;
    }
}

```

C++ Code 18: *main.cpp "Yaw Pitch Roll for 3D Cube"*

Then create the Vertex Shader, Fragment Shader, Utils file and header too.

```
#version 330
```

```

out vec4 color;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

in vec4 varyingColor;

void main(void) {
    color = varyingColor;
}

```

C++ Code 19: *fragShader.gsls "Yaw Pitch Roll for 3D Cube"*

```

#include <GL/glew.h>
#include <GLFW/glfw3.h>
//#include <SOIL2/SOIL2.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cmath>
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp> // glm::value_ptr
#include <glm/gtc/matrix_transform.hpp> // glm::translate, glm::
rotate, glm::scale, glm::perspective
#include "Utils.h"
using namespace std;

Utils::Utils() {}

string Utils::readShaderFile(const char *filePath) {
    string content;
    ifstream fileStream(filePath, ios::in);
    string line = "";
    while (!fileStream.eof()) {
        getline(fileStream, line);
        content.append(line + "\n");
    }
    fileStream.close();
    return content;
}

bool Utils::checkOpenGLError() {
    bool foundError = false;
    int glErr = glGetError();
    while (glErr != GL_NO_ERROR) {
        cout << "glError: " << glErr << endl;
        foundError = true;
        glErr = glGetError();
    }
}

```

```

        return foundError;
    }

    void Utils::printShaderLog(GLuint shader) {
        int len = 0;
        int chWrittn = 0;
        char *log;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &len);
        if (len > 0) {
            log = (char *)malloc(len);
            glGetShaderInfoLog(shader, len, &chWrittn, log);
            cout << "Shader Info Log: " << log << endl;
            free(log);
        }
    }

    GLuint Utils::prepareShader(int shaderTYPE, const char *
        shaderPath)
{
    GLint shaderCompiled;
    string shaderStr = readShaderFile(shaderPath);
    const char *shaderSrc = shaderStr.c_str();
    GLuint shaderRef = glCreateShader(shaderTYPE);
    glShaderSource(shaderRef, 1, &shaderSrc, NULL);
    glCompileShader(shaderRef);
    checkOpenGLError();
    glGetShaderiv(shaderRef, GL_COMPILE_STATUS, &
        shaderCompiled);
    if (shaderCompiled != 1)
    {
        if (shaderTYPE == 35633) cout << "Vertex ";
        if (shaderTYPE == 36488) cout << "Tess Control ";
        if (shaderTYPE == 36487) cout << "Tess Eval ";
        if (shaderTYPE == 36313) cout << "Geometry ";
        if (shaderTYPE == 35632) cout << "Fragment ";
        cout << "shader compilation error." << endl;
        printShaderLog(shaderRef);
    }
    return shaderRef;
}

void Utils::printProgramLog(int prog) {
    int len = 0;
    int chWrittn = 0;
    char *log;
    glGetProgramiv(prog, GL_INFO_LOG_LENGTH, &len);
    if (len > 0) {
        log = (char *)malloc(len);

```

```

        glGetProgramInfoLog(prog, len, &chWrittn, log);
        cout << "Program Info Log: " << log << endl;
        free(log);
    }

    int Utils::finalizeShaderProgram(GLuint sprogram)
    {
        GLint linked;
        glLinkProgram(sprogram);
        checkGLError();
        glGetProgramiv(sprogram, GL_LINK_STATUS, &linked);
        if (linked != 1)
        {
            cout << "linking failed" << endl;
            printProgramLog(sprogram);
        }
        return sprogram;
    }

    GLuint Utils::createShaderProgram(const char *vp, const char *fp) {
        GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
        GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
        GLuint vfprogram = glCreateProgram();
        glAttachShader(vfprogram, vShader);
        glAttachShader(vfprogram, fShader);
        finalizeShaderProgram(vfprogram);
        return vfprogram;
    }

    GLuint Utils::createShaderProgram(const char *vp, const char *gp, const char *fp) {
        GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
        GLuint gShader = prepareShader(GL_GEOMETRY_SHADER, gp);
        GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
        GLuint vgfprogram = glCreateProgram();
        glAttachShader(vgfprogram, vShader);
        glAttachShader(vgfprogram, gShader);
        glAttachShader(vgfprogram, fShader);
        finalizeShaderProgram(vgfprogram);
        return vgfprogram;
    }

    GLuint Utils::createShaderProgram(const char *vp, const char *tCS, const char* tES, const char *fp) {
        GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
        GLuint tcShader = prepareShader(GL_TESS_CONTROL_SHADER,

```

```

        tCS);
    GLuint teShader = prepareShader(
        GL_TESS_EVALUATION_SHADER, tES);
    GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
    GLuint vtfprogram = glCreateProgram();
    glAttachShader(vtfprogram, vShader);
    glAttachShader(vtfprogram, tcShader);
    glAttachShader(vtfprogram, teShader);
    glAttachShader(vtfprogram, fShader);
    finalizeShaderProgram(vtfprogram);
    return vtfprogram;
}

GLuint Utils::createShaderProgram(const char *vp, const char *
tCS, const char* tES, char *gp, const char *fp) {
    GLuint vShader = prepareShader(GL_VERTEX_SHADER, vp);
    GLuint tcShader = prepareShader(GL_TESS_CONTROL_SHADER,
        tCS);
    GLuint teShader = prepareShader(
        GL_TESS_EVALUATION_SHADER, tES);
    GLuint gShader = prepareShader(GL_GEOMETRY_SHADER, gp);
    GLuint fShader = prepareShader(GL_FRAGMENT_SHADER, fp);
    GLuint vtgfprogram = glCreateProgram();
    glAttachShader(vtgfprogram, vShader);
    glAttachShader(vtgfprogram, tcShader);
    glAttachShader(vtgfprogram, teShader);
    glAttachShader(vtgfprogram, gShader);
    glAttachShader(vtgfprogram, fShader);
    finalizeShaderProgram(vtgfprogram);
    return vtgfprogram;
}

// GOLD material — ambient, diffuse, specular, and shininess
float* Utils::goldAmbient() { static float a[4] = { 0.2473f,
    0.1995f, 0.0745f, 1 }; return (float*)a; }
float* Utils::goldDiffuse() { static float a[4] = { 0.7516f,
    0.6065f, 0.2265f, 1 }; return (float*)a; }
float* Utils::goldSpecular() { static float a[4] = { 0.6283f,
    0.5559f, 0.3661f, 1 }; return (float*)a; }
float Utils::goldShininess() { return 51.2f; }

// SILVER material — ambient, diffuse, specular, and shininess
float* Utils::silverAmbient() { static float a[4] = { 0.1923f,
    0.1923f, 0.1923f, 1 }; return (float*)a; }
float* Utils::silverDiffuse() { static float a[4] = { 0.5075f,
    0.5075f, 0.5075f, 1 }; return (float*)a; }
float* Utils::silverSpecular() { static float a[4] = { 0.5083f,
    0.5083f, 0.5083f, 1 }; return (float*)a; }

```

```

float Utils::silverShininess() { return 51.2f; }

// BRONZE material — ambient, diffuse, specular, and shininess
float* Utils::bronzeAmbient() { static float a[4] = { 0.2125f,
    0.1275f, 0.0540f, 1 }; return (float*)a; }
float* Utils::bronzeDiffuse() { static float a[4] = { 0.7140f,
    0.4284f, 0.1814f, 1 }; return (float*)a; }
float* Utils::bronzeSpecular() { static float a[4] = { 0.3936f,
    0.2719f, 0.1667f, 1 }; return (float*)a; }
float Utils::bronzeShininess() { return 25.6f; }

```

C++ Code 20: *Utils.cpp "Yaw Pitch Roll for 3D Cube"*

```

#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <string>
#include <iostream>
#include <fstream>
#include <cmath>
#include <vector>
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>

class Utils
{
private:
    static std::string readShaderFile(const char *filePath);
    static void printShaderLog(GLuint shader);
    static void printProgramLog(int prog);
    static GLuint prepareShader(int shaderTYPE, const char *
        shaderPath);
    static int finalizeShaderProgram(GLuint sprogram);

public:
    Utils();
    static bool checkOpenGLError();
    static GLuint createShaderProgram(const char *vp, const
        char *fp);
    static GLuint createShaderProgram(const char *vp, const
        char *gp, const char *fp);
    static GLuint createShaderProgram(const char *vp, const
        char *tCS, const char* tES, const char *fp);
    static GLuint createShaderProgram(const char *vp, const
        char *tCS, const char* tES, char *gp, const char *fp
    );
    static GLuint loadTexture(const char *texImagePath);
    static GLuint loadCubeMap(const char *mapDir);

```

```

        static float* goldAmbient();
        static float* goldDiffuse();
        static float* goldSpecular();
        static float goldShininess();

        static float* silverAmbient();
        static float* silverDiffuse();
        static float* silverSpecular();
        static float silverShininess();

        static float* bronzeAmbient();
        static float* bronzeDiffuse();
        static float* bronzeSpecular();
        static float bronzeShininess();
    };

```

C++ Code 21: *Utils.h "Yaw Pitch Roll for 3D Cube"*

```

#ifndef version 330

layout (location=0) in vec3 position;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

out vec4 varyingColor;

void main(void) {
    gl_Position = proj_matrix * mv_matrix * vec4(position,
        1.0);
    varyingColor = vec4(position, 1.0) * 0.5 + vec4(0.5,
        0.5, 0.5, 0.5);
}

```

C++ Code 22: *vertShader.gsls "Yaw Pitch Roll for 3D Cube"*

After finish create all those files, now open the terminal at the current working directory, and type:

```
g++ *.cpp -o result -lGLFW -lglfw -lGL
./result
```

III. KNOW-HOW IN C++

1. First, for the compiler we use **g++** that is more suitable for C++ codes, to compile **.c** files you better use **gcc**.
2. To show warning messages type:
g++ -o main main.cpp -Wall

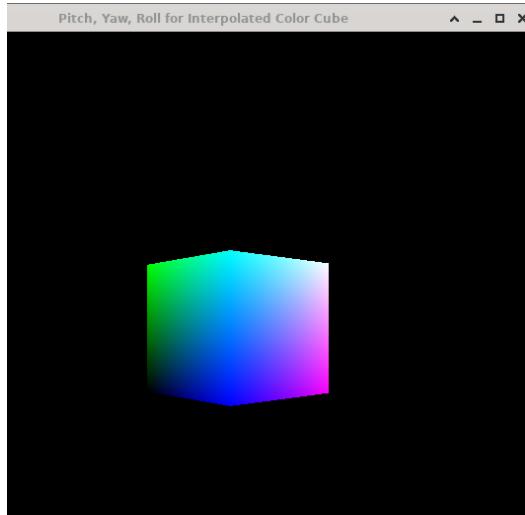


Figure 2.13: You can move the box by using keyboard' keys W / S for pitch movement, keys A / D for yaw movement, and keys Z / C for roll movement. (all files for this example are in the folder: `DFSimulatorC/Source Codes/C++/ch2-example7-Yaw Pitch Roll for 3D Cube`).

3. If you want to specify the name of the compiled executable file, do so by using the **-o** flag:
`g++ -o [name] [file to compile]`
4. If you want to compile object files **object-1.o** and **object-2.o** into a **main** executable file, type:
`g++ -o main object-1.o object-2.o`
5. If you want to specify a root directory, where libraries and headers can be found, use the **-sysroot** flag:
`g++ -o [name] -sysroot [directory] main.cpp`
6. To create a static library, start by compiling an object file:
`g++ -o obj.o main.cpp`

use the ar utility with rcs to create an archive (**.a**) file:
`ar rcs archive.a obj.o`

Finally, use g++:
`g++ -o final example.cpp archive.a`

7. You will always include header file from cpp source code file. For example inside **main.cpp** there will be `#include "Mesh.h"`.
8. To be able to reuse functions, we can create separate **.cpp** file (and an associated **.h**) instead of putting a lot of codes inside the main source code (**main.cpp**).
9. Consider a function `createShaderProgram()` that can be defined as:
 - **GLuint Utils::createShaderProgram(const char *vp, const char *fp)**
Supports shader programs which utilize a vertex and fragment shader.

- **GLuint Utils::createShaderProgram(const char *vp, const char *gp, const char *fp)**
Supports shader programs which utilize a vertex, geometry and fragment shader.
- **GLuint Utils::createShaderProgram(const char *vp, const char *tCS, const char *tES, const char *fp)**
Supports shader programs which utilize a vertex, tessellation and fragment shader.
- **GLuint Utils::createShaderProgram(const char *vp, const char *tCS, const char *tES, const char *gp, const char *fp)**
Supports shader programs which utilize a vertex, tessellation, geometry and fragment shader.

The parameters accepted in each case are pathnames for the GLSL files containing the shader code. An example of a completed program that is placed in the variable "renderingProgram":
renderingProgram = Utils::createShaderProgram("vertShader.gsl", "fragShader.gsl")
given that you put the two files "vertShader.gsl", "fragShader.gsl" in the working directory of your C++ project.

10. Learn about every libraries that are used, like **SOIL**, or **stb-image**, because some of them need different configuration at the C++ source code, for example you have to add: **#define STB_IMAGE_IMPLEMENTATION** to your code before all the other include (as the first line). This can be known from their documentation. To avoid getting this when compiling: **undefined reference to stbi_load stbi_image_free stbi_set_flip_vertically**

Chapter 3

Mathematics and Physics in Computer Graphics

*"I'll rise up like the Eagle, and I will soar with you, your spirit lifts me on, with the power of Your Love" -
The Power of Your Love song*

In this chapter I will only put minimal explanations, if you are interested and want to dig more you can read books related to Mathematics and Physics.

I. MATHEMATICS

The mathematics that will be used in computer graphics are (but not limited to):

1. Geometry (Coordinate Systems, Space transformations)
 2. Linear Algebra (Vectors, Matrices)
- i. Geometry

When we see the image on our desktop computer, what we see are actually made of pixel or point with different color. The monitor or display that is still used in 2023 is flat, but it can trick our brain so we can see 3-dimensional world in there, when we play 3-D game or designing 3-D image or watching a movie.

In computer graphics term, we will call points as vertices. The first step of creating object is to draw the vertices, and then connecting them to create triangle, then group of triangles that will become object. A cube itself can be seen as combination of 12 triangles.

[DF*] 3D Coordinate System

In order to define a position and movement, we will need a 3D coordinate system that can help us compute and determine the state or condition of the object at certain time period.

[DF*] Points

With the defined coordinate system, we can specify the position of the object in 3D correctly. The origin is located at $(0, 0, 0)$.

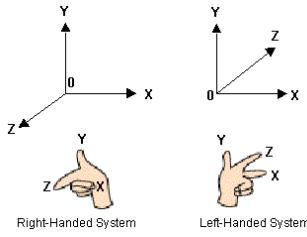


Figure 3.1: The right-handed coordinate system is used by OpenGL and Vulkan, while the left-handed coordinate system is used by DirectX and Direct3D.

[DF*] Polygon

In computer graphics, especially with OpenGL, all the rendered graphics are based on vertices that will be connected into triangle, then combining with other triangle that will become another polygon such as rectangle, cube, circle, sphere, etc. Since OpenGL conventionally uses a right-handed coordinate system, thus creating triangle vertices, they will have a counter-clockwise ordering.

ii. Linear Algebra

When an object is moving then we need to learn about vector, vector is a quantity that has direction and magnitude. The velocity is an example of vector.

[DF*] Vectors

Vectors can be added, multiplied and subtracted. Suppose we have vector \vec{v} and \vec{w} , where $\vec{v} = (v_x, v_y, v_z)$ and $\vec{w} = (w_x, w_y, w_z)$, thus

$$\begin{aligned}\vec{v} + \vec{w} &= (v_x + w_x, v_y + w_y, v_z + w_z) \\ \vec{v} - \vec{w} &= (v_x - w_x, v_y - w_y, v_z - w_z)\end{aligned}$$

To calculate the magnitude of the vector (or the length of the vector):

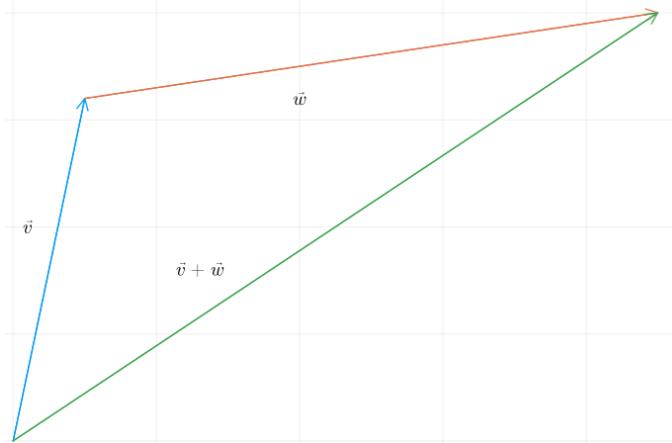


Figure 3.2: The addition of two vectors in 2-dimension (ch3-linalg-vectoradd.jl).

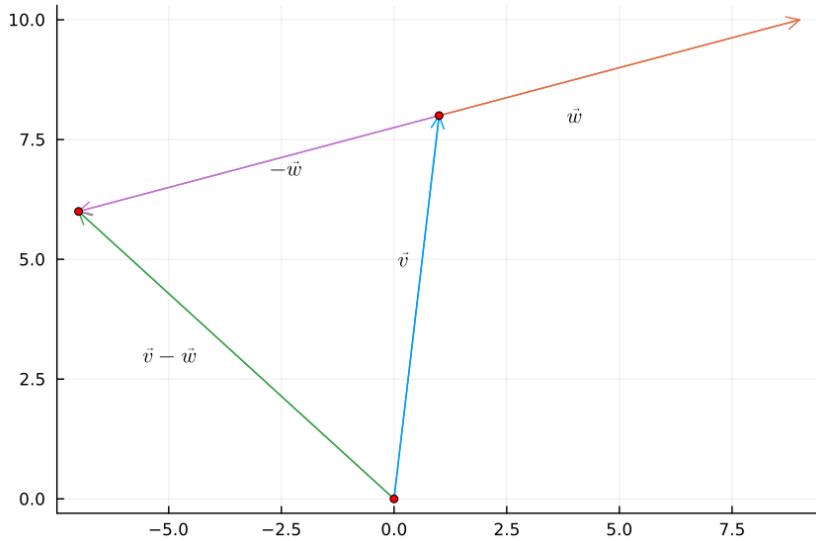


Figure 3.3: The subtraction of two vectors in 2-dimension (*ch3-linalg-vectorsubtraction.jl*).

$$\|\vec{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

with $\|\vec{v}\|$ is sometimes called norm, the magnitude of a vector is always greater than or equal to zero. In a lot of cases, we might only need the direction of the vector, thus we can convert all vectors into unit vector by:

$$\hat{v} = \frac{\vec{v}}{\|\vec{v}\|} = \left(\frac{v_x}{\|\vec{v}\|}, \frac{v_y}{\|\vec{v}\|}, \frac{v_z}{\|\vec{v}\|} \right)$$

If we want to know the angle between two vectors, we will use dot product:

$$\begin{aligned} \vec{v} \cdot \vec{w} &= v_x w_x + v_y w_y + v_z w_z \\ &= \|\vec{v}\| \|\vec{w}\| \cos \theta \end{aligned}$$

thus

$$\theta = \cos^{-1} \frac{v_x w_x + v_y w_y + v_z w_z}{\|\vec{v}\| \|\vec{w}\|}$$

- If $\vec{v} \cdot \vec{w} = 0$, then \vec{v} is perpendicular to \vec{w} .
- If $\vec{v} \cdot \vec{w} = \|\vec{v}\| \|\vec{w}\|$, then the two vectors are parallel to each other.
- If $\vec{v} \cdot \vec{w} < 0$, then the angle between the two vectors is greater than 90° .
- If $\vec{v} \cdot \vec{w} > 0$, then the angle between the two vectors is less than 90°
- The commutative law applies to a scalar product, so

$$\vec{v} \cdot \vec{w} = \vec{w} \cdot \vec{v}$$

We can perform a cross product(vector product) between two vectors only in 3-dimension.
Consider

$$\vec{f} = \vec{v} \times \vec{w}$$

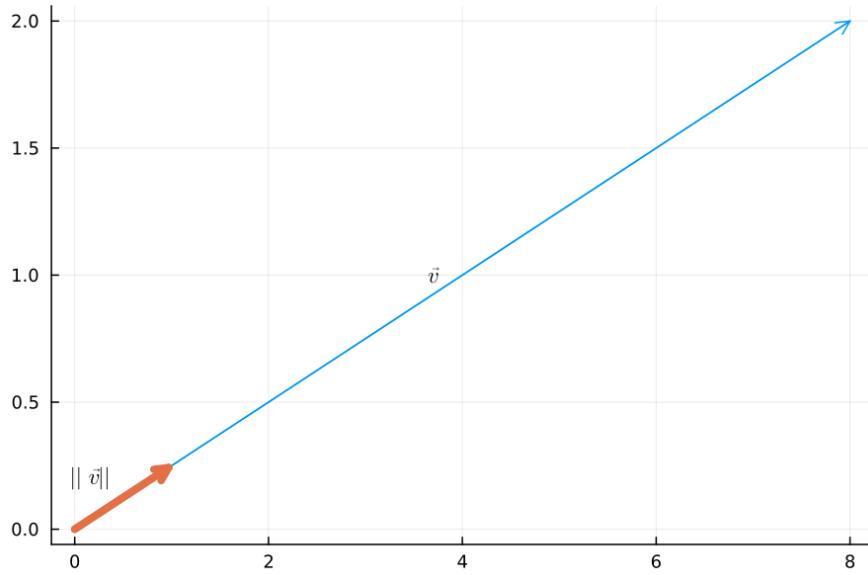


Figure 3.4: The norm of vector \vec{v} is bolded with orange color (`ch3-linalg-vectornorm.jl`).

if we expand the cross product in unit-vector notation, we will obtain

$$\begin{aligned}\vec{v} \times \vec{w} &= (v_x \hat{i} + v_y \hat{j} + v_z \hat{k}) \times (w_x \hat{i} + w_y \hat{j} + w_z \hat{k}) \\ &= (v_y w_z - v_z w_y) \hat{i} + (v_z w_x - v_x w_z) \hat{j} + (v_x w_y - v_y w_x) \hat{k}\end{aligned}$$

To obtain the magnitude of \vec{f} :

$$\vec{f} = |\vec{v} \times \vec{w}| = v \cdot w \sin \phi$$

where ϕ is the smallest angle between \vec{v} and \vec{w} .

We can obtain the direction for \vec{f} by using our right hand, point all four fingers toward one direction and the thumb will be perpendicular toward the four fingers, then imagine that we sweep all our four fingers from the direction of \vec{v} toward the direction of \vec{w} , the thumb will show the direction of \vec{f} .

Cross product is used widely for computer graphics (in lighting effects), since the resulting cross product is normal (perpendicular) to the plane defined by the original two vectors. How people walk on a surface, no matter how bumpy the road is, they will stand tall toward the normal of the surface while moving forward or do any activities.

- If \vec{v} and \vec{w} are parallel or antiparallel, then $\vec{v} \times \vec{w} = 0$.
- The magnitude of $\vec{v} \times \vec{w}$ is maximum when \vec{v} and \vec{w} are perpendicular to each other.
- The commutative law does not apply to a cross product, so

$$\vec{v} \times \vec{w} = -(\vec{w} \times \vec{v})$$

[DF*] Matrices In computer graphics, matrices are used to calculate object transforms such as translation (movement), scaling in the x, y, z -axis and rotation around the x, y, z -axis.

Matrices have rows and columns. A matrix A with m number of rows and n number of columns is a matrix of size $m \times n$, thus

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

each element of a matrix is represented as indices ij , thus it will be written a_{ij} to represent matrix index at i th-row, j th-column.

What you need to know and how to compute:

- Matrix addition and subtraction
- Matrix multiplication to a matrix with correct size
- Matrix multiplication to a vector with correct size and dimension
- Identity matrix, a square matrix of size n with indices $a_{ii} = 1$ with $i = 1, 2, \dots, n$, and 0 elsewhere.
- Determine the transpose of a matrix
- Determine the inverse of a matrix

[DF*] In computer graphics, matrices are used for performing transformations on objects, such as:

- Translation
- Rotation
- Scale
- Projection
- Look-At

All transformation matrices have the size of 4×4 .

Translation

Consider the initial point of an object in 3D is (x, y, z) , and want to be translated to $(x + t_x, y + t_y, z + t_z)$ then the translation matrix transform is

$$\begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (3.1)$$

The functions for building translation matrices with GLM are:

**glm::translate(x,y,z)
mat4 * vec4**

If we want to translate a vector of $(1, 0, 0)$ by $(1, 2, 3)$ the code is:

```
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <iostream>

int main()
```

```

{
    glm::vec4 vec(1.0f, 0.0f, 0.0f, 1.0f); // vector (1,0,0)
    glm::mat4 translation = glm::mat4(1.0f);
    translation = glm::translate(translation, glm::vec3(1.0f
        , 2.0f, 3.0f));
    vec = translation * vec;
    std::cout << "(" << vec.x << "," << vec.y << "," << vec.z
        << ")" << std::endl;
}

```

You can easily compile and run it from terminal with:

```
g++ main.cpp -o result
./result
```

Scaling

A scale matrix is used to change the size of objects or move points toward or away from the origin.

Consider the initial point of an object in 3D is (x, y, z) , and want to be scaled to (s_x, s_y, s_z) then the scaling matrix transform is

$$\begin{bmatrix} x * s_x \\ y * s_y \\ z * s_z \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (3.2)$$

If we want to scale a vector of $(4, 3, 2)$ by $(3, 3, 3)$ / enlarging it 3 times its' original size, the code is:

```

#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <iostream>

int main()
{
    glm::vec4 vec(4.0f, 3.0f, 2.0f, 1.0f); // vector (4,3,2)
    glm::mat4 scaling = glm::mat4(1.0f);
    scaling = glm::scale(scaling, glm::vec3(3.0f, 3.0f, 3.0f
        )); // enlarge 3 times
    vec = scaling * vec;
    std::cout << "(" << vec.x << "," << vec.y << "," << vec.z
        << ")" << std::endl;
}

```

You can easily compile and run it from terminal with:

```
g++ main.cpp -o result
./result
```

Scaling can also be used to switch coordinate system, by negating the z coordinate, thus the scale matrix transform is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

To rotate an object in 3D space requires specifying:

1. An axis of rotation (toward x axis will be called yaw, toward y axis will be called roll, and toward z axis will be called pitch)
2. A rotation amount in degrees or radians

The rotation matrix transform around x by θ :

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (3.3)$$

The rotation matrix transform around y by ϕ :

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \phi & 0 & \sin \phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \phi & 0 & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (3.4)$$

The rotation matrix transform around z by φ :

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \varphi & -\sin \varphi & 0 & 0 \\ \sin \varphi & \cos \varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (3.5)$$

From a simple observation above, the rotation toward x axis won't change the value of the x part, the same for rotation toward y axis, the value of y stays, and it occurs as well for rotation towards z axis.

If you want to rotate a vector of $(1, 2, 3)$ 90 degree toward the x, y , and z axis separately, the code is

```
#include <glm/glm.hpp>
#include <glm/gtc/type_ptr.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include <iostream>

int main()
{
    glm::vec4 vecx(1.0f, 2.0f, 3.0f, 1.0f); // vector
    (1,2,3)
```

```

glm::vec4 vecy(1.0f, 2.0f, 3.0f, 1.0f); // vector
(1,2,3)
glm::vec4 vecz(1.0f, 2.0f, 3.0f, 1.0f); // vector
(1,2,3)

glm::mat4 rotationx = glm::mat4(1.0f);
glm::mat4 rotationy = glm::mat4(1.0f);
glm::mat4 rotationz = glm::mat4(1.0f);
rotationx = glm::rotate(rotationx, glm::radians(90.0f),
    glm::vec3(1.0f, 0.0f, 0.0f)); // rotate 90 degrees
    toward x axis
vecx = rotationx * vecx;

rotationy = glm::rotate(rotationy, glm::radians(90.0f),
    glm::vec3(0.0f, 1.0f, 0.0f)); // rotate 90 degrees
    toward y axis
vecy = rotationy * vecy;

rotationz = glm::rotate(rotationz, glm::radians(90.0f),
    glm::vec3(0.0f, 0.0f, 1.0f)); // rotate 90 degrees
    toward z axis
vecz = rotationz * vecz;

std::cout << "Original_vector:_(1,2,3)" << std::endl;
std::cout << "Rotation_90_degrees_toward_x_axis:" << std
    ::endl;
std::cout << "(" << vecx.x << "," << vecx.y << "," <<
    vecx.z << ")" << std::endl;
std::cout << "Rotation_90_degrees_toward_y_axis:" << std
    ::endl;
std::cout << "(" << vecy.x << "," << vecy.y << "," <<
    vecy.z << ")" << std::endl;
std::cout << "Rotation_90_degrees_toward_z_axis:" << std
    ::endl;
std::cout << "(" << vecz.x << "," << vecz.y << "," <<
    vecz.z << ")" << std::endl;
}

```

You can easily compile and run it from terminal with:

```

g++ main.cpp -o result
./result

```

[DF*]

II. PHYSICS

The physics that will be used in computer graphics are (but not limited to):

1. Gravity ($g = 9.8 \text{ m/s}^2 = 32.17 \text{ ft/s}^2$)

2. Collision, Newton's Laws

3. Oscillations

4. Angular Rotation, velocity

5. Moment of inertia and torque

To make the object realistic, we need to add the physics otherwise without physics libraries / engine, the cube we render with OpenGL can just walk through a wall, is it possible for such ghost collision in this real life? That's why we need a Physics engine when we create an object after we render the shape and giving the color or texture to it. More detail on the physics formula and explanations can be read at the corresponding chapter, along with the C++ codes to create the simulation.

Chapter 4

Computer Graphics: OpenGL, SFML, GLFW, GLEW, And All That

"Love is composed of a single soul inhabiting two bodies/entities" - Aristotle

"The best and most beautiful things in the world cannot be seen or even touched - they must be felt with the heart" - Helen Keller

Computer graphics and the numerical computing power are used everywhere in all kinds of industry, to improve design, quality of products, reduce defections, carrying out varieties of tests, like stress test or fuel consumption test. One of the most notable successes of computer graphics is the automobile industry, where the computer model can be viewed from different angles to obtain the most pleasing and popular style, we can test the vehicle's strength of components, for roadability, for seating comfort, and for safety in a crash. The same happens for flight simulator, where we can design and engineer a better airplane. Create a better fast train like Shinkansen or TGV to move a load of cargo and goods for mining or export import.

The packages used in this book to make the simulator are available in my github repository:
<https://github.com/glanzkaiser/DFSimulatorC/Source Codes/Libraries>

I. OpenGL

OpenGL is an API(Application Programming Interface) that provides us with a large set of functions to manipulate graphics and images. Each graphics card that you buy supports specific versions of OpenGL, in my example with Dell Precision 6510, I use OpenGL 3.3 that is supported by my graphics card.

OpenGL can't stand alone to show the image or animation that we create with C++ source codes. OpenGL only renders to a frame buffer, we need additional library to draw the contents of the frame buffer onto a window on the screen.

OpenGL's API is written in C, the calls are directly compatible with C and C++. C++ application / source code that calls for OpenGL will run on CPU.

The good news is all future versions of OpenGL starting from 3.3 add extra useful features to OpenGL without changing OpenGL's core mechanics; the newer versions just introduce slightly more efficient ways to accomplish the same tasks.

For example, OpenGL 4.0 has a new addition of tessellator that can generate a large number of triangles, typically as a grid, we are able to create square area surface or curved surface as a terrain.

In all three major desktop platforms (Linux, macOS, and Windows), OpenGL more or less comes with the system. However, you will need to ensure that you have downloaded and installed a recent driver for your graphics hardware.

OpenGL is entirely hardware and operating system independent, whether you are using nVIDIA GeForce or AMD Radeon graphics card, it will work the same on both hardware. The way in which OpenGL's features work is defined by a specification that is used by graphics hardware manufacturers while they're developing the drivers for their hardware. This is why we sometimes have to update the graphics hardware drivers if something doesn't look right.

Since GFreya OS is based on Linux, graphics on Linux is almost exclusively implemented using the X Window system. Supporting OpenGL on Linux involves using GLX extensions to the X Server.

GFreya OS installs OpenGL through MESA version-21.2.1, Mesa is an OpenGL compatible 3D graphics library, you can read more here:

<https://www.linuxfromscratch.org/blfs/view/11.0/x/mesa.html>

OpenGL by itself is only capable of drawing a few primitives: points, lines, and triangles. Most 3D models made by OpenGL are made up from lots of those primitives. Primitives are consisted of vertices.

Since we have to display 3D world on a 2D monitor, OpenGL do that job with vertices, triangles, color. The 2D monitor screen is made up of a raster - a rectangular array of pixels. When a 3D object is rasterized, OpenGL converts the primitives in the object into fragments. A fragment holds the information associated with a pixel. Rasterization determines the locations of pixels that need to be drawn in order to produce the triangle specified by its three vertices.

Facts about OpenGL:

- All OpenGL functions start with the `gl` prefix.
- For an object to be drawn, its vertices must be sent to the vertex shader. Vertices are usually sent by putting them in a buffer with a vertex attribute declared in the shader.

Some that need to be done once (typically in `init()`):

1. Create a buffer (or created in a function called by `init()`).
2. Copy the vertices into the buffer.

Some that need to be done at each frame (typically in `display()`):

1. Enable the buffer containing the vertices.
2. Associate the buffer with a vertex attribute.

3. Enable the vertex attribute.

4. Use **glDrawArrays(...)** to draw the object.

- In OpenGL, a buffer is contained in a Vertex Buffer Object (VBO), which is declared or instantiated in the C++/OpenGL application. Vertex Buffer Object (VBO) is the geometrical information, it includes attributes such as position, color, normal, and texture coordinates. These are stored on a per vertex basis on the GPU. We can also see VBO as the manner in which we load the vertices of a model into a buffer.
- Element Buffer Object (EBO) is used to store the index of each vertex and will be used while drawing the mesh.
- Vertex Array Object (VAO) is a helper container object that stores all the VBOs and attributes. This is used as you may have more than one VBO per object, and it would be tedious to bind the VBOs all over again when you render each frame. One VAO is required by OpenGL to be created, it is good as a way of organizing buffers and making them easier to manipulate in complex scenes.

For example, suppose that we wish to display two objects, then:

```

GLuint vao[1];
GLuint vbo[2];
...
glGenVertexArrays(1,vao);
 glBindVertexArray(vao[0]);
 glGenBuffers(2,vbo);

```

The command **glGenVertexArrays(1,vao)** creates VAOs, and the command **glGenBuffers(2,vbo)** creates VBOs, both commands return integer IDs for them.

A buffer needs to have a corresponding vertex attribute variable declared in the vertex shader. Vertex attributes are generally the first variables declared in a shader.

- Buffers are used to store information in the GPU memory for fast and efficient access to the data. Modern GPUs have a memory bandwidth of approximately 600 GB/s, quite enormous compared to the current high-end CPUs that only have approximately 12 GB/s. Buffer objects are used to store, retrieve, and move data. It is very easy to generate a buffer object in OpenGL. You can generate one by calling **glGenBuffers()**.
- OpenGL has its own data types, they are prefixed with GL, followed by the data type.

C Type	Bitdepth	Description	Common Enum
GLboolean	1+	A boolean value, either <code>GL_TRUE</code> or <code>GL_FALSE</code>	
GLbyte	8	Signed, 2's complement binary integer	<code>GL_BYTE</code>
GLubyte	8	Unsigned binary integer	<code>GL_UNSIGNED_BYTE</code>
GLshort	16	Signed, 2's complement binary integer	<code>GL_SHORT</code>
GLushort	16	Unsigned binary integer	<code>GL_UNSIGNED_SHORT</code>
GLint	32	Signed, 2's complement binary integer	<code>GL_INT</code>
GLuint	32	Unsigned binary integer	<code>GL_UNSIGNED_INT</code>
GLfixed	32	Signed, 2's complement 16.16 integer	<code>GL_FIXED</code>
GLint64	64	Signed, 2's complement binary integer	
GLuint64	64	Unsigned binary integer	
GLsizei	32	A non-negative binary integer, for sizes.	
GLenum	32	An OpenGL enumerator value	
GLintptr	<code>ptrbits¹</code>	Signed, 2's complement binary integer	
GLsizeiptr	<code>ptrbits¹</code>	Non-negative binary integer size, for memory offsets and ranges	
GLsync	<code>ptrbits¹</code>	Sync Object handle	
GLbitfield	32	A bitfield value	
GLfloat	16	An IEEE-754 floating-point value	<code>GL_HALF_FLOAT</code>
GLfloat	32	An IEEE-754 floating-point value	<code>GL_FLOAT</code>
GLclampf	32	An IEEE-754 floating-point value, clamped to the range [0,1]	
GLdouble	64	An IEEE-754 floating-point value	<code>GL_DOUBLE</code>
GLclampd	64	An IEEE-754 floating-point value, clamped to the range [0,1]	

Figure 4.1: The data types of OpenGL.

- To draw a 3D object, for example a cube, you will need to at least send the following items:
 - The vertices for the cube model
 - The transformation matrices to control the appearance of the cube's orientation in 3D space.

There are two ways of sending data through the OpenGL pipeline:

- Through a buffer to a vertex attribute
- Directly to a uniform variable

Rendering a scene to make it appears 3D requires building appropriate transformation matrices and applying them to each of the models' vertices. It is most efficient to apply the required matrix operations in the vertex shader, and it is customary to send these matrices from the C++/OpenGL application to the shader in a uniform variable.

Some OpenGL commands:

- **glDrawArrays(GLenum mode, GLint first, GLsizei count);**

To draw primitive, mode is the type of primitive, first indicates which vertex to start with, usually vertex 0, count specifies the total number of vertices to be drawn. For example:

`glDrawArrays(GL_POINTS,0,1)` (Draw a point, start from vertex 0, one vertices is drawn)

`glDrawArrays(GL_TRIANGLE,0,3)` (Draw a triangle, start from vertex 0, three vertices are drawn).

When `glDrawArrays()` is executed, the data in the buffer starts flowing, sequentially from the beginning of the buffer, through the vertex shader. The vertex shader executes once per vertex.

- **glGenVertexArrays()**

Create VAOs and return integer ID.

- **glGenBuffers()**

Create VBOs and return integer ID.

- **glBindBuffer(GL_ARRAY_BUFFER, vbo[0])** is about activating the 0th buffer

`glBufferData(GL_ARRAY_BUFFER, sizeof(vPositions), vPositions, GL_STATIC_DRAW)` is about copying the array containing the vertices into the active buffer.

Together they will copy the values of the vertices that are stored in a float array named **vPositions** into the 0th VBO.

- **glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0)** is to associate 0th attribute with buffer
glEnableVertexAttribArray(0) is to enable 0th vertex attribute.
- **glBindVertexArrays()**
To make the specified VAO active so that the generated buffers will be associated with that VAO.
- **glClear(GL_DEPTH_BUFFER_BIT)**
glClearColor(0.0, 0.0, 0.0, 1.0)
glClear(GL_COLOR_BUFFER_BIT)
Clear the background to black.

II. GLAD

In simple words, GLAD manages function pointers for OpenGL. It is useful because OpenGL is only really a standard/specification it is up to the driver manufacturer to implement the specification to a driver that the specific graphics card supports. Since there are many different versions of OpenGL drivers, the location of most of its functions is not known at compile-time and needs to be queried at run-time. GLFW helps us at compile time only.

The "GLAD file" is generally referred to as the "OpenGL Loading Library" which is generally the library which loads the various types of the pointers, to the various types of the "OpenGL functions" at the time of the runtime respectively, which includes the various types of the "Core" as well as the "Extensions."

The include file for GLAD includes the required OpenGL headers behind the scenes (like GL/gl.h) so be sure to include GLAD before other header files that require OpenGL (like GLFW).

TO download GLAD, you can go to this website and specify your OpenGL version:
<https://glad.dav1d.de/>

you will get header files (.h) and C file as well (.c). Put it in your include file (e.g. /usr/include) and the **glad.c** can be stored somewhere in your main OpenGL project directory.

Glad

Generated files. These files are not permanent!

Name	Last modified	Size
include	2023-10-19 13:46:27	-
src	2023-10-19 13:46:27	-
glad.zip	2023-10-19 13:46:27	15 MB

Permalink:

<https://glad.dav1d.de/#language=c&specification=gl&api=gl%3D3.3&api=gles1%3Dnone&api=gles2%3Dnone&api=glsc2%3Dnone&profile=>

Figure 4.2: The src and include files that is generated from <https://glad.dav1d.de/>.

Advantages of using GLAD:

- Being able to select only the extensions you use, leading to (slightly) faster compile times and initialization at runtime.
- No additional dependency for your project.

III. GLEW

OpenGL Extension Wrangler (GLEW) can be used to try newer OpenGL functions.

You can find GLEW version-2.2.0 in my github repository for this book and DF Simulator.

This command is used to build and install GLEW:

```
sed -i 's%lib64%lib%g' config/Makefile.linux &&
sed -i -e '/glew.lib.static:/d' \
-e '/0644..*STATIC/d' \
-e 's/glew.lib.static//' Makefile &&
make
```

then as super user or root type:

make install.all

Advantages of using GLEW:

- Adding GLEW as a dependency to, e.g., your CMakeLists.txt is enough to make it work.
- No large additional header and source files in your repository.
- GLEW can detect which extensions are available at runtime.
- It allows your program to adapt to the available extensions. For example, it can select a fallback path if a certain extension is not available on older hardware.

IV. GLFW

Graphics Library Framework (GLFW) is a free, Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan application development. It provides a simple, platform-independent API for creating windows, contexts and surfaces, reading input, handling events, etc.

GLFW is a C library specifically designed for use with OpenGL. It includes a class called **GLFWwindow** on which we can draw 3D scenes. Unlike SDL and SFML, GLFW only comes with the absolute necessities: window and context creation and input management. It offers the most control over the OpenGL context creation out of these three libraries.

You can download GLFW here:
<https://www.glfw.org/download>

Some GLFW commands:

- **glfwInit()**
To initialize GLFW
- **glfwSwapInterval(), glfwSwapBuffers()**
Enable Vertical synchronization (VSync), GLFW windows are by default double-buffered.
- **glfwGetCurrentContext()**
Make the associated OpenGL context current.
- **glfwPollEvents()**
Handles other window-related events, such as a key-press event.
- **glfwGetTime()**
Returns the elapsed time since GLFW was initialized.

SDL, SFML, and GLFW are for creating the context and handling input

V. GLM

GLM (OpenGL Mathematics) is a header-only C++ mathematics library that will help us to do the computation necessary for our project. GLM provides classes and basic math functions related to graphics concepts, such as vector, matrix, and quaternion.

GLM able to create points, perform vector operations (addition, subtraction), carry out matrix transforms, generate random numbers, and generate noise. GLM includes a class called **mat4** for instantiating and storing 4×4 matrices.

The manual can be read here:
<https://github.com/g-truc/glm/blob/master/manual.md>

How to use GLM:

- First at the top of the source code we need to include GLM by adding:
`#include <glm/glm.hpp>`

To do translation and rotation add:
`#include <glm/ext.hpp>`

- To define a 2D point:
`glm::vec2 p1 = glm::vec2(2.0f, 8.0f);`
- To define a 3D point:
`glm::vec3 p2 = glm::vec3(1.0f, 8.0f, 2.0f);`
- To create identity square matrix of size 4:
`glm::mat4 matrix = glm::mat4(1.0f);`
- The constructor call to build the identity matrix in the variable *m*
`glm::mat4 m(1.0f)`
- To normalize vector:
`normalize(vec3) or normalize(vec4)`
- To obtain dot product:
`dot(vec3,vec3) or dot(vec4,vec4)`
- To obtain cross product:
`cross(vec3,vec3)`
- In GLSL and GLM the data type **vec3** / **vec4** can be used to hold either points or vectors.
- To obtain the magnitude of a vector:
`length()`
- Reflection and refraction are available in GLSL and GLM.

VI. GLSL

OpenGL Shading Language. On the hardware side, OpenGL provides a multi-stage graphics pipeline that is partially programmable using GLSL. GLSL intended to run on GPU / Graphics Card, since it is related with graphics pipeline.

GLSL language includes a data type called **mat4** that can be used for storing 4×4 matrices.

VII. SDL

SDL is also a cross-platform multimedia library, but targeted at C. That makes it a bit rougher to use for C++ programmers, but it's an excellent alternative to SFML. It supports more exotic platforms and most importantly, offers more control over the creation of the OpenGL context than SFML.

VIII. SFML

SFML is a cross-platform C++ multimedia library that provides access to graphics, input, audio, networking and the system. The downside of using this library is that it tries hard to be an all-in-one solution. You have little to no control over the creation of the OpenGL context, as it was designed to be used with its own set of drawing functions. SFML is written in C++, and has bindings for various languages such as C, .Net, Ruby, Python.

You can get the latest official release on SFML's website:
<https://www.sfml-dev.org/download.php>

In my github repository, you can find SFML version-2.5.1, along with FLAC and OpenAL (the required packages to be able to install SFML successfully). To build and install SFML, you need to

build and install FLAC and OpenAL first, all of them are using cmake thus it won't be so hard to build and install.

First start with FLAC, extract the file:

```
tar -xvf flac-1.4.2.tar.xz
```

go to the extracted directory then type:

```
mkdir build
```

```
cd build
```

```
ccmake ..
```

Press c on keyboard then press e, then choose ON for BUILD_SHARED_LIBS (if available) and set CMAKE_INSTALL_PREFIX=/usr, then press c then e again then press g. Then back at terminal type:

```
make
```

as root type:

```
make install
```

Repeat the process for installing OpenAL and SFML, they are pretty much the same.

Now if you check **/usr/lib**, you will see **libsfml-audio.so**, **libsfml-graphics.so**, **libsfml-network.so**, **libsfml-system.so**, **libsfml-window.so**. Then you can use SFML.

IX. SOIL

Simple OpenGL Image Loader (SOIL) is an image loading library for OpenGL, it can be used to load image that we can process or do transformation onto that image. This library is very useful, when I remember the Elementary linear Algebra book with application to warps and morphs photograph of a person and predict how that person will look like after 50 years, then to be able to do the prediction we might use SOIL as one of the many solutions.

In order to build and install it, almost the same as SFML, we can use CMake, extract the package then go to the directory and type:

```
mkdir build
```

```
cd build
```

```
ccmake ..
```

Press c on keyboard then press e, then set CMAKE_BUILD_TYPE as RELEASE for this option. No need to build Tests, so SOIL_BUILD_TESTS=OFF, and set CMAKE_INSTALL_PREFIX=/usr.

Then press c then e again then press g. Then back at terminal type:

```
make
```

as root type:

```
make install
```

You shall see inside **/usr/lib** there is **libSOIL.a**, it is a static library. Different than dynamic library that has extension of **.so**.

X. GNUPLOT

Gnuplot is a portable command-line driven graphing utility for Linux, OS/2, MS Windows, OSX, VMS, and many other platforms. The source code is copyrighted but freely distributed (i.e., you don't have to pay for it). It was originally created to allow scientists and students to visualize mathematical functions and data interactively, but has grown to support many non-interactive uses such as web scripting. It is also used as a plotting engine by third-party applications like Octave. Gnuplot has been supported and under active development since 1986.

Gnuplot that is being used in this book is Gnuplot version 5.4 patchlevel 3 (last modified 2021-12-24).

XI. CMAKE

CMake is a tool that can generate project / solution file from a collection of source code files using pre-defined CMake scripts.

Chapter 5

Box2D, Bullet3, and ReactPhysics3D

"You know someone is the best in Science and Engineering, when one chose the path of solitary, more focus, less distraction, like Leonardo Da Vinci and Isaac Newton who never marry, the proof will be her/his achievements, books written and innovations created." - DS Glanzsche

There are 3 physics library / engine that I will use and try here. You can learn from all these and perhaps create your own Physics library, or another science branch library of your interest like Astronomy library or Biology library. All the testing here is done with GFreya OS 1.8 with OpenGL 3.3, starting on October 2023.

I. Box2D

Box2D is using imgui for the testbed GUI, it uses OpenGL for the graphics API, and for taking care of mouse and keyboard events it uses GLFW.

i. Install Box2D Library and Include Files / Headers

To download it, open the terminal then type:

git clone https://github.com/erincatto/box2d.git

Enter the directory then type:

./build.sh

Results are in the build sub-folder. To build with CMake:

```
$ mkdir build &&
$ cd build &&
$ cmake --DCMAKE_INSTALL_PREFIX=/usr \
$ --DBUILD_SHARED_LIBS=ON .. &&
$ make
```

After that, type:

make install

With the dynamic library has been installed into **/usr/lib**, we are able to build all kinds of physics simulations by modifying the physics world and the physical objects we want to simulate

with Box2D.

Now for the headers, go to the downloaded repository of Box2D `../box2d/extern/`, where you will see 4 folders of **glad**, **glfw**, **imgui**, and **sajson**. Since you should have installed and get GLAD by yourself and GLFW probably have been installed. You only need to copy **imgui** and **sajson** into **/usr/include**.

Now to test it, let's try the famous Hello World for Box2D, it shows the falling object from the height of 4 with gravity of 10 (not 9.8).

Create this simple HelloBox2D example:

```
#include "box2d/box2d.h"
#include <stdio.h>

int main(int argc, char** argv)
{
    B2_NOT_USED(argc);
    B2_NOT_USED(argv);

    // Define the gravity vector.
    b2Vec2 gravity(0.0f, -10.0f);

    // Construct a world object, which will hold and simulate the rigid
    // bodies.
    b2World world(gravity);

    // Define the ground body.
    b2BodyDef groundBodyDef;
    groundBodyDef.position.Set(0.0f, -10.0f);

    // Call the body factory which allocates memory for the ground body
    // from a pool and creates the ground box shape (also from a pool).
    // The body is also added to the world.
    b2Body* groundBody = world.CreateBody(&groundBodyDef);

    // Define the ground box shape.
    b2PolygonShape groundBox;

    // The extents are the half-widths of the box.
    groundBox.SetAsBox(50.0f, 10.0f);

    // Add the ground fixture to the ground body.
    groundBody->CreateFixture(&groundBox, 0.0f);

    // Define the dynamic body. We set its position and call the body
    // factory.
    b2BodyDef bodyDef;
    bodyDef.type = b2_dynamicBody;
```

```
bodyDef.position.Set(0.0f, 4.0f);
b2Body* body = world.CreateBody(&bodyDef);

// Define another box shape for our dynamic body.
b2PolygonShape dynamicBox;
dynamicBox.SetAsBox(1.0f, 1.0f);

// Define the dynamic body fixture.
b2FixtureDef fixtureDef;
fixtureDef.shape = &dynamicBox;

// Set the box density to be non-zero, so it will be dynamic.
fixtureDef.density = 1.0f;

// Override the default friction.
fixtureDef.friction = 0.3f;

// Add the shape to the body.
body->CreateFixture(&fixtureDef);

// Prepare for simulation. Typically we use a time step of 1/60 of a
// second (60Hz) and 10 iterations. This provides a high quality
// simulation
// in most game scenarios.
float timeStep = 1.0f / 60.0f;
int32 velocityIterations = 6;
int32 positionIterations = 2;

// This is our little game loop.
for (int32 i = 0; i < 60; ++i)
{
    // Instruct the world to perform a single step of simulation.
    // It is generally best to keep the time step and iterations
    // fixed.
    world.Step(timeStep, velocityIterations, positionIterations);

    // Now print the position and angle of the body.
    b2Vec2 position = body->GetPosition();
    float angle = body->GetAngle();

    printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle);
}

// When the world destructor is called, all bodies and joints are
// freed. This can
// create orphaned pointers, so be careful about your world
// management.
```

```
    return 0;
}
```

C++ Code 23: main.cpp "Hello Box2D"

To compile it type:

```
g++ main.cpp -o result -lbox2d  
.result
```

(-lbox2d is linking to shared library of Box2D' libbox2d.so)

```
xterm
root [ ~/SourceCodes/CPP/Box2D>Hello World ]# ./result
0.00 4.00 0.00
0.00 3.99 0.00
0.00 3.98 0.00
0.00 3.97 0.00
0.00 3.96 0.00
0.00 3.94 0.00
0.00 3.92 0.00
0.00 3.90 0.00
0.00 3.87 0.00
0.00 3.85 0.00
0.00 3.82 0.00
0.00 3.78 0.00
0.00 3.75 0.00
0.00 3.71 0.00
0.00 3.67 0.00
0.00 3.62 0.00
0.00 3.57 0.00
0.00 3.52 0.00
0.00 3.47 0.00
0.00 3.42 0.00
0.00 3.36 0.00
0.00 3.30 0.00
0.00 3.23 0.00
0.00 3.17 0.00
0.00 3.10 0.00
0.00 3.02 0.00
0.00 2.95 0.00
0.00 2.87 0.00
0.00 2.79 0.00
0.00 2.71 0.00
0.00 2.62 0.00
```

Figure 5.1: The running HelloBox2D shows decreasing height from 4 to 1. The computation is the core of Box2D, to shows the visualization like at the testbed, by default Box2D uses imgui, but you can also use your own GUI besides imgui (file for this example in folder: DFSimulatorC/Source Codes/C++/ch5-box2d-helloworld).

ii. Build Box2D Testbed

For simplicity instead of putting all the codes for Box2D testbed, I advise you to copy from my repository <https://github.com/glanzkaiser/DFSimulatorC/Source Codes/C++/ch5-box2d-testbed>, then go inside the directory and open the terminal then type:

```
mkdir build
```

```
cd build
```

```
cmake ..
```

```
make
```

./testbed You can configure the **CMakeLists.txt** so you can only generate the example you are concern about, less time compiling. The C++ source codes for every examples are located in **.. tests/**, how to modify them, you need to comprehend C++ around medium-level at least, and you can read the manual here:

<https://box2d.org/documentation/>

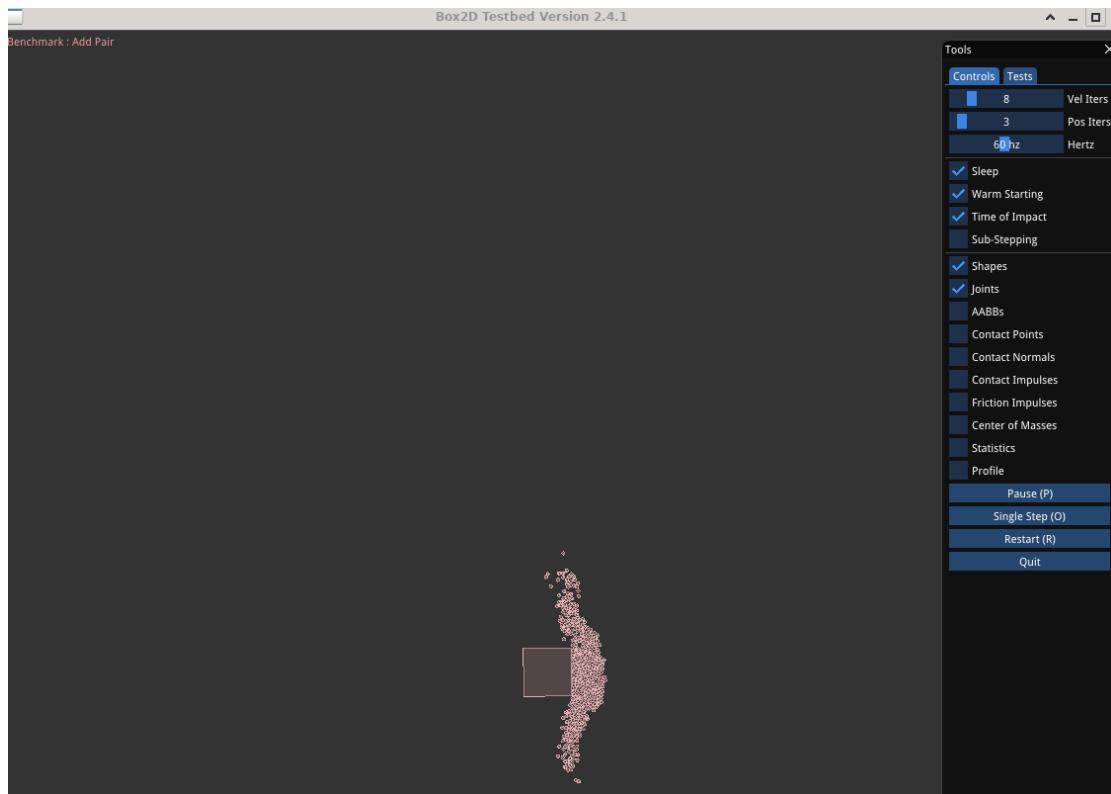


Figure 5.2: The testbed of Box2D, one of the beauty of imgui is that it can offers parameters changing and a lot of modification on the scene (files for this example in folder: `DFSimulatorC/Source Codes/C++/ch5-box2d-testbed`).

iii. Know-How in Box2D

- There are 3 body types in Box2D: Static, Kinematic, and Dynamic
- Box2D support motion and collisions with
 1. Body class provides the motion.
 2. Fixture and Shape classes are for collisions.
- Properties in class Body: Position, Linear Velocity, Angular Velocity, Body Type. In the header file (`./box2d/include/box2d/b2_body.h`) you can see all the body definitions under `struct B2_API b2BodyDef...`, you can edit this and recompile to recreate the library to adjust to your modification.

```

struct B2_API b2BodyDef
{
    /// This constructor sets the body definition default
     values.
b2BodyDef()
{
    position.Set(0.0f, 0.0f);
    angle = 0.0f;
    linearVelocity.Set(0.0f, 0.0f);
    angularVelocity = 0.0f;
    linearDamping = 0.0f;
    angularDamping = 0.0f;
    allowSleep = true;
    awake = true;
    fixedRotation = false;
    bullet = false;
    type = b2_staticBody;
    enabled = true;
    gravityScale = 1.0f;
}
...
}
```

- In Box2D we have impulse as force times 1 second.
- Motion depends on Force, current velocity, and mass

$$\begin{aligned}
 \Delta s &= v \Delta t \\
 &= v_0 \Delta t + \frac{1}{2} a (\Delta t)^2 \\
 &= v_0 \Delta t + \frac{1}{2} \left(\frac{F}{m} \right) (\Delta t)^2
 \end{aligned}$$

The mass comes from the Fixture class. Fixture gives volume to the body.

- Four ways to move a Dynamic body:
 1. Forces: `applyForce` (linear), `applyTorque` (angular).
For joints, complex shapes. Hard to control.
 2. Impulses: `applyLinearImpulse` (linear), `applyAngularImpulse` (angular).
Great for joints, complex shapes. Extremely hard to control.

- 3. Velocity: **setLinearVelocity** (linear), **setAngularVelocity** (angular)
Very easy to control, not for joints, complex shapes.
- 4. Translation: **setTransform**
- Shape stores the object geometry (either boxes, circles or polygons, it must be convex). Shape also stores object density, which mass is area times density. The higher the density, the higher the mass.

```

bodydef = newBodyDef();
bodydef.type = type;
bodydef.position.set(position);
bodydef.angle = angle;

body1 = world.createBody(bodydef);

bodydef.position.set(position2);
body2 = world.createBody(bodydef);

shape1 = new PolygonShape();
shape2 = new PolygonShape();
shape1.set(verts1);
shape2.set(verts2);

fixdef = new FixtureDef();
fixdef.density = density;

fixdef.shape = shape1;
fixture1 = body1.createFixture(fixdef);
fixdef.shape = shape2;
fixture2 = body1.createFixture(fixdef);

```

- Size in Box2D:
 - 1. 1 Box2D unit = 1 meter
 - 2. 1 density = 1 kg/m^2
- Fixture has only one body, while bodies have many fixtures.
- Properties in Box2D:
 - 1. Friction: stickiness
 - 2. Restitution: bounciness

Both value should be within 0 to 1.
- For custom collisions you can use **ContactListeners**, with two primary methods in interface:
`beginContact`: When objects first collide
`endContact`: When objects no longer collide

It can be used for color changing in Box2D.

- Collision filtering can be used to define what can collide with certain fixture.
- Joints connect bodies, and they are affected by fixtures. Joints must use force or impulse, manual velocity might violate constraints.

- Box2D tends to allocate a large number of small objects (around 50-300 bytes). Using the system heap through malloc or new for small objects is inefficient and can cause fragmentation. Box2D's solution is to use a small object allocator (SOA). Since Box2D uses a SOA, you should never new or malloc a body, fixture, or joint. However, you do have to allocate a **b2World** on your own.

II. BULLET

Bullet Physics is a professional open source collision detection, rigid body and soft body dynamics library. The library is free for commercial use under the ZLib license.

You can read more about Bullet here: bulletphysics.org

i. Install Bullet Library and Include Files / Headers

To get the library, open the terminal and type:

```
git clone https://github.com/bulletphysics/bullet3.git
```

Enter the directory, then type:

```
mkdir build  
cd build  
cmake ..  
make -j4 (build with 4 cores, faster than just make)
```

If you want to use manual typing, from the bullet directory, type:

```
mkdir build &&  
$ cd build &&  
$ cmake --DCMAKE_INSTALL_PREFIX=/opt/hamzstlib/Physics/bulletinstall \  
$ -DBUILD_SHARED_LIBS=ON .. &&  
$ make
```

after finish compiling then type:

```
make install
```

It will install Bullet' libraries and include files at the install prefix place.

To get more information on the installation directory, where Bullet installs the libraries and search for the include files go to `./lib/cmake/bullet/` and read the **BulletConfig.cmake**, make sure it is not clashing with our own **LIBRARY_PATH**. You can see below on create an example for Bullet, how I set the path the same as the **BulletConfig.cmake** to find the libraries and include files for Bullet.

ii. Know-How in Bullet

- Bullet can do discrete and continuous collision detection including ray and convex sweep test. Collision shapes include concave and convex meshes and all basic primitives.
- The main task of a physics engine is to perform collision detection, resolve collisions and other constraints, and provide the updated world transform for all the objects.

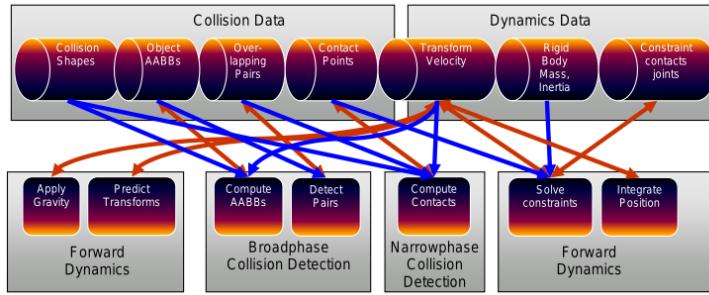


Figure 5.3: The pipeline for Bullet physics is executed from left to right, starting by applying gravity, and ending by position integration, updating the world transform.

- The most important data structures and computation stages for rigid body physics: The entire physics pipeline computation and its data structures are represented in Bullet by a dynamics world. When performing **stepSimulation** on the dynamics world, all the above stages are executed. The default dynamics world implementation is the **btDiscreteDynamicsWorld**.
- How to create a **btDiscreteDynamicsWorld**, **btCollisionShape**, **btMotionState**, and **btRigidBody**. Each frame call the **stepSimulation** on the dynamics world, and synchronize the world transform for your graphics object, the requirements:
 1. Put `#include "btBulletDynamicsCommon.h"` in your source file.
 2. Required include path: **Bullet/src** folder.
 3. Required libraries: **BulletDynamics**, **BulletCollision**, **LinearMath**.
- The derivations from **btDynamicsWorld** are **btDiscreteDynamicsWorld** and **btSoftRigidDynamicsWorld** will provide a high level interface that manages your physics objects and constraints. It also implements the update of all objects each frame.
- To construct a **btRigidBody** or **btCollisionObject** you need to provide:
 1. Mass, positive for dynamics moving objects and 0 for static objects
 2. Collision shape (Box, Sphere, Cone, Convex Hull, Triangle Mesh)
 3. Material properties (friction and restitution)

Then update each frame with **stepSimulation**, call it on the dynamics world. The **btDiscreteDynamicsWorld** automatically takes into account variable timestep by performing interpolation instead of simulation for small timesteps. It uses an internal fixed timestep of 60 Hertz. The **stepSimulation** will perform collision detection and physics simulation. It updates the world transform for active objects by calling **btMotionState's setWorldTransform**.

iii. Create an Example with Bullet

First, in order to link the library correctly and find the include files for Bullet, you will need to modify the export file by typing at terminal in :

```
cd
vim export
```

```
export prefix="/usr"
export hamzstlib="/opt/hamzstlib"
```

```

export physics="$hamzstlib/Physics"

# For library (.so, .a)
export LIBRARY_PATH="/usr/lib:$hamzstlib/lib:$physics/bulletinstall/lib"

export PKG_CONFIG_PATH="$physics/bulletinstall/lib/pkgconfig:$hamzstlib/lib
/pkgconfig:"

```

The **bulletinstall** is the folder containing the **include**, **lib**, **output** directories after compiling Bullet.

Thus, it will be able to find the Bullet' libraries that have been built / compiled in GFreya OS. We are going to use the libraries **-lBulletDynamics -lBulletCollision -lLinearMath**.

Now, in a working directory (assumed still empty) create a file as usual, type:
vim main.cpp

C++ Code 24: *main.cpp "Bullet Example"*

III. REACTPHYSICS3D

ReactPhysics3D is an open source C++ physics engine library that can be used in 3D simulations and games. The library is released under the Zlib license. ReactPhysics3D is using OpenGL.

i. Install ReactPhysics3D Library and Include Files / Headers

To get the library, open the terminal and type:

git clone https://github.com/DanielChappuis/reactphysics3d.git

Now to build it, enter the downloaded repository, then type:
mkdir build

cd build

ccmake ..

Choose to build the library, then install it at the **/usr/lib**, so it can be used later on.

ii. Know-How in ReactPhysics3D

- The first thing you need to do when you want to use ReactPhysics3D is to instantiate the **PhysicsCommon** class. This class is used as a factory to instantiate one or multiple physics worlds and other objects. It is also responsible for logging and memory management. To create a physics world:

PhysicsWorld* world = physicsCommon.createPhysicsWorld();

This method will return a pointer to the physics world that has been created. How to create the world settings:

```

PhysicsWorld::WorldSettings settings;
settings.defaultVelocitySolverNbIterations = 20;
settings.isSleepEnabled = false;
settings.gravity = Vector(0, -9.81, 0)

PhysicsWorld* world = physicsCommon.createPhysicsWorld();

```

this will create the physics world with your own settings.

- You probably only need one physics world with multiple objects.
- Simulating many bodies is cost expensive, thus sleeping technique is used to deactivate resting bodies. A body is put to sleep when its' linear and angular velocity stay under a given velocity threshold for certain amount of time.
- The base memory allocations in ReactPhysics3D are done using the `std::malloc()` and `std::free()` methods.
- Some methods that you can use:

1. **testOverlap()**

This group of methods can be used to test whether the colliders of two bodies overlap or not

2. **testCollision()**

This group of methods will give you the collision information (contact points, normals, etc) for colliding bodies.

3. **testPointInside()**

This method will tell you if a 3D point is inside a given CollisionBody, RigidBody, or Collider.

- Types of a Rigid Body:

1. **Static body**

A static body has infinite mass, zero velocity but its position can be changed manually. A static body does not collide with other static or kinematic bodies. You can use this as a floor or ground.

2. **Kinematic body**

A kinematic body has infinite mass, its' velocity can be changed manually and its' position computed by the physics engine. A kinematic body does not collide with other static or kinematic bodies.

3. **Dynamic body**

A dynamic body has non-zero mass, with velocity determined by forces and its' position is determined by the physics engine. A dynamic body can collide with other dynamic, static or kinematic bodies. You can use this as a rock or apple or any kind of object in the physics world.

You can create a rigid body then change the type of RigidBody with:

```

Vector3 position(0.0, 3.0, 0.0);
Quaternion orientation = Quaternion::identity();
Transform transform(position, orientation);

RigidBody* body = world -> createRigidBody(transform);
body->setType(BodyType::KINEMATIC);

```

- Damping is the effect of reducing the velocity of the rigid body during the simulation effects like air friction for instance. By default, no damping is applied. Without angular damping a pendulum will never comes to rest. You can choose to damp the linear or/and the angular velocity of a rigid body. You need to use:

RigidBody::setLinearDamping()

RigidBody::setAngularDamping()

the damping value has to be positive. The value of 0 means no damping at all.

Chapter 6

DFSimulatorC++ 0: C+++ = C++ + Gnuplot + SymbolicC++

"Bring that book, the one that is written your name on it, replacing the real author name (on College Geometry book)" - Sentinel

I put this chapter before the real physics simulation, because after we do simulation and want to analyze more, we need to be able to plot the data, clean the data, manipulate the data if there are lots of NaN or outliers, I think Gnuplot that can be called directly from C++ offers a great promise in the future of scientific computing.

In this chapter, we also want to introduce a library for C++, SymbolicC++ [7] that uses C++ and object-oriented programming to develop a computer algebra system. SymbolicC++ is a general purpose computer algebra system written in the programming language C++. It is free software released under the terms of the GNU General Public License. SymbolicC++ is used by including a C++ header file or by linking against a library.

Object-oriented programming is an approach to software design that is based on classes rather than procedures. This approach maximizes modularity and information hiding. Object-oriented design provides many advantages. For example, it combines both the data and the functions that operate on that data into a single unit. Such a unit (abstract data type) is called a class. This library can be used for symbolic computations in mathematics, like determining a derivative or integral of a function. If there is SymPy for Python, then SymbolicC++ is for C++.

In the first version of SymbolicC++ the main data type for symbolic computation was the Sum class. The list of available classes included:

- Verylong : An unbounded integer implementation
- Rational : A template class for rational numbers
- Quaternion : A template class for quaternions
- Derive : A template class for automatic differentiation
- Vector : A template class for vectors (see vector space)
- Matrix : A template class for matrices (see matrix (mathematics))
- Sum : A template class for symbolic expressions

The second version of SymbolicC++ featured new classes such as the Polynomial class and initial support for simple integration. The third version features a complete rewrite of SymbolicC++ and was released in 2008. This version encapsulates all symbolic expressions in the Symbolic class.

The greatness of SymbolicC++ combined with Gnuplot and C++ is that all that I have done with Julia to create Riemann sum plot, plot integral of a function, plot derivative of a function, can also be done in C++, more codes but more rewarding, since it is very fast, and not taking too many storage, I compare this with my Julia installation, I add a lot of packages while in Qemu for GFreya OS, do Rsync to move the downloaded packages into GFreya OS in dual boot (that can't be connected to internet so people using GFreya OS can focus more on either coding or writing books, not finding something to buy in internet), turns out the `./julia` folder is 14 GB with packages that I have added including: **SymPy**, **PrettyTables**, **OrdinaryDiffEq**, **Optim**, **CairoMakie**, **Makie**, and more. Now let see **gnuplot_i.hpp**, a header file that is needed to plot with Gnuplot with C++ code is only 58 kb, and all the SymbolicC++ headers are 294 kb only.

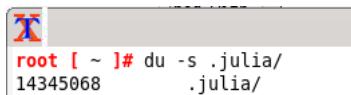


Figure 6.1: The size of `.julia` in GFreyaOS is 14 GB as of November 16, 2023.

All the codes related to this chapter can be obtained in the repository:
<https://github.com/glanzkaiser/DFSimulatorC>
 (Find them under this directory `../DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++`)

I have put the libraries in the repository as well, but you can also find it yourself in internet:

- To obtain **gnuplot_i.hpp**

You need to download the interfacing code from here: <https://code.google.com/archive/p/gnuplot-cpp/>.

- To obtain **gnuplot-iostream**

You can get it here: <https://github.com/dstahlke/gnuplot-iostream>.

For more advanced usage of Gnuplot you need to install Boost library. Always remember **gnuplot-iostream** relies on the Boost library, we will always link to Boost library when using this header. Boost is a very common library, but it doesn't come together with the C++ compiler, so make sure it is properly installed. Boost package compilation and installation is covered in BLFS book, see Linux From Scratch book. Fortunately, Boost is installed in GFreya OS used in this book, so we are all covered.

- To obtain **SymbolicC++** headers

You can find it at the official website: <http://issc.uj.ac.za> and to download it directly you can go here: <https://symboliccpp.sourceforge.net/>.

(Users of SymbolicC++ with GCC on 64-bit may need to use the `-fno-elide-constructors` flag.)

I. PLOT A SLOPE WITH GNUPLOT FROM C++

We start with Gnuplot waltz with C++, since it only takes a simple header to be included in the working directory to plot any kind of function even in 3 dimension. First, we will learn how to plot a slope $y = x$, first create the file name **main.cpp**

```
#include <iostream>
#include "gnuplot_i.hpp" //Gnuplot class handles POSIX-Pipe-
                      communication with Gnuplot

#if defined(WIN32) || defined(_WIN32) || defined(__WIN32__)
    #include <conio.h> //for getch(), needed in wait_for_key()
    #include <windows.h> //for Sleep()
    void sleep(int i) { Sleep(i*1000); }
#endif

#define SLEEP_LGTH 2 //sleep time in seconds
#define NPOINTS 50 //length of array

void wait_for_key(); // Programm halts until keypress

using std::cout;
using std::endl;

int main(int argc, char* argv[])
{
    cout << "*** example of gnuplot control through C++ ***" <<
        endl << endl;

    //
    // Using the GnuplotException class
    //
    try
    {
        Gnuplot g1("lines");

        //
        // Slopes
        //
        cout << "*** plotting slopes" << endl;
        g1.set_title("Slope Plotting");

        cout << "y = x" << endl;
        g1.plot_slope(1.0,0.0,"y=x");
        g1.showonscreen(); // window output

        wait_for_key();
    }
}
```

```

        }

        catch (GnuplotException ge)
        {
            cout << ge.what() << endl;
        }

        cout << endl << "*** end of gnuplot example" << endl;

        return 0;
    }

void wait_for_key ()
{
    #if defined(WIN32) || defined(_WIN32) || defined(__WIN32__)
        defined(__TOS_WIN__)
        // every keypress registered, also
        arrow keys
    cout << endl << "Press any key to continue..." << endl;

    FlushConsoleInputBuffer(GetStdHandle(STD_INPUT_HANDLE));
    _getch();
    #elif defined(unix) || defined(__unix) || defined(__unix__)
        defined(__APPLE__)
    cout << endl << "Press ENTER to continue..." << endl;

    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
    std::cin.get();
    #endif
    return;
}

```

C++ Code 25: main.cpp "Slope plotting Gnuplot C++"

Now copy **gnuplot_i.hpp** from the repository under the include folder, you will find it here:
./Source Codes/include/

```

#ifndef _GNUPLOT_PIPES_H_
#define _GNUPLOT_PIPES_H_


#include <iostream>
#include <string>
#include <vector>
#include <fstream>
#include <sstream> // for std::ostringstream
#include <stdexcept>

```

```

#include <cstdio>
#include <cstdlib> // for getenv()
#include <list> // for std::list

#if defined(WIN32) || defined(_WIN32) || defined(__WIN32__)
    __TOS_WIN__
#endif //defined for 32 and 64-bit environments
#include <iostream> // for cout, endl
#define GP_MAX_TMP_FILES 27 // 27 temporary files it's Microsoft restriction
#elif defined(unix) || defined(__unix) || defined(__unix__)
    __APPLE__
#endif //all UNIX-like OSs (Linux, *BSD, Mac OSX, Solaris, ...)
#include <unistd.h> // for access(), mkstemp()
#define GP_MAX_TMP_FILES 64
#else
#error unsupported or unknown operating system
#endif

//declare classes in global namespace

class GnuplotException : public std::runtime_error
{
public:
    GnuplotException(const std::string &msg) : std::runtime_error(msg){}
};

...
...

void Gnuplot::remove_tmpfiles(){
    if ((tmpfile_list).size() > 0)
    {
        for (unsigned int i = 0; i < tmpfile_list.size(); i++)
            remove( tmpfile_list[i].c_str() );

        Gnuplot::tmpfile_num -= tmpfile_list.size();
    }
}
#endif

```

C++ Code 26: gnuplot_i.hpp

I only put the head and tail of the source code since it took 50 pages of this book just to show the full codes of **gnuplot_i.hpp**, those who wrote the source codes is amazing.

Put **gnuplot_i.hpp** and **main.cpp** under one same folder / working directory then type:
g++ -o main main.cpp
./main

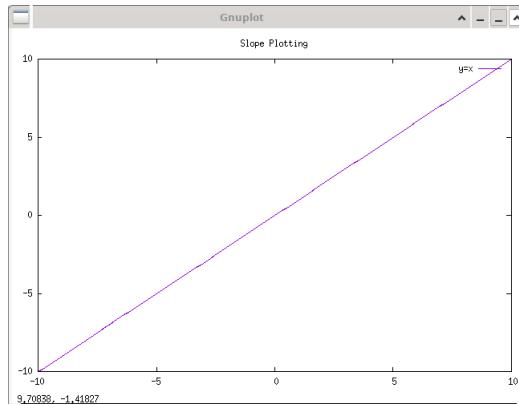


Figure 6.2: The plot of a slope $y = x$ with Gnuplot inside C++ you can move the view from the GUI with arrow key press on your keyboard (the plot code can be located in: *DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++/C++ Gnuplot/Slope Plotting/main.cpp*).

I get two warnings but can still run the example. If you want to use Makefile create a Makefile in the current working directory:

```
CFLAGS = -ggdb
DEFINES = -DDEBUGGA
INCLUDES =
LIBS = -lstdc++
MAIN = main.o
CC=g++

.cc.o:
$(CC) -c $(CFLAGS) $(DEFINES) $(INCLUDES) <

all:: main

gnuplot_i.o: gnuplot_i.hpp
main.o: main.cpp

main: $(MAIN)
$(CC) -o $@ $(CFLAGS) $(MAIN) $(LIBS)

clean:
rm -f $(MAIN) main
```

C++ Code 27: Makefile "Gnuplot Slope Makefile"

now you can type:

make
./make

That, is another way to compile besides **g++ -o main main.cpp**, sometimes for big projects Makefile is more preferred.

II. PLOT A 2D FUNCTION WITH GNUPLOT FROM C++

We will be able to learn how to plot function, not only trigonometric, but also transcendental functions (exponential and logarithm). First create the file name **main.cpp** in a new working directory

```
#include <iostream>
#include "gnuplot_i.hpp" //Gnuplot class handles POSIX-Pipe-communication
with Gnuplot

#if defined(WIN32) || defined(_WIN32) || defined(__WIN32__)
__TOS_WIN__
#endif
#include <conio.h> //for getch(), needed in wait_for_key()
#include <windows.h> //for Sleep()
void sleep(int i) { Sleep(i*1000); }
#endif

#define SLEEP_LGTH 2 // sleep time in seconds
#define NPOINTS 50 // length of array

void wait_for_key(); // Programm halts until keypress

using std::cout;
using std::endl;

int main(int argc, char* argv[])
{
    cout << "*** example of gnuplot control through C++ ***" << endl <<
    endl;

    try
    {
        Gnuplot g1("lines");

        cout << "*** plotting slopes" << endl;
        g1.set_title("Function Plotting");

        //cout << "y = sin(x)" << endl;
        //g1.plot_equation("sin(x)","sine");

        //cout << "y = log(x)" << endl;
        //g1.plot_equation("log(x)","logarithm");

        //cout << "y = exp(x) * tan(2*x)" << endl;
        //g1.plot_equation("exp(x)*tan(2*x)","exponential and
        //trigonometric product");

        cout << "y = sin(x) * cos(2*x)" << endl;
    }
}
```

```

        g1.plot_equation("sin(x)*cos(2*x)","sine product");

        g1.showonscreen(); // window output

        wait_for_key();

    }

    catch (GnuplotException ge)
    {
        cout << ge.what() << endl;
    }

    cout << endl << "*** end of gnuplot example" << endl;

    return 0;
}

void wait_for_key ()
{
    #if defined(WIN32) || defined(_WIN32) || defined(__WIN32__)
    // every keypress registered, also arrow keys
    cout << endl << "Press any key to continue..." << endl;

    FlushConsoleInputBuffer(GetStdHandle(STD_INPUT_HANDLE));
    _getch();
    #elif defined(unix) || defined(__unix) || defined(__unix__)
    //APPLE__
    cout << endl << "Press ENTER to continue..." << endl;

    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
    std::cin.get();
    #endif
    return;
}

```

C++ Code 28: main.cpp "Function plotting Gnuplot C++"

Put **gnuplot_i.hpp** and **main.cpp** under one same folder / working directory then type:
g++ -o main main.cpp
./main

There are four choices here, and you can uncomment three of them and only plot 1 out of the three that you want to plot:

```

//cout << "y = sin(x)" << endl;
//g1.plot_equation("sin(x)","sine");

//cout << "y = log(x)" << endl;
//g1.plot_equation("log(x)","logarithm");

```

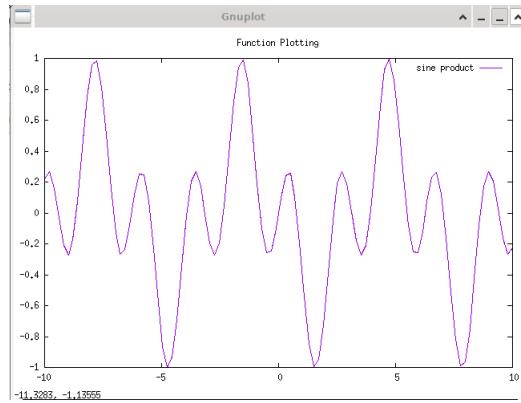


Figure 6.3: The plot of a function $\sin(x) \cos(2x)$ with Gnuplot inside C++ (the plot code can be located in: **DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++/C++ Gnuplot/Function Plotting/main.cpp**).

```
//cout << "y = exp(x) * tan(2*x)" << endl;
//g1.plot_equation("exp(x)*tan(2*x)","exponential and trigonometric product
 //");
cout << "y = sin(x) * cos(2*x)" << endl;
g1.plot_equation("sin(x)*cos(2*x)","sine product");
```

The makefile from previous section is also working on this code.

III. PLOT A 3D FUNCTION WITH GNUPLOT FROM C++

After some warming up, now how about plotting a surface plot? Building Information Modeling like AutoDesk and AutoCad are all in 3 dimension, so the user able to simulate a building or highway and to apply a lot of tests for the standard quality determination. This is the very first step before one day maybe we can create something better than AutoDesk and AutoCad for all engineering and science branches.

From this section there are two important things you need to know:

1. Whenever you want to call for Gnuplot, always remember that you need to have **gnuplot_i.hpp** in your working directory, along with the C++ source codes.
2. the Makefile from the last two sections can be used in this section and below related to calling Gnuplot from C++.

Now, in a new working directory, create a **main.cpp**.

```
#include <iostream>
#include "gnuplot_i.hpp" //Gnuplot class handles POSIX-Pipe-communication
with Gnuplot

#define SLEEP_LGTH 2 // sleep time in seconds
#define NPOINTS 50 // length of array
```

```

void wait_for_key(); // Programm halts until keypress

using std::cout;
using std::endl;

int main(int argc, char* argv[])
{
    cout << "*** example of gnuplot control through C++ ***" << endl <<
        endl;

    try
    {
        Gnuplot g1("lines");
        cout << "window: splot with hidden3d" << endl;
        g1.set_isosamples(25).set_hidden3d();
        g1.plot_equation3d("x**2+y*y");

        wait_for_key();

    }
    catch (GnuplotException ge)
    {
        cout << ge.what() << endl;
    }

    cout << endl << "*** end of gnuplot example" << endl;

    return 0;
}

void wait_for_key ()
{
    cout << endl << "Press ENTER to continue..." << endl;

    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
    std::cin.get();
    return;
}

```

C++ Code 29: *main.cpp "3D Surface plotting Gnuplot C++"*

From the working directory type:

g++ -o main main.cpp
./main

Compared to the last two source codes, since this book is using GFreya OS, thus I delete the

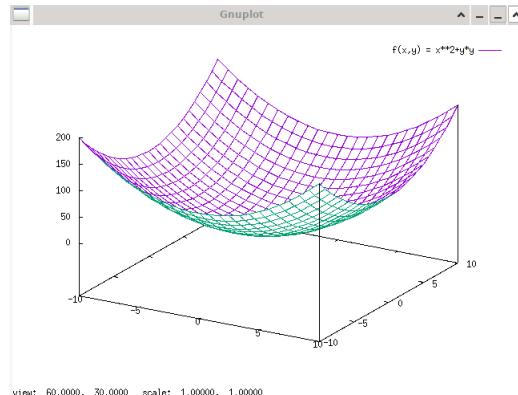


Figure 6.4: The 3-dimensional plot of a surface $f(x,y) = x^2y^2$ with Gnuplot inside C++ (the plot code can be located in: `DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++/C++ Gnuplot/3D Surface Plotting/main.cpp`).

commands that are used for Windows OS to save more space, thus if you are using Microsoft Windows, see the last two source codes and copy what I have deleted into your source codes.

IV. PLOT A 3D CURVE / LINE FROM USER DEFINED POINTS WITH GNUPLOT FROM C++

Now, this is more fun, since in this section, we learn if we have an input points of x, y, z , then we can plot a line or curve out of them.

In a new working directory, create **main.cpp** file.

```
#include <iostream>
#include "gnuplot_i.hpp" //Gnuplot class handles POSIX-Pipe-communication
with Gnuplot

#define SLEEP_LGTH 2 // sleep time in seconds
#define NPOINTS 50 // length of array

void wait_for_key(); // Programm halts until keypress

using std::cout;
using std::endl;

int main(int argc, char* argv[])
{
    cout << "*** example of gnuplot control through C++ ***" << endl <<
    endl;

    try
    {
```

```

        std::vector<double> x, y, z;

        for (int i = 0; i < NPOINTS; i++) // fill double arrays x, y,
        {
            z
            {
                x.push_back((double)i); // x[i] = i
                y.push_back((double)i * (double)i); // y[i] = i^2
                z.push_back( x[i]*y[i] ); // z[i] = x[i]*y[i] = i^3
            }
            Gnuplot g1("lines");
            cout << endl << endl << "*** user-defined lists of points (x
                ,y,z)" << endl;
            g1.unset_grid();
            g1.plot_xyz(x,y,z,"user-defined points 3d");

            wait_for_key();

        }
        catch (GnuplotException ge)
        {
            cout << ge.what() << endl;
        }

        cout << endl << "*** end of gnuplot example" << endl;

        return 0;
    }

    void wait_for_key ()
    {
        cout << endl << "Press ENTER to continue..." << endl;

        std::cin.clear();
        std::cin.ignore(std::cin.rdbuf()->in_avail());
        std::cin.get();
        return;
    }
}

```

C++ Code 30: *main.cpp "3D Curve plotting Gnuplot C++"*

From the working directory type:

```
g++ -o main main.cpp
./main
```

The **NPOINTS** defined at the beginning is the number of points that are going to be plotted or the length of array, the higher the number, the smoother the curve.

If you want to plot the points only instead of curve connecting the curve, you can replace

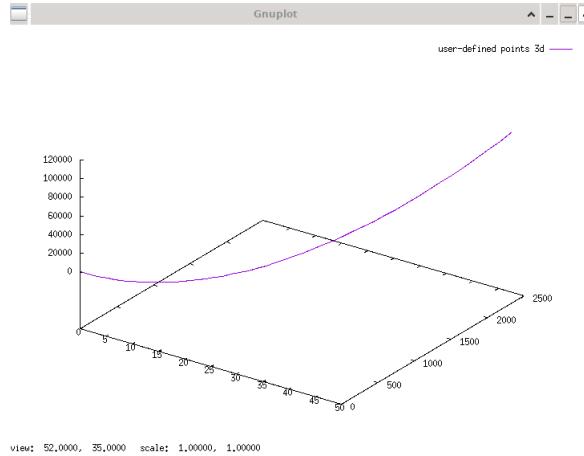


Figure 6.5: The 3-dimensional plot of a line connecting 50 points defined by user with Gnuplot inside C++ (the plot code can be located in: *DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++/C++ Gnuplot/3D Line from Points/main.cpp*).

```
cout << endl << endl << "*** user-defined lists of points (x,y,z)" << endl
;
g1.unset_grid();
g1.plot_xyz(x,y,z,"user-defined points 3d");
```

with

```
g1.set_style("points").plot_xy(x,y,"user-defined points 2d");
cout << endl << endl << "*** user-defined lists of points (x,y,z)" << endl
;
g1.unset_grid();
g1.plot_xyz(x,y,z,"user-defined points 3d");
```

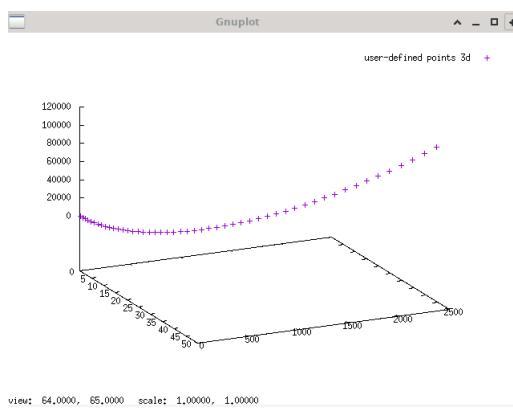


Figure 6.6: The 3-dimensional plot of 50 points defined by user with Gnuplot inside C++ (the plot code can be located in: *DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++/C++ Gnuplot/3D Plot xyz Points/main.cpp*).

There are a lot of other Gnuplot examples that can be called from C++ source codes in the

repository in `../DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++/C++ Gnuplot/`, you can check it out yourselves.

V. PLOT DATA FROM TEXTFILE IN 2D WITH GNUPLOT

Now, let the fun begins. When we play with Box2D and we can print the output of the velocity of the object, or the kinetic energy, or other parameter, it comes in textfile (.txt), there are two ways to plot the result from txt:

1. By calling gnuplot directly from terminal at the working directory containing the textfile
2. By using C++ code to plot with Gnuplot through C++

We are going to examine the second option, it is longer, but compared to the possibilities it opens, it has tremendous applications. First, we can combine with SymbolicC++ and thus interpolate the graph with Fourier Series or Cubic Spline after plotting the data, we can also find the derivative of the interpolated function. Option 1 is the easiest, it only takes 1 line to plot the velocity of an object, but then to do deeper analysis on the variable/s related, that will come to differential equation or partial differential equation, like solving double pendulum problem, option 2 is way to go for that.

First, in a new working directory, create a textfile **matrix.txt**

```
1.1
2.4
3.5
1.4
5.5
6.7
3.2
4.7
7.5
-9.7
-20
```

C++ Code 31: *matrix.txt "2D Plot from Textfile with Gnuplot through C++"*

It is only an example, you can print your own output from Box2D simulation or another Physics Engine / simulation.

Then, create the C++ code **main.cpp**.

```
// Use fstream to read from a file, and ofstream to write to a file.
// g++ main.cpp

#include <iostream>
#include <fstream>
#include <string>
#include "gnuplot_i.hpp" //Gnuplot class handles POSIX-Pipe-communication
with Gnuplot
```

```

#define SLEEP_LGTH 2 // sleep time in seconds
#define NPOINTS 11 // length of array

const int numRows=11;
const int numCols=1;

void wait_for_key(); // Programm halts until keypress

using std::cout;
using std::endl;
using namespace std;

int main(int argc, char* argv[])
{
    cout << "*** Example of Gnuplot plot data from txt through C++ ***"
        << endl << endl;
    std::ifstream in("matrix.txt");
    float tiles[numRows][numCols];
    std::vector<double> y;
    for (int i = 0; i < numRows; i++)
    {
        for (int j = 0; j < numCols; j++)
        {
            in >> tiles[i][j]; // get data row i column j from the
            // textfile
            cout << tiles[i][j] << ' ' << endl;
        }
        y.push_back((float)tiles[i][0]); // Thanks for telling me
        // this trick Freya.. I owe you a lot.
    }

    try
    {
        std::vector<double> x;
        for (int i = 0; i < NPOINTS; i++) // fill double array x
        {
            x.push_back((double)i); // x[i] = i
        }
        Gnuplot g1("lines");
        cout << endl << endl << "*** user-defined lists of points (x
            ,y)" << endl;
        g1.set_grid();
        g1.set_style("linespoints").plot_xy(x,y,"(x,y)");

        wait_for_key();

    }
    catch (GnuplotException ge)
}

```

```

    {
        cout << ge.what() << endl;
    }

    cout << endl << "*** end of gnuplot example" << endl;

    return 0;
}

void wait_for_key ()
{
    cout << endl << "Press ENTER to continue..." << endl;

    std::cin.clear();
    std::cin.ignore(std::cin.rdbuf()->in_avail());
    std::cin.get();
    return;
}

```

C++ Code 32: *main.cpp* "2D Plot from Textfile with Gnuplot through C++"

Compile it by typing from terminal:

```
g++ -o result main.cpp
./result
```

You can also compile by using **Makefile**, create one in the current working directory.

```

CFLAGS = -ggdb
DEFINES = -DDEBUGGA
INCLUDES =
LIBS = -lstdc++
MAIN = main.o
CC=g++

.cc.o:
$(CC) -c $(CFLAGS) $(DEFINES) $(INCLUDES) $<

all:: main

gnuplot_i.o: gnuplot_i.hpp
main.o: main.cpp

main: $(MAIN)
$(CC) -o $@ $(CFLAGS) $(MAIN) $(LIBS)

clean:
rm -f $(MAIN) main

```

C++ Code 33: *Makefile* "2D Plot from Textfile with Gnuplot through C++"

Compile with Makefile by typing:

```
make
./main
```

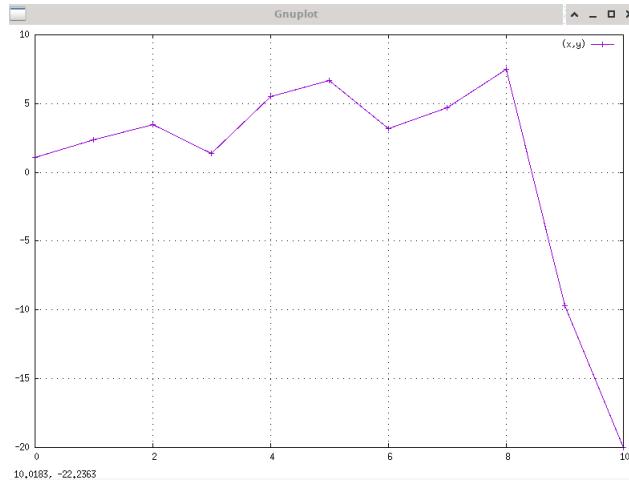


Figure 6.7: The plot from "matrix.txt" with Gnuplot through C++ (the code can be located in: *DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++/C++ Gnuplot/2D Plot from Textfile*).

VI. PLOT DATA AND FITTING CURVE FROM TEXTFILE IN 2D WITH "GNUPLOT-IOSTREAM.H"

In the previous sections, all examples are using header of "gnuplot_i.hpp", now we are going to use "gnuplot-iostream.h" to plot data from textfile input and then plot a fitting curve (interpolation method) of the form

$$f(x) = a + bx + cx^2$$

with a , b , and c are constants that can be determined by the fitting formula, it is built in from Gnuplot.

First, in a new working directory put "gnuplot-iostream.h" there. Then create the main C++ source code, **main.cpp**.

```
#include <fstream>
#include <vector>
#include <cmath>

#include "gnuplot-iostream.h"

const int numRows=11;
const int numCols=1;

using std::cout;
```

```

using std::endl;
using namespace std;

int main() {
    std::fstream in("matrix.txt");
    float tiles[numRows][numCols];
    std::vector<std::pair<double, double>> xy_pts;

    for (int i = 0; i < numRows; i++)
    {
        for (int j = 0; j < numCols; j++)
        {
            in >> tiles[i][j]; // get data row i column j from the
            // textfile
            cout << tiles[i][j] << ' ' << endl;
        }
        xy_pts.emplace_back(i, tiles[i][0]); // Thanks a lot
        // Beautiful Goddess, Freya!!!
    }
    Gnuplot gp;

    gp << "set xrange [0:12]\n";
    gp << "f(x) = a + b*x + c*x*x\n";
    gp << "fit f(x) '-' via a,b,c\n";
    gp.send1d(xy_pts);
    gp << "plot '-' with points title 'input', f(x) with lines title '"
        "fit'\n";
    gp.send1d(xy_pts);
}

```

C++ Code 34: *main.cpp* "2D Plot and Fit Curve from Textfile with Gnuplot through C++"

Now, you can create again the **matrix.txt** that containing 11 points of data (11×1 matrix or column vector with size of 11), read previous section for the corresponding textfile.

You can compile it by typing:

```
g++ -o main main.cpp -lboost_iostreams -lboost_system -lboost_filesystem
./main
```

Another alternative is by creating Makefile, so you can just type:
make
./main

Clean the object file with **make clean**, but you have to manually remove the log by typing:
rm -rf fit.log

```

CFLAGS = -ggdb
DEFINES = -DDEBUGGA
INCLUDES = gnuplot-iostream.h

```

```

LIBS = -lstdc++ -lboost_iostreams -lboost_system -lboost_filesystem
MAIN = main.o
CC=g++

.cc.o:
$(CC) -c $(CFLAGS) $(DEFINES) $(INCLUDES) $<

all:: main

gnuplot_i.o:
main.o: main.cpp

main: $(MAIN)
$(CC) -o $@ $(CFLAGS) $(MAIN) $(LIBS)

clean:
rm -f $(MAIN) main

```

C++ Code 35: Makefile "2D Plot and Fit Curve from Textfile with Gnuplot through C++"

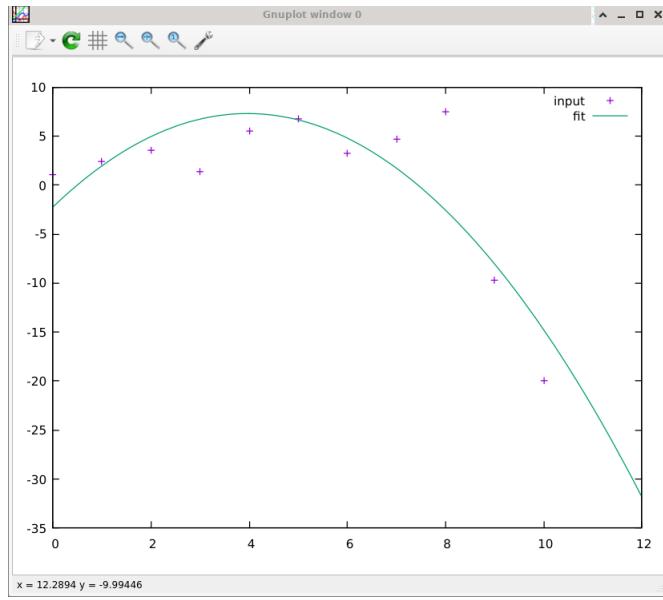


Figure 6.8: The plot and fitting curve from "matrix.txt" with Gnuplot through C++ with "gnuplot-iostream.h" and Boost (the code can be located in: *DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++/C++ Gnuplot/2D Plot and Fit Curve from Textfile*).

```
*****
Fri Nov 17 14:01:50 2023

FIT: data read from '-'
format = z
x range restricted to [0.00000 : 12.0000]
#datapoints = 11
residuals are weighted equally (unit weight)

function used for fitting: f(x)
f(x) = a + b*x + c*x*x
fitted parameters initialized with current variable values

iter  chisq  delta(lm) lambda a      b      c
0 3.6889999987e+04 0.00e+00 2.79e+01 1.000000e+00 1.000000e+00 1.000000e+00
5 1.8786273982e+02 -7.62e-06 2.79e-04 -2.267133e+00 4.812028e+00 -6.062937e-01

After 5 iterations the fit converged.
final sum of squares of residuals : 187.863
rel. change during last iteration : -7.61745e-11

degrees of freedom (FIT_NDF) : 8
rms of residuals (FIT_STDFIT) = sqrt(WSSR/ndf) : 4.84591
variance of residuals (reduced chisquare) = WSSR/ndf : 23.4828

Final set of parameters           Asymptotic Standard Error
=====                      =====
a      = -2.26713   +/- 3.692   (162.8%)
b      = 4.81203   +/- 1.718   (35.7%)
c      = -0.606294  +/- 0.1654  (27.29%)

correlation matrix of the fit parameters:
   a   b   c
a  1.000
b -0.816 1.000
c  0.672 -0.963 1.000
```

Figure 6.9: The fit.log file containing the details for the function and parameters used for curve fitting.

VII. COMPILE AND INSTALL SYMBOLICC++ LIBRARY

Download SymbolicC++ that can be used to be compiled into library (this file name is **SymbolicC++3-3.35-ac.tar.gz**, you can type from terminal:

wget http://sourceforge.net/projects/symboliccpp/files/
SymbolicC%2B%2B%203.35/
SymbolicC%2B%2B3-3.35-ac.tar.gz/download
 (to create the library)

or if you only want the headers file, type:

wget http://sourceforge.net/projects/symboliccpp/files/
SymbolicC%2B%2B%203.35/
SymbolicC%2B%2B3-3.35.tar.gz/download

all the files are also available in my repository under the directory **./DFSimulatorC/Source Codes/Libraries/**, we are going to proceed by creating the library, thus extract it by typing:

```
tar -xvf SymbolicC++3-3.35-ac.tar.gz
cd SymbolicC++3-3.35
./configure --prefix=/usr
make
make install
```

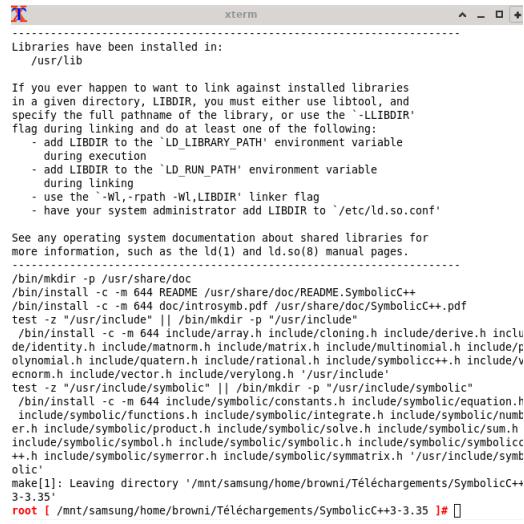
The include files installed from the method above produce error while compiling. The solution is (if you are using GFreya OS) make sure in **/usr/include/symbolic** you replace all of them from the include in my repository **DFSimulatorC/Source Codes/include/symbolic**, this include in my repository is obtained from the downloaded SymbolicC++ that you can find here:

DFSimulatorC/Source Codes/Libraries/SymbolicC++3-3.35.zip.

or from:

http://sourceforge.net/projects/symboliccpp/files/
SymbolicC%2B%2B%203.35/
SymbolicC%2B%2B3-3.35.tar.gz/download

(this zip/tar only contains lisp, examples and headers file for SymbolicC++ and can't be compiled



```

xterm
-----
Libraries have been installed in:
  /usr/lib

If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the '-LLIBDIR'
flag during linking and do at least one of the following:
  add LIBDIR to the 'LD_LIBRARY_PATH' environment variable
    during execution
  - add LIBDIR to the 'LD_RUN_PATH' environment variable
    during linking
  - use the '-WL,-rpath -WL,LIBDIR' linker flag
  - have your system administrator add LIBDIR to '/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.

/bin/mkdir -p /usr/share/doc
/bin/install -c -m 644 README /usr/share/doc/README.SymbolicC++
/bin/install -c -m 644 doc/introsymb.pdf /usr/share/doc/SymbolicC++.pdf
test -z "/usr/include" || /bin/mkdir -p "/usr/include"
/bin/install -c -m 644 include/array.h include/cloning.h include/derive.h include/
de/identity.h include/matnorm.h include/matrix.h include/multinomial.h include/p
olynomial.h include/quatern.h include/rational.h include/symbolicc++.h include/v
ectors.h include/vector.h include/verlong.h '/usr/include'
test -z "/usr/include/symbolic" || /bin/mkdir -p "/usr/include/symbolic"
/bin/install -c -m 644 include/symbolic/constants.h include/symbolic/equation.h
include/symbolic/functions.h include/symbolic/integrate.h include/symbolic/numb
er.h include/symbolic/product.h include/symbolic/solve.h include/symbolic/sym
++_h include/symbolic/syerror.h include/symbolic/symmatrix.h '/usr/include/sym
olic'
make[1]: Leaving directory '/mnt/samsung/home/browni/Téléchargements/SymbolicC++
3-3.35'
root [ /mnt/samsung/home/browni/Téléchargements/SymbolicC++3-3.35 ]# []

```

Figure 6.10: If you are successful building/compiling then install *SymbolicC++*, it will look like this.

to library)

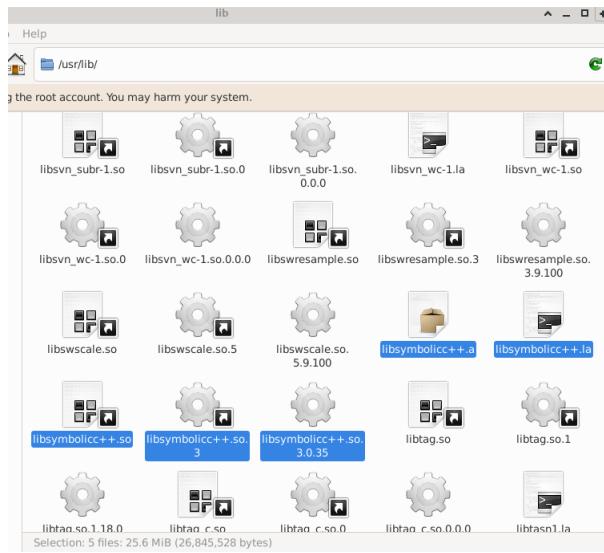


Figure 6.11: The libraries will be installed in /usr/lib.

VIII. SYMBOLIC DERIVATION AND INTEGRATION WITH SYMBOLICC++ LIBRARY

Now, for the real challenge we want to derivate and integrate a simple function $f(x) = x^3 + x^2$ with respect to x , it is a function of one variable, so it won't be too hard.

Create a file name **derivation.cpp**.

```
#include <iostream>
#include "symbolicc++.h"
using namespace std;

int main(void)
{
    Symbolic x("x");
    Symbolic y, dy;
    y = (x*x*x) + (x*x);
    cout << "y = " << y << endl;
    dy = df(y, x);

    cout << "derivative of y = " << dy << endl;
    return 0;
}
```

C++ Code 36: *derivation.cpp* "Symbolic Derivation for Function in One Variable C++"

In the working directory, open terminal and type:

```
g++ -o result derivation.cpp -lsymbolicc++
./result
```

Afterwards, create a file name **integral.cpp**.

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C++ Gnuplot SymbolicC++/SymbolicC++ ]# g++ -o result derivation.cpp -lsymbolic++
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C++ Gnuplot SymbolicC++/SymbolicC++ ]# ./result
y = x^(3)+x^(2)
derivative of y = 3*x^(2)+2*x
```

Figure 6.12: The result of differentiating $f(x) = x^3 + x^2$ with respect to x (the code can be located in: *DFSimulator-C/Source Codes/C++/C++ Gnuplot SymbolicC++/SymbolicC++/derivation.cpp*).

```
#include <iostream>
#include "symbolic++.h"
using namespace std;

int main(void)
{
    Symbolic x("x"), y;
    y = (x*x*x) + (x*x);
    cout << "y = " << y << endl;
    y = integrate(y,x);
    cout << "integral of y = " << y << endl;

    return 0;
}
```

C++ Code 37: *integral.cpp* "Symbolic Integration for Function in One Variable C++"

Now in the working directory, open terminal and type:

```
g++ -o integralresult integral.cpp -lsymbolic++
./integralresult
```

```
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C++ Gnuplot SymbolicC++/SymbolicC++ ]# g++ -o integralresult integral.cpp -lsymbolic++
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C++ Gnuplot SymbolicC++/SymbolicC++ ]# ./integralresult
y = x^(3)+x^(2)
integral of y = 1/4*x^(4)+1/3*x^(3)
```

Figure 6.13: The result of integrating $f(x) = x^3 + x^2$ with respect to x (the code can be located in: *DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++/SymbolicC++/integral.cpp*).

You can create a Makefile for this example, rename the cpp file into **main.cpp**, then create the Makefile.

```
CFLAGS = -ggdb
DEFINES = -DDEBUGGA
INCLUDES =
LIBS = -lstdc++ -lsymbolic++
MAIN = main.o
CC=g++

.ccpp.o:
$(CC) -c $(CFLAGS) $(DEFINES) $(INCLUDES) $<
```

```
all:: main

main.o: main.cpp

main: $(MAIN)
$(CC) -o $@ $(CFLAGS) $(MAIN) $(LIBS)

clean:
rm -f $(MAIN) main
```

C++ Code 38: Makefile "SymbolicC++ Example"

Now in the working directory, open terminal and type:

make
./main

To clean all files created after compiling (.o files and the binary executable file) type:
make clean

Chapter 7

DFSimulatorC++ I: Motion in Two Dimensions

"There's no need to rush things, there's much to explain." - Maria Traydor (Star Ocean 3)

I. POSITION, DISPLACEMENT, VELOCITY, AND ACCELERATION

The general way of locating a particle is starting with a position vector, that determined where the particle is located at certain time. If we denote \hat{r} as the position vector, it can be written in the unit-vector notation:

$$\hat{r} = x\hat{i} + y\hat{j} + z\hat{k}$$

where $x\hat{i}$, $y\hat{j}$, $z\hat{k}$ are the vector components of \hat{r} , and the coefficients x , y , and z are its scalar components.

If a particle undergoes a displacement $\Delta \vec{r}$ in time interval Δt , its average velocity \vec{v}_{avg} for that time interval is

$$\vec{v}_{avg} = \frac{\Delta \vec{r}}{\Delta t}$$

As Δt goes to 0, \vec{v}_{avg} reaches a limit called either the velocity or the instantaneous velocity \vec{v} :

$$\vec{v} = \frac{d\vec{r}}{dt}$$

which can be written in unit-vector notation as

$$\vec{v} = v_x\hat{i} + v_y\hat{j} + v_z\hat{k}$$

where the scalar components of \vec{v} are

$$v_x = \frac{dx}{dt}, \quad v_y = \frac{dy}{dt}, \quad v_z = \frac{dz}{dt}$$

If a particle's velocity changes from \vec{v}_1 to \vec{v}_2 in time interval Δt , its average acceleration during Δt is

$$\vec{a}_{avg} = \frac{\vec{v}_2 - \vec{v}_1}{\Delta t}$$

As Δt goes to 0, \overrightarrow{a}_{avg} reaches a limiting value called the instantaneous acceleration \overrightarrow{a} :

$$\overrightarrow{a} = \frac{d\overrightarrow{v}}{dt}$$

which can be written in unit-vector notation as

$$\overrightarrow{a} = a_x \hat{i} + a_y \hat{j} + a_z \hat{k}$$

where the scalar components of \overrightarrow{a} are

$$a_x = \frac{dv_x}{dt}, \quad a_y = \frac{dv_y}{dt}, \quad a_z = \frac{dv_z}{dt}$$

The basic equations for motion with constant acceleration are

$$v = v_0 + at \quad (7.1)$$

$$x - x_0 = v_0 t + \frac{1}{2} a t^2 \quad (7.2)$$

Another equations that can be used to solve any constant acceleration problem:

$$v^2 = v_0^2 + 2a(x - x_0) \quad (7.3)$$

$$x - x_0 = \frac{1}{2}(v_0 + v)t \quad (7.4)$$

$$x - x_0 = vt - \frac{1}{2}at^2 \quad (7.5)$$

To prove equation (7.1), we can write the indefinite integral from the definition of acceleration

$$\begin{aligned} dv &= a \, dt \\ \int dv &= \int a \, dt \\ \int dv &= a \int dt \\ v &= at + C \end{aligned}$$

To evaluate the constant C , we let $t = 0$, at which $v(0) = v_0$ (the initial value), thus

$$v_0 = a(0) + C$$

$$v_0 = C$$

hence we will obtain $v = v_0 + at$.

Now let's take the indefinite integral for the definition of velocity with respect to time

$$\begin{aligned} dx &= v \, dt \\ \int dx &= \int v \, dt \\ \int dx &= \int (v_0 + at) \, dt \\ \int dx &= v_0 \int dt + a \int t \, dt \\ x &= v_0 t + \frac{1}{2} a t^2 + K \end{aligned}$$

To find K , we set the initial value at $t = 0$ with $x(0) = x_0$, thus

$$x_0 = v_0(0) + \frac{1}{2}a(0)^2 + K$$

$$x_0 = K$$

we will obtain $x - x_0 = v_0 t + \frac{1}{2}at^2$, this proves equation (7.2).

II. PROJECTILE MOTION

We consider a special case in two-dimensional motion: when a particle moves in a vertical plane with some initial velocity \vec{v}_0 but its acceleration is always the freefall acceleration \vec{g} , which is downward. That particle is called projectile, mean that it is being projected or launched, and its motion is called projectile motion, like throwing a baseball into its ring. Ballistic, missile, a lot of sports like golf, tennis involve the study of projectile motion.

In projectile motion, a particle is launched into the air with a speed of \vec{v}_0 and at an angle θ_0

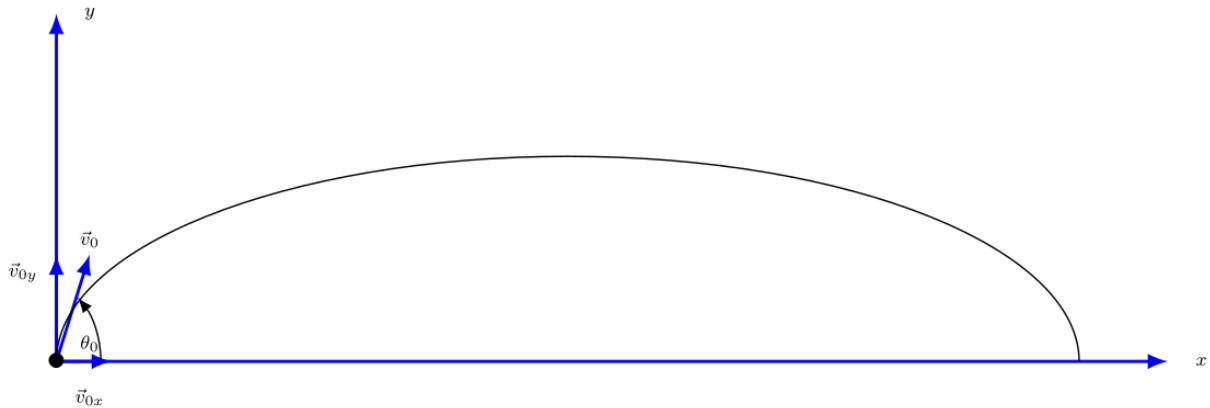


Figure 7.1: The illustration of projectile motion, an object is launched into the air at the origin of the coordinate system with launch velocity \vec{v}_0 at angle θ_0 .

(as measured from a horizontal x axis). Its horizontal acceleration during flight is zero, while its vertical acceleration is $-g$. Thus, the equation of motion for the particle can be written as

$$x - x_0 = (v_0 \cos \theta_0)t \quad (7.6)$$

$$y - y_0 = (v_0 \sin \theta_0)t - \frac{1}{2}gt^2 \quad (7.7)$$

$$v_y = v_0 \sin \theta_0 - gt \quad (7.8)$$

$$v_y^2 = (v_0 \sin \theta_0)^2 - 2g(y - y_0) \quad (7.9)$$

III. SIMULATION FOR PROJECTILE MOTION WITH Box2D

You need to copy from my repository' directory **../Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:

```
mkdir build
cd build
cmake ..
make
./testbed
```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

```
#include "test.h"
#include <iostream>

class ProjectileMotion: public Test
{
public:

    ProjectileMotion()
    {
        m_world->SetGravity(b2Vec2(0.0f, -9.8f));
        b2Timer timer;
        // Perimeter Ground body
        {
            b2BodyDef bd;
            b2Body* ground = m_world->CreateBody(&bd);

            b2EdgeShape shapeGround;
            shapeGround.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
                (30.0f, 0.0f));
            ground->CreateFixture(&shapeGround, 0.0f);

            b2EdgeShape shapeTop;
            shapeTop.SetTwoSided(b2Vec2(-40.0f, 30.0f), b2Vec2
                (30.0f, 30.0f));
            ground->CreateFixture(&shapeTop, 0.0f);

            b2EdgeShape shapeLeft;
            shapeLeft.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
                (-40.0f, 30.0f));
            ground->CreateFixture(&shapeLeft, 0.0f);

            b2EdgeShape shapeRight;
            shapeRight.SetTwoSided(b2Vec2(30.0f, 0.0f), b2Vec2
                (30.0f, 30.0f));
            ground->CreateFixture(&shapeRight, 0.0f);
        }

        // Create the ball
        b2CircleShape ballShape;
        ballShape.m_p.SetZero();
```

```

ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.15f; // the bounciness
ballFixtureDef.density = 7.3f; // this will affect the ball
    mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(-25.0f, 0.5f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&
    ballFixtureDef);

m_createTime = timer.GetMilliseconds();
}

b2Body* bodybbox;
b2Body* m_ball;

void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_S:
            int theta = 45;
            float v0 = 20.0f;
            m_ball ->SetLinearVelocity(b2Vec2(v0*cos(theta), v0*
                sin(theta)));
            break;
    }
}

void Step(Settings& settings) override
{
    b2MassData massData = m_ball->GetMassData();
    g_debugDraw.DrawString(5, m_textLine, "Ball Mass = %.6f",
        massData.mass);
    m_textLine += m_textIncrement;

    b2Vec2 position = m_ball->GetPosition();
    g_debugDraw.DrawString(5, m_textLine, "Ball Position, x = %.6
        f", position.x);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Ball Position, y = %.6
        f", position.y);
}

```

```

        f", position.y);
m_textLine += m_textIncrement;

b2Vec2 velocity = m_ball->GetLinearVelocity();
g_debugDraw.DrawString(5, m_textLine, "Ball velocity, x = %.6
f", velocity.x);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Ball velocity, y = %.6
f", velocity.y);
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "create time = %6.2f ms
",
m_createTime);
m_textLine += m_textIncrement;

printf("%4.2f %4.2f \n", velocity.x, velocity.y);

Test::Step(settings);
}

static Test* Create()
{
    return new ProjectileMotion;
}

float m_createTime;
};

static int testIndex = RegisterTest("Motion in 2D", "Projectile Motion",
ProjectileMotion::Create);

```

C++ Code 39: tests/projectile_motion.cpp "Projectile Motion Box2D"

Some explanations for the codes:

- Create a ball that is staying still on the ground.

```

b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.15f; // the bounciness
ballFixtureDef.density = 7.3f; // this will affect the ball
mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;

```

```

ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(-25.0f, 0.5f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
);

```

You can change the restitution, density, friction for different kind of ball for your own research.

- Create keyboard event, when pressed 'S' the ball is launched with initial velocity, $v_0 = 20$ and the initial angle of $\theta_0 = 45^\circ$.

```

void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_S:
            int theta = 45;
            float v0 = 20.0f;
            m_ball ->SetLinearVelocity(b2Vec2(v0*cos(theta),
                                              v0*sin(theta)));
            break;
    }
}

```

- To show the data of the position, velocity and the mass of the ball for our simulation, then print the velocity data into xterm / terminal.

```

void Step(Settings& settings) override
{
    b2MassData massData = m_ball->GetMassData();
    g_debugDraw.DrawString(5, m_textLine, "Ball Mass = %.6f"
                           , massData.mass);
    m_textLine += m_textIncrement;

    b2Vec2 position = m_ball->GetPosition();
    g_debugDraw.DrawString(5, m_textLine, "Ball Position, x
                           = %.6f", position.x);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Ball Position, y
                           = %.6f", position.y);
    m_textLine += m_textIncrement;

    b2Vec2 velocity = m_ball->GetLinearVelocity();
    g_debugDraw.DrawString(5, m_textLine, "Ball velocity, x
                           = %.6f", velocity.x);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Ball velocity, y
                           = %.6f", velocity.y);
}

```

```

    m_textLine += m_textIncrement;

    g_debugDraw.DrawString(5, m_textLine, "create time =
        %6.2f ms",
    m_createTime);
    m_textLine += m_textIncrement;

    printf("%4.2f %4.2f \n", velocity.x, velocity.y);
    //printf("%4.2f %4.2f \n", position.x, position.y);
    Test::Step(settings);
}

```

After recompiling this, you can save it into textfile by opening the testbed with this command:
./testbed > projectileoutput.txt

After you close the testbed to record the result, then see it at the current working directory where **testbed** is located and see **projectileoutput.txt**. You need to clean up a bit and delete the strings at the beginning of the textfile so it will only left you with numbers only in 2 columns.

To plot the velocity in x and y axis for the projectile motion of the ball with respect to time, now open terminal at the directory containing the **projectileoutput.txt** and type:

```

gnuplot
set xlabel "time"
set ylabel "v_{x}"
plot "projectileoutput.txt" using 1 title "" with lines
set ylabel "v_{y}"
plot "projectileoutput.txt" using 2 title "" with lines

```

Now you can go back to the source code of the projectile motion and uncomment the line to print the position and comment the line to print the velocity:

```

//printf("%4.2f %4.2f \n", velocity.x, velocity.y);
printf("%4.2f %4.2f \n", position.x, position.y);

```

Now, we can plot the position of the ball with gnuplot, recompile the testbed to make the change occurs then type:

./testbed > projectileoutput.txt

Plot it with gnuplot from the working directory:

```

gnuplot
set xlabel "time"
set ylabel "v_{x}"
plot "projectileoutput.txt" using 1:2 title "Ball trajectory" with lines

```

IV. SIMULATION FOR PROJECTILE DROPPED FROM ABOVE WITH Box2D

This is coming from a sample problem in [6], but I modify it a little bit. Suppose that Europe is now in war and get help of food supplies from its allies, if there are two rescue drones, the first

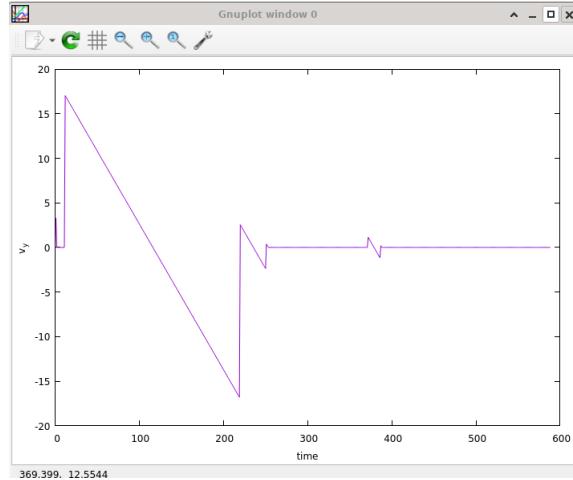


Figure 7.2: The "projectileoutput.txt" is being plotted by using gnuplot for the data of the v_x (the horizontal velocity).

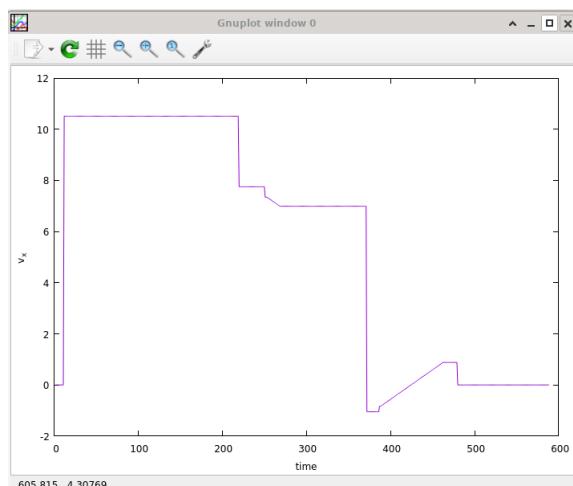


Figure 7.3: The "projectileoutput.txt" is being plotted by using gnuplot for the data of the v_y (the horizontal velocity).

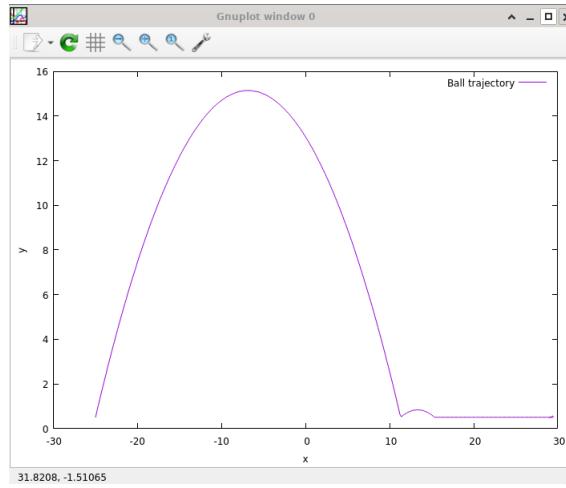


Figure 7.4: The "projectileoutput.txt" is being plotted by using gnuplot for the data of the x and y (position of the ball). You can see that the ball is still sliding forward if you see the Box2D simulation, since we make the restitution for the ball 0.15, a bit bouncy. This simulation is very useful to design a better golf ball or galleon canon for spaceship to hit the target faster and accurate.

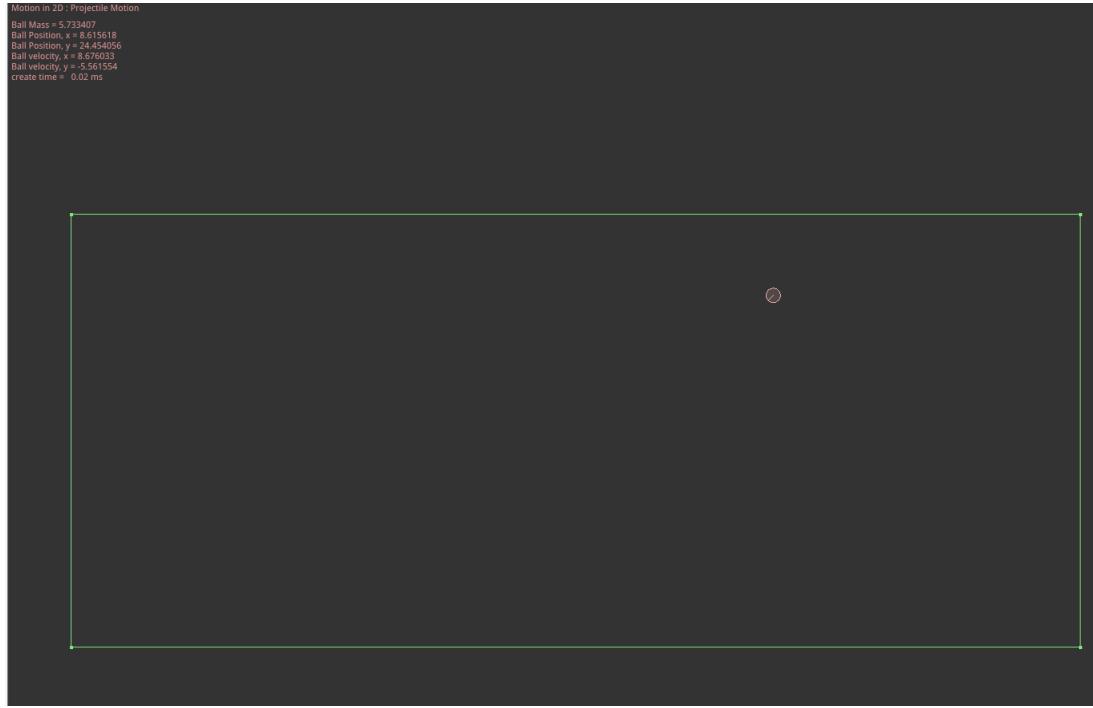


Figure 7.5: The modified Box2D simulation of a ball being projected with initial angle $\theta_0 = 45^\circ$ and initial velocity of $v_0 = 20$ (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/projectile_motion.cpp`).

drone is sending a cheese ball (shape of a circle in 2 dimension) and the second drone is sending boxes full of chocolates and medicine. Don't ask why they don't send vegetables and fruits that are healthier, it is at war now. Suppose both flying at different height, constant height of $h_1 = 40$ and $h_2 = 30$, but their speeds are the same, at certain speed of $v_0 = 20$. What should be the angle ϕ_1 and ϕ_2 of the first and second drone line of sight to the net below (the height of the net is 5, with width of 10, centered at $x = 0$) that will capture the cheese ball and boxes full of chocolate and medicine?

Solution:

We need to know that ϕ_1 is given by

$$\phi_1 = \tan^{-1} \left(\frac{x_1}{h_1} \right)$$

and ϕ_2 is given by

$$\phi_2 = \tan^{-1} \left(\frac{x_2}{h_2} \right)$$

where x_1 and x_2 are the horizontal coordinates of the net to capture the cheese ball and the chocolate boxes with medicine respectively from where the supplies being drop by each of the drones. We will have two cases and should be able to find x_1 and x_2 for each of the case with

$$x_1 - x_0 = (v_0 \cos \theta_0) t_1$$

$$x_2 - x_0 = (v_0 \cos \theta_0) t_2$$

we know that $x_0 = 0$ because the origin is placed at the point of release from each of the drone. v_0 is the drone velocity, and both drones fly at the same velocity of $v_0 = 20$, with the angle $\theta_0 = 0^0$ measured relative to the positive direction of the x axis, since the supplies are being dropped not being projected / launched.

To find t , the time for the supply arrive at the net, we can use the vertical motion displacement equation for the first drone:

$$y_1 - y_0 = (v_0 \sin \theta_0) t_1 - \frac{1}{2} g t_1^2$$

and for the second drone vertical displacement equation:

$$y_2 - y_0 = (v_0 \sin \theta_0) t_2 - \frac{1}{2} g t_2^2$$

The vertical displacement for each of the drone will be negative, the negative value indicates that the supplies moves downward. Thus, solving for t for the first drone

$$\begin{aligned} y_1 - y_0 &= (v_0 \sin \theta_0) t_1 - \frac{1}{2} g t_1^2 \\ -40 &= (20 \sin 0^0) t_1 - \frac{1}{2} (9.8) t_1^2 \\ -40 &= -\frac{1}{2} (9.8) t_1^2 \\ t_1 &= \sqrt{\frac{80}{9.8}} \\ t_1 &= 2.8571 \end{aligned}$$

then for the second drone

$$\begin{aligned}
 y_2 - y_0 &= (v_0 \sin \theta_0)t_2 - \frac{1}{2}gt_2^2 \\
 -30 &= (20 \sin 0^\circ)t_2 - \frac{1}{2}(9.8)t_2^2 \\
 -30 &= -\frac{1}{2}(9.8)t_2^2 \\
 t_2 &= \sqrt{\frac{60}{9.8}} \\
 t_2 &= 2.4743
 \end{aligned}$$

After we obtain t for both drones, we can estimate the horizontal distance between the drone to the net and the drone' line of sight

$$\begin{aligned}
 x_1 - x_0 &= (v_0 \cos \theta_0)t_1 \\
 x_1 - 0 &= (20 \cos 0^\circ)(2.8571) \\
 x_1 &= 57.141999
 \end{aligned}$$

$$\phi_1 = \tan^{-1} \left(\frac{x_1}{h_1} \right) = \tan^{-1} \left(\frac{57.141999}{40} \right) = \tan^{-1}(1.428549) = 0.960063 \text{ rad} \approx 55.007575^\circ$$

The horizontal distance for the second drone

$$\begin{aligned}
 x_2 - x_0 &= (v_0 \cos \theta_0)t_2 \\
 x_2 - 0 &= (20 \cos 0^\circ)(2.4743) \\
 x_2 &= 49.48716
 \end{aligned}$$

$$\phi_2 = \tan^{-1} \left(\frac{x_2}{h_2} \right) = \tan^{-1} \left(\frac{49.48716}{30} \right) = \tan^{-1}(1.649572) = 1.0258174 \text{ rad} \approx 58.775^\circ$$

Either the angles of depression ϕ_1 and ϕ_2 or the horizontal distance between the center of the net and the drone can be used as the data input to release or drop the supplies, as sensor for drone or mechatronics that can calculate horizontal distance toward a target and altitude is available along with one that can calculate angle of depression toward a designated target.

For the C++ simulation, you can copy from my repository' directory **./Source Codes/C++/DianFreya-box2d-testbed**, then go inside the directory and open the terminal then type:

```

mkdir build
cd build
cmake ..
make
./testbed

```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

```

#include "test.h"
#include <iostream>

// This is used to test sensor shapes.

```

```

class ProjectileDropped : public Test
{
public:

enum
{
    e_count = 10
};

ProjectileDropped()
{
    b2Body* ground = NULL;
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
            0.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(5.6f, 0.0f), b2Vec2(5.6f,
            10.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
    // Create the net centering at x=0
    {
        b2PolygonShape shape;
        shape.SetAsBox(0.5f, 0.125f);

        b2FixtureDef fd;
        fd.shape = &shape;
        fd.density = 20.0f;
        fd.friction = 0.2f;

        b2RevoluteJointDef jd;

        b2Body* prevBody = ground;
        for (int32 i = 0; i < e_count; ++i)
        {
            b2BodyDef bd;
            bd.type = b2_dynamicBody;
            bd.position.Set(-4.5f + 1.0f * i, 5.0f); // 5.0
        }
    }
}

```

```

        f is the height
        b2Body* body = m_world->CreateBody(&bd);
        body->CreateFixture(&fd);

        b2Vec2 anchor(-5.0f + 1.0f * i, 5.0f); //create
            the chain ball anchor
        jd.Initialize(prevBody, body, anchor);
        m_world->CreateJoint(&jd);

        if (i == (e_count >> 1))
        {
            m_middle = body;
        }
        prevBody = body;
    }

    b2Vec2 anchor(-5.0f + 1.0f * e_count, 5.0f); // the
        right anchor
    jd.Initialize(prevBody, ground, anchor);
    m_world->CreateJoint(&jd);
}

// Create the ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.15f; // the bounciness
ballFixtureDef.density = 7.3f; // this will affect the ball
    mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(-52.14199f, 40.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&
    ballFixtureDef);
int theta = 0;
float v0 = 20.0f;
m_ball ->SetLinearVelocity(b2Vec2(v0*cos(theta), 0));

// Create Breakable dynamic body
{
    b2BodyDef bd;
}

```

```

        bd.type = b2_dynamicBody;
        bd.position.Set(-44.48716f, 30.0f);
        bd.angle = 0.85f * b2_pi;
        m_body1 = m_world->CreateBody(&bd);

        m_shape1.SetAsBox(0.5f, 0.5f, b2Vec2(-0.5f, 0.0f), 0.0
                           f);
        m_piece1 = m_body1->CreateFixture(&m_shape1, 1.0f);

        m_shape2.SetAsBox(0.5f, 0.5f, b2Vec2(0.5f, 0.0f), 0.0f
                           );
        m_piece2 = m_body1->CreateFixture(&m_shape2, 1.0f);

        m_body1 ->SetLinearVelocity(b2Vec2(v0*cos(theta), 0));
    }

    m_break = false;
    m_broke = false;
}
b2Body* m_middle;
b2Body* m_ball;

b2Body* m_body1;
b2Vec2 m_velocity;
float m_angularVelocity;
b2PolygonShape m_shape1;
b2PolygonShape m_shape2;
b2Fixture* m_piece1;
b2Fixture* m_piece2;
bool m_broke;
bool m_break;

void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse)
    override
{
    if (m_broke)
    {
        // The body already broke.
        return;
    }

    // Should the body break?
    int32 count = contact->GetManifold()->pointCount;

    float maxImpulse = 0.0f;
    for (int32 i = 0; i < count; ++i)
    {
        maxImpulse = b2Max(maxImpulse, impulse->normalImpulses

```

```

        [i]);
    }

    if (maxImpulse > 40.0f)
    {
        // Flag the body for breaking.
        m_break = true;
    }
}

void Break()
{
    // Create two bodies from one.
    b2Body* body1 = m_piece1->GetBody();
    b2Vec2 center = body1->GetWorldCenter();

    body1->DestroyFixture(m_piece2);
    m_piece2 = NULL;

    b2BodyDef bd;
    bd.type = b2_dynamicBody;
    bd.position = body1->GetPosition();
    bd.angle = body1->GetAngle();
    b2Body* body2 = m_world->CreateBody(&bd);
    m_piece2 = body2->CreateFixture(&m_shape2, 1.0f);

    // Compute consistent velocities for new bodies based on
    // cached velocity.
    b2Vec2 center1 = body1->GetWorldCenter();
    b2Vec2 center2 = body2->GetWorldCenter();

    b2Vec2 velocity1 = m_velocity + b2Cross(m_angularVelocity,
                                              center1 - center);
    b2Vec2 velocity2 = m_velocity + b2Cross(m_angularVelocity,
                                              center2 - center);

    body1->SetAngularVelocity(m_angularVelocity);
    body1->SetLinearVelocity(velocity1);

    body2->SetAngularVelocity(m_angularVelocity);
    body2->SetLinearVelocity(velocity2);
}

void Step(Settings& settings) override
{
    b2MassData massData = m_ball->GetMassData();
    g_debugDraw.DrawString(5, m_textLine, "Ball Mass = %.6f",
                          massData.mass);
}

```

```

    m_textLine += m_textIncrement;

    b2Vec2 position = m_ball->GetPosition();
    g_debugDraw.DrawString(5, m_textLine, "Ball Position, x = %.6
        f", position.x);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Ball Position, y = %.6
        f", position.y);
    m_textLine += m_textIncrement;

    b2Vec2 positionbox1 = m_body1->GetPosition();
    g_debugDraw.DrawString(5, m_textLine, "Box 1 Position, x =
        %.6f", positionbox1.x);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Box 1 Position, y =
        %.6f", positionbox1.y);
    m_textLine += m_textIncrement;

    printf("%4.2f %4.2f %4.2f %4.2f \n", position.x, position.y,
        positionbox1.x, positionbox1.y);

    if (m_break)
    {
        Break();
        m_broke = true;
        m_break = false;
    }

    // Cache velocities to improve movement on breakage.
    if (m_broke == false)
    {
        m_velocity = m_body1->GetLinearVelocity();
        m_angularVelocity = m_body1->GetAngularVelocity();
    }

    Test::Step(settings);
}

static Test* Create()
{
    return new ProjectileDropped;
}

};

static int testIndex = RegisterTest("Motion in 2D", "Projectile Dropped",
    ProjectileDropped::Create);

```

C++ Code 40: *tests/projectile_dropped.cpp* "Projectile Motion Box2D"

Some explanations for the codes:

- Inside the **ProjectileDropped()**, to create the ground and the wall at the right side of the net

```

b2Body* ground = NULL;
{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
        0.0f));
    ground->CreateFixture(&shape, 0.0f);
}
{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(5.6f, 0.0f), b2Vec2(5.6f, 10.0f
        ));
    ground->CreateFixture(&shape, 0.0f);
}
// Create the net centering at x=0
{
    b2PolygonShape shape;
    shape.SetAsBox(0.5f, 0.125f);

    b2FixtureDef fd;
    fd.shape = &shape;
    fd.density = 20.0f;
    fd.friction = 0.2f;

    b2RevoluteJointDef jd;

    b2Body* prevBody = ground;
    for (int32 i = 0; i < e_count; ++i)
    {
        b2BodyDef bd;
        bd.type = b2_dynamicBody;
        bd.position.Set(-4.5f + 1.0f * i, 5.0f); // 5.0f
        is the height
        b2Body* body = m_world->CreateBody(&bd);
        body->CreateFixture(&fd);

        b2Vec2 anchor(-5.0f + 1.0f * i, 5.0f); //create
        the chain ball anchor
        jd.Initialize(prevBody, body, anchor);
        m_world->CreateJoint(&jd);
    }
}

```

```

        if (i == (e_count >> 1))
    {
        m_middle = body;
    }
    prevBody = body;
}

b2Vec2 anchor(-5.0f + 1.0f * e_count, 5.0f); // the
right anchor
jd.Initialize(prevBody, ground, anchor);
m_world->CreateJoint(&jd);
}

```

- To create the cheese ball that is being dropped by the first drone at height of 40, the reason why I set the drop point as $52.14199f$ instead of $57.14199f$ is because the net is centered at $x = 0$ and has width of 10, 10 divided by 2 is 5, so the drone shall move forward 5 unit more before dropping the cheese ball, thus it can land at the center of the net, not too early or late. If you watch the Box2D simulation for this, both the cheese ball and chocolates with medicine boxes arrive exactly at the center of the net.

```

// Create the ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.15f; // the bounciness
ballFixtureDef.density = 7.3f; // this will affect the ball
mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(-52.14199f, 40.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
);
int theta = 0;
float v0 = 20.0f;
m_ball ->SetLinearVelocity(b2Vec2(v0*cos(theta), 0));

```

- To create the 2 boxes of chocolates and medicine, the boxes are attached and prone to break

```

// Create Breakable dynamic body
{
    b2BodyDef bd;

```

```

        bd.type = b2_dynamicBody;
        bd.position.Set(-44.48716f, 30.0f);
        bd.angle = 0.85f * b2_pi;
        m_body1 = m_world->CreateBody(&bd);

        m_shape1.SetAsBox(0.5f, 0.5f, b2Vec2(-0.5f, 0.0f), 0.0f)
        ;
        m_piece1 = m_body1->CreateFixture(&m_shape1, 1.0f);

        m_shape2.SetAsBox(0.5f, 0.5f, b2Vec2(0.5f, 0.0f), 0.0f);
        m_piece2 = m_body1->CreateFixture(&m_shape2, 1.0f);

        m_body1 ->SetLinearVelocity(b2Vec2(v0*cos(theta), 0));
    }

    m_break = false;
    m_broke = false;

```

- To declare the body, polygon shape, fixtures, break boolean variable.

```

b2Body* m_middle;
b2Body* m_ball;

b2Body* m_body1;
b2Vec2 m_velocity;
float m_angularVelocity;
b2PolygonShape m_shape1;
b2PolygonShape m_shape2;
b2Fixture* m_piece1;
b2Fixture* m_piece2;
bool m_broke;
bool m_break;

```

- The solver, function **PostSolve()** for the breaking apart of the boxes into two.

```

void PostSolve(b2Contact* contact, const b2ContactImpulse*
    impulse) override
{
    if (m_broke)
    {
        // The body already broke.
        return;
    }

    // Should the body break?
    int32 count = contact->GetManifold()->pointCount;

    float maxImpulse = 0.0f;
    for (int32 i = 0; i < count; ++i)
    {

```

```

        maxImpulse = b2Max(maxImpulse, impulse->
            normalImpulses[i]);
    }

    if (maxImpulse > 40.0f)
    {
        // Flag the body for breaking.
        m_break = true;
    }
}

```

- We initially have two boxes attached into one, then after landing on the net, get impulses, it breaks apart into two, that will each has its own velocity, position, and become different bodies in Box2D Physics world for this simulation. With function **Break()** we define how to separate **m_body1** into two separate bodies

```

void Break()
{
    // Create two bodies from one.
    b2Body* body1 = m_piece1->GetBody();
    b2Vec2 center = body1->GetWorldCenter();

    body1->DestroyFixture(m_piece2);
    m_piece2 = NULL;

    b2BodyDef bd;
    bd.type = b2_dynamicBody;
    bd.position = body1->GetPosition();
    bd.angle = body1->GetAngle();
    b2Body* body2 = m_world->CreateBody(&bd);
    m_piece2 = body2->CreateFixture(&m_shape2, 1.0f);

    // Compute consistent velocities for new bodies based on
    // cached velocity.
    b2Vec2 center1 = body1->GetWorldCenter();
    b2Vec2 center2 = body2->GetWorldCenter();

    b2Vec2 velocity1 = m_velocity + b2Cross(
        m_angularVelocity, center1 - center);
    b2Vec2 velocity2 = m_velocity + b2Cross(
        m_angularVelocity, center2 - center);

    body1->SetAngularVelocity(m_angularVelocity);
    body1->SetLinearVelocity(velocity1);

    body2->SetAngularVelocity(m_angularVelocity);
    body2->SetLinearVelocity(velocity2);
}

```

- This will show the cheese ball mass, the position, and the boxes position for the first box, along with the condition if the boxes break apart then the function **Break()** will occur

```

        void Step(Settings& settings) override
        {
            b2MassData massData = m_ball->GetMassData();
            g_debugDraw.DrawString(5, m_textLine, "Ball Mass = %.6f"
                , massData.mass);
            m_textLine += m_textIncrement;

            b2Vec2 position = m_ball->GetPosition();
            g_debugDraw.DrawString(5, m_textLine, "Ball Position, x"
                = %.6f", position.x);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Ball Position, y"
                = %.6f", position.y);
            m_textLine += m_textIncrement;

            b2Vec2 positionbox1 = m_body1->GetPosition();
            g_debugDraw.DrawString(5, m_textLine, "Box 1 Position, x"
                = %.6f", positionbox1.x);
            m_textLine += m_textIncrement;
            g_debugDraw.DrawString(5, m_textLine, "Box 1 Position, y"
                = %.6f", positionbox1.y);
            m_textLine += m_textIncrement;

            printf("%4.2f %4.2f %4.2f %4.2f \n", position.x,
                position.y,positionbox1.x, positionbox1.y);

            if (m_break)
            {
                Break();
                m_broke = true;
                m_break = false;
            }

            // Cache velocities to improve movement on breakage.
            if (m_broke == false)
            {
                m_velocity = m_body1->GetLinearVelocity();
                m_angularVelocity = m_body1->GetAngularVelocity()
                    ;
            }

            Test::Step(settings);
        }
    
```

Now, to plot the position of the cheese ball, chocolates and medicine supplies with gnuplot, recompile the testbed to make the change occurs then type:

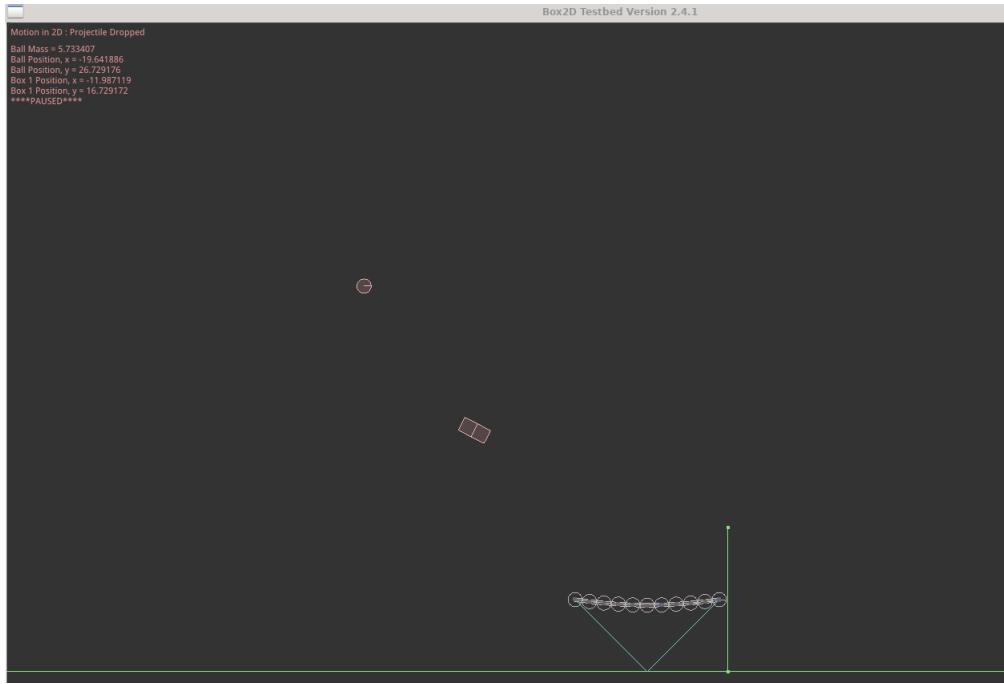


Figure 7.6: The modified Box2D simulation of a cheese ball and chocolates with medicine boxes being dropped from two drones with same velocity of $v_0 = 20$ but flying at different heights (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/projectile_dropped.cpp`).

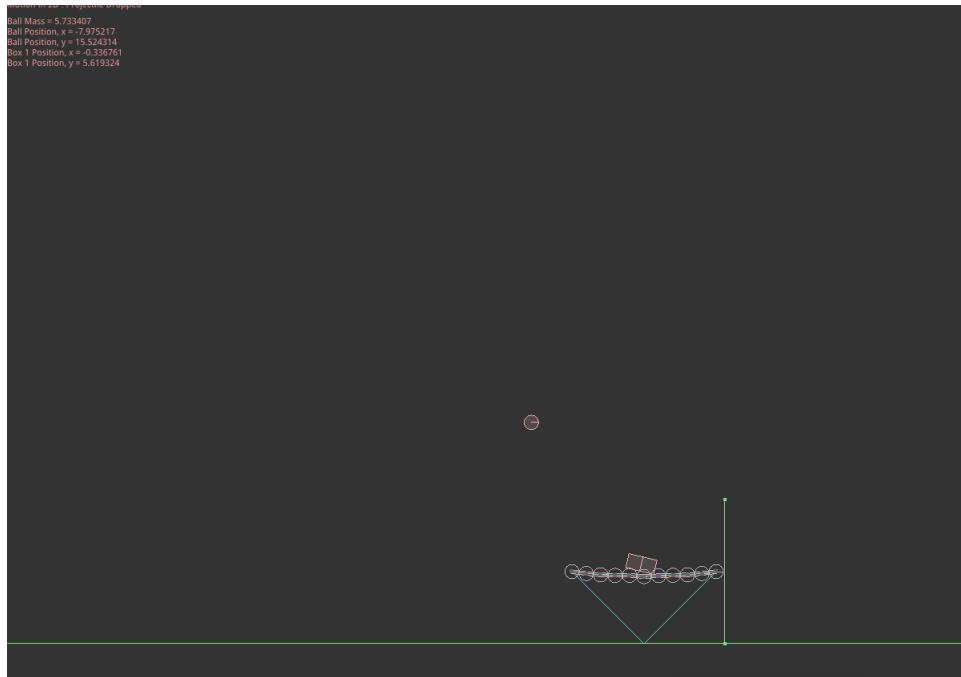


Figure 7.7: The modified Box2D simulation clearly shows that both the supplies will arrive at the middle of the net.

./testbed > projectiledropped.txt

Plot it with gnuplot from the working directory:

```
gnuplot
set xlabel "x"
set ylabel "y"
plot "projectiledropped.txt" using 1:2 title "Cheese Ball" with lines, "projectiledropped.txt"
using 3:4 title "Chocolates and Medicine" with lines
```

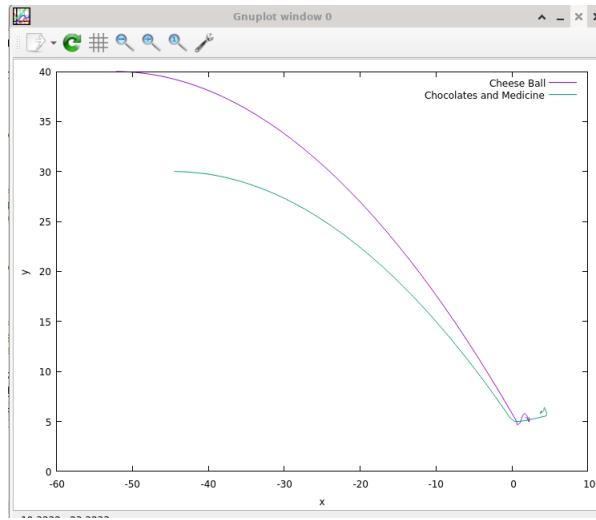


Figure 7.8: The gnuplot of the position of the supplies that are dropped from different height by each drone

V. UNIFORM CIRCULAR MOTION

A particle is in uniform circular motion if it travels around a circle path or a circular arc at constant (uniform) speed. Although the speed does not vary, the particle is accelerating because the velocity changes in direction.

The velocity and acceleration vectors for uniform circular motion have constant magnitude but their directions change continuously. The velocity is always directed tangent to the circle in the direction of motion, while the acceleration is always directed radially inward, to the center of the circle. That is why it is called centripetal acceleration.

The scalar components of \vec{v} can be written as

$$\vec{v} = v_x \hat{i} + v_y \hat{j} = (-v \sin \theta) \hat{i} + (v \cos \theta) \hat{j} \quad (7.10)$$

remember that since sine is an odd function and cosine is an even function, we will have $\sin(\theta) = -\sin(\theta)$ and $\cos(\theta) = \cos(\theta)$. Now, we can substitute

$$\sin \theta = \frac{y_p}{r}$$

$$\cos \theta = \frac{x_p}{r}$$

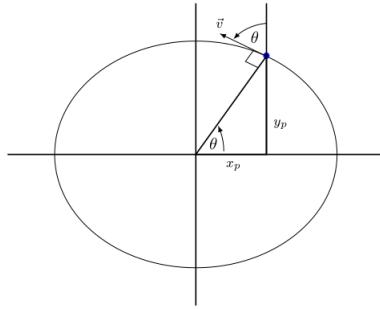


Figure 7.9: The particle p moves in counter-clockwise uniform circular motion with its position and velocity \vec{v} at a certain instant.

thus

$$\begin{aligned}\vec{v} &= v_x \hat{i} + v_y \hat{j} \\ &= \left(-\frac{vy_p}{r} \right) \hat{i} + \left(\frac{vx_p}{r} \right) \hat{j}\end{aligned}$$

To find the acceleration of the particle, we must derivate the velocity with respect to time. Noting that speed v and radius r do not change with time, we obtain

$$\begin{aligned}\vec{a} &= \frac{d\vec{v}}{dt} \\ &= \left(-\frac{v}{r} \frac{dy_p}{dt} \right) \hat{i} + \left(\frac{v}{r} \frac{dx_p}{dt} \right) \hat{j} \\ &= \left(-\frac{v^2}{r} \cos \theta \right) \hat{i} + \left(-\frac{v^2}{r} \sin \theta \right) \hat{j}\end{aligned}$$

with $\frac{dy_p}{dt} = v_y = v \cos \theta$ and $\frac{dx_p}{dt} = v_x = -v \sin \theta$. Now, to calculate the acceleration magnitude, we will have

$$\begin{aligned}a &= \sqrt{a_x^2 + a_y^2} \\ &= \frac{v^2}{r} \sqrt{(\cos \theta)^2 + (\sin \theta)^2} \\ &= \frac{v^2}{r} (1) \\ a &= \frac{v^2}{r}\end{aligned}$$

Now to orient a , we want to prove that the angle ϕ will direct the acceleration toward the circle's center

$$\tan \phi = \frac{a_y}{a_x} = \frac{-(v^2/r) \sin \theta}{-(v^2/r) \cos \theta} = \tan \theta \quad (7.11)$$

Thus, $\phi = \theta$.

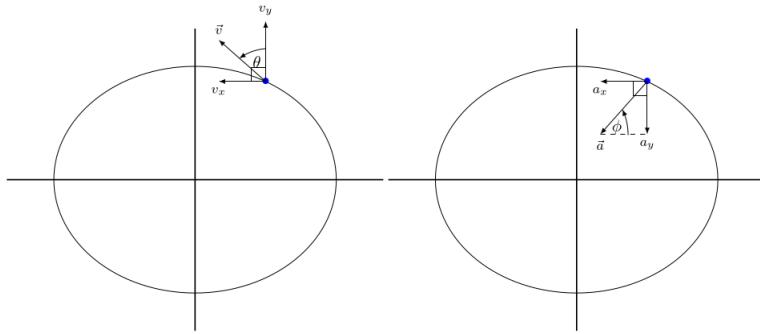


Figure 7.10: The velocity \vec{v} and acceleration \vec{a} of a particle in uniform circular motion that is moving in counter-clockwise direction.

VI. SIMULATION FOR UNIFORM CIRCULAR MOTION WITH Box2D

For the C++ simulation, you can copy from my repository' directory [..//Source Codes/C++/DianFreya-box2d-testbed](#), then go inside the directory and open the terminal then type:

```
mkdir build
cd build
cmake ..
make
./testbed
```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Circular Motion**.

```
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#include "test.h"
#include <fstream>

class CircularMotion : public Test
{
public:
    CircularMotion()
    {
        m_world->SetGravity(b2Vec2(0.0f, 0.0f));
        b2BodyDef bd;
        b2Body* ground = m_world->CreateBody(&bd);
        b2Body* b1;
        {
            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
                0.0f));
        }
    }
};
```

```

        b2BodyDef bd;
        b1 = m_world->CreateBody(&bd);
        b1->CreateFixture(&shape, 0.0f);
    }
    // Create the trail path
    {
        b2CircleShape shape;
        shape.m_radius = 3.0f;
        shape.m_p.Set(-3.0f, 25.0f);

        b2FixtureDef fd;
        fd.shape = &shape;
        fd.isSensor = true;
        m_sensor = ground->CreateFixture(&fd);
    }

    // Create the ball
    b2CircleShape ballShape;
    ballShape.m_p.SetZero();
    ballShape.m_radius = 0.5f;

    b2FixtureDef ballFixtureDef;
    ballFixtureDef.restitution = 0.75f;
    ballFixtureDef.density = 3.3f; // this will affect the ball
        mass
    ballFixtureDef.friction = 0.1f;
    ballFixtureDef.shape = &ballShape;

    b2BodyDef ballBodyDef;
    ballBodyDef.type = b2_dynamicBody;
    ballBodyDef.position.Set(0.0f, 25.0f);

    m_ball = m_world->CreateBody(&ballBodyDef);
    b2Fixture *ballFixture = m_ball->CreateFixture(&
        ballFixtureDef);
    m_ball->SetAngularVelocity(1.0f);
    m_time = 0.0f;
}
b2Body* m_ball;
float m_time;
b2Fixture* m_sensor;
void Step(Settings& settings) override
{
    b2Vec2 v = m_ball->GetLinearVelocity();
    float r = 3.0f;
    float omega = m_ball->GetAngularVelocity();
    float angle = m_ball->GetAngle();
    b2MassData massData = m_ball->GetMassData();
}

```

```

b2Vec2 position = m_ball->GetPosition();
float sin = sinf(angle);
float cos = cosf(angle);
float ball_vel = 3.0f;
float a = (ball_vel*ball_vel) / r;
m_time += 1.0f / 60.0f; // assuming we are using frequency of
                         60 Hertz

float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f *
massData.I * omega * omega;

m_ball->SetLinearVelocity(b2Vec2(-ball_vel*sinf(angle),
ball_vel*cosf(angle)));

g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
f", m_time);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Ball position, x = %.6
f", position.x);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Ball position, y = %.6
f", position.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Kinetic energy = %.6f"
, ke);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Linear velocity = %.6f
", v);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Linear velocity, x =
%.6f", v.x);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Linear velocity, y =
%.6f", v.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Acceleration, a = %.6f
", a);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Angle (in degrees) =
%.6f", angle*RADTODEG);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "sin (angle) = %.6f",
sin);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "cos (angle) = %.6f",
cos);

```

```

        cos);
    m_textLine += m_textIncrement;
    // Print the result in every time step then plot it into
    graph with either gnuplot or anything

    printf("%4.2f %4.2f %4.2f %4.2f %4.2f %4.2f %4.2f\n",
           position.x, position.y, angle*RADTODEG, v.x, v.y, sin,
           cos);

    Test::Step(settings);
}

static Test* Create()
{
    return new CircularMotion;
}

};

static int testIndex = RegisterTest("Motion in 2D", "Circular Motion",
    CircularMotion::Create);

```

C++ Code 41: *tests/circular_motion.cpp* "Uniform Circular Motion Box2D"

Some explanations for the codes:

- First, we create the physics world with gravity of 0 (not realistic on earth, but it can be used to show uniform circular motion on space), then we create the trail path for the particle that is moving, a circle shape with color of green.

```

m_world->SetGravity(b2Vec2(0.0f,0.0f));
b2BodyDef bd;
b2Body* ground = m_world->CreateBody(&bd);
b2Body* b1;
{
    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
        0.0f));

    b2BodyDef bd;
    b1 = m_world->CreateBody(&bd);
    b1->CreateFixture(&shape, 0.0f);
}

// Create the trail path
{
    b2CircleShape shape;
    shape.m_radius = 3.0f;
    shape.m_p.Set(-3.0f, 25.0f);

    b2FixtureDef fd;

```

```

        fd.shape = &shape;
        fd.isSensor = true;
        m_sensor = ground->CreateFixture(&fd);
    }
}

```

- To create the circle (funny that we always call this circle as a ball in the code) with every details necessary, set the density, radius, restitution, friction, starting position. At the end we set the value for a new variable called **m_time** as 0 to calculate time since the beginning of the simulation.

```

// Create the ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.75f;
ballFixtureDef.density = 3.3f; // this will affect the ball
mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(0.0f, 25.0f);

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
);
m_ball->SetAngularVelocity(1.0f);
m_time = 0.0f;

```

- The variable declarations

```

b2Body* m_ball;
float m_time;
b2Fixture* m_sensor;

```

- We are showing some useful information on the moving particle that is in uniform circular motion, we can get the velocity and the position for the particle in x and y axis. The trick to make it revolving around a fixed center / in uniform circular motion, is to set the velocity to be changing with time to follow the formula from equation (7.10), with $v = 3$, since the sine and cosine of an angle will always be bounded and periodic.

```

void Step(Settings& settings) override
{
    b2Vec2 v = m_ball->GetLinearVelocity();
    float r = 3.0f;
    float omega = m_ball->GetAngularVelocity();
    float angle = m_ball->GetAngle();
    b2MassData massData = m_ball->GetMassData();
}

```

```

b2Vec2 position = m_ball->GetPosition();
float sin = sinf(angle);
float cos = cosf(angle);
float ball_vel = 3.0f;
float a = (ball_vel*ball_vel) / r;
m_time += 1.0f / 60.0f; // assuming we are using
frequency of 60 Hertz

float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f *
massData.I * omega * omega;

m_ball->SetLinearVelocity(b2Vec2(-ball_vel*sinf(angle),
ball_vel*cosf(angle)));

g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)
= %.6f", m_time);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Ball position, x
= %.6f", position.x);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Ball position, y
= %.6f", position.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f",
massData.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Kinetic energy =
%.6f", ke);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Linear velocity =
%.6f", v);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Linear velocity,
x = %.6f", v.x);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Linear velocity,
y = %.6f", v.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Acceleration, a =
%.6f", a);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Angle (in degrees
) = %.6f", angle*RADTODEG);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "sin (angle) = %.6
f", sin);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "cos (angle) = %.6
f", cos);

```

```

        f", cos);
m_textLine += m_textIncrement;
// Print the result in every time step then plot it into
graph with either gnuplot or anything

printf("%4.2f %4.2f %4.2f %4.2f %4.2f %4.2f %4.2f\n",
position.x, position.y, angle*RADTODEG, v.x, v.y,
sin, cos);

Test::Step(settings);
}

```

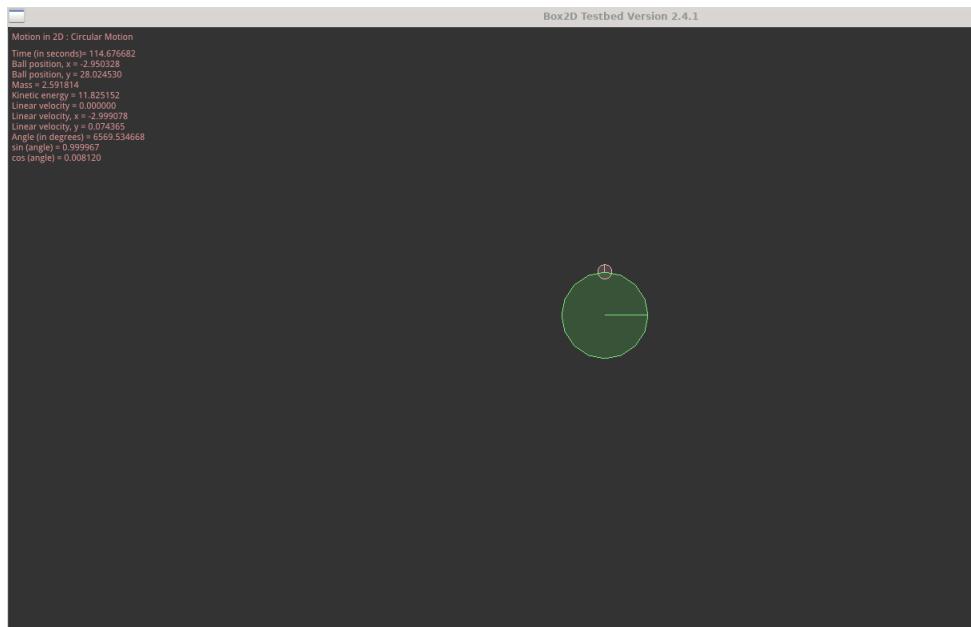


Figure 7.11: The simulation of a particle doing circular motion with speed of $\vec{v} = 3$ (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/circular_motion.cpp`).

After recompiling this, you can save it into textfile by opening the testbed with this command:
`./testbed > circularoutput.txt`

After you close the testbed to record the result, then see it at the current working directory where `testbed` is located and see `circularoutput.txt`. You need to clean up a bit and delete the strings at the beginning of the textfile so it will only left you with numbers only.

To plot the velocity in x and y axis for the uniform circular motion of the ball with respect to time, now open terminal at the directory containing the `circularoutput.txt` and type:

```

gnuplot
set xlabel "time"
plot "circularoutput.txt" using 4 title "v_{x}" with lines, "circularoutput.txt" using 5 title "v_{y}"
with lines

```

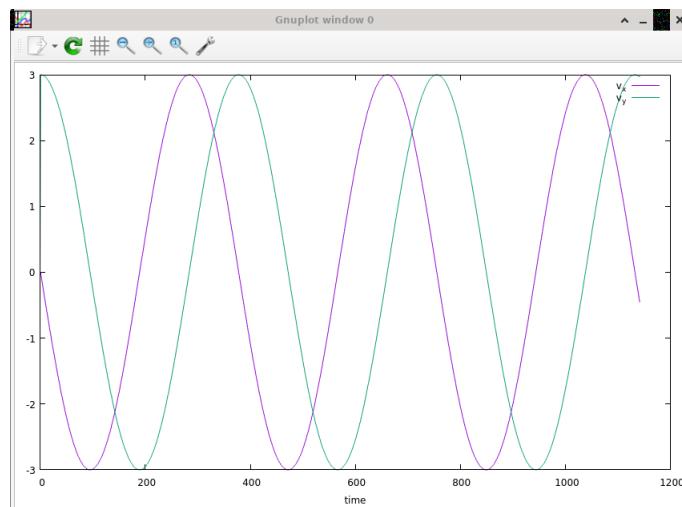


Figure 7.12: The gnuplot of the velocity of the ball particle is a negative sine function for v_y and a cosine function for v_x

Chapter 8

DFSimulatorC++ II: Force and Motion

"Remember how it all started by Force? You(Freya) asked me to create something better than BLAS and LAPACK with FORTRAN, or sleep 24/7, but why am I writing this now?" - DS Glanzsche

What cause an object to move? The answer is Force. I was forced to study FORTRAN, thus I take a FORTRAN book and read it then study, there it is one good example, since the alternative is if I don't want to study FORTRAN, I better sleep 24/7, well it is not good for my soul, happiness and body to sleep 24/7 unless I am in a comma, so I read FORTRAN based on Force. But, C++ comes first before FORTRAN, as there is always a prelude of something, appetizer before the main course.

I. NEWTONIAN MECHANICS

The relation between a force and the acceleration it causes was first understood by Isaac Newton(1642-1727), it is known as Newtonian mechanics. But, if the speed of the interacting bodies are too large, converging to the speed of light, we will use Einstein's theory of relativity instead.

The velocity of an object can change (the object can accelerate or decelerate) when the object is acted on by one or more forces (pushes or pulls) from other objects. Newtonian mechanics relates accelerations and forces.

Forces are vector quantities. Masses are scalar quantities. Their magnitudes are defined in terms of the acceleration times the mass of the object. A force that accelerates the standard body by exactly 1 m/s^2 is defined to have a magnitude of 1 N . The direction of a force is the direction of the acceleration it causes. Forces are combined according to the rules of vector algebra. The net force on a body is the vector sum of all the forces acting on the body.

If there is no net force on a body, the body remains at rest. The body is called to be in motion when it moves in a straight line at constant speed.

Reference frames in which Newtonian mechanics holds are called inertial reference frames or inertial frames.

Theorem 8.1: Newton's Laws**Newton's First Law**

If no net force acts on a body ($F_{net} = 0$), the body's velocity cannot change; that is, the body cannot accelerate.

Newton's Second Law

The net force on a body is equal to the product of the body's mass and its acceleration.

In equation form,

$$\vec{F}_{net} = m\vec{a} \quad (8.1)$$

The acceleration component along a given axis is caused only by the sum of the force components along that same axis, and not by force components along any other axis.

Newton's Third Law

When two bodies interact, the forces on the bodies from each other are always equal in magnitude and opposite in direction.

If a force \vec{F}_{BC} acts on body B due to body C , then there is a force \vec{F}_{CB} on body C due to body B :

$$\vec{F}_{BC} = -\vec{F}_{CB}$$

The forces are equal in magnitude but opposite in direction.

II. SIMULATION FOR NEWTON'S FIRST LAW WITH Box2D

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class NewtonFirstlaw : public Test
{
public:
    NewtonFirstlaw()
    {
        b2Body* ground = NULL;
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2
                (-46.0f, 46.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
    }
}
```

```

b2BodyDef bd;
ground = m_world->CreateBody(&bd);

b2EdgeShape shape;
shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2
    (46.0f, 46.0f));
ground->CreateFixture(&shape, 0.0f);
}

{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2
        (46.0f, 0.0f));
    ground->CreateFixture(&shape, 0.0f);
}

{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(-46.0f, 12.0f), b2Vec2
        (46.0f, 12.0f));
    ground->CreateFixture(&shape, 0.0f);
}

{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(-46.0f, 23.0f), b2Vec2
        (46.0f, 23.0f));
    ground->CreateFixture(&shape, 0.0f);
}

// Create the box as the movable object with friction
    0
b2PolygonShape boxShape;
boxShape.SetAsBox(0.5f, 0.5f);

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.75f;
boxFixtureDef.density = 7.3f; // this will affect the
    box mass
boxFixtureDef.friction = 0.0f;
boxFixtureDef.shape = &boxShape;

```

```
b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(5.0f, 0.5f);

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&
    boxFixtureDef);

// Create the box 2 with friction 0.5
b2PolygonShape boxShape2;
boxShape2.SetAsBox(0.5f, 0.5f);

b2FixtureDef boxFixtureDef2;
boxFixtureDef2.restitution = 0.75f;
boxFixtureDef2.density = 7.3f; // this will affect the
    box mass
boxFixtureDef2.friction = 0.5f;
boxFixtureDef2.shape = &boxShape2;

b2BodyDef boxBodyDef2;
boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(5.0f, 12.5f);

m_box2 = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_box2->CreateFixture(&
    boxFixtureDef2);

// Create the box 3 with friction 1
b2PolygonShape boxShape3;
boxShape3.SetAsBox(0.5f, 0.5f);

b2FixtureDef boxFixtureDef3;
boxFixtureDef3.restitution = 0.75f;
boxFixtureDef3.density = 7.3f; // this will affect the
    box mass
boxFixtureDef3.friction = 1.0f;
boxFixtureDef3.shape = &boxShape3;

b2BodyDef boxBodyDef3;
boxBodyDef3.type = b2_dynamicBody;
boxBodyDef3.position.Set(5.0f, 23.5f);

m_box3 = m_world->CreateBody(&boxBodyDef3);
b2Fixture *boxFixture3 = m_box3->CreateFixture(&
    boxFixtureDef3);
m_time = 0.0f;
}

b2Body* m_box;
```

```

b2Body* m_box2;
b2Body* m_box3;
float m_time;

void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_D:
            m_box->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
            m_box2->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
            m_box3->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
            break;
        case GLFW_KEY_A:
            m_box->SetLinearVelocity(b2Vec2(-10.0f, 0.0f))
                ;
            m_box2->SetLinearVelocity(b2Vec2(-10.0f, 0.0f)
                );
            m_box3->SetLinearVelocity(b2Vec2(-10.0f, 0.0f)
                );
            break;
        case GLFW_KEY_F:
            m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                true);
            m_box2->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                true);
            m_box3->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                true);
            break;
        case GLFW_KEY_I:
            m_box->ApplyLinearImpulseToCenter(b2Vec2(250.0f,
                0.0f), true);
            m_box2->ApplyLinearImpulseToCenter(b2Vec2(250.0f,
                0.0f), true);
            m_box3->ApplyLinearImpulseToCenter(b2Vec2(250.0f,
                0.0f), true);
            break;
    }
}

void Step(Settings& settings) override
{
    m_time += 1.0f / 60.0f; // assuming we are using
                           // frequency of 60 Hertz
    b2MassData massData = m_box->GetMassData();
    b2Vec2 position = m_box->GetPosition();
    b2Vec2 position2 = m_box2->GetPosition();
    b2Vec2 position3 = m_box3->GetPosition();
    b2Vec2 velocity1 = m_box->GetLinearVelocity();
}

```

```

        b2Vec2 velocity2 = m_box2->GetLinearVelocity();
        b2Vec2 velocity3 = m_box3->GetLinearVelocity();

        g_debugDraw.DrawString(5, m_textLine, "Time (in
            seconds)= %.6f", m_time);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box 1 position
            = (%4.1f, %4.1f)", position.x, position.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box 1 velocity
            = (%4.1f, %4.1f)", velocity1.x, velocity1.y);
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Box 2 position
            = (%4.1f, %4.1f)", position2.x, position2.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box 2 velocity
            = (%4.1f, %4.1f)", velocity2.x, velocity2.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box 3 position
            = (%4.1f, %4.1f)", position3.x, position3.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box 3 velocity
            = (%4.1f, %4.1f)", velocity3.x, velocity3.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Box 1 / 2 / 3
            Mass = %.6f", massData.mass);
        m_textLine += m_textIncrement;
        Test::Step(settings);

        printf("%4.2f %4.2f \n", velocity1.x, velocity1.y);

    }

    static Test* Create()
    {
        return new NewtonFirstlaw;
    }

};

static int testIndex = RegisterTest("Force and Motion", "Newton's
First Law", NewtonFirstlaw::Create);

```

C++ Code 42: *tests/newton_firstlaw.cpp* "Newton's First Law Box2D"

Some explanations for the codes:

- We are creating a closed 3 different grounds each at different y axis value

```
b2Body* ground = NULL;
```

```

{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(-46.0f,
        46.0f));
    ground->CreateFixture(&shape, 0.0f);
}
{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2(46.0f,
        46.0f));
    ground->CreateFixture(&shape, 0.0f);
}
{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
        0.0f));
    ground->CreateFixture(&shape, 0.0f);
}
{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(-46.0f, 12.0f), b2Vec2(46.0f,
        12.0f));
    ground->CreateFixture(&shape, 0.0f);
}
{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(-46.0f, 23.0f), b2Vec2(46.0f,
        23.0f));
    ground->CreateFixture(&shape, 0.0f);
}

```

- Next, we create three boxes with different coefficient of friction: 0, 0.5, and 1.

```
// Create the box as the movable object with friction 0
```

```
b2PolygonShape boxShape;
boxShape.SetAsBox(0.5f, 0.5f);

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.75f;
boxFixtureDef.density = 7.3f; // this will affect the box mass
boxFixtureDef.friction = 0.0f;
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(5.0f, 0.5f);

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef);

// Create the box 2 with friction 0.5
b2PolygonShape boxShape2;
boxShape2.SetAsBox(0.5f, 0.5f);

b2FixtureDef boxFixtureDef2;
boxFixtureDef2.restitution = 0.75f;
boxFixtureDef2.density = 7.3f; // this will affect the box mass
boxFixtureDef2.friction = 0.5f;
boxFixtureDef2.shape = &boxShape2;

b2BodyDef boxBodyDef2;
boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(5.0f, 12.5f);

m_box2 = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_box2->CreateFixture(&boxFixtureDef2
);

// Create the box 3 with friction 1
b2PolygonShape boxShape3;
boxShape3.SetAsBox(0.5f, 0.5f);

b2FixtureDef boxFixtureDef3;
boxFixtureDef3.restitution = 0.75f;
boxFixtureDef3.density = 7.3f; // this will affect the box mass
boxFixtureDef3.friction = 1.0f;
boxFixtureDef3.shape = &boxShape3;

b2BodyDef boxBodyDef3;
boxBodyDef3.type = b2_dynamicBody;
boxBodyDef3.position.Set(5.0f, 23.5f);
```

```
m_box3 = m_world->CreateBody(&boxBodyDef3);
b2Fixture *boxFixture3 = m_box3->CreateFixture(&boxFixtureDef3
);
m_time = 0.0f;
```

Don't forget to declare the variable of the boxes, and the time as well (**m_box**, **m_box2**, **m_box3**, **m_time**).

- We create keyboard press events when we press "A" the boxes will suddenly have velocity of 10, and when we press "D" the boxes will have velocity of -10 (in the direction of negative x axis). If we press "F" then all the boxes will be given force toward the negative x axis with power of 250, while if we press "G" the boxes will be given force toward the positive x axis with power of 250, you can notice that box with highest friction barely move with such high force. Learn the difference of setting constant velocity on an object or particle with giving it some kind of force from the rest state.

```
void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_D:
            m_box->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
            m_box2->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
            m_box3->SetLinearVelocity(b2Vec2(10.0f, 0.0f));
            break;
        case GLFW_KEY_A:
            m_box->SetLinearVelocity(b2Vec2(-10.0f, 0.0f));
            m_box2->SetLinearVelocity(b2Vec2(-10.0f, 0.0f));
            m_box3->SetLinearVelocity(b2Vec2(-10.0f, 0.0f));
            break;
        case GLFW_KEY_F:
            m_box->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f),
                true);
            m_box2->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f)
                , true);
            m_box3->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f)
                , true);
            break;
        case GLFW_KEY_G:
            m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                true);
            m_box2->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                true);
            m_box3->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                true);
            break;
    }
}
```

- Next, we will showing some position data of all the boxes, all of them share the same mass.

As usual, in the end we print the velocity of box 1, so that we can plot the graph of the velocity and the acceleration.

```

void Step(Settings& settings) override
{
    m_time += 1.0f / 60.0f; // assuming we are using
                           // frequency of 60 Hertz
    b2MassData massData = m_box->GetMassData();
    b2Vec2 position = m_box->GetPosition();
    b2Vec2 position2 = m_box2->GetPosition();
    b2Vec2 position3 = m_box3->GetPosition();
    b2Vec2 velocity1 = m_box->GetLinearVelocity();
    b2Vec2 velocity2 = m_box2->GetLinearVelocity();
    b2Vec2 velocity3 = m_box3->GetLinearVelocity();

    g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)
        = %.6f", m_time);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Box 1 position =
        (%4.1f, %4.1f)", position.x, position.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Box 1 velocity =
        (%4.1f, %4.1f)", velocity1.x, velocity1.y);
    m_textLine += m_textIncrement;

    g_debugDraw.DrawString(5, m_textLine, "Box 2 position =
        (%4.1f, %4.1f)", position2.x, position2.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Box 2 velocity =
        (%4.1f, %4.1f)", velocity2.x, velocity2.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Box 3 position =
        (%4.1f, %4.1f)", position3.x, position3.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Box 3 velocity =
        (%4.1f, %4.1f)", velocity3.x, velocity3.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Box 1 / 2 / 3
        Mass = %.6f", massData.mass);
    m_textLine += m_textIncrement;
    Test::Step(settings);

    printf("%4.2f %4.2f \n", velocity1.x, velocity1.y);
}

```

Now, we can plot the velocity of the box with gnuplot, recompile the testbed to make the change occurs then type:

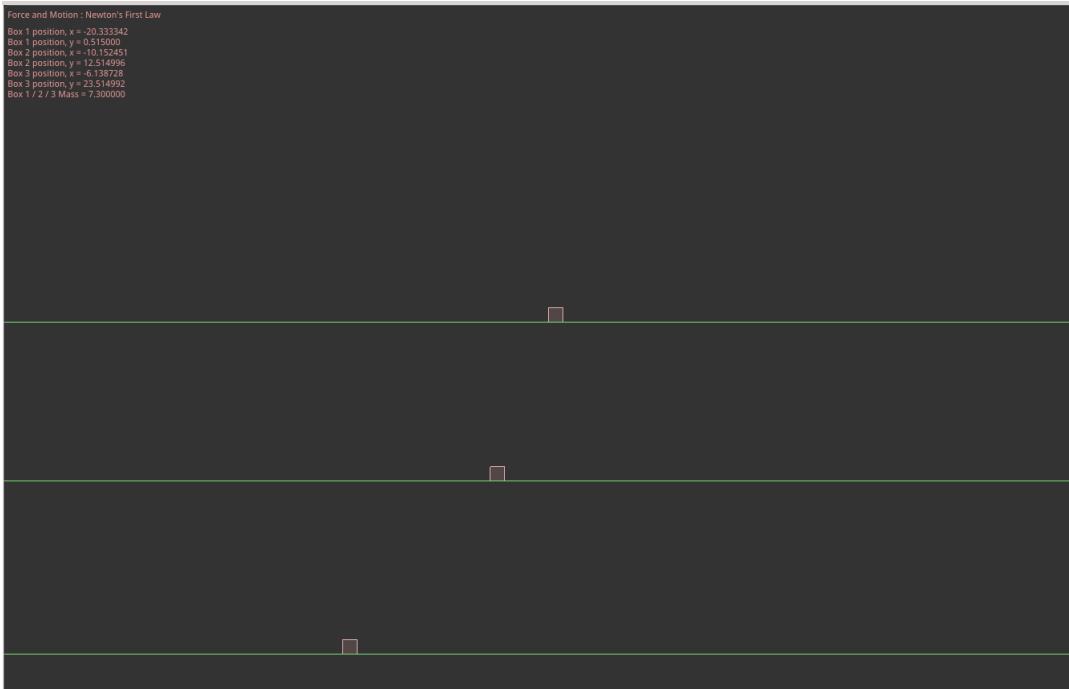


Figure 8.1: The simulation of three boxes being pulled to the left at the speed of $\vec{v} = 10$ (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/newton_firstlaw.cpp`).

```
./testbed > newtonfirstoutput.txt
```

Plot it with gnuplot from the working directory:

```
gnuplot
set xlabel "time"
set ylabel ""
delta_v(x) = (vD = x - old_v, old_v = x, vD)
old_v = NaN
plot "newtonfirstoutput.txt" using 1 with lines title "v_x", "newtonfirstoutput.txt" using0:(delta_v($1))
with lines title "a_{x}"
```

Figure it would be trickier to plot the acceleration, compared to the previous chapter. We have velocity in x axis, then we remember that instantaneous acceleration is defined as

$$\vec{a} = \frac{d\vec{v}}{dt}$$

and the average acceleration during Δt is:

$$\vec{a}_{avg} = \frac{v_2 - v_1}{\Delta t}$$

Thus while the box is moving at constant velocity, the acceleration is read as 0 in the graph.

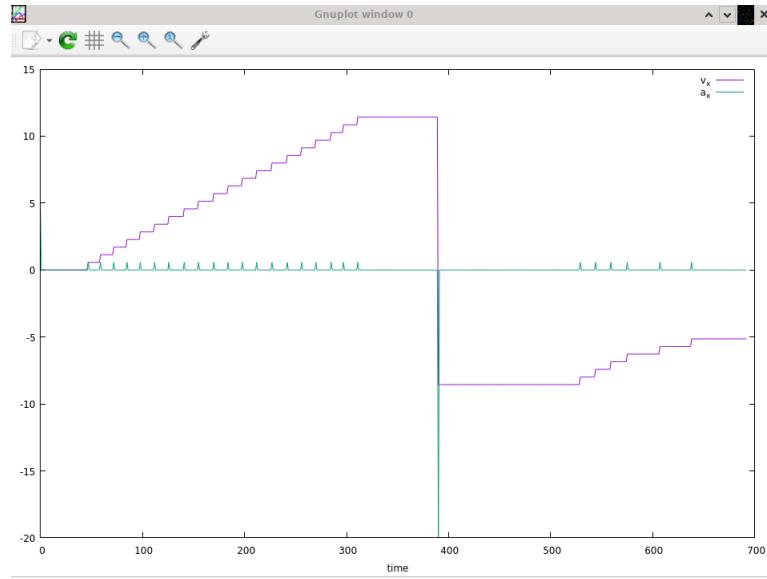


Figure 8.2: The gnuplot of the velocity and the acceleration of box 1 that is being push with force of 250 toward positive x axis then hit the wall and go back then push with force of 250 again toward positive x axis.

III. SIMULATION FOR NEWTON'S FIRST LAW AND 2 DIMENSIONAL FORCES WITH Box2D

We are going to simulate the movement of a box with forces from different directions:

1. South West
2. North East
3. North West
4. South East

Remember carefully that

$$\cos(\theta) = \cos(-\theta)$$

and

$$\cos(\theta) = -\cos(180^\circ - \theta)$$

```
#define DEGTORAD 0.0174532925199432957f
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class NewtonFirstlaw2dforces : public Test
{
public:
    NewtonFirstlaw2dforces()
```

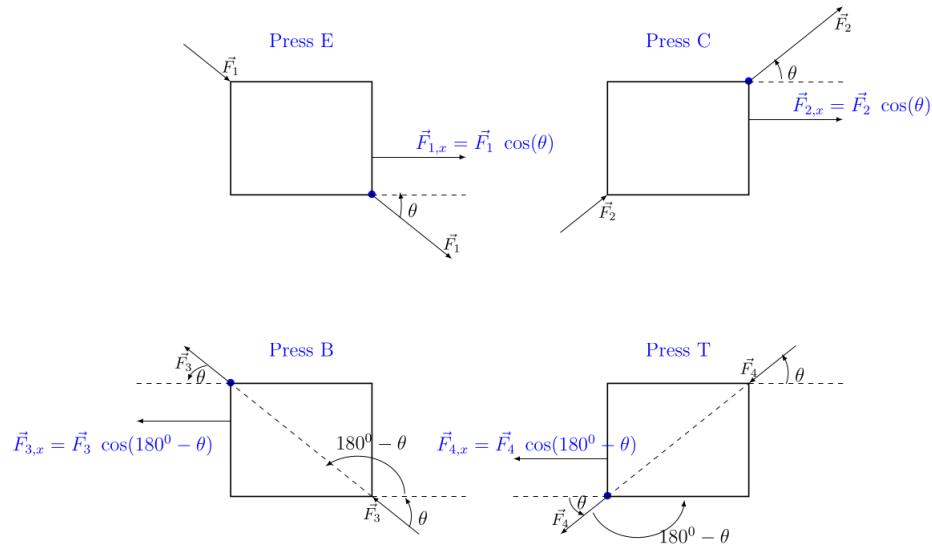


Figure 8.3: The schemes of forces that are coming from 4 different directions, a force from North West is coming by pressing "E", a force from South West is coming by pressing "C" and both will push the box to the right. A force from South East is coming by pressing "B", a force from North East is coming by pressing "T" and both will push the box to the left. This is more of a geometry trick.

```
{
    m_world->SetGravity(b2Vec2(0.0f, -9.8f));
    b2Body* ground = NULL;
{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(-46.0f,
        , 23.0f));
    ground->CreateFixture(&shape, 0.0f);
}
{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2(46.0f,
        , 23.0f));
    ground->CreateFixture(&shape, 0.0f);
}
{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);
```

```

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
0.0f));
        ground->CreateFixture(&shape, 0.0f);
    }

    // Create the box as the movable object with friction 0.3
    b2PolygonShape boxShape;
    boxShape.SetAsBox(0.5f, 0.5f);

    b2FixtureDef boxFixtureDef;
    boxFixtureDef.restitution = 0.75f;
    boxFixtureDef.density = 3.3f; // this will affect the box
        mass
    boxFixtureDef.friction = 0.3f;
    boxFixtureDef.shape = &boxShape;

    b2BodyDef boxBodyDef;
    boxBodyDef.type = b2_dynamicBody;
    boxBodyDef.position.Set(5.0f, 0.5f);

    m_box = m_world->CreateBody(&boxBodyDef);
    b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
        ;

    m_time = 0.0f;
}
b2Body* m_box;
float m_time;
int theta = 30;
float deginrad = theta*DEGTORAD;
float deginrad2 = (180-theta)*DEGTORAD;
void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_F:
            m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f), true);

            break;
        case GLFW_KEY_G:
            m_box->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f), true
                );
            break;
        case GLFW_KEY_E:
            m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(-
                deginrad), 0.0f), true);
            break;
    }
}

```

```

        case GLFW_KEY_C:
            m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(deginrad)
                , 0.0f), true);
            break;
        case GLFW_KEY_T:
            m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(-
                deginrad2), 0.0f), true);
            break;
        case GLFW_KEY_B:
            m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(deginrad2
                ), 0.0f), true);
            break;
    }
}

void Step(Settings& settings) override
{
    m_time += 1.0f / 60.0f; // assuming we are using frequency of
                           // 60 Hertz
    b2MassData massData = m_box->GetMassData();
    b2Vec2 position = m_box->GetPosition();
    b2Vec2 velocity1 = m_box->GetLinearVelocity();

    g_debugDraw.DrawString(5, m_textLine, "Press E, C, T, or B
        for adding force from 4 different directions");
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
        f", m_time);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Box position = (%4.1f,
        %4.1f)", position.x, position.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Box velocity = (%4.1f,
        %4.1f)", velocity1.x, velocity1.y);
    m_textLine += m_textIncrement;

    g_debugDraw.DrawString(5, m_textLine, "Box Mass = %.6f",
        massData.mass);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "cos(150) = %.6f", cosf
        (150*DEGTORAD));
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "cos(-150) = %.6f",
        cosf(-150*DEGTORAD));
    m_textLine += m_textIncrement;
    Test::Step(settings);

    printf("%4.2f %4.2f \n", velocity1.x, velocity1.y);
}

```

```

        static Test* Create()
    {
        return new NewtonFirstlaw2dforces;
    }

};

static int testIndex = RegisterTest("Force and Motion", "Newton's First Law
2D Forces", NewtonFirstlaw2dforces::Create);

```

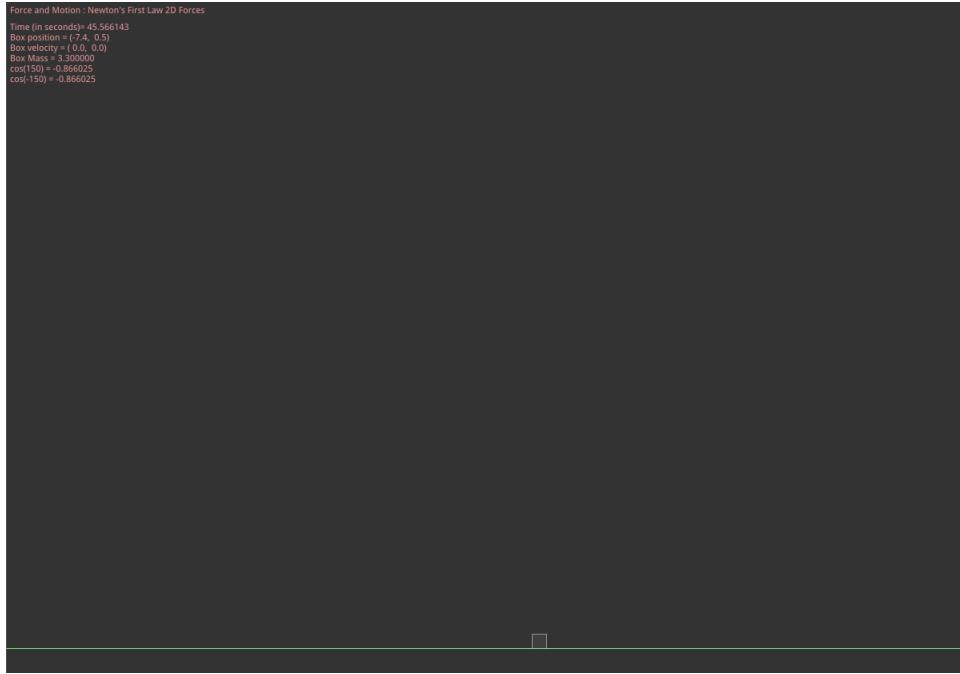
C++ Code 43: tests/newton_firstlaw2dforces.cpp "Newton's First Law and 2D Forces Box2D"

Figure 8.4: The simulation of a box that we can push from 4 different directions of South West, North West, North East and South East, we set the friction to 0.3 thus the box will stop eventually (the current simulation code can be located in: *DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/newton_firstlaw2dforces.cpp*).

Some explanations for the codes:

- Create a box as dynamic body with friction 0.3 and will be located at $x = 5$, $y = 0.5$, it will fall to the ground at $y = 0$ due to the gravity.

```

// Create the box as the movable object with friction 0.3
b2PolygonShape boxShape;
boxShape.SetAsBox(0.5f, 0.5f);

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.75f;
boxFixtureDef.density = 3.3f; // this will affect the box mass

```

```

boxFixtureDef.friction = 0.3f;
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(5.0f, 0.5f);

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef);

m_time = 0.0f;

```

- Declare class variable for the box, the time and the degree of the forces, then we convert it into radian.

```

b2Body* m_box;
float m_time;
int theta = 30;
float deginrad = theta*DEGTORAD;
float deginrad2 = (180-theta)*DEGTORAD;

```

- To manage the keyboard press events,
 1. When we press "F", a force of 250 N is going toward the positive x direction.
 2. When we press "G", a force of 250 N is going toward the negative x direction, thus the force is written as negative.
 3. When we press "E", a force of 250 N is coming from the North West direction going toward South East. Thus, we multiply the force with cosine of the angle between the force and the positive x axis. The **cosf(-deginrad)** means we are computing the degree in clockwise direction, since the positive degree should represent computing degree in counter clockwise direction. Imagine that we are computing the degree from from positive x axis, use vector $(1, 0)$, then rotate it clockwise direction till it become \vec{F}_1 .
 4. When we press "C", a force of 250 N is coming from the South West direction going toward North East. Thus, we multiply the force with cosine of the angle between the force and the positive x axis. The **cosf(deginrad)** means we are computing the degree in counter clockwise direction. Imagine that we are computing the degree from from positive x axis, use vector $(1, 0)$, then rotate it counter clockwise direction till it become \vec{F}_2 .
 5. When we press "B", a force of 250 N is coming from the South East direction going toward North West. Thus, we multiply the force with cosine of $180 - \theta$ - the angle between the force and the positive x axis. The **cosf(deginrad2)** means we are computing the degree from from positive x axis, since we are using the input of positive vector force ($F = 250$, **ApplyForceToCenter(b2Vec2(250.0f*cosf(deginrad2), 0.0f), true)**), thus use vector $(1, 0)$, then rotate it counter clockwise direction till it become \vec{F}_3 that is going toward North West. Remember that cosine on the second quadrant ($90^\circ \leq \theta \leq 180^\circ$) is always negative, thus when we compute **cosf(deginrad2)**, we will get negative number times the force that is positive, will become negative, meaning the force coming from South East will push the box to go toward the negative x axis.

6. When we press "T", a force of 250 N is coming from the North East direction going toward South West. Thus, we multiply the force with cosine of 180 - the angle between the force and the positive x axis. The `cosf(-deginrad2)` means we are computing the degree from from positive x axis, since we are using the input of positive vector force ($F = 250$, `ApplyForceToCenter(b2Vec2(250.0f*cosf(deginrad), 0.0f), true)`), thus use vector $(1, 0)$, then rotate it clockwise direction till it become \vec{F}_4 that is going toward South West. Hence, we will get negative number times the force that is positive, will become negative, meaning the force coming from North East will push the box to go toward the negative x axis.

This command:

`ApplyForceToCenter(b2Vec2(250.0f*cosf(deginrad), 0.0f), true)`

is equal with

`ApplyForceToCenter(b2Vec2(-250.0f*cosf(deginrad), 0.0f), true)`

with `deginrad = 300` and `deginrad2 = 1500`

$$F \times \cos(30) = -F \times \cos(150)$$

```
void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_F:
            m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                                       true);
            break;
        case GLFW_KEY_G:
            m_box->ApplyForceToCenter(b2Vec2(-250.0f, 0.0f),
                                       true);
            break;
        case GLFW_KEY_E:
            m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(-
                deginrad), 0.0f), true);
            break;
        case GLFW_KEY_C:
            m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(
                deginrad), 0.0f), true);
            break;
        case GLFW_KEY_T:
            m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(-
                deginrad2), 0.0f), true);
            break;
        case GLFW_KEY_B:
            m_box->ApplyForceToCenter(b2Vec2(250.0f*cosf(
                deginrad2), 0.0f), true);
            break;
    }
}
```

- To print the time, mass, position and velocity of the box on the GUI

```

void Step(Settings& settings) override
{
    m_time += 1.0f / 60.0f; // assuming we are using
                           frequency of 60 Hertz
    b2MassData massData = m_box->GetMassData();
    b2Vec2 position = m_box->GetPosition();
    b2Vec2 velocity1 = m_box->GetLinearVelocity();

    g_debugDraw.DrawString(5, m_textLine, "Press E, C, T, or
        B for adding force from 4 different directions");
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)
        = %.6f", m_time);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Box position =
        (%4.1f, %4.1f)", position.x, position.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Box velocity =
        (%4.1f, %4.1f)", velocity1.x, velocity1.y);
    m_textLine += m_textIncrement;

    g_debugDraw.DrawString(5, m_textLine, "Box Mass = %.6f",
        massData.mass);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "cos(150) = %.6f",
        cosf(150*DEGTORAD));
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "cos(-150) = %.6f"
        , cosf(-150*DEGTORAD));
    m_textLine += m_textIncrement;
    Test::Step(settings);

    printf("%4.2f %4.2f \n", velocity1.x, velocity1.y);
}

```

IV. SOME PARTICULAR FORCES

[DF*] The Gravitational Force

A gravitational force \vec{F}_g on a body is a certain type of pull that is directed toward a second body. Thus, \vec{F}_g means a force that pulls that body toward the center of the earth. We shall assume that the ground is an inertial frame. The magnitude of the force is

$$F_g = mg \quad (8.2)$$

The same gravitational force still acts on the body even when the body is not in free fall, or at rest.

[DF*] Weight

The weight W of a body is the magnitude of the upward force needed to balance the gravitational force on the body. A body's weight is related to the body's mass by

$$W = mg \quad (8.3)$$

[DF*] The Normal Force

A normal force \vec{F}_N is the force on a body from a surface against which the body presses. The normal force is always perpendicular to the surface.

If there is a block of mass m presses down on a table, deforming the table because of the gravitational force F_g on the block, then we can write Newton's second law for a positive y axis as

$$\begin{aligned} F_N - F_g &= ma_y \\ F_N - mg &= ma_y \\ F_N &= mg + ma_y \\ F_N &= m(g + a_y) \end{aligned}$$

If the block is not accelerated then $a = 0$ and $F_N = mg$. The acceleration can be toward positive y axis or negative y axis.

[DF*] Friction

If we attempt to slide a body on a surface, the motion is resisted by bonding between the body and the surface. That's why after we push a box will stop, but if the surface is smooth or frictionless, it can keeps going, like sliding a puck on an ice rink.

[DF*] Tension

When a cord (or a rope, cable) is attached to a body and pulled taut, the cord pulls on the body with a force \vec{T} directed away from the body and along the cord.

For a massless cord (a cord with negligible mass), the pulls at both ends of the cord have the same magnitude T , even if the cord runs around a massless, frictionless pulley (a pulley with negligible mass and negligible friction on its axle to oppose its rotation).

V. SIMULATION FOR APPLYING NEWTON'S LAW WITH Box2D

I took the example from chapter 5' problem no 57 of [6]. Why? this reminds me of my high school year physics, Now we can simulate how the block moves, it is amazing, like a dream comes true.

Here it is the problem:

A box of mass $m_1 = 3.70 \text{ kg}$ on a frictionless plane inclined at angle $\theta = 30^\circ$ is connected by a cord over a massless, frictionless pulley to a second box of mass $m_2 = 2.30 \text{ kg}$. What are

- (a) The magnitude of the acceleration of each box
- (b) The direction of the acceleration of the hanging box
- (c) The tension in the cord

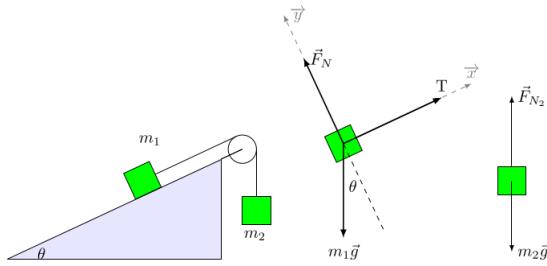


Figure 8.5: A box with mass m_1 on inclined plane of $\theta = 30^\circ$, is connected by a cord to hanging box on the right with mass m_2 (T is the tension in the cord). The middle picture is the body diagram of the first box, while the right picture is the body diagram of the second box.

Solution:

- (a) We take the positive x direction to be up the incline and the positive y direction to be in the direction of the normal force \vec{F}_N that the plane exerts on the box.

For the second box, we take the positive y direction to be down. In this way, the accelerations of the two blocks can be represented by the same symbol a , without ambiguity. Applying Newton's second law to the x and y axes for box 1 and to the y axis of box 2, we obtain

$$\begin{aligned} \sum F_{x_1} &= 0 \\ T_1 - m_1 g \sin \theta &= m_1 a \\ \sum F_{y_1} &= 0 \\ F_N - m_1 g \cos \theta &= 0 \\ \sum F_{y_2} &= 0 \\ m_2 g - T_2 &= m_2 a \end{aligned}$$

with $F_{N_2} = T_2$, the second equation $\sum F_{y_1} = F_N - m_1 g \cos \theta = 0$ is not needed in this problem, since the normal force is neither asked nor needed as part of the computation. We only use the first and third equations to obtain the values of a and T .

The tension for the cord connecting the first and second box will be the same, $T_1 = T_2 = T$, by applying Newton's laws, we add the first and third equations:

$$\begin{aligned} T_1 &= T_2 \\ m_2g - m_1g \sin \theta &= m_1a + m_2a \\ (m_2 - m_1 \sin \theta)g &= (m_1 + m_2)a \\ a &= \frac{(m_2 - m_1 \sin \theta)g}{m_1 + m_2} \\ a &= 0.735 \end{aligned}$$

the magnitude of the acceleration of the box is 0.735 m/s^2

- (b) Since the value of a is positive, the direction of the acceleration of the hanging box is indeed up the incline plane and bringing box 2 vertically down.
- (c) After we obtain the value of a , we can substitute to the equation 1 or 3 from part (a), thus

$$\begin{aligned} T &= m_1a + m_1g \sin \theta \\ &= (3.70)(0.735) + (3.70)(9.80) \sin 30^\circ \\ T &= 20.8 \end{aligned}$$

the tension in the cord is 20.8 N .

Now, the fun begins when we can see it simulated with computer, C++ and Box2D save our day.

```
#define DEGTORAD 0.0174532925199432957f
#include <iostream>
#include "test.h"

class PulleyJointTriangle : public Test
{
public:
    PulleyJointTriangle()
    {
        m_world->SetGravity(b2Vec2(0.0f, -9.8f));
        float L = 10.0f;
        float a = 1.0f;
        float b = 1.5f;

        b2Body* ground = NULL;
        // Create the pulley
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2CircleShape circle;
        circle.m_radius = 1.0f;

        circle.m_p.Set(0.0f, b + L); // circle with center of
                                    (0,b+L)
```

```

        ground->CreateFixture(&circle, 0.0f);
    }
    // Create the triangle
    b2ChainShape chainShape;
    b2Vec2 vertices[] = {b2Vec2(-17.3205,0), b2Vec2(0,0), b2Vec2
        (0,10)};
    chainShape.CreateLoop(vertices, 3);

    b2FixtureDef groundFixtureDef;
    groundFixtureDef.density = 0;
    groundFixtureDef.shape = &chainShape;

    b2BodyDef groundBodyDef;
    groundBodyDef.type = b2_staticBody;

    b2Body *groundBody = m_world->CreateBody(&groundBodyDef);
    b2Fixture *groundBodyFixture = groundBody->CreateFixture(&
        groundFixtureDef);

    {
        // Create the box on a triangle
        b2PolygonShape boxShape;
        boxShape.SetAsBox(a, b); // width and length of the
            box

        b2FixtureDef boxFixtureDef;
        boxFixtureDef.restitution = 0.75f;
        boxFixtureDef.density = 0.616985f; // this will affect
            the box mass
        boxFixtureDef.friction = 0.0f; // frictionless plane
        boxFixtureDef.shape = &boxShape;

        b2BodyDef boxBodyDef;
        boxBodyDef.type = b2_dynamicBody;
        boxBodyDef.position.Set(-3.0f, L);
        // boxBodyDef.fixedRotation = true;

        m_boxl = m_world->CreateBody(&boxBodyDef);
        b2Fixture *boxFixture = m_boxl->CreateFixture(&
            boxFixtureDef);

        // Create the box hanging on the right
        b2PolygonShape boxShape2;
        boxShape2.SetAsBox(a, b); // width and length of the
            box

        b2FixtureDef boxFixtureDef2;
        boxFixtureDef2.restitution = 0.75f;
    }
}

```

```

        boxFixtureDef2.density = 0.3835f; // this will affect
            the box mass, mass = density*5.9969
        boxFixtureDef2.friction = 0.3f;
        boxFixtureDef2.shape = &boxShape;

        b2BodyDef boxBodyDef2;
        boxBodyDef2.type = b2_dynamicBody;
        boxBodyDef2.position.Set(3.0f, L);
        // boxBodyDef2.fixedRotation = true;

        m_boxr = m_world->CreateBody(&boxBodyDef2);
        b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
            boxFixtureDef2);

        // Create the Pulley
        b2PulleyJointDef pulleyDef;
        b2Vec2 anchor1(-3.0f, L-1.5f); // the position of the
            end string of the left cord is (-3.0f,L-1.5)
        b2Vec2 anchor2(2.0f, L); // the position of the end
            string of the right cord is (2.0f,L)
        b2Vec2 groundAnchor1(-1.0f, b + L ); // the string of
            the cord is tightened at (-1.0f,b+L)
        b2Vec2 groundAnchor2(1.5f, b + L); // the string of
            the cord is tightened at (1.5f, b+L)
        // the last float is the ratio
        pulleyDef.Initialize(m_boxl, m_boxr, groundAnchor1,
            groundAnchor2, anchor1, anchor2, 1.0f);

        m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&
            pulleyDef);
    }
}

b2Body* m_boxl;
b2Body* m_boxr;
b2PulleyJoint* m_joint1;
void Step(Settings& settings) override
{
    Test::Step(settings);
    b2MassData massData1 = m_boxl->GetMassData();
    b2Vec2 position1 = m_boxl->GetPosition();
    b2Vec2 velocity1 = m_boxl->GetLinearVelocity();
    b2MassData massData2 = m_boxr->GetMassData();
    b2Vec2 position2 = m_boxr->GetPosition();
    b2Vec2 velocity2 = m_boxr->GetLinearVelocity();
    float m1 = massData1.mass;
    float m2 = massData2.mass;
    float g = 9.8f;
}

```

```

        float a = (m2-m1*sinf(30*DEGTORAD))*g / (m1+m2);
        float T = (m1*a) + (m1*g*sinf(30*DEGTORAD));

        g_debugDraw.DrawString(5, m_textLine, "Left Box position =
            (%4.1f, %4.1f)", position1.x, position1.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Left Box velocity =
            (%4.1f, %4.1f)", velocity1.x, velocity1.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Left Box Mass = %.6f",
            massData1.mass);
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Right Box position =
            (%4.1f, %4.1f)", position2.x, position2.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Right Box velocity =
            (%4.1f, %4.1f)", velocity2.x, velocity2.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Right Box Mass = %.6f"
            , massData2.mass);
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Acceleration = %.6f",
            a);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "The tension of the
            cord = %.6f", T);
        m_textLine += m_textIncrement;

        float ratio = m_joint1->GetRatio();
        float L = m_joint1->GetCurrentLengthA() + ratio * m_joint1->
            GetCurrentLengthB();
        g_debugDraw.DrawString(5, m_textLine, "L1 + %4.2f * L2 = %4.2
            f", (float) ratio, (float) L);
        m_textLine += m_textIncrement;
    }

    static Test* Create()
    {
        return new PulleyJointTriangle;
    }
};

static int testIndex = RegisterTest("Force and Motion", "Pulley Tension 1",
    PulleyJointTriangle::Create);

```

C++ Code 44: *tests/pulley_jointtriangle.cpp* "Applying Newton's Law Box2D"

Now go figures, when we play game with amazing graphics and physics engine like GTA series, Star Ocean, The Quarry, we then realize details are very important, to make it more realistic, you need to add more physics in every single objects in the game. The book does not state the length and width of the box, thus I input whatever I think is correct in the C++ codes, now when you run the simulation the boxes are rolling first at the initial state, then stabilize. By putting the masses as stated, box 1 has no friction with the inclined plane, the simulation is correct, the box 1 is being pulled up and the box 2 is going down even if the mass of box 2 is less than that of box 1. The equation can explain the world look-like / the graphics / the geometry of our system. Algebra solves first, then we can construct the geometry view.

If you try to pull the box 1 down the plane, with mouse click, it will sliding upward again.

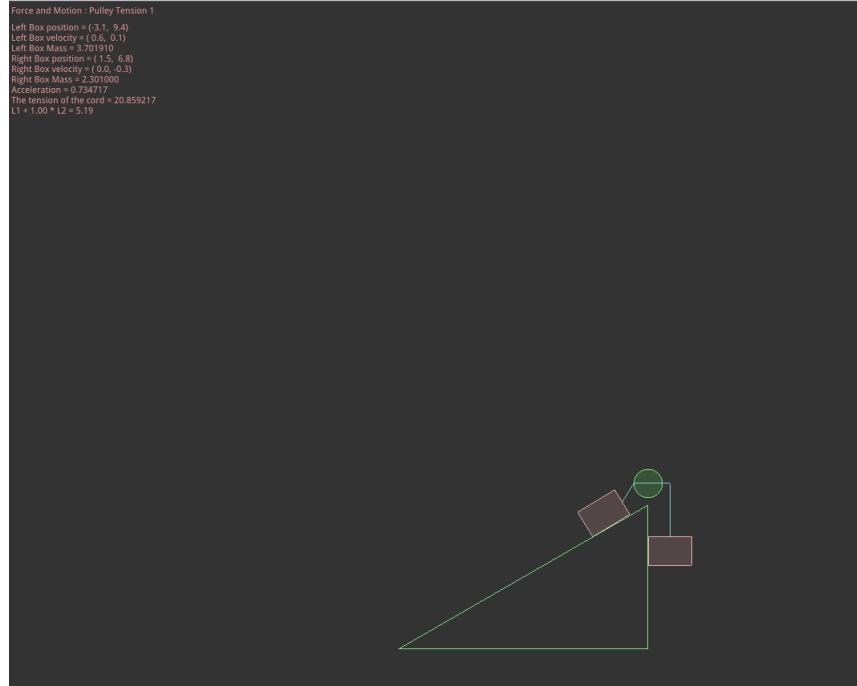


Figure 8.6: The simulation of a box on an inclined plane of $\theta = 30^\circ$ being connected with a cord and a pulley to another box on the right hanging at the right side of the plane (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/pulley_jointtriangle.cpp`).

Some explanations for the codes:

- To create the physics world with gravity going toward negative y axis of magnitude 9.8, the float variables a and b are to define the width and length of the boxes, while L is used to define the y positions of the cord connected to the left box and the right box, as well to the position of the cord where it will be tightened and connected to each of the box.

```
m_world->SetGravity(b2Vec2(0.0f,-9.8f));
float L = 10.0f;
float a = 1.0f;
float b = 1.5f;
```

- Create the pulley with a shape of a circle.

```
b2Body* ground = NULL;
// Create the pulley
b2BodyDef bd;
ground = m_world->CreateBody(&bd);

b2CircleShape circle;
circle.m_radius = 1.0f;

circle.m_p.Set(0.0f, b + L); // circle with center of
(0,b+L)
ground->CreateFixture(&circle, 0.0f);
}
```

- Create the inclined plane that has a shape of a triangle.

```
// Create the triangle
b2ChainShape chainShape;
b2Vec2 vertices[] = {b2Vec2(-17.3205,0), b2Vec2(0,0), b2Vec2
(0,10)};
chainShape.CreateLoop(vertices, 3);

b2FixtureDef groundFixtureDef;
groundFixtureDef.density = 0;
groundFixtureDef.shape = &chainShape;

b2BodyDef groundBodyDef;
groundBodyDef.type = b2_staticBody;

b2Body *groundBody = m_world->CreateBody(&groundBodyDef);
b2Fixture *groundBodyFixture = groundBody->CreateFixture(&
groundFixtureDef);
```

- Create the boxes and then the PulleyJoint, we are connecting 2 boxes, 2 pulley' anchor, and 2 pulley' ground anchor.

The **groundAnchor()** is used to show the pixel position of the cord where it is tightened or become static before connected to the box. While the **anchor()** is used to define pixel position for the end string of the cord to the box.

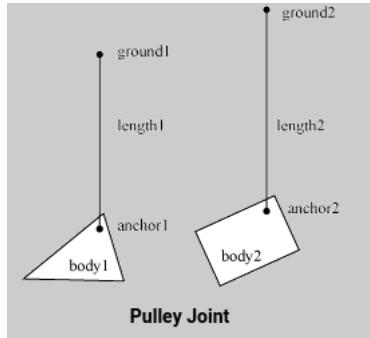


Figure 8.7: If the ratio is 2, then *length1* will vary at twice the rate of *length2*. The force in the rope attached to *body1* will have half the constraint force as the rope attached to *body2*.

```
{
    // Create the box on a triangle
    b2PolygonShape boxShape;
    boxShape.SetAsBox(a, b); // width and length of the box

    b2FixtureDef boxFixtureDef;
    boxFixtureDef.restitution = 0.75f;
    boxFixtureDef.density = 0.616985f; // this will affect
        the box mass
    boxFixtureDef.friction = 0.0f; // frictionless plane
    boxFixtureDef.shape = &boxShape;

    b2BodyDef boxBodyDef;
    boxBodyDef.type = b2_dynamicBody;
    boxBodyDef.position.Set(-3.0f, L);
    // boxBodyDef.fixedRotation = true;

    m_box1 = m_world->CreateBody(&boxBodyDef);
    b2Fixture *boxFixture = m_box1->CreateFixture(&
        boxFixtureDef);

    // Create the box hanging on the right
    b2PolygonShape boxShape2;
    boxShape2.SetAsBox(a, b); // width and length of the box

    b2FixtureDef boxFixtureDef2;
    boxFixtureDef2.restitution = 0.75f;
    boxFixtureDef2.density = 0.3835f; // this will affect
        the box mass, mass = density*5.9969
    boxFixtureDef2.friction = 0.3f;
    boxFixtureDef2.shape = &boxShape;

    b2BodyDef boxBodyDef2;
```

```

boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(3.0f, L);
// boxBodyDef2.fixedRotation = true;

m_boxr = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
    boxFixtureDef2);

// Create the Pulley
b2PulleyJointDef pulleyDef;
b2Vec2 anchor1(-3.0f, L-1.5f); // the position of the
    end string of the left cord is (-3.0f,L-1.5)
b2Vec2 anchor2(2.0f, L); // the position of the end
    string of the right cord is (2.0f,L)
b2Vec2 groundAnchor1(-1.0f, b + L ); // the string of
    the cord is tightened at (-1.0f,b+L)
b2Vec2 groundAnchor2(1.5f, b + L); // the string of the
    cord is tightened at (1.5f, b+L)
// the last float is the ratio
pulleyDef.Initialize(m_boxl, m_boxr, groundAnchor1,
    groundAnchor2, anchor1, anchor2, 1.0f);

m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&
    pulleyDef);
}

```

A pulley is used to create an idealized pulley. It connects two bodies to ground and to each other. As one body goes up, the other goes down.

- To show the mass, position, linear velocity, acceleration of the box, and the tension in the cord.

```

void Step(Settings& settings) override
{
    Test::Step(settings);
    b2MassData massData1 = m_boxl->GetMassData();
    b2Vec2 position1 = m_boxl->GetPosition();
    b2Vec2 velocity1 = m_boxl->GetLinearVelocity();
    b2MassData massData2 = m_boxr->GetMassData();
    b2Vec2 position2 = m_boxr->GetPosition();
    b2Vec2 velocity2 = m_boxr->GetLinearVelocity();
    float m1 = massData1.mass;
    float m2 = massData2.mass;
    float g = 9.8f;

    float a = (m2-m1*sinf(30*DEGTORAD))*g / (m1+m2);
    float T = (m1*a) + (m1*g*sinf(30*DEGTORAD));

    g_debugDraw.DrawString(5, m_textLine, "Left Box position
        = (%4.1f, %4.1f)", position1.x, position1.y);
}

```

```

    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Left Box velocity
        = (%4.1f, %4.1f)", velocity1.x, velocity1.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Left Box Mass =
        %.6f", massData1.mass);
    m_textLine += m_textIncrement;

    g_debugDraw.DrawString(5, m_textLine, "Right Box
        position = (%4.1f, %4.1f)", position2.x, position2.y
    );
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Right Box
        velocity = (%4.1f, %4.1f)", velocity2.x, velocity2.y
    );
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Right Box Mass =
        %.6f", massData2.mass);
    m_textLine += m_textIncrement;

    g_debugDraw.DrawString(5, m_textLine, "Acceleration =
        %.6f", a);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "The tension of
        the cord = %.6f", T);
    m_textLine += m_textIncrement;

    float ratio = m_joint1->GetRatio();
    float L = m_joint1->GetCurrentLengthA() + ratio *
        m_joint1->GetCurrentLengthB();
    g_debugDraw.DrawString(5, m_textLine, "L1 + %4.2f * L2 =
        %4.2f", (float) ratio, (float) L);
    m_textLine += m_textIncrement;
}

```

VI. FRICTION, DRAG FORCE, AND CENTRIPETAL FORCE

- [DF*] Frictional forces are unavoidable in our daily lives. It can stop every moving object and bring to a halt every rotating wheel, given a sufficient friction coefficient. If friction is absent, everything will be on the loose, like a puck sliding non-stop till forever on an ice rink.
- [DF*] When a force \vec{F} tends to slide a body along a surface, a frictional force from the surface acts on the body. The frictional force is parallel to the surface and directed so as to oppose the sliding. It is due to bonding between the atoms on the body and the atoms on the surface, an effect called cold-welding.
- [DF*] If the body does not slide, the frictional force is a static frictional force \vec{f}_s . If there is sliding, the frictional force is a kinetic frictional force \vec{f}_k .
- [DF*] Properties of Friction:

1. If a body does not move, the static frictional force, \vec{f}_s and the component of \vec{F} parallel to the surface are equal in magnitude, and \vec{f}_s is directed opposite that component. If the component increases, \vec{f}_s also increases.
2. The magnitude of \vec{f}_s has a maximum value $f_{s,\max}$ given by

$$f_{s,\max} = \mu_s F_N \quad (8.4)$$

where μ_s is the coefficient of static friction and F_N is the magnitude of the normal force. If the component of \vec{F} parallel to the surface exceeds $f_{s,\max}$, the static friction is overwhelmed and the body slides on the surface. In other words, a force that is parallel to the surface should be bigger than the coefficient of static friction times the normal force to make the body move from its' rest position.

3. If the body begins to slide on the surface, the magnitude of the frictional force rapidly decreases to a constant value f_k given by

$$f_k = \mu_k F_N \quad (8.5)$$

[DF*] When an applied force has overwhelmed the static frictional force. The object slides and accelerates.

VII. SIMULATION FOR STATIC FRICTIONAL FORCES WITH Box2D

I took the example from chapter 6' sample problem no 6.01 of [6].

Suppose there is a force coming from North West / tilted applied force. The force and the positive x axis makes an angle of 30° , the magnitude of the force is 50 N. The box has a mass of 3.3 kg, the coefficient of static friction between the box and the ground gloor is $\mu_s = 0.3$. What is the minimum magnitude of force needed to move the box from its' rest position?

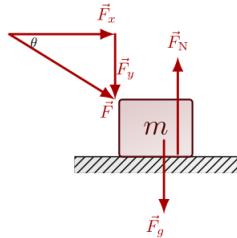


Figure 8.8: Angled force is applied to an initially stationary box from North West direction. These are the vertical force components: \vec{F}_N , \vec{F}_g and \vec{F}_y

Solution:

To see if the block slides, we must compare the applied force component F_x with the maximum magnitude $f_{s,\max}$ that the static friction can have. We see that

$$\begin{aligned} F_x &= F \cos \theta \\ &= 50(\cos 30) \\ F_x &= 43.30 \end{aligned}$$

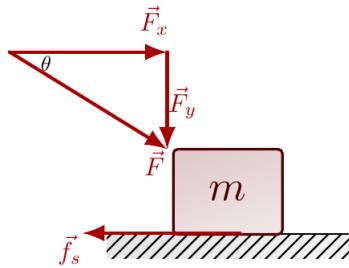


Figure 8.9: The horizontal force components will include \vec{f}_s and \vec{F}_x .

To calculate $f_{s,\max}$, we need the magnitude F_N of the normal force. Thus,

$$\begin{aligned}\sum F_y &= ma_y \\ F_N - F_g - F \sin \theta &= ma_y \\ F_N &= mg + F \sin \theta + m(0) \\ F_x &= mg + F \sin \theta\end{aligned}$$

Now, we can evaluate $f_{s,\max}$,

$$\begin{aligned}f_{s,\max} &= \mu_s F_N \\ &= (0.3)(mg + F \sin \theta) \\ &= (0.3)((3.3)(9.8) + 50 \sin 30^\circ) \\ &= 34.702\end{aligned}$$

The minimum force needed to push the box to move is 34.702 N, now since the force is 50 N tilted with angle of 30° , we are going to use F_x and see whether it will move or not (when $F_x - f_{s,\max} > 0$ the box will move),

$$\begin{aligned}\sum F_x &= ma_x \\ F_x - f_{s,\max} &= 43.30 - 34.702 \\ 43.30 - 34.702 &> 0\end{aligned}$$

The box will move with that force, since we will obtain $a_x > 0$. Now, we can try to prove the physics equation above for another case, if we push the box with another force that is parallel toward the surface, it will be easier to make the box move, as the $f_{s,\max}$ will be smaller due to $\sin \theta = 0$ for parallel force. We can see it from the simulation with Box2D.

```
#define DEGTORAD 0.0174532925199432957f
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class FrictionalForces : public Test
{
public:
```

```

FrictionalForces()
{
    b2Body* ground = NULL;
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(-46.0f,
                                                       46.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2(46.0f,
                                                       46.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                                                       0.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
    // Create the box as the movable object with friction 0.3
    b2PolygonShape boxShape;
    boxShape.SetAsBox(0.5f, 0.5f);

    b2FixtureDef boxFixtureDef;
    boxFixtureDef.restitution = 0.75f;
    boxFixtureDef.density = 3.3f; // this will affect the box
                                 mass
    boxFixtureDef.friction = 0.3f;
    boxFixtureDef.shape = &boxShape;

    b2BodyDef boxBodyDef;
    boxBodyDef.type = b2_dynamicBody;
    boxBodyDef.position.Set(5.0f, 0.5f);

    m_box = m_world->CreateBody(&boxBodyDef);
    b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
;
}

```

```

        m_time = 0.0f;
    }
    b2Body* m_box;
    float m_time;
    float F = 50.0f;
    int theta = 30;
    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_A:
                m_box->ApplyForceToCenter(b2Vec2(F*cosf(theta*
                    DEGTORAD), F*sinf(theta*DEGTORAD)), true);
                break;
            case GLFW_KEY_S:
                m_box->ApplyForceToCenter(b2Vec2(50.0f, 0.0f), true);

                break;
            case GLFW_KEY_D:
                m_box->ApplyForceToCenter(b2Vec2(100.0f, 0.0f), true);

                break;
            case GLFW_KEY_F:
                m_box->ApplyForceToCenter(b2Vec2(150.0f, 0.0f), true);

                break;
            case GLFW_KEY_G:
                m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f), true);

                break;
        }
    }
    void Step(Settings& settings) override
    {
        m_time += 1.0f / 60.0f; // assuming we are using frequency of
        // 60 Hertz
        b2MassData massData = m_box->GetMassData();
        b2Vec2 position = m_box->GetPosition();
        b2Vec2 velocity1 = m_box->GetLinearVelocity();
        float m = massData.mass;
        float fs_max = (0.3 * m * 9.8) + (F*sinf(theta*DEGTORAD));

        g_debugDraw.DrawString(5, m_textLine, "Press A to push the
            box with magnitude 50 N at a downward angle of 30 degree"
            );
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Press S, D, F, G to

```

```

        push the box with magnitude 50 N, 100 N, 150 N, 250 N
        respectively, with direction toward positive x axis");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
f", m_time);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "The maximum static
frictional force (f_{s,max})= %.6f", fs_max);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Box 1 position = (%4.1
f, %4.1f)", position.x, position.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Box 1 velocity = (%4.1
f, %4.1f)", velocity1.x, velocity1.y);
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "Box Mass = %.6f",
massData.mass);
m_textLine += m_textIncrement;
Test::Step(settings);

printf("%4.2f %4.2f \n", velocity1.x, velocity1.y);
}
static Test* Create()
{
    return new FrictionalForces;
}

};

static int testIndex = RegisterTest("Force and Motion", "Frictional Forces"
, FrictionalForces::Create);

```

C++ Code 45: *tests/frictional_forces.cpp* "Static Frictional Forces Box2D"

Some explanations for the codes:

- For the keyboard press events:

1. If you press "A" you will push the box from North West, with tilted angle of $\theta = 30^\circ$ toward the positive x axis direction.
2. If you press "S" you will push the box with force of 50 N parallel to the surface toward the positive x axis direction.
3. If you press "D" you will push the box with force of 150 N parallel to the surface toward the positive x axis direction.
4. If you press "F" you will push the box with force of 150 N parallel to the surface toward the positive x axis direction.
5. If you press "G" you will push the box with force of 250 N parallel to the surface toward the positive x axis direction.

Notice that you need to push the key repeatedly to apply the force more than once so it will keep moving, since **ApplyForceToCenter()** only give force in an instant not constantly giving it power to move / translate toward x and y axis all the time like **SetLinearVelocity()**.

```
void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_A:
            m_box->ApplyForceToCenter(b2Vec2(F*cosf(theta*
                DEGTORAD), F*sinf(theta*DEGTORAD)), true);
            break;
        case GLFW_KEY_S:
            m_box->ApplyForceToCenter(b2Vec2(50.0f, 0.0f),
                true);
            break;
        case GLFW_KEY_D:
            m_box->ApplyForceToCenter(b2Vec2(100.0f, 0.0f),
                true);
            break;
        case GLFW_KEY_F:
            m_box->ApplyForceToCenter(b2Vec2(150.0f, 0.0f),
                true);
            break;
        case GLFW_KEY_G:
            m_box->ApplyForceToCenter(b2Vec2(250.0f, 0.0f),
                true);
            break;
    }
}
```

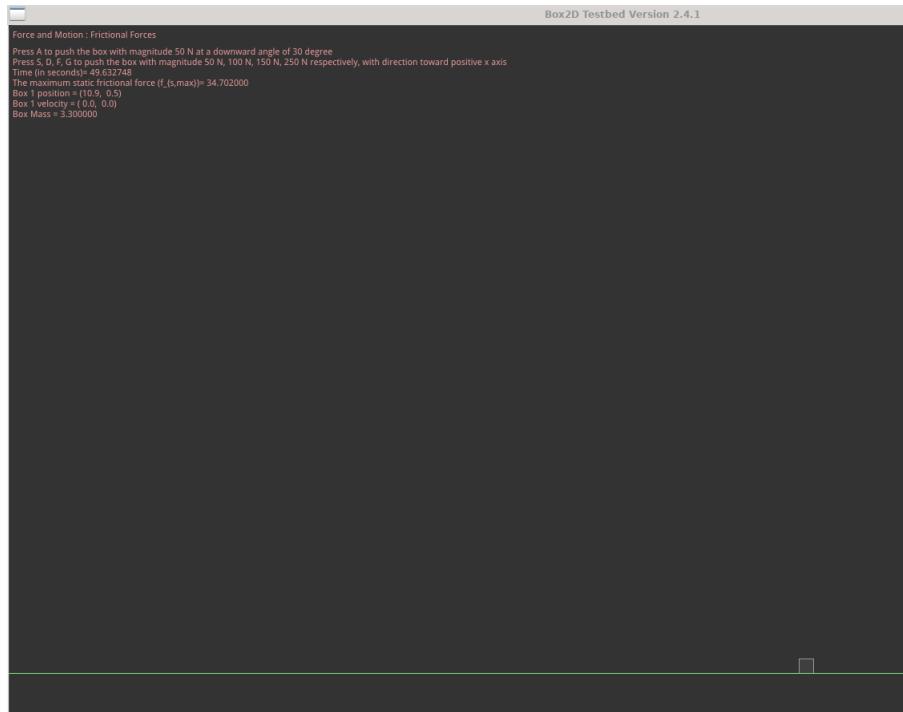


Figure 8.10: The simulation of a box being pushed with different forces, first when pressing "A" with force coming from North West with a tilted angle of $\theta = 30^\circ$, then another key presses of "S", "D", "F", "G" to push the box parallel to the surface toward positive x axis with force of 50, 100, 150, 250 N respectively (the current simulation code can be located in: **DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/pulley_jointtriangle.cpp**).

VIII. SIMULATION FOR KINETIC FRICTION WITH Box2D

Taken from problem no 23 chapter 6 of [6], where there are 3 boxes, one hanging on the left, one on the table, and the third box is hanging on the right, the left and center box is connected with a cord and a pulley on the left side of the table, while the center and the right box is connected with a cord and the pulley on the right side of the table.

This simulation wants to show that boxes that are moving will make our first concern toward kinetic friction, since we already pass the force to make the resting bodies move, say goodbye to static friction and hello kinetic friction.

When the three boxes are released from rest, they accelerate with a magnitude of 0.500 m/s^2 . Box 1(on the left) has mass M , box 2(the center box) has mass $2M$, and box 3 (on the right) has mass $2M$. What is the coefficient of kinetic friction between box 2 and the table?

Solution:

Let the tensions on the strings connecting m_2 and m_3 be T_{23} and that connecting m_2 and m_1 be T_{12} ,respectively. Applying Newton's second law to the system we have

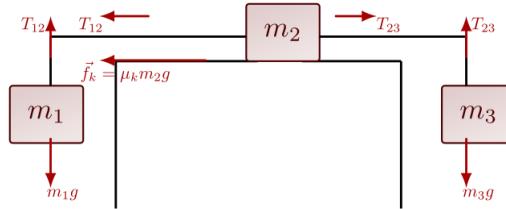


Figure 8.11: The force and kinetic friction components on the system.

For the box on the left:

$$\begin{aligned}\sum F_y &= m_1 a \\ T_{12} - m_1 g &= m_1 a\end{aligned}$$

For the box on the table

$$\begin{aligned}\sum F_x &= m_2 a \\ T_{23} - \mu_k m_2 g - T_{12} &= m_2 a\end{aligned}$$

For the box on the right:

$$\begin{aligned}\sum F_y &= m_3 a \\ m_3 g - T_{23} &= m_3 a\end{aligned}$$

Since this is a system, all 3 boxes connected, we know that a has one value, which is $a = 0.500$, thus by adding three equations for each boxes,

$$\begin{aligned}
 T_{12} - m_1g + (T_{23} - \mu_k m_2g - T_{12}) + (m_3g - T_{23}) &= m_1a + m_2a + m_3a \\
 -Mg - 2\mu_k Mg + 2Mg &= 5Ma \\
 2g - g - 2\mu_k g &= 5a \\
 2(9.8) - 9.8 - 2(\mu_k)(9.8) &= 5(0.500) \\
 \mu_k &= 0.3724
 \end{aligned}$$

Thus, the coefficient of kinetic friction in this system is $\mu_k = 0.3724$. Now for the simulation with Box2D:

```

#include <iostream>
#include "test.h"

class PulleyKineticFriction : public Test
{
public:
    PulleyKineticFriction()
    {
        m_world->SetGravity(b2Vec2(0.0f, -9.8f));
        float L = 10.0f;
        float a = 1.0f;
        float b = 1.5f;

        b2Body* ground = NULL;
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shapeGround;
            shapeGround.SetTwoSided(b2Vec2(-30.0f, 0.0f), b2Vec2
                (30.0f, 0.0f));
            ground->CreateFixture(&shapeGround, 0.0f);
        }
        // Create the pulley on the right
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2CircleShape circle;
        circle.m_radius = 1.0f;

        circle.m_p.Set(0.0f, b + L); // circle with center of
        (0, b+L)
        ground->CreateFixture(&circle, 0.0f);
    }
    // Create the pulley on the left
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);
}

```

```

        b2CircleShape circle;
        circle.m_radius = 1.0f;

        circle.m_p.Set(-17.0f, b + L); // circle with center
                                      of (0,b+L)
        ground->CreateFixture(&circle, 0.0f);
    }

    // Create the table
    b2ChainShape chainShape;
    b2Vec2 vertices[] = {b2Vec2(-17.3205,0), b2Vec2(0,0), b2Vec2
    (0,10),b2Vec2(-17.3205,10)};
    chainShape.CreateLoop(vertices, 4);

    b2FixtureDef groundFixtureDef;
    groundFixtureDef.density = 0;
    groundFixtureDef.shape = &chainShape;

    b2BodyDef groundBodyDef;
    groundBodyDef.type = b2_staticBody;

    b2Body *groundBody = m_world->CreateBody(&groundBodyDef);
    b2Fixture *groundBodyFixture = groundBody->CreateFixture(&
        groundFixtureDef);

    {

        // Create the box hanging on the left
        b2PolygonShape boxShape0;
        boxShape0.SetAsBox(a/2, a/2); // width and length of
                                     the box

        b2FixtureDef boxFixtureDef0;
        boxFixtureDef0.restitution = 0.75f;
        boxFixtureDef0.density = 2.001034f; // this will
                                         affect the box mass
        boxFixtureDef0.friction = 0.0f; // frictionless plane
        boxFixtureDef0.shape = &boxShape0;

        b2BodyDef boxBodyDef0;
        boxBodyDef0.type = b2_dynamicBody;
        boxBodyDef0.position.Set(-18.0f, L-a);
        // boxBodyDef.fixedRotation = true;

        m_boxl = m_world->CreateBody(&boxBodyDef0);
        b2Fixture *boxFixture0 = m_boxl->CreateFixture(&
            boxFixtureDef0);

        // Create the box on a table
    }
}

```

```

b2PolygonShape boxShape;
boxShape.SetAsBox(a, a); // width and length of the
box

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.75f;
boxFixtureDef.density = 1.0005169f; // this will
affect the box mass
boxFixtureDef.friction = 0.0f; // frictionless plane
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(-10.0f, L);
// boxBodyDef.fixedRotation = true;

m_boxc = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_boxc->CreateFixture(&
boxFixtureDef);

// Create the box hanging on the right
b2PolygonShape boxShape2;
boxShape2.SetAsBox(a,a); // width and length of the
box

b2FixtureDef boxFixtureDef2;
boxFixtureDef2.restitution = 0.75f;
boxFixtureDef2.density = 1.0005169f; // this will
affect the box mass, mass depends on dimension too
boxFixtureDef2.friction = 0.3f;
boxFixtureDef2.shape = &boxShape;

b2BodyDef boxBodyDef2;
boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(3.0f, L);
// boxBodyDef2.fixedRotation = true;

m_boxr = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
boxFixtureDef2);

// Create the Pulley on the right
b2PulleyJointDef pulleyDef;
b2Vec2 anchor1(-9.1f, L+a/2); // the position of the
end string of the left cord is (-3.0f, L-1.5)
b2Vec2 anchor2(2.0f, L); // the position of the end
string of the right cord is (2.0f, L)
b2Vec2 groundAnchor1(-1.0f, b + L ); // the string of

```

```

        the cord is tightened at (-1.0f, b+L)
        b2Vec2 groundAnchor2(1.5f, b + L); // the string of
        the cord is tightened at (1.5f, b+L)
        // the last float is the ratio
        pulleyDef.Initialize(m_boxc, m_boxr, groundAnchor1,
            groundAnchor2, anchor1, anchor2, 1.0f);

        m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&
            pulleyDef);

        // Create the Pulley on the left
        b2PulleyJointDef pulleyDef1;
        b2Vec2 anchor3(-18.0f, L-1.5f);
        b2Vec2 anchor4(-11.0f, L+a/2);
        b2Vec2 groundAnchor3(-18.5f, b + L );
        b2Vec2 groundAnchor4(-15.5f, b + L);
        pulleyDef1.Initialize(m_boxl, m_boxc, groundAnchor3,
            groundAnchor4, anchor3, anchor4, 1.0f);

        m_joint2 = (b2PulleyJoint*)m_world->CreateJoint(&
            pulleyDef1);
    }

    b2Body* m_boxc;
    b2Body* m_boxl;
    b2Body* m_boxr;
    b2PulleyJoint* m_joint1;
    b2PulleyJoint* m_joint2;

    void Step(Settings& settings) override
    {
        Test::Step(settings);
        b2MassData massData0 = m_boxl->GetMassData();
        b2Vec2 position0 = m_boxl->GetPosition();
        b2Vec2 velocity0 = m_boxl->GetLinearVelocity();
        b2MassData massData1 = m_boxc->GetMassData();
        b2Vec2 position1 = m_boxc->GetPosition();
        b2Vec2 velocity1 = m_boxc->GetLinearVelocity();
        b2MassData massData2 = m_boxr->GetMassData();
        b2Vec2 position2 = m_boxr->GetPosition();
        b2Vec2 velocity2 = m_boxr->GetLinearVelocity();
        float m1 = massData1.mass;
        float m2 = massData2.mass;
        float g = 9.8f;
        float a = 0.500;
        float fk = (5*a-g)/(-2*g);

        g_debugDraw.DrawString(5, m_textLine, "Left Box position =

```

```

        (%4.1f, %4.1f)", position0.x, position0.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Left Box velocity =
        (%4.1f, %4.1f)", velocity0.x, velocity0.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Left Box Mass = %.6f",
        massData0.mass);
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "Center Box position =
        (%4.1f, %4.1f)", position1.x, position1.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Center Box velocity =
        (%4.1f, %4.1f)", velocity1.x, velocity1.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Center Box Mass = %.6f"
        , massData1.mass);
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "Right Box position =
        (%4.1f, %4.1f)", position2.x, position2.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Right Box velocity =
        (%4.1f, %4.1f)", velocity2.x, velocity2.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Right Box Mass = %.6f"
        , massData2.mass);
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "Acceleration = %.6f",
        a);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Coefficient Kinetic
        Friction= %.6f", fk);
m_textLine += m_textIncrement;

float ratio = m_joint1->GetRatio();
float L = m_joint1->GetCurrentLengthA() + ratio * m_joint1->
        GetCurrentLengthB();
g_debugDraw.DrawString(5, m_textLine, "L1 + %4.2f * L2 = %4.2
        f", (float) ratio, (float) L);
m_textLine += m_textIncrement;
}

static Test* Create()
{
    return new PulleyKineticFriction;
}

```

```

};

static int testIndex = RegisterTest("Force and Motion", "Pulley Kinetic
Friction", PulleyKineticFriction::Create);

```

C++ Code 46: *tests/pulley_kineticfriction.cpp* "Pulley Kinetic Friction Box2D"

What interesting is coefficient of kinetic friction for this system is not depending on the floating number of each mass. Since the mass cancel itself, it depends on the proportion between the masses in the system.

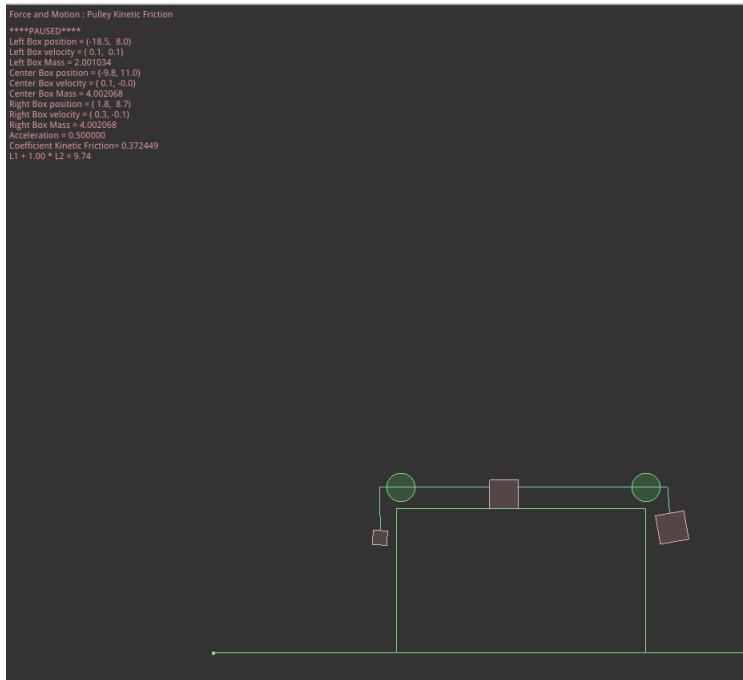


Figure 8.12: The simulation of a physical system, with 3 boxes, one hanging on the left, one on the table, and one hanging on the right, all connected with cords and pulleys (the current simulation code can be located in: *DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/pulley_kineticfriction.cpp*).

Some explanations for the codes:

- To create the correct Pulley Joint, for the left and right pulley and the anchor as well, I need to measure the location of the box on the table, thus know the *x* position needs to put on **anchor1()** and **anchor4()** that will connect the cord from the left (**anchor4()**) and the right (**anchor1()**) to the box on the table. Quite trial and error while coding this for Box2D.

```

// Create the Pulley on the right
b2PulleyJointDef pulleyDef;
b2Vec2 anchor1(-9.1f, L+a/2); // the position of the end
string of the left cord is (-3.0f, L-1.5)
b2Vec2 anchor2(2.0f, L); // the position of the end string of
the right cord is (2.0f, L)
b2Vec2 groundAnchor1(-1.0f, b + L ); // the string of the cord
is tightened at (-1.0f, b+L)

```

```

b2Vec2 groundAnchor2(1.5f, b + L); // the string of the cord is
// tightened at (1.5f, b+L)
// the last float is the ratio
pulleyDef.Initialize(m_boxc, m_boxr, groundAnchor1,
                     groundAnchor2, anchor1, anchor2, 1.0f);

m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&pulleyDef);

// Create the Pulley on the left
b2PulleyJointDef pulleyDef1;
b2Vec2 anchor3(-18.0f, L-1.5f);
b2Vec2 anchor4(-11.0f, L+a/2);
b2Vec2 groundAnchor3(-18.5f, b + L );
b2Vec2 groundAnchor4(-15.5f, b + L );
pulleyDef1.Initialize(m_boxl, m_boxc, groundAnchor3,
                     groundAnchor4, anchor3, anchor4, 1.0f);

m_joint2 = (b2PulleyJoint*)m_world->CreateJoint(&pulleyDef1);

```

Most of the codes are already explained on the previous chapters, I believe readers will already be familiar even far smarter than me and wish for something better than repeating the same explanations. We will proceed with less repetition and more substances. This is also the main reason to read step by step from beginning not jumping to chapter 10 directly.

IX. DRAG FORCE AND TERMINAL SPEED

A fluid is anything that can flow, either a gas or a liquid. When there is a relative motion between air (or some other fluid) and a body, the body experiences a drag force \vec{D} that opposes the relative motion and points in the direction in which the fluid flows relative to the body. The magnitude of \vec{D} is related to the relative speed v by an experimentally determined drag coefficient C according to

$$D = \frac{1}{2}C\rho Av^2 \quad (8.6)$$

where ρ is the fluid density (mass per unit volume) and A is the effective cross-sectional area of the body (the area of a cross section taken perpendicular to the relative velocity \vec{v}).

When a blunt object has fallen far enough through air, the magnitude of the drag force \vec{D} and the gravitational force \vec{F}_g on the body become equal. The body then falls at a constant terminal speed v_t given by

$$v_t = \sqrt{\frac{2F_g}{C\rho A}} \quad (8.7)$$

X. SIMULATION FOR DOWNHILL SKIING WITH Box2D

Taken from example problem no 40 on chapter 6 of [6].

In downhill speed skiing a skier is retarded by both the air drag force on the body and the kinetic frictional force on the skis. Suppose the slope angle is $\theta = 40^\circ$, the snow is dry snow with

a coefficient of kinetic friction $\mu_k = 0.0400$, the mass of the skier and equipment is $m = 85.0 \text{ kg}$, the cross-sectional area of the (tucked) skier is $A = 1.30 \text{ m}^2$, the drag force coefficient is $C = 0.150$, and the air density is 1.20 kg/m^3 .

- (a) What is the terminal speed?
- (b) If a skier can vary C by a slight amount dC by adjusting, say, the hand positions, what is the corresponding variation in the terminal speed?

Solution:

- (a) The force along the slope is given by

$$\begin{aligned} F_g &= mg \sin \theta - \mu F_N \\ &= mg \sin \theta - \mu mg \cos \theta \\ &= mg(\sin \theta - \mu \cos \theta) \\ &= (85)(9.8)(\sin 40^\circ - (0.0400)\cos 40^\circ) \\ F_g &= 510 \end{aligned}$$

Thus, the terminal speed of the skier is

$$\begin{aligned} v_t &= \sqrt{\frac{2F_g}{C\rho A}} \\ &= \sqrt{\frac{2(510)}{(0.150)(1.20)(1.30)}} \\ &= 66 \end{aligned}$$

the terminal speed is 66 m/s .

- (b) In order to know how much speed is changed when we change C for a small amount / infinitesimal amount of dC , we will use differentiation. Differentiating v_t with respect to C , we obtain

$$\begin{aligned} \frac{dv_t}{dC} &= \frac{d}{dC} \sqrt{\frac{2F_g}{C\rho A}} \\ &= \frac{d}{dC} \sqrt{\frac{2F_g}{\rho A} C^{-1/2}} \\ \frac{dv_t}{dC} &= -\frac{1}{2} \sqrt{\frac{2F_g}{\rho A}} C^{-3/2} \\ dv_t &= -\frac{1}{2} \sqrt{\frac{2F_g}{\rho A}} C^{-3/2} dC \\ &= -\frac{1}{2} \sqrt{\frac{2(510)}{(1.20)(1.30)}} dC \\ &= -(2.20 \times 10^2) dC \end{aligned}$$

a slight change of dC will result in change of terminal speed of $-(2.20 \times 10^2) \text{ m/s}$. The minus sign means the speed is decreasing, thus the perfect body position when skiing has to

be maintained to reach highest terminal speed, and not moving too much, since not only decreasing the terminal speed, changing position or gesture while skiing might make you get into accident or hit a tree.

```
#include "test.h"
#include <iostream>

class DragforceSkier : public Test
{
public:
    DragforceSkier()
    {
        b2Body* ground = NULL;
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            float const PlatformWidth = 8.0f;

            /*
            First angle is from the horizontal and should be
            negative for a downward slope.
            Second angle is relative to the preceding slope, and
            should be positive, creating a kind of
            loose 'Z'-shape from the 3 edges.
            If A1 = -10, then A2 <= ~1.5 will result in the
            collision glitch.
            If A1 = -30, then A2 <= ~10.0 will result in the
            glitch.
            */
            float const Angle1Degrees = -40.0f;
            float const Angle2Degrees = 0.0f;

            /*
            The larger the value of SlopeLength, the less likely
            the glitch will show up.
            */
            float const SlopeLength = 5100.0f;

            float const SurfaceFriction = 0.04f;

            // Convert to radians
            float const Slope1Incline = -Angle1Degrees * b2_pi /
                180.0f;
            float const Slope2Incline = Slope1Incline -
                Angle2Degrees * b2_pi / 180.0f;
        }
    }
}
```

```

m_platform_width = PlatformWidth;

// Horizontal platform
b2Vec2 v1(-PlatformWidth, 0.0f);
b2Vec2 v2(0.0f, 0.0f);
b2Vec2 v3(SlopeLength * cosf(Slope1Incline), -
           SlopeLength * sinf(Slope1Incline));
b2Vec2 v4(v3.x + SlopeLength * cosf(Slope2Incline), v3
           .y - SlopeLength * sinf(Slope2Incline));
b2Vec2 v5(v4.x, v4.y - 1.0f);

b2Vec2 vertices[5] = { v5, v4, v3, v2, v1 };

b2ChainShape shape;
shape.CreateLoop(vertices, 5);
b2FixtureDef fd;
fd.shape = &shape;
fd.density = 0.0f;
fd.friction = SurfaceFriction;

ground->CreateFixture(&fd);
}

float const BodyWidth = 1.0f;
float const BodyHeight = 2.5f;
float const SkiLength = 3.0f;

/*
Larger values for this seem to alleviate the issue to some
extent.
*/
float const SkiThickness = 0.3f;
float const SkiFriction = 0.04f;
float const SkiRestitution = 0.15f;

b2BodyDef skiBodyDef;
b2Body* skier = m_world->CreateBody(&skiBodyDef);

// Create the ski geometry from vertices
b2PolygonShape ski;
b2Vec2 verts[4];
verts[0].Set(-SkiLength / 2 - SkiThickness, -BodyHeight /
            2);
verts[1].Set(-SkiLength / 2, -BodyHeight / 2 - SkiThickness
            );
verts[2].Set(SkiLength / 2, -BodyHeight / 2 - SkiThickness);
verts[3].Set(SkiLength / 2 + SkiThickness, -BodyHeight / 2);
ski.Set(verts, 4);

```

```
b2FixtureDef skiFixtureDef;
skiFixtureDef.restitution = SkiRestitution;
skiFixtureDef.density = 85.8586019936297f; // density = mass
*1.010101199925053
skiFixtureDef.friction = SkiFriction;
skiFixtureDef.shape = &ski;

skiBodyDef.type = b2_dynamicBody;
float initial_y = BodyHeight / 2 + SkiThickness;
skiBodyDef.position.Set(-m_platform_width / 2, initial_y);

m_skier = m_world->CreateBody(&skiBodyDef);
b2Fixture *skiFixture = m_skier->CreateFixture(&
    skiFixtureDef);
m_skier->SetLinearVelocity(b2Vec2(4.5f, 0.0f));

b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.75f;
ballFixtureDef.density = 3.3f; // this will affect the ball
mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(-m_platform_width / 2 - 1.0f,
    initial_y);

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&
    ballFixtureDef);
m_ball->SetAngularVelocity(1.0f);
m_ball->SetLinearVelocity(b2Vec2(4.5f, 0.0f));

g_camera.m_center = b2Vec2(m_platform_width / 2.0f, 0.0f);
g_camera.m_zoom = 1.1f;
m_fixed_camera = true;
m_time = 0.0f;
}

b2Body* m_ball;
b2Body* m_skier;
float m_platform_width;
float m_time;
bool m_fixed_camera;
```

```

void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_C:
            m_fixed_camera = !m_fixed_camera;
            if(m_fixed_camera)
            {
                g_camera.m_center = b2Vec2(m_platform_width /
                    2.0f, 0.0f);
            }
            break;
    }
}

void Step(Settings& settings) override
{
    b2Vec2 v = m_skier->GetLinearVelocity();
    b2MassData massData1 = m_skier->GetMassData();
    b2Vec2 position = m_skier->GetPosition();
    m_time += 1.0f / 60.0f; // assuming we are using frequency of
                           60 Hertz
    float omega = m_skier->GetAngularVelocity();
    float ke = 0.5f * massData1.mass * b2Dot(v, v) + 0.5f *
               massData1.I * omega * omega;

    g_debugDraw.DrawString(5, m_textLine, "Press C = Camera fixed
        /tracking");
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
        f", m_time);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Position = (%4.1f,
        %4.1f)", position.x, position.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Velocity = (%4.1f,
        %4.1f)", v.x, v.y); // the velocity for x stop at 91.9
                           then not increasing anymore
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Mass = %.1f",
        massData1.mass);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Kinetic energy = %.6f"
        , ke);
    m_textLine += m_textIncrement;

    printf("%4.2f %4.2f %4.2f %4.2f %4.2f\n", position.x,

```

```

        position.y, v.x, v.y, ke);
    if(!m_fixed_camera)
    {
        g_camera.m_center = m_skier->GetPosition();
    }
    Test::Step(settings);
}

static Test* Create()
{
    return new DragforceSkier;
}

};

static int testIndex = RegisterTest("Force and Motion", "Drag Force Skier",
    DragforceSkier::Create);

```

C++ Code 47: *tests/dragforce_skier.cpp* "Drag Force Skier Box2D"

The simulation run well, imagine you are playing ski on Mt Logan, Yukon, Canada with your beloved wife. This is just the simulation before the real life comes in. I add a ball as an extra to see what will happens if there is a ball rolling down like a snowball from top of the mountain along with a skier. Turns out the ball can be outrun by the skier, since the ball needs to rotating while moving, and by cross section A , ski is slimmer, thus skier makes the terminal velocity higher than the ball.

What went wrong is the velocity stop at 91.9 m/s toward x axis and -77.1 m/s toward y axis, this would make the velocity magnitude of 119.9584 m/s toward South East / downhill.

Some explanations for the codes:

- To create the skier from vertices, in Box2D this is another way to create an object / body, besides calling for Box or Circle shape object.

```

float const BodyWidth = 1.0f;
float const BodyHeight = 2.5f;
float const SkiLength = 3.0f;

/*
Larger values for this seem to alleviate the issue to some
extent.
*/
float const SkiThickness = 0.3f;
float const SkiFriction = 0.04f;
float const SkiRestitution = 0.15f;

b2BodyDef skiBodyDef;
b2Body* skier = m_world->CreateBody(&skiBodyDef);

```

```

// Create the ski geometry from vertices
b2PolygonShape ski;
b2Vec2 verts[4];
verts[0].Set(-SkiLength / 2 - SkiThickness, -BodyHeight / 2);
verts[1].Set(-SkiLength / 2, -BodyHeight / 2 - SkiThickness);
verts[2].Set(SkiLength / 2, -BodyHeight / 2 - SkiThickness);
verts[3].Set(SkiLength / 2 + SkiThickness, -BodyHeight / 2);
ski.Set(verts, 4);

b2FixtureDef skiFixtureDef;
skiFixtureDef.restitution = SkiRestitution;
skiFixtureDef.density = 85.8586019936297f; // density = mass
*1.010101199925053
skiFixtureDef.friction = SkiFriction;
skiFixtureDef.shape = &ski;

skiBodyDef.type = b2_dynamicBody;
float initial_y = BodyHeight / 2 + SkiThickness;
skiBodyDef.position.Set(-m_platform_width / 2, initial_y);

m_skier = m_world->CreateBody(&skiBodyDef);
b2Fixture *skiFixture = m_skier->CreateFixture(&skiFixtureDef)
;
m_skier->SetLinearVelocity(b2Vec2(4.5f, 0.0f));

```

- A keyboard press event to track the movement of the skier.

```

void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_C:
            m_fixed_camera = !m_fixed_camera;
            if(m_fixed_camera)
            {
                g_camera.m_center = b2Vec2(m_platform_width
                    / 2.0f, 0.0f);
            }
            break;
    }
}

```

- To print out the data of mass, velocity, and position of the skier, and when "C" is pressed the camera will follow the movement of the skier by using the **GetPosition()** function.

```

void Step(Settings& settings) override
{
    b2Vec2 v = m_skier->GetLinearVelocity();
    b2MassData massData1 = m_skier->GetMassData();
    b2Vec2 position = m_skier->GetPosition();
}

```

```

m_time += 1.0f / 60.0f; // assuming we are using
    frequency of 60 Hertz
float omega = m_skier->GetAngularVelocity();
float ke = 0.5f * massData1.mass * b2Dot(v, v) + 0.5f *
    massData1.I * omega * omega;

g_debugDraw.DrawString(5, m_textLine, "Press C = Camera
    fixed/tracking");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)
    = %.6f", m_time);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Position = (%4.1f
    , %4.1f)", position.x, position.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Velocity = (%4.1f
    , %4.1f)", v.x, v.y); // the velocity for x stop at
    91.9 then not increasing anymore
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Mass = %.1f",
    massData1.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Kinetic energy =
    %.6f", ke);
m_textLine += m_textIncrement;

printf("%4.2f %4.2f %4.2f %4.2f %4.2f\n", position.x,
    position.y, v.x, v.y, ke);
if(!m_fixed_camera)
{
    g_camera.m_center = m_skier->GetPosition();
}
Test::Step(settings);
}

```

If you know the chapter in Donald Duck comic, where his cousin Didi Duck rolling fresh milk from top of the mountain to the bottom, so he can then sell the milk to the store in the village at the bottom of the mountain. Imagine that the ball in this simulation is the milk' jar, we can add more complexity and details, filled with milk or anything else, and see maximum velocity based on the cross section of the milk to preserve the milk, so the milk can still be fresh when arrived at the village.

The story from Donald Duck is a tragedy actually, what happened when milk is being rolled down from top of the mountain continuously? At the store, the shopkeeper checks, it becomes butter, thus it can't be sold. What a waste! Maybe reader of this book has ever encountered that Donald Duck chapter. A great comic, my favorite is Uncle Scrooge.

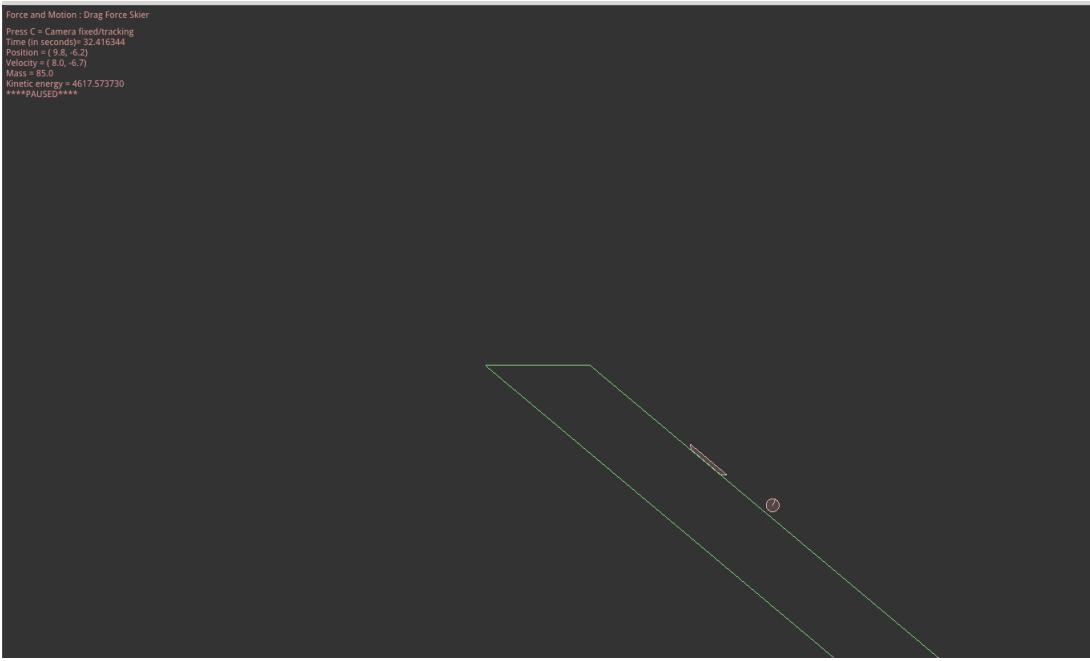


Figure 8.13: The simulation of a skier going downhill with a ball on top of it, then the ball goes downhill first but the skier can catch up with the ball (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/dragforce_skier.cpp`).

XI. UNIFORM CIRCULAR MOTION WITH FORCE

We have discussed about uniform circular motion, that is, a body moves in a circle (or a circular arc) at constant speed v , with a centripetal acceleration of

$$a = \frac{v^2}{R} \quad (8.8)$$

where R is the radius of the circle. A centripetal force is not a new kind of force. It can, in fact, be a frictional force, a gravitational force, the force from a car wall or a string, or any other force.

A centripetal force accelerates a body by changing the direction of the body's velocity without changing the body's speed.

From Newton's second law, we can write the magnitude F of a centripetal force as

$$F = m \frac{v^2}{R} \quad (8.9)$$

You need to remember that

- The speed v is constant, the magnitudes of the acceleration and the force are also constant.
- The directions of the centripetal acceleration and force are not constant. They vary continuously so as to always point toward the center of the circle.
- The force and acceleration vectors are sometimes drawn along a radial axis r that moves with the body and always extends from the center of the circle to the body.

- The positive direction of the axis is radially outward, but the acceleration and force vectors point radially inward.

XII. SIMULATION FOR AIRPLANE UNIFORM CIRCULAR MOTION WITH Box2D

Taken from problem no 51 chapter 6 of [6].

An airplane is flying in a horizontal circle at a speed of 480 km/h . If its wings are tilted at angle $\theta = 40^\circ$ to the horizontal, what is the radius of the circle in which the plane is flying? Assume that the required force is provided by an "aerodynamic lift" that is perpendicular to the wing surface.

Solution:

First, you have to be able to imagine an airplane flying in a horizontal circle. Then tilted at angle $\theta = 40^\circ$ means that airplane is doing roll movement toward the circle's center of $\theta = 40^\circ$. I asked a lot of people who are the best in science, the key is their imagination is the best, they can imagine without the need for textbooks full of pictures. Isn't Einstein like this as well?

We note that \vec{F}_1 is the force of aerodynamic lift and \vec{a} points rightwards in the figure. We also note that $|\vec{a}| = \frac{v^2}{R}$.

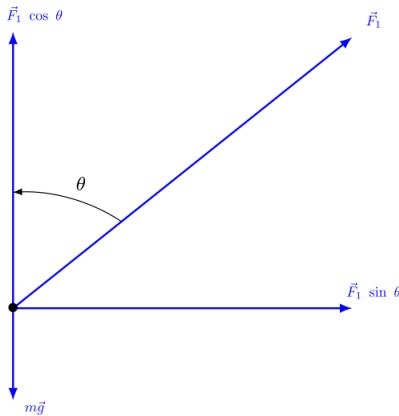


Figure 8.14: The free-body diagram for the airplane of mass m .

The key to comprehend this is by imagining from Figure 8.14, on the left the airplane is shaped like that when tilted, tilted itself means roll movement toward the circle center (for uniform circular motion) about the angle of θ , we play some geometry tricks then we know since it is tilted, the force is tilted as well, that's explains on the right side, where I only draw the body diagram, we need to decompose \vec{F}_1 into the x and y components to be connected with the gravitational force and centripetal force, thus able to use Newton's law. The force \vec{F}_1 can be seen as the normal force \vec{F}_N that is tilted about the angle of θ .

Moving counterclockwise or clockwise does not matter since it is only a matter of orientation, negative sign or not, the force still will be computed the same. Now, by applying Newton's law to

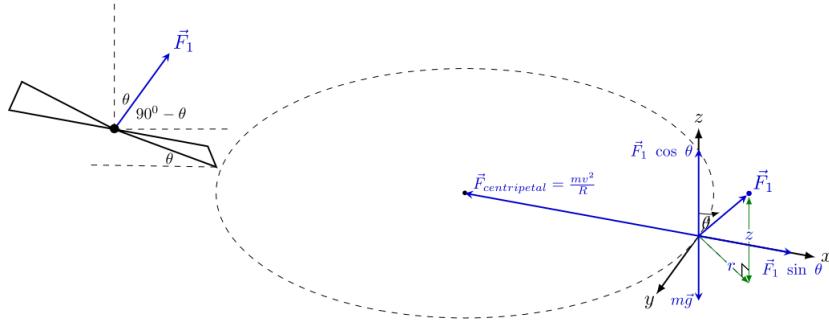


Figure 8.15: The three dimensional airplane diagram moving in circular motion with mass of m . The creation of image is guided by Hamzst.

the axes of the problem ($+x$ rightward and $+y$ upward), we obtain

$$\begin{aligned}\sum F_x &= 0 \\ F_1 \sin \theta - m \frac{v^2}{R} &= 0 \\ F_1 \sin \theta &= m \frac{v^2}{R}\end{aligned}$$

The equation above means the balance of the force in the horizontal axis, between $F_1 \sin \theta$ and the centripetal force that is always directed toward the circle' center for circular motion. This equation preserve the condition of circular motion since the force pulling the object inside (centripetal force) and the force pushing the object away from the circle' center is the same, thus the object(the airplane) will keep moving in circular motion with same radius.

$$\begin{aligned}\sum F_y &= 0 \\ F_1 \cos \theta - mg &= 0 \\ F_1 \cos \theta &= mg\end{aligned}$$

The equation above means the balance of the force in the vertical axis, between $F_1 \cos \theta$ and the gravitational force for the airplane. This equation preserves the height of the airplane, since the force that bring the airplane to that height is the same as the gravitational force, thus the airplane won't go higher or lower if this equation applied.

Afterwards, by eliminating mass from these equations leads to

$$\tan \theta = \frac{v^2}{gR}$$

The equation allows us to solve for the radius R . With $v = 480 \text{ km/h} = 133 \text{ m/s}$ and $\theta = 40^\circ$, we

find

$$\begin{aligned} R &= \frac{v^2}{g \tan \theta} \\ &= \frac{133^2}{(9.8)(\tan 40^\circ)} \\ &= 2151 \end{aligned}$$

the radius R is 2151 m. If you are wondering why the radius is quite big? It is natural and logical since R and v has linear relationship. If you are using Box2D and try the circular motion simulation, try to make the velocity very high 133 for example, but let the trail path stay with small radius (around 3), what will happens? The object rotating wider and move very fast, since the speed is increasing highly. This is the basic of rocket science, satellite and beyond.

We can also say that tilted airplane will have wider radius than airplane that is moving in circular motion without tilting anywhere. For untilted airplane it will be like this:

$$\begin{aligned} \sum F_x &= 0 \\ F_{untitled} - m \frac{v^2}{R} &= 0 \\ F_{untitled} &= m \frac{v^2}{R} \\ \sum F_y &= 0 \\ F_{untitled} - mg &= 0 \\ F_{untitled} &= mg \\ R_{untitled} &= \frac{v^2}{g} \\ &= \frac{133^2}{9.8} \\ &= 1805 \end{aligned}$$

```
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#include "test.h"
#include <fstream>

class CircularMotionairplane : public Test
{
public:

    CircularMotionairplane()
    {
        m_world->SetGravity(b2Vec2(0.0f, 0.0f));
        b2BodyDef bdtrail;
        b2Body* bodytrail = m_world->CreateBody(&bdtrail);

        // Create the trail path
    }
}
```

```

{
    b2CircleShape shape;
    shape.m_radius = 2.1510f;
    shape.m_p.Set(-3.0f, 25.0f);

    b2FixtureDef fd;
    fd.shape = &shape;
    fd.isSensor = true;
    m_sensor = bodytrail->CreateFixture(&fd);
}
// Create shape
b2Transform xf1;
xf1.q.Set(1.3496f * b2_pi); // rotate the shape 1.3496 pi
xf1.p = xf1.q.GetAxis();

b2Vec2 vertices[3];
vertices[0] = b2Mul(xf1, b2Vec2(-1.0f, 0.0f));
vertices[1] = b2Mul(xf1, b2Vec2(1.0f, 0.0f));
vertices[2] = b2Mul(xf1, b2Vec2(0.0f, 0.5f));

b2PolygonShape poly1;
poly1.Set(vertices, 3);

b2FixtureDef sd1;
sd1.shape = &poly1;
sd1.density = 200.0f;

b2Transform xf2;
xf2.q.Set(-1.3496f * b2_pi); // rotate the shape - 1.3496 pi
xf2.p = -xf2.q.GetAxis();

vertices[0] = b2Mul(xf2, b2Vec2(-1.0f, 0.0f));
vertices[1] = b2Mul(xf2, b2Vec2(1.0f, 0.0f));
vertices[2] = b2Mul(xf2, b2Vec2(0.0f, 0.5f));

b2PolygonShape poly2;
poly2.Set(vertices, 3);

b2FixtureDef sd2;
sd2.shape = &poly2;
sd2.density = 200.0f;

b2BodyDef bd1;
bd1.type = b2_dynamicBody;

bd1.position.Set(-5.1540f, 24.0f);
bd1.angle = b2_pi;
bd1.allowSleep = false;

```

```

m_body = m_world->CreateBody(&bd1);
m_body->CreateFixture(&sd1);
m_body->CreateFixture(&sd2);
m_body->SetAngularVelocity(1.0f); // Counter clockwise
movement
// I Love my Wife!!!
m_time = 0.0f;
}
b2Body* m_body;
float m_time;
b2Fixture* m_sensor;
void Step(Settings& settings) override
{
    b2Vec2 v = m_body->GetLinearVelocity();
    float r = 2.1510f;
    float omega = m_body->GetAngularVelocity();
    float angle = m_body->GetAngle();
    b2MassData massData = m_body->GetMassData();
    b2Vec2 position = m_body->GetPosition();
    float sin = sinf(angle);
    float cos = cosf(angle);
    float body_vel = 2.1510f;
    float a = (body_vel*body_vel) / r;
    m_time += 1.0f / 60.0f; // assuming we are using frequency of
    60 Hertz

    float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f *
    massData.I * omega * omega;

    m_body->SetLinearVelocity(b2Vec2(-body_vel*sinf(angle),
    body_vel*cosf(angle)));

    g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
    f", m_time);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Airplane position =
    (%4.1f, %4.1f)", position.x, position.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
    .mass);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Kinetic energy = %.6f"
    , ke);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Total linear velocity=
    %.6f", v);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Linear velocity =

```

```

        ("%4.1f, %4.1f)", v.x, v.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Acceleration, a = %.6f",
                           ", a);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Angle (in degrees) =",
                           %.6f", angle*RADTODEG);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "sin (angle) = %.6f",
                           sin);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "cos (angle) = %.6f",
                           cos);
    m_textLine += m_textIncrement;
    // Print the result in every time step then plot it into
    graph with either gnuplot or anything

    printf("%4.2f %4.2f %4.2f %4.2f %4.2f %4.2f %4.2f\n",
           position.x, position.y, angle*RADTODEG, v.x, v.y, sin,
           cos);

    Test::Step(settings);
}

static Test* Create()
{
    return new CircularMotionairplane;
}

static int testIndex = RegisterTest("Motion in 2D", "Circular Motion for
Airplane", CircularMotionairplane::Create);

```

C++ Code 48: *tests/circular_motionairplane.cpp* "Circular Motion Airplane Box2D"

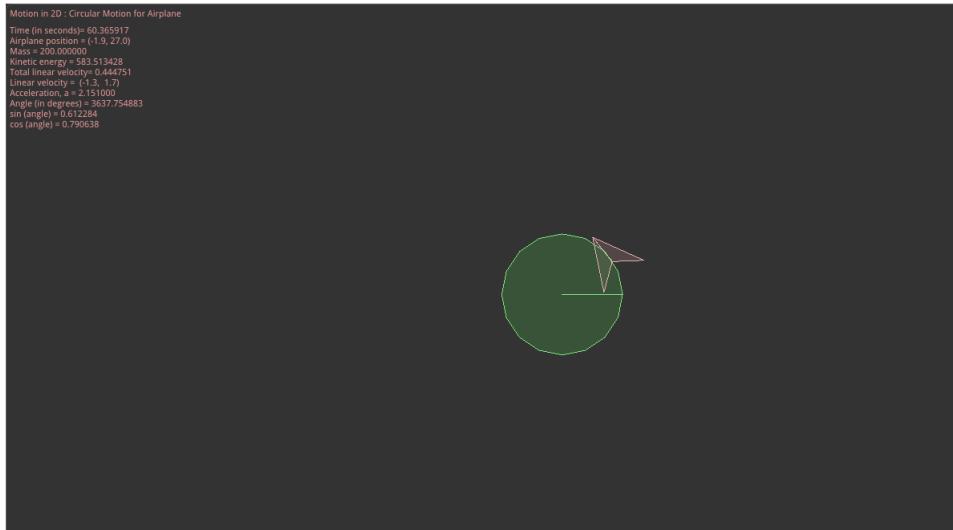


Figure 8.16: The simulation of an airplane circling in 2-dimension (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/circular_motionairplane.cpp`).

Some explanations for the codes:

- To create the airplane shape we are combining 3 vertices into triangle as one shape, then create similar copy. Rotate the first shape clockwise, and the other counterclockwise in the same degree. Et Voila. The function `CreateFixture()` able to join two shapes into one.

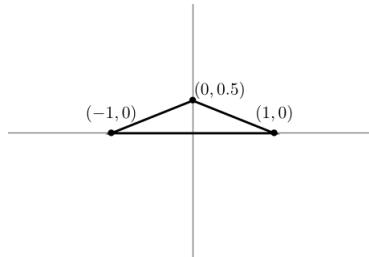


Figure 8.17: The triangle shape is made out of 3 vertices, we choose it $(1, 0)$, $(-1, 0)$, $(0, 0.5)$.

```
b2Transform xf1;
xf1.q.Set(1.3496f * b2_pi); // rotate the shape 1.3496 pi
xf1.p = xf1.q.GetXAxis();

b2Vec2 vertices[3];
vertices[0] = b2Mul(xf1, b2Vec2(-1.0f, 0.0f));
vertices[1] = b2Mul(xf1, b2Vec2(1.0f, 0.0f));
vertices[2] = b2Mul(xf1, b2Vec2(0.0f, 0.5f));

b2PolygonShape poly1;
poly1.Set(vertices, 3);
```

```

b2FixtureDef sd1;
sd1.shape = &poly1;
sd1.density = 200.0f;

b2Transform xf2;
xf2.q.Set(-1.3496f * b2_pi); // rotate the shape - 1.3496 pi
xf2.p = -xf2.q.GetXAxis();

vertices[0] = b2Mul(xf2, b2Vec2(-1.0f, 0.0f));
vertices[1] = b2Mul(xf2, b2Vec2(1.0f, 0.0f));
vertices[2] = b2Mul(xf2, b2Vec2(0.0f, 0.5f));

b2PolygonShape poly2;
poly2.Set(vertices, 3);

b2FixtureDef sd2;
sd2.shape = &poly2;
sd2.density = 200.0f;

b2BodyDef bd1;
bd1.type = b2_dynamicBody;

bd1.position.Set(-5.1540f, 24.0f);
bd1.angle = b2_pi;
bd1.allowSleep = false;
m_body = m_world->CreateBody(&bd1);
m_body->CreateFixture(&sd1);
m_body->CreateFixture(&sd2);
m_body->SetAngularVelocity(1.0f);

```

- When create the trail path for the circular movement, I set the radius to be in ratio, like when you see a map, the distance between two point in a map is a ratio of real life distance. Thus instead of making 2,151 meter of radius, I prefer to use 2.151 meter of radius. Still limited with computer monitor, one day we can use better graphics display.

```

{
    b2CircleShape shape;
    shape.m_radius = 2.1510f;
    shape.m_p.Set(-3.0f, 25.0f);

    b2FixtureDef fd;
    fd.shape = &shape;
    fd.isSensor = true;
    m_sensor = bodytrail->CreateFixture(&fd);
}

```

- The code for the simulation is not perfect in my opinion for this problem, someday I will come back to this, but for the mean time I want to proceed to other physics problem.

Chapter 9

DFSimulatorC++ III: Kinetic Energy and Work

"Ouais bla.. tu me connais" - Sweden Sexy (after being wild running around then comes to me and say that words)

When we talk about kinetic energy we will think of something that moves will require energy. Everything moves, even if you sit still the earth is moving / rotating for 24 hours till it face the same side of the sun again, and revolving toward the sun for 365 days till it back to its original position on its' own orbit.

Energy can be transformed from one type to another and transferred from one object to another, but the total amount is always the same. That is the conservation law of energy.

I. KINETIC ENERGY

Kinetic energy K is energy associated with the state of motion of an object, the faster the object moves, the greater is its kinetic energy. When the object is stationary, its kinetic energy is zero.

The formula for kinetic energy

$$K = \frac{1}{2}mv^2 \quad (9.1)$$

with v is well below the speed of light.

II. SIMULATION FOR KINETIC ENERGY, TRAIN CRASH WITH Box2D

This is taken form sample problem 7.01 from [6].

In 1896 in Waco, Texas, William Crush parked two locomotives at opposite ends of a 6.4 km-long track, fired them up, tied their throttles, and then allowed them to crash head-on at full speed in front of 30,000 spectators. Hundreds of people were hurt by flying debris; several were killed. Assuming each locomotive weighed 1.2×10^6 N and its acceleration was a constant 0.26 m/s^2 , what was the total kinetic energy of the two locomotives just before the collision?

Solution:

We can assume each locomotive had constant acceleration, to find its speed before the collision:

$$\begin{aligned} v^2 &= v_0^2 + 2a(x - x_0) \\ &= 0 + 2(0.26)(3.2 \times 10^3) \\ v &= 40.8 \end{aligned}$$

the speed before the collision is 40.8 m/s or 147 km/h.

We can find the mass of each locomotive by

$$m = \frac{w}{g} = \frac{1.2 \times 10^6}{9.8} = 1.22 \times 10^5$$

The total kinetic energy of two locomotives just before the collision is

$$K = \frac{1}{2}mv^2 = (1.22 \times 10^5)(40.8)^2 = 2 \times 10^8$$

This collision with kinetic energy of 200 million Joule (2×10^8) was like an exploding bomb.

You need to copy from my repository' directory `../Source Codes/C++/DianFreya-box2d-testbed`, then go inside the directory and open the terminal then type:

```
mkdir build
cd build
cmake ..
make
./testbed
```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Kinetic Energy and Work/Kinetic Energy: Collision**.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class KineticenergyCollision : public Test
{
public:
    KineticenergyCollision()
    {
        b2Body* ground = NULL;
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
```

```
shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(-46.0f,
    , 46.0f));
ground->CreateFixture(&shape, 0.0f);
}
{
b2BodyDef bd;
ground = m_world->CreateBody(&bd);

b2EdgeShape shape;
shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2(46.0f,
    46.0f));
ground->CreateFixture(&shape, 0.0f);
}
{
b2BodyDef bd;
ground = m_world->CreateBody(&bd);

b2EdgeShape shape;
shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
    0.0f));
ground->CreateFixture(&shape, 0.0f);
}
// Create the middle train body
b2BodyDef bd1;
bd1.type = b2_dynamicBody;
bd1.angularDamping = 0.1f;

bd1.position.Set(1.0f, 0.5f);
b2Body* springbox = m_world->CreateBody(&bd1);

b2PolygonShape shape1;
shape1.SetAsBox(0.5f, 0.5f);
springbox->CreateFixture(&shape1, 5.0f);
// Create the last train body
b2BodyDef bd2;
bd2.type = b2_dynamicBody;
bd2.angularDamping = 0.1f;

bd2.position.Set(-1.0f, 0.5f);
b2Body* springbox2 = m_world->CreateBody(&bd2);

b2PolygonShape shape2;
shape2.SetAsBox(0.5f, 0.5f);
springbox2->CreateFixture(&shape2, 5.0f);

// Create the first train / the head as the movable object
// going to positive x axis
b2PolygonShape boxShape;
```

```

boxShape.SetAsBox(0.5f, 0.5f);

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.75f;
boxFixtureDef.density = 122000.0f; // this will affect the
                                box mass
boxFixtureDef.friction = 0.1f;
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(3.0f, 0.5f);

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
;
m_box->SetLinearVelocity(b2Vec2(40.8f, 0.0f));

// Make a distance joint for the head train with the middle
// train
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f),
              boxBodyDef.position);
jd.collideConnected = true; // In this case we decide to
                            allow the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f;
m_maxLength = 2.0f;
b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
                   m_dampingRatio, jd.bodyA, jd.bodyB);

m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
m_joint->SetMinLength(m_minLength);
m_joint->SetMaxLength(m_maxLength);

// Create distance joint / chain that connect the last train
// to the middle
b2DistanceJointDef jd2;
jd2.Initialize(springbox2, springbox2, b2Vec2(1.0f, 0.5f), bd2
               .position);
jd2.collideConnected = true; // In this case we decide to
                            allow the bodies to collide.
m_length = jd2.length;
m_minLength = 2.0f;
m_maxLength = 2.0f;

```

```

        b2LinearStiffness(jd2.stiffness, jd2.damping, m_hertz,
                           m_dampingRatio, jd2.bodyA, jd2.bodyB);
        m_joint2 = (b2DistanceJoint*)m_world->CreateJoint(&jd2);
        m_joint2->SetMinLength(m_minLength);
        m_joint2->SetMaxLength(m_maxLength);

        // For the second train going toward negative x axis
        // Create the middle train body
        b2BodyDef bd3;
        bd3.type = b2_dynamicBody;
        bd3.angularDamping = 0.1f;

        bd3.position.Set(31.0f, 0.5f);
        b2Body* springbox3 = m_world->CreateBody(&bd3);

        b2PolygonShape shape3;
        shape3.SetAsBox(0.5f, 0.5f);
        springbox3->CreateFixture(&shape3, 5.0f);
        // Create the last train body
        b2BodyDef bd4;
        bd4.type = b2_dynamicBody;
        bd4.angularDamping = 0.1f;

        bd4.position.Set(29.0f, 0.5f);
        b2Body* springbox4 = m_world->CreateBody(&bd4);

        b2PolygonShape shape4;
        shape4.SetAsBox(0.5f, 0.5f);
        springbox4->CreateFixture(&shape4, 5.0f);

        // Create the first train / the head
        b2PolygonShape boxShape2;
        boxShape2.SetAsBox(0.5f, 0.5f);

        b2FixtureDef boxFixtureDef2;
        boxFixtureDef2.restitution = 0.75f;
        boxFixtureDef2.density = 122000.0f; // this will affect the
                                         box mass
        boxFixtureDef2.friction = 0.1f;
        boxFixtureDef2.shape = &boxShape2;

        b2BodyDef boxBodyDef2;
        boxBodyDef2.type = b2_dynamicBody;
        boxBodyDef2.position.Set(33.0f, 0.5f);

        m_box2 = m_world->CreateBody(&boxBodyDef2);
        b2Fixture *box2Fixture = m_box2->CreateFixture(&
                                                       boxFixtureDef2);
    
```

```

m_box2->SetLinearVelocity(b2Vec2(-40.8f, 0.0f));

// Make a distance joint for the head train with the middle
// train
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd3;
jd3.Initialize(springbox3, m_box2, b2Vec2(31.0f, 0.5f),
               boxBodyDef2.position);
jd3.collideConnected = true; // In this case we decide to
                           // allow the bodies to collide.
m_length = jd3.length;
m_minLength = 2.0f;
m_maxLength = 2.0f;
b2LinearStiffness(jd3.stiffness, jd3.damping, m_hertz,
                   m_dampingRatio, jd3.bodyA, jd3.bodyB);

m_joint3 = (b2DistanceJoint*)m_world->CreateJoint(&jd3);
m_joint3->SetMinLength(m_minLength);
m_joint3->SetMaxLength(m_maxLength);

// Make a distance joint that connect the last train to the
// middle
b2DistanceJointDef jd4;
jd4.Initialize(springbox3, springbox4, b2Vec2(31.0f, 0.5f),
               bd4.position);
jd4.collideConnected = true; // In this case we decide to
                           // allow the bodies to collide.
m_length = jd4.length;
m_minLength = 2.0f;
m_maxLength = 0.0f;
b2LinearStiffness(jd4.stiffness, jd4.damping, m_hertz,
                   m_dampingRatio, jd4.bodyA, jd4.bodyB);
m_joint4 = (b2DistanceJoint*)m_world->CreateJoint(&jd4);
m_joint4->SetMinLength(m_minLength);
m_joint4->SetMaxLength(m_maxLength);

m_time = 0.0f;
}

b2Body* m_box;
b2Body* m_box2;
b2DistanceJoint* m_joint;
b2DistanceJoint* m_joint2;
b2DistanceJoint* m_joint3;
b2DistanceJoint* m_joint4;
float m_length;
float m_minLength;

```

```

        float m_maxLength;
        float m_hertz;
        float m_dampingRatio;
        float m_time;
        float F = 50.0f;
        int theta = 30;

    void Step(Settings& settings) override
    {
        m_time += 1.0f / 60.0f; // assuming we are using frequency of
        // 60 Hertz
        b2MassData massData = m_box->GetMassData();
        b2MassData massData2 = m_box2->GetMassData();
        b2Vec2 position = m_box->GetPosition();
        b2Vec2 position2 = m_box2->GetPosition();
        b2Vec2 velocity = m_box->GetLinearVelocity();
        b2Vec2 velocity2 = m_box2->GetLinearVelocity();

        g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
            f", m_time);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Train 1 position =
            (%4.1f, %4.1f)", position.x, position.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Train 2 position =
            (%4.1f, %4.1f)", position2.x, position2.y);
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Train 1 velocity =
            (%4.1f, %4.1f)", velocity.x, velocity.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Train 2 velocity =
            (%4.1f, %4.1f)", velocity2.x, velocity2.y);
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Train 1 Mass = %.6f",
            massData.mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Train 2 Mass = %.6f",
            massData2.mass);
        m_textLine += m_textIncrement;

        Test::Step(settings);

        printf("%4.2f %4.2f %4.2f %4.2f \n", velocity.x, velocity.y,
            velocity2.x, velocity2.y);
    }
    static Test* Create()
}

```

```

    {
        return new KineticenergyCollision;
    }

};

static int testIndex = RegisterTest("Kinetic Energy and Work", "Kinetic
Energy: Collision", KineticenergyCollision::Create);

```

C++ Code 49: tests/kineticenergy_collision.cpp "Kinetic Energy Train Crash Box2D"

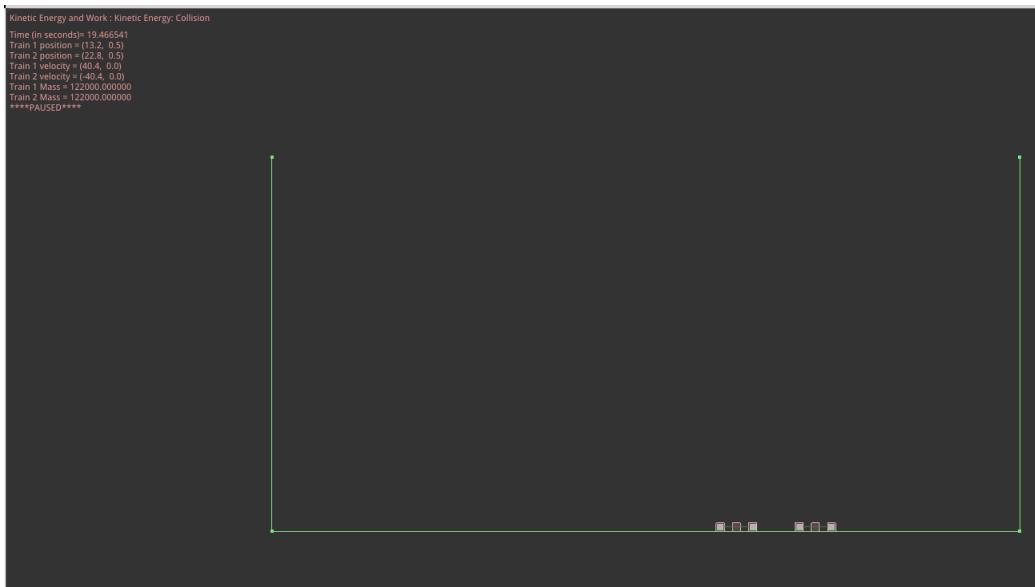


Figure 9.1: The simulation of two trains heading toward each other at speed of $v = 40.8 \text{ m/s}$ (the current simulation code can be located in: *DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/kineticenergy_collision.cpp*).

The simulation is very deterministic since we can repeat it and how the train collides then being thrown into the air is pretty much the same all the time. If in 1896, Box2D already exists, maybe there will be no need to make the demonstration of real locomotives collide into each other, if it is only for science discovery satisfaction.

III. WORK AND KINETIC ENERGY

[DF*] Work, W , is energy transferred to or from an object via a force acting on the object. Energy transferred to the object is positive work, while energy transferred from the object is negative work.

[DF*] The work done on a particle by a constant force \vec{F} during displacement \vec{d} is

$$W = Fd \cos \phi = \vec{F} \cdot \vec{d}$$

in which ϕ is the constant angle between the direction of \vec{F} and \vec{d} .

Only the component of \vec{F} that is along the displacement \vec{d} can do work on the object.

[DF*] When two or more forces act on an object, their net work is the sum of the individual works done by the forces, which is also equal to the work that would be done on the object by the net force \vec{F}_{net} of those forces.

[DF*] For a particle, a change ΔK in the kinetic energy equals the net work, W , done on the particle:

$$\Delta K = K_f - K_i = W$$

in which K_i is the initial kinetic energy of the particle and K_f is the kinetic energy after the work is done. The equation rearrange gives us

$$K_f = K_i + W$$

[DF*] Work has the SI unit of the joule, the same as the kinetic energy. The corresponding unit in the British system is the foot-pound ($ft \cdot lb$)

$$1 J = 1 kg \cdot m/s^2 = 1 N \cdot m = 0.738 ft \cdot lb \quad (9.2)$$

IV. SIMULATION FOR WORK BY TWO CONSTANT FORCES WITH Box2D

I take this from sample problem 7.02 of [6], but I modify everything.

Two inventors-engineers sliding an initially stationary 230 kg cutting tool that is just arrive from the front yard near their garden and ranch toward their home laboratory, for a displacement \vec{d} of magnitude 46 m . This newly wed spouse in Mt Logan, Yukon, Canada are very happy to be able to build their dream to be engineers, inventors and scientists together, not to mention they are supported by the government of Canada along with the community that is very friendly and open-minded to same-sex couple like them. The push \vec{F}_1 of DS Glanzsche is 23.0 N at an angle 30.0° downward from the horizontal; the pull \vec{F}_2 of DS Glanzsche' wife is 24.0 N at 33.0° above the horizontal. The magnitudes and directions of these forces do not change as the cutting tool moves, and the floor and the cutting tool make frictionless contact.

What is the net work done on the cutting tool by forces \vec{F}_1 and \vec{F}_2 during the displacement \vec{d} .

Solution:

The net work, W , done on the cutting tool by the two forces is the sum of the works they do individually.

We can treat the cutting tool as a particle and the forces are constant in both magnitude and direction.

The work done by \vec{F}_1 is

$$\begin{aligned} W_1 &= F_1 d \cos \phi_1 \\ &= (30)(46) \cos(30.0^\circ) \\ &= 1195.12 \end{aligned}$$

The work done by \vec{F}_2 is

$$\begin{aligned} W_2 &= F_2 d \cos \phi_2 \\ &= (24)(46) \cos(33.0^\circ) \\ &= 925.89 \end{aligned}$$

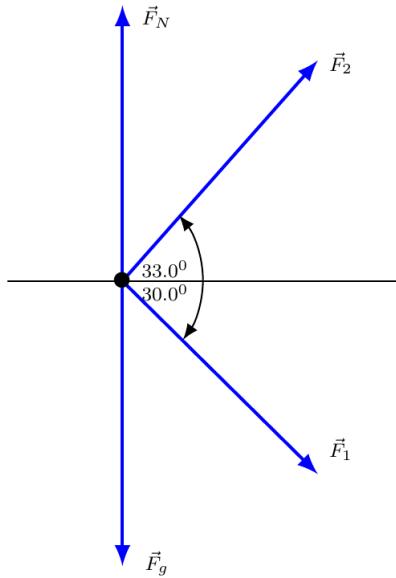


Figure 9.2: The free-body diagram for the cutting tool.

Thus, the net work W is

$$\begin{aligned} W &= W_1 + W_2 \\ &= 1195.12 + 925.89 \\ &= 2121.01 \end{aligned}$$

During the 46 m displacement, the newly wed transfer 2121.01 J of energy to the kinetic energy of the cutting tool.

$$\begin{aligned} W &= \Delta K \\ 2121.01 &= K_f - K_i \\ 2121.01 &= K_f - 0 \\ 2121.01 &= \frac{1}{2}mv_f^2 \\ 2121.01 &= \frac{1}{2}230v_f^2 \\ v_f^2 &= \frac{4242.02}{230} \\ v_f &= \sqrt{18.4436} \\ v_f &= 4.3 \end{aligned}$$

K_i is the initial kinetic energy, since the cutting tool at rest initially it will be 0, thus the cutting tool will be moving 4.3 m/s , thus in less than 11 seconds the cutting tool will be arriving in the home laboratory.

From v_f we can do backward checking to see if the speed is valid, from equation (7.3):

$$\begin{aligned} v_f^2 &= v_i^2 + 2a(x_f - x_i) \\ 4.3^2 &= 0 + 2a(46) \\ a &= 0.200978 \end{aligned}$$

then by using Newton's Second Law

$$\begin{aligned} F &= ma \\ F &= (230)0.200978 \\ F &= 46.225 \end{aligned}$$

With the known force, we can find the work from the result above:

$$\begin{aligned} W &= Fs \\ W &= (46.225)46 \\ W &= 2126.35 \end{aligned}$$

There's a bit of error margin here, when we compute from the push and pull forces we obtain work of 2121.01 J, when we compute from the velocity v_f , the obtain a then F and we can compute W it becomes 2126.35 J, the relative error compared to the $W = 2121.01$ is 0.0025.

```
#define DEGTORAD 0.0174532925199432957f
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class KineticenergyWork : public Test
{
public:
    KineticenergyWork()
    {
        b2Body* ground = NULL;
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(-46.0f,
                , 46.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(46.0f, 0.0f), b2Vec2(46.0f,
                46.0f));
        }
    }
};
```

```

        ground->CreateFixture(&shape, 0.0f);
    }
{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
        0.0f));
    ground->CreateFixture(&shape, 0.0f);
}
// Create the cutting tool going to positive x axis
b2PolygonShape boxShape;
boxShape.SetAsBox(0.5f, 0.5f);

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.75f;
boxFixtureDef.density = 230.0f; // this will affect the box
mass
boxFixtureDef.friction = 0.0f; // frictionless
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(-23.0f, 0.5f);

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
;
//m_box->SetLinearVelocity(b2Vec2(40.8f, 0.0f));

m_time = 0.0f;
}
b2Body* m_box;
float m_time;
float F1 = 30.0f;
float F2 = 24.0f;
float d = 46.0f;
int theta = 30;
int theta2 = 33;

float fsum = F1*d*cosf(theta*DEGTORAD) + F2*d*cosf(theta2*DEGTORAD);
float vf = sqrt(2*fsum/230);
void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_D:

```

```

        for (int i = 0; i <= 46.0f; ++i)
        {
            //m_box->ApplyForceToCenter(b2Vec2(2121.0f, 0.0
            f), true);
            m_box->ApplyForceToCenter(b2Vec2(F1*i*cosf(
                theta*DEGTORAD) + F2*i*cosf(theta2*DEGTORAD
            ), 0.0f), true); // resulting in same work
            for 46 meter displacement
        }
        break;
    case GLFW_KEY_F:
        m_box->SetLinearVelocity(b2Vec2(vf, 0.0f));
        break;
    }
}
void Step(Settings& settings) override
{
    m_time += 1.0f / 60.0f; // assuming we are using frequency of
    60 Hertz
    b2MassData massData = m_box->GetMassData();
    b2Vec2 position = m_box->GetPosition();
    b2Vec2 velocity = m_box->GetLinearVelocity();

    g_debugDraw.DrawString(5, m_textLine, "Press D to make the
        cutting tool move toward positive x axis with force");
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Press F to make the
        cutting tool move toward positive x axis with constant
        velocity");
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
        f", m_time);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Cutting tool position
        = (%4.1f, %4.1f)", position.x, position.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Cutting tool velocity
        = (%4.1f, %4.1f)", velocity.x, velocity.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Work done to move the
        cutting tool 46 meter = %.4f", fsum);
    m_textLine += m_textIncrement;

    g_debugDraw.DrawString(5, m_textLine, "Cutting tool Mass =
        %.6f", massData.mass);
    m_textLine += m_textIncrement;

    Test::Step(settings);
}

```

```

        printf("%4.2f %4.2f \n", position.x, position.y);
    }
    static Test* Create()
    {
        return new KineticenergyWork();
    }

};

static int testIndex = RegisterTest("Kinetic Energy and Work", "Kinetic
Energy: Work", KineticenergyWork::Create);

```

C++ Code 50: *tests/kineticenergy_work.cpp* "Kinetic Energy Work Box2D"

Some explanations for the codes:

- In the keyboard press events, I create two options, one by using **ApplyForceToCenter()** and the other is **SetLinearVelocity()**. By manual computation we obtain that the kinetic energy shall give velocity of $v_f = 4.3 \text{ m/s}$, but when using the **ApplyForceToCenter()**, the speed of movement is different as it produces another number for the constant velocity that is working on the body. This might be caused by the force is converted into Newton's second Law that gives the acceleration to the body, then the velocity will be obtained from it. The formula behind the **ApplyForceToCenter()** should be look into more. Why it produces different velocity than what it should be ? For a while, the keypress "D" is still under thinking to make it right. The keypress "F" is the right one for this problem.

```

void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_D:
            for (int i = 0; i <= 46.0f; ++i)
            {
                //m_box->ApplyForceToCenter(b2Vec2(2121.0f
                //, 0.0f), true);
                m_box->ApplyForceToCenter(b2Vec2(F1*i*cosf
                    (theta*DEGTORAD) + F2*i*cosf(theta2*
                    DEGTORAD), 0.0f), true); // resulting
                in same work for 46 meter displacement
            }
            break;
        case GLFW_KEY_F:
            m_box->SetLinearVelocity(b2Vec2(vf, 0.0f));
            break;
    }
}

```

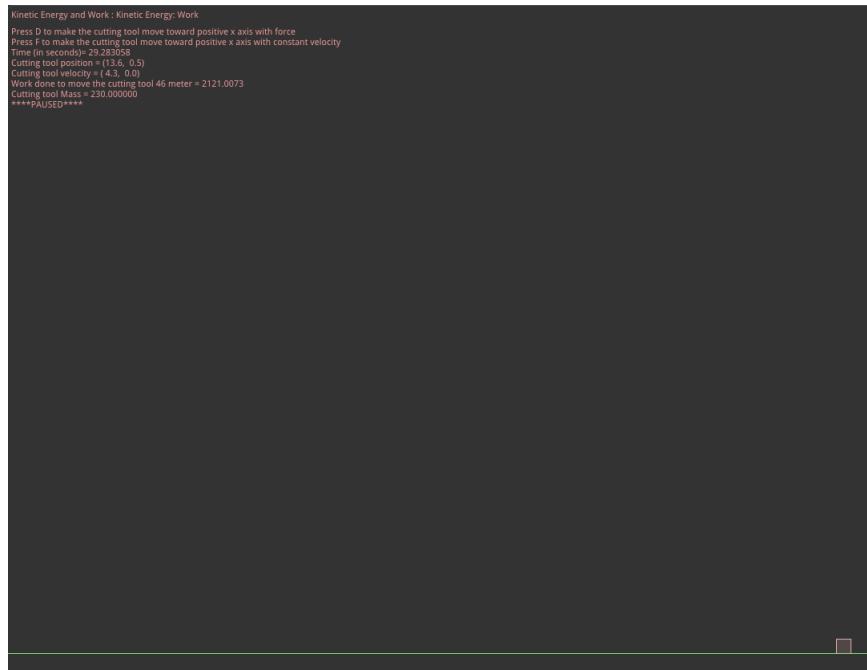


Figure 9.3: The simulation of a cutting tool being moved for 46 m by two forces of push and pull with different angles, thus resulting in constant velocity of 4.3 m/s (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/kineticenergy_work.cpp`).

Now, we can plot the position of the cutting tool with gnuplot, recompile the testbed to make the change occurs then type:

./testbed > work.txt

Plot it with gnuplot from the working directory:

```
gnuplot
set xlabel "time"
plot "work.txt" using 1 with lines title "x_t"
```

V. WORK DONE BY THE GRAVITATIONAL FORCE

[DF*] The work W_g done by the gravitational force \vec{F}_g on a particle-like object of mass m as the object moves through a displacement d is given by

$$W_g = mgd \cos \phi \quad (9.3)$$

in which ϕ is the angle between \vec{F}_g and \vec{d} .

[DF*] The work W_a done by an applied force as a particle-like object is either lifted or lowered is related to the work W_g done by the gravitational force and the change ΔK in the object's kinetic energy by

$$\Delta K = K_f - K_i = W_a + W_g \quad (9.4)$$

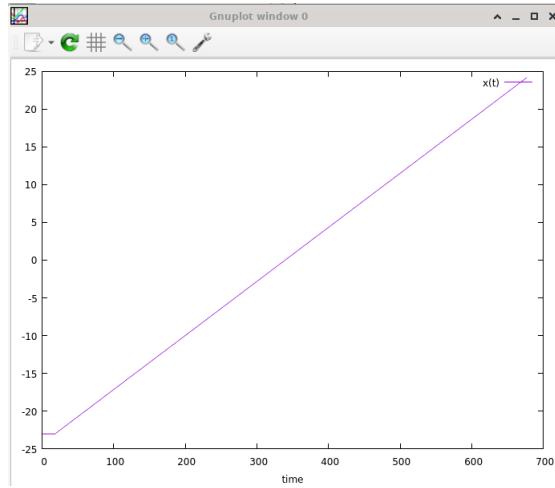


Figure 9.4: The gnuplot of the position toward x axis for the cutting tool till it reach the distance of 46 m. The time is in Hertz, thus $60 \text{ time} = 1 \text{ seconds}$, around 11 seconds will the cutting tool arrives in the home laboratory.

If $K_f = K_i$, then the equation reduces to

$$W_a = -W_g$$

which tells us that the applied force transfers as much energy to the object as the gravitational force transfers from it.

We can rewrite it as

$$W_a = -mgd \cos \phi \quad (9.5)$$

it is the work done in lifting and lowering, with ϕ being the angle between \vec{F}_g and \vec{d} . If the displacement is vertically upward, then $\phi = 180^0$ and the work done by the applied force equals mgd . If the displacement is vertically downward, then $\phi = 0^0$ and the work done by the applied force equals $-mgd$.

VI. SIMULATION FOR WORK IN PULLING AND PUSHING A SACK UP A FRICTIONLESS SLOPE

Taken from sample problem 7.04 and Problem no 24 chapter 7 of [6], I modify it a lot.

In Valhalla Projection, it divided into two parts: Midgard Area (where villagers with low education and high laziness throw trashes there when they pass by), and the other one is Asgard Area, the forest where DS Glanzsche has been contracted by Berlin to work as Grass cutter, 7 chakras builder, maintaining the forest. The Midgard area is descending from main road, that probably why based on Feng Shui, you should not have a home below the main road. While the Asgard area is ascending from the main road, meaning will lift you higher, like heaven and sky.

Now after cleaning up Midgard area, DS Glanzsche has a problem to bring a sack full of trashes that is too heavy into the main road, to be transported then by motorcycle to the trash collection point down at the city, the sack is filled with diapers, leftover clothes, etc that make the

sack has mass of 50 kg . Now, learning physics should not come to a waste, she has to learn how this sack full of trashes can be brought to the main road, is it better to pull it from above? if it is pulled along a ramp, the sack starts and ends at rest. She can create the ramp frictionless from glass material with angle of 30° , through distance $d = 20 \text{ m}$.

Another option is to push the sack through the ramp, with a horizontal force F_a of minimum magnitude that we can determine, the sack has zero kinetic energy at the start of the displacement.

Calculate how much work is done by each force acting on the sack full of trashes with pulling method and the minimum amount of force in pushing method.

Solution:

The first thing to do with most physics problems involving forces is to draw a free-body diagram so we can organize our thoughts.

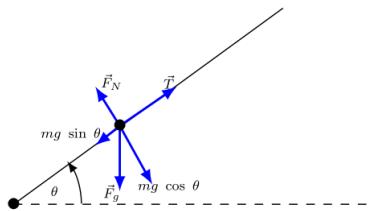


Figure 9.5: The free-body diagram of the sack being pulled from above with a rope.

Pull the Sack

1. Work W_N by the normal force:

The normal force is perpendicular to the slope and thus also to the sack's displacement. Thus the normal force does not affect the sleigh's motion and does zero work.

$$W_N = F_N d \cos 90^\circ = 0$$

2. Work W_g by the gravitational force:

We use the full gravitational force \vec{F}_g , the angle between \vec{F}_g and \vec{d} is 120° . This gives us

$$\begin{aligned} W_g &= F_g d \cos 120^\circ \\ &= mgd \cos 120^\circ \\ &= (50)(9.8)(20)(-0.5) \\ &= -4900 \end{aligned}$$

3. Work W_T by the rope's force:

The quickest way to calculate this is by using the work-kinetic energy theorem ($\Delta K = W$), where the net work W done by the forces is $W_N + W_g + W_T$ and the change ΔK in the kinetic

energy is just zero (because the initial and final kinetic energies are the same). This gives us

$$\begin{aligned}\Delta K &= W_N + W_g + W_T \\ 0 &= 0 - 4900 + W_T \\ W_T &= 4900\end{aligned}$$

The work needed to pull the sack is 4900 J.

The second way to calculate W_T is by applying Newton's second law for motion along the x axis to find the magnitude F_T of the rope's force. Assuming that the acceleration along the slope is zero,

$$\begin{aligned}F_{net,x} &= ma_x \\ F_T - mg \sin 30^0 &= m(0) \\ F_T &= mg \sin 30^0\end{aligned}$$

We have the magnitude of the force F_T , now we can find W_T . Because the force and the displacement are both have the same direction, up the slope, the angle between \vec{F}_T and \vec{d} is zero. To find the work done by the rope's force:

$$\begin{aligned}W_T &= F_T d \cos 0^0 \\ &= (mg \sin 30^0)d \cos 0^0 \\ &= (50)(9.8)(0.5)(20)(1) \\ &= 4900\end{aligned}$$

Confirmed with the second method, we need the same work of 4900 J to pull the sack. Combine with nutrition and biology, what kind of intake needed for that amount of energy in calorie terms, is a Koko Krunch cereal enough? or DS Glanzsche' favorite Choipanku enough to pull that sack for dinner the previous night before working in the morning for this? Tell you what, Big Tree is happy and want to give you all with that peanuts that you love, Kacang Bali and Cashew along with Polong. You get more than symbiosis mutualism when working with Nature. All science are related and can be fun to work on, all we do are all still science, even how people fall in love. Still based on astronomical places of planets and asteroids.

Another way of thinking, this is in Mechanical Engineering area, but can be noted here since this might be useful for all kind of applications. Instead of pulling the rope with human' labour or animal labour, my dogs will be wild and won't do any of this, we can use a mechanical rotation wheel so the rope will be pulled when the wheel rotating, just tie the rope to the wheel, and make it rotating automatically to the same direction till the sack arrived. One day this can be created and demonstrated, with minimal cost. It is called rotational leverage.

Push the Sack

Now for the pushing part, we will calculate how much force is needed to push that sack to the main road.

1. Work W_g by the gravitational force

This work bring down the sack to the bottom of the ramp due to gravity.

$$\begin{aligned} W_g &= F_g d \sin \theta \\ &= mgd \sin 30^0 \\ &= (50)(9.8)(20)(0.5) \\ &= 4900 \end{aligned}$$

The work is still the same in the pull method, since this work by gravitational force will stay the same with same condition.

2. Work W_a by the pushing horizontal force

To push the sack till it went up the ramp we can use Newton's second law

$$\begin{aligned} \sum F &= 0 \\ F_a - F_g &= 0 \\ F_a &= mg \sin \theta \\ &= (50)(9.8)(0.5) \\ &= 245 \end{aligned}$$

The minimum horizontal force, F_a needed is 245 N. The work from this is

$$\begin{aligned} W_a &= F_a d \\ &= 245(20) \\ &= 4900 \end{aligned}$$

The same amount of work of 4900 J with the pulling method is needed to bring the sack to the main road.

```
#define DEGTORAD 0.0174532925199432957f
#include <iostream>
#include "test.h"

class WorkGravitationalforce : public Test
{
public:
    WorkGravitationalforce()
    {
        m_world->SetGravity(b2Vec2(0.0f, -9.8f));
        float L = 10.0f;
        float a = 1.0f;
        float b = 1.5f;
        m_world->SetGravity(b2Vec2(0.0f, -9.8f));
        b2Body* ground = NULL;
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);
```

```

        b2EdgeShape shape;
        shape.SetTwoSided(b2Vec2(-46.0f,0.0f), b2Vec2(46.0f,
0.0f));
        ground->CreateFixture(&shape, 0.0f);
    }
{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(0.0f,8.0f), b2Vec2(46.0f, 8.0
f));
    ground->CreateFixture(&shape, 0.0f);
}
// Create the pulley
b2BodyDef bd;
ground = m_world->CreateBody(&bd);

b2CircleShape circle;
circle.m_radius = 1.0f;

circle.m_p.Set(0.0f, b + L); // circle with center of
(0,b+L)
ground->CreateFixture(&circle, 0.0f);
}
// Create the triangle
b2ChainShape chainShape;
b2Vec2 vertices[] = {b2Vec2(-17.3205,0), b2Vec2(0,0), b2Vec2
(0,10)};
chainShape.CreateLoop(vertices, 3);

b2FixtureDef groundFixtureDef;
groundFixtureDef.density = 0;
groundFixtureDef.shape = &chainShape;

b2BodyDef groundBodyDef;
groundBodyDef.type = b2_staticBody;

b2Body *groundBody = m_world->CreateBody(&groundBodyDef);
b2Fixture *groundBodyFixture = groundBody->CreateFixture(&
groundFixtureDef);

{
    // Create the sack on a triangle
    b2PolygonShape boxShape;
    boxShape.SetAsBox(a, b); // width and length of the
box

```

```

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.75f;
boxFixtureDef.density = 8.334f; // this will affect
    the box mass
boxFixtureDef.friction = 0.0f; // frictionless plane
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(-17.0f, 0.5);
// boxBodyDef.fixedRotation = true;

m_boxl = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_boxl->CreateFixture(&
    boxFixtureDef);

// Create the box hanging on the right
b2PolygonShape boxShape2;
boxShape2.SetAsBox(a, b); // width and length of the
    box

b2FixtureDef boxFixtureDef2;
boxFixtureDef2.restitution = 0.75f;
boxFixtureDef2.density = 16.667f; // this will affect
    the box mass, mass = density*5.9969
boxFixtureDef2.friction = 0.3f;
boxFixtureDef2.shape = &boxShape2;

b2BodyDef boxBodyDef2;
boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(3.0f, L);
// boxBodyDef2.fixedRotation = true;

m_boxr = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
    boxFixtureDef2);

// Create the Pulley
b2PulleyJointDef pulleyDef;
b2Vec2 anchor1(-16.0f, 0.5f); // the position of the
    end string of the left cord to connect to the sack
b2Vec2 anchor2(2.0f, L); // the position of the end
    string of the right cord to connect to the right
    puller
b2Vec2 groundAnchor1(-1.0f, b + L); // the string of
    the cord is tightened at (-1.0f, b+L)
b2Vec2 groundAnchor2(1.5f, b + L); // the string of
    the cord is tightened at (1.5f, b+L)

```

```

        // the last float is the ratio
        pulleyDef.Initialize(m_boxl, m_boxr, groundAnchor1,
            groundAnchor2, anchor1, anchor2, 1.0f);

        m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&
            pulleyDef);
    }
}

b2Body* m_boxl;
b2Body* m_boxr;
float F1 = 50.0f;
int theta = 30;
b2PulleyJoint* m_joint1;
void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_D:
            m_boxr->ApplyForceToCenter(b2Vec2(20000.0f, 0.0f),
                true);
            break;
        case GLFW_KEY_F:
            m_boxr->SetLinearVelocity(b2Vec2(14.0f, 0.0f));
            break;
        case GLFW_KEY_G:
            for (int i = 0; i <= 46.0f; ++i)
            {
                m_boxl->ApplyForceToCenter(b2Vec2(F1*i*cosf(
                    theta*DEGTORAD), 0.0f), true); // resulting
                in same work for 46 meter displacement
            }
            break;
    }
}
void Step(Settings& settings) override
{
    Test::Step(settings);
    b2MassData massData1 = m_boxl->GetMassData();
    b2Vec2 position1 = m_boxl->GetPosition();
    b2Vec2 velocity1 = m_boxl->GetLinearVelocity();
    b2MassData massData2 = m_boxr->GetMassData();
    b2Vec2 position2 = m_boxr->GetPosition();
    b2Vec2 velocity2 = m_boxr->GetLinearVelocity();
    float m1 = massData1.mass;
    float m2 = massData2.mass;
    float g = 9.8f;

    float ke1 = 0.5*m1*((velocity1.x)*(velocity1.x));
}

```

```

float ke2 = 0.5*m2*((velocity2.x)*(velocity2.x));
float a = (m2-m1*sinf(30*DEGTORAD))*g / (m1+m2);
float T = (m1*a) + (m1*g*sinf(30*DEGTORAD));

g_debugDraw.DrawString(5, m_textLine, "Press D to pull the
    sack with force of 20,000 N");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Press F to pull the
    sack with linear velocity of 14 m/s");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Press G to push the
    sack with force of 50 N");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Left Box position =
    (%4.1f, %4.1f)", position1.x, position1.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Left Box velocity =
    (%4.1f, %4.1f)", velocity1.x, velocity1.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Left Box Mass = %.6f",
    massData1.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Left Box Kinetic
    Energy = %.6f", ke1);
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "Right Box position =
    (%4.1f, %4.1f)", position2.x, position2.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Right Box velocity =
    (%4.1f, %4.1f)", velocity2.x, velocity2.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Right Box Mass = %.6f"
    , massData2.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Right Box Kinetic
    Energy = %.6f", ke2);
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "Acceleration = %.6f",
    a);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "The tension of the
    cord = %.6f", T);
m_textLine += m_textIncrement;

float ratio = m_joint1->GetRatio();
float L = m_joint1->GetCurrentLengthA() + ratio * m_joint1->

```

```

        GetCurrentLengthB();
        g_debugDraw.DrawString(5, m_textLine, "L1 + %4.2f * L2 = %4.2
            f", (float) ratio, (float) L);
        m_textLine += m_textIncrement;

        printf("%4.2f %4.2f %4.2f %4.2f \n", position1.y, position2.x
            , ke1, ke2);
    }

    static Test* Create()
    {
        return new WorkGravitationalforce;
    }
};

static int testIndex = RegisterTest("Kinetic Energy and Work", "
    Gravitational Force", WorkGravitationalforce::Create);

```

C++ Code 51: *tests/work_gravitationalforce.cpp "Push and Pull along a Ramp Box2D"*

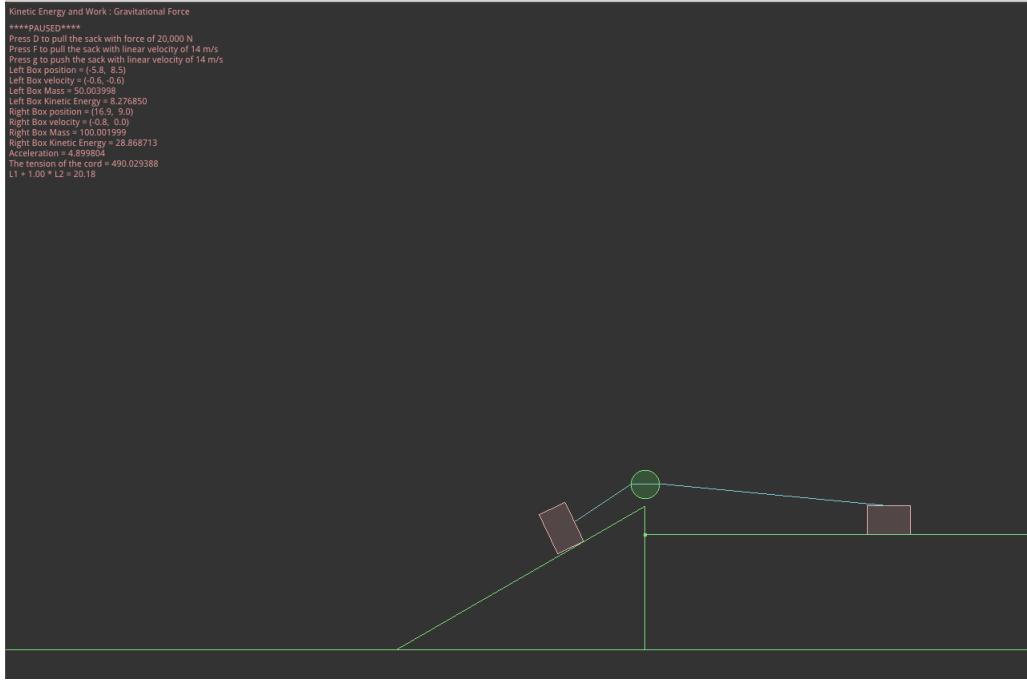


Figure 9.6: The simulation of sack of trashes being push from below or being pulled from above along a frictionless ramp with angle of elevation of $\theta = 30^\circ$ (the current simulation code can be located in: *DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/work_gravitationalforce.cpp*).

Some explanations for the codes:

- For the push force with key press "G", I use the force of $F1 = 50\text{ N}$, because if I put 245 N , the box will jump out of sudden, yes we need that amount of 245 N based on the calculation

from above, but it should be gained after repeated pushing, as when we push anything it is continuous giving of forces, not only once.

For the pull with key press "D" and "F", we can set the analogy for "D" as pull with a mechanical machine, and for "F" it is like tie up the end of the cord into a DC motor, motorcycle or a horse then make it run with velocity of 14 m/s for certain meter to pull the sack up.

```
void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_D:
            m_boxr->ApplyForceToCenter(b2Vec2(20000.0f, 0.0f),
                                         true);
            break;
        case GLFW_KEY_F:
            m_boxr->SetLinearVelocity(b2Vec2(14.0f, 0.0f));

            break;
        case GLFW_KEY_G:
            for (int i = 0; i <= 46.0f; ++i)
            {
                m_boxl->ApplyForceToCenter(b2Vec2(F1*i *
                                         cosf(theta*DEGTORAD), 0.0f), true); // resulting in same work for 46 meter displacement
            }
            break;
    }
}
```

VII. WORK DONE BY A SPRING FORCE

Spring will be explained extensively in Oscillation chapter far below. We will only put minimal needed to know. This is the introduction on Spring Force.

[DF*] The force \vec{F}_s from a spring is proportional to the displacement \vec{d} of the free end from its position when the spring is in the relaxed state (neither compressed nor extended). The spring force is given by

$$\vec{F}_s = -k\vec{d} \quad (9.6)$$

which is known as Hooke's law. The minus sign indicates that the direction of the spring force is always opposite the displacement of the spring's free end. The constant k is the spring constant to measure the spring's stiffness.

Now if we put the x axis parallel to the length of the spring, with the origin ($x = 0$) at the position of the free end when the spring is in its relaxed state. Then we will have

$$F_x = -kx \quad (9.7)$$

If x is positive (the spring is stretched toward the positive x axis), then F_x is negative. If x is negative (the spring is compressed toward the negative x axis), then F_x is positive.

Hooke's law is a linear relationship between F_x and x

[DF*] A spring force is a variable force. It varies with the displacement of the spring's free end.

[DF*] Two assumptions commonly used in spring problem:

1. The spring is massless; its mass is negligible relative to the block's mass.
2. It is an ideal spring; that is, it obeys Hooke's law.

[DF*] Let a block attached to a spring' free end, the block' initial position be x_i and its later position be x_f . We can create partitions between the initial and later position of same width. Thus, we can approximate the force magnitude as being constant within the partition. Then, we can find the work done within each segment

$$\begin{aligned} W_s &= \sum -F_{xj}\Delta x \\ &= \int_{x_i}^{x_f} -F_x dx \\ &= \int_{x_i}^{x_f} -kx dx \\ &= -k \int_{x_i}^{x_f} x dx \\ &= \left(-\frac{1}{2}k\right) [x^2]_{x_i}^{x_f} \\ &= \left(-\frac{1}{2}k\right) (x_f^2 - x_i^2) \end{aligned}$$

thus

$$W_s = \frac{1}{2}kx_i^2 - \frac{1}{2}kx_f^2 \quad (9.8)$$

VIII. SIMULATION FOR SPRING WORK WITH Box2D

Taken from sample problem 7.06 from [6].

A box of mass $m = 14 \text{ kg}$ located at $x = 15$ slides across a horizontal frictionless counter with speed $v = 10 \text{ m/s}$. It then runs into and compresses a spring of spring constant $k = 330 \text{ N/m}$, with its' free end located at $x = 5$. When the box is momentarily stopped by the spring, by what distance d is the spring compressed?

Solution:

The work W_s done on the box by the spring force is related to the requested distance d

$$W_s = -\frac{1}{2}kx^2$$

with d replacing x . The work, W_s is also related to the kinetic energy of the box by

$$K_f - K_i = W$$

The box's kinetic energy has an initial value of

$$K = \frac{1}{2}mv^2$$

and a value of zero when the box is momentarily at rest.

Now, we write the work-kinetic energy theorem for the box as

$$\begin{aligned} K_f - K_i &= -\frac{1}{2}kd^2 \\ 0 - \frac{1}{2}mv^2 &= -\frac{1}{2}kd^2 \\ d &= v\sqrt{\frac{m}{k}} \\ &= (10)\sqrt{\frac{14}{330}} \\ d &= 2.0597 \end{aligned}$$

the spring is compressed till 2.0597 m.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class SpringWork : public Test
{
public:
    SpringWork()
    {
        b2Body* ground = NULL;
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                0.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        // Create a static body as the box for the spring
        b2BodyDef bd1;
        bd1.type = b2_staticBody;
        bd1.angularDamping = 0.1f;

        bd1.position.Set(1.0f, 0.5f);
        b2Body* springbox = m_world->CreateBody(&bd1);

        b2PolygonShape shape1;
```

```
shape1.SetAsBox(0.5f, 0.5f);
springbox->CreateFixture(&shape1, 5.0f);

// Create the box as the movable object
b2PolygonShape boxShape1;
boxShape1.SetAsBox(0.5f, 0.5f);

b2FixtureDef boxFixtureDef1;
boxFixtureDef1.restitution = 0.75f;
boxFixtureDef1.density = 14.0f; // this will affect the box
mass
boxFixtureDef1.friction = 0.10;
boxFixtureDef1.shape = &boxShape1;

b2BodyDef boxBodyDef1;
boxBodyDef1.type = b2_dynamicBody;
boxBodyDef1.position.Set(15.0f, 0.5f);

m_box1 = m_world->CreateBody(&boxBodyDef1);
b2Fixture *boxFixture1 = m_box1->CreateFixture(&
boxFixtureDef1);

// Create the box attached to the spring free end
b2PolygonShape boxShape;
boxShape.SetAsBox(0.1f, 0.5f);

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.75f;
boxFixtureDef.density = 7.3f; // this will affect the box
mass
boxFixtureDef.friction = 0.0f;
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(5.0f, 0.5f);

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
;
//m_box->SetGravityScale(-7); // negative means it will goes
upward, positive it will goes downward
// Make a distance joint for the box / ball with the static
box above
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
```

```

        jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f),
                      boxBodyDef.position);
        jd.collideConnected = true; // In this case we decide to
                                    allow the bodies to collide.
        m_length = jd.length;
        m_minLength = 2.0f;
        m_maxLength = 10.0f;
        b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
                           m_dampingRatio, jd.bodyA, jd.bodyB);

        m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
        m_joint->SetMinLength(m_minLength);
        m_joint->SetMaxLength(m_maxLength);

        m_time = 0.0f;
    }
    b2Body* m_box;
    b2Body* m_box1;
    b2DistanceJoint* m_joint;
    float m_length;
    float m_time;
    float m_minLength;
    float m_maxLength;
    float m_hertz;
    float m_dampingRatio;

    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_A:
                m_box1->SetLinearVelocity(b2Vec2(-10.0f, 0.0f));
                break;
            case GLFW_KEY_S:
                m_box1->SetLinearVelocity(b2Vec2(-5.0f, 0.0f));
                break;
        }
    }
    void UpdateUI() override
    {
        ImGui::SetNextWindowPos(ImVec2(10.0f, 150.0f));
        ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
        ImGui::Begin("Joint Controls", nullptr,
                    ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

        if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0
                               f"))
        {

```

```

        m_length = m_joint->SetLength(m_length);
    }

    if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"))
    {
        float stiffness;
        float damping;
        b2LinearStiffness(stiffness, damping, m_hertz,
                           m_dampingRatio, m_joint->GetBodyA(), m_joint->
                           GetBodyB());
        m_joint->SetStiffness(stiffness);
        m_joint->SetDamping(damping);
    }

    if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
                           , 2.0f, "%.1f"))
    {
        float stiffness;
        float damping;
        b2LinearStiffness(stiffness, damping, m_hertz,
                           m_dampingRatio, m_joint->GetBodyA(), m_joint->
                           GetBodyB());
        m_joint->SetStiffness(stiffness);
        m_joint->SetDamping(damping);
    }

    ImGui::End();
}

void Step(Settings& settings) override
{
    b2MassData massData1 = m_box1->GetMassData();
    b2MassData massData = m_box->GetMassData();
    b2Vec2 position = m_box->GetPosition();
    b2Vec2 position1 = m_box1->GetPosition();
    b2Vec2 velocity = m_box->GetLinearVelocity();
    b2Vec2 velocity1 = m_box1->GetLinearVelocity();
    m_time += 1.0f / 60.0f; // assuming we are using frequency of
                           60 Hertz

    g_debugDraw.DrawString(5, m_textLine, "Press A/S to apply
                                different speed to the box");
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                                f", m_time);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Spring Mass position =
                                (%4.1f, %4.1f)", position.x, position.y);
}

```

```

        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Spring Mass velocity = "
            "(%4.1f, %4.1f)", velocity.x, velocity.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Spring Block Mass = "
            "%.6f", massData.mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Moving Box position = "
            "(%4.1f, %4.1f)", position1.x, position1.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Moving Box velocity = "
            "(%4.1f, %4.1f)", velocity1.x, velocity1.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Moving Box Mass = %.6f"
            ", massData1.mass");
        m_textLine += m_textIncrement;
        // Print the result in every time step then plot it into
        // graph with either gnuplot or anything

        printf("%4.2f\n", position.x);

        Test::Step(settings);
    }
    static Test* Create()
    {
        return new SpringWork;
    }

};

static int testIndex = RegisterTest("Kinetic Energy and Work", "Spring Work"
    ", SpringWork::Create");

```

C++ Code 52: *tests/spring_work.cpp* "Work Done by A Spring Force Box2D"

Now, we can plot the position of the spring free end with gnuplot, recompile the testbed to make the change occurs then type:
./testbed > springwork.txt

Plot it with gnuplot from the working directory:

```

gnuplot
set xlabel "time"
plot "springwork.txt" using 1 with lines title "d"

```

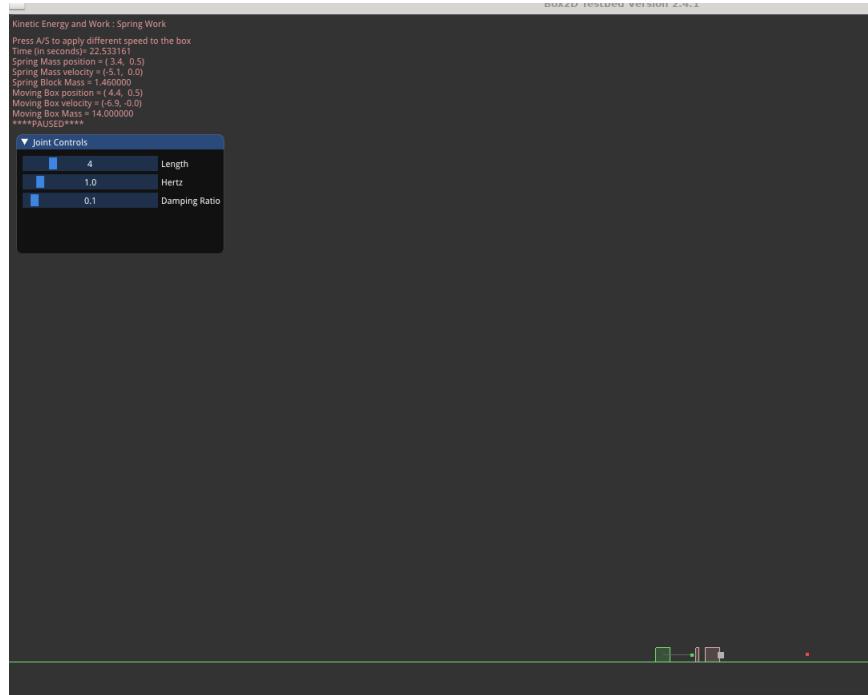


Figure 9.7: The spring is compressed for d distance when a box hit it with speed of $v = 10 \text{ m/s}$ on the frictionless floor (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/spring_work.cpp`).

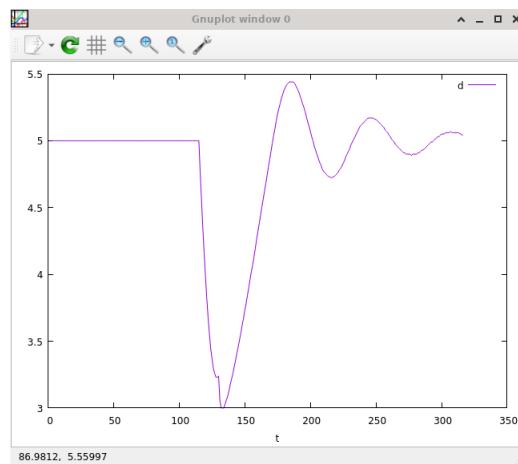


Figure 9.8: The plot of the spring position after being hit by a box with speed of $v = 10 \text{ m/s}$, it will goes to negative x axis then to positive x axis, oscillates a bit then goes back to its' equilibrium position.

IX. WORK DONE BY A GENERAL VARIABLE FORCE

[DF*] When the force \vec{F} on a particle-like object depends on the position of the object, the work done by \vec{F} on the object while the object moves from an initial position r_i with coordinates (x_i, y_i, z_i) to a final position r_f with coordinates (x_f, y_f, z_f) must be found by integrating the force.

If we assume that component F_x may depend on x but not on y or z , component F_y may depend on y but not on x or z , and component F_z may depend on z but not on x or y , then the work is

$$W = \int_{x_i}^{x_f} F_x \, dx + \int_{y_i}^{y_f} F_y \, dy + \int_{z_i}^{z_f} F_z \, dz + \quad (9.9)$$

[DF*] If \vec{F} has only an x component, then this reduces to

$$W = \int_{x_i}^{x_f} F(x) \, dx \quad (9.10)$$

[DF*] Example: Work, Two Dimensional Integration

Force $\vec{F} = (3x^2 N)\hat{i} + (4N)\hat{j}$, with x in meters, acts on a particle, changing only the kinetic energy of the particle. How much work is done on the particle as it moves from coordinates $(2 \text{ m}, 3 \text{ m})$ to $(3 \text{ m}, 0 \text{ m})$? Does the speed of the particle increasee, decrease or remain the same?

Solution:

The force is a variable force because its x component depends on the value of x . Thus, we use

$$\begin{aligned} W &= \int_2^3 3x^2 \, dx + \int_3^0 4 \, dy \\ &= 3 \int_2^3 x^2 \, dx + 4 \int_3^0 dy \\ &= 3 \left[\frac{1}{3}x^3 \right]_2^3 + 4 [y]_3^0 \\ &= [3^3 - 2^3] + 4[0 - 3] \\ &= 7 \end{aligned}$$

The positive 7 J result means that energy is transferred to the particle by force \vec{F} . Thus, the kinetic energy of the particle increases and, because $K = \frac{1}{2}mv^2$, its speed must also increase. If the work had come out negative, the kinetic energy and speed would have decreased.

X. SIMULATION FOR WORK DONE BY A GENERAL VARIABLE FORCE WITH Box2D

Taken from problem number 42, chapter 7 from [6].

A cord is attached to a card that can slide along a frictionless horizontal rail aligned along an x axis. The left end of the cord is pulled over a pulley, of negligible mass and friction and at cord height $h = 1.20 \text{ m}$, so the cart slides from $x_1 = 3.00 \text{ m}$ to $x_2 = 1.00 \text{ m}$. During the move, the tension in the cord is a constant 25.0 N. What is the change in the kinetic energy of the cart during

the move?

Solution:

We solve the problem using the work-kinetic energy theorem, which states that the change in kinetic energy is equal to the work done by the applied force, $\Delta K = W$. In our problem, the work done is

$$W = Fd$$

where F is the tension in the cord and d is the length of the cord pulled as the cart slides from x_1 to x_2 . We have

$$\begin{aligned} d &= \sqrt{x_1^2 + h^2} - \sqrt{x_2^2 + h^2} \\ &= \sqrt{(3)^2 + (1.2)^2} - \sqrt{(1)^2 + (1.2)^2} \\ &= 3.23 - 1.56 \\ &= 1.67 \end{aligned}$$

which yields

$$\begin{aligned} \Delta K &= Fd \\ &= (25)(1.67) \\ &= 41.7 \end{aligned}$$

the change in the kinetic energy of the cart is 41.7 J.

```
#define DEGTORAD 0.0174532925199432957f
#include <iostream>
#include "test.h"

class WorkGeneralvariable : public Test
{
public:
    WorkGeneralvariable()
    {
        m_world->SetGravity(b2Vec2(0.0f, -9.8f));
        float L = 10.0f;
        float a = 1.0f;
        float b = 1.5f;
        m_world->SetGravity(b2Vec2(0.0f, -9.8f));
        b2Body* ground = NULL;
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                0.0f));
        }
    }
};
```

```

        ground->CreateFixture(&shape, 0.0f);
    }
{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(0.0f, 8.8f), b2Vec2(46.0f, 8.8
        f));
    ground->CreateFixture(&shape, 0.0f);
}
// Create the pulley
b2BodyDef bd;
ground = m_world->CreateBody(&bd);

b2CircleShape circle;
circle.m_radius = 1.0f;

circle.m_p.Set(0.0f, b + L); // circle with center of
// (0,b+L)
ground->CreateFixture(&circle, 0.0f);
}
// Create the triangle
b2ChainShape chainShape;
b2Vec2 vertices[] = {b2Vec2(-23,10), b2Vec2(0,0), b2Vec2
(0,10)};
chainShape.CreateLoop(vertices, 3);

b2FixtureDef groundFixtureDef;
groundFixtureDef.density = 0;
groundFixtureDef.shape = &chainShape;

b2BodyDef groundBodyDef;
groundBodyDef.type = b2_staticBody;

b2Body *groundBody = m_world->CreateBody(&groundBodyDef);
b2Fixture *groundBodyFixture = groundBody->CreateFixture(&
groundFixtureDef);

{
    // Create the box on the left
    b2PolygonShape boxShape;
    boxShape.SetAsBox(a, b); // width and length of the
    box

    b2FixtureDef boxFixtureDef;
    boxFixtureDef.restitution = 0.75f;
    boxFixtureDef.density = 8.334f; // this will affect
}

```

```

        the box mass
boxFixtureDef.friction = 0.3f; // frictionless plane
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(-7.0f, 11.5);
// boxBodyDef.fixedRotation = true;

m_boxl = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_boxl->CreateFixture(&
    boxFixtureDef);

// Create the cart on the right
b2PolygonShape boxShape2;
boxShape2.SetAsBox(a, b); // width and length of the
    box

b2FixtureDef boxFixtureDef2;
boxFixtureDef2.restitution = 0.75f;
boxFixtureDef2.density = 16.667f; // this will affect
    the box mass, mass = density*5.9969
boxFixtureDef2.friction = 0.0f;
boxFixtureDef2.shape = &boxShape2;

b2BodyDef boxBodyDef2;
boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(5.0f, L);
// boxBodyDef2.fixedRotation = true;

m_boxr = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
    boxFixtureDef2);

// Create the Pulley
b2PulleyJointDef pulleyDef;
b2Vec2 anchor1(-6.0f, 11.5f); // the position of the
    end string of the left cord to connect to the sack
b2Vec2 anchor2(4.0f, L); // the position of the end
    string of the right cord to connect to the right
    puller
b2Vec2 groundAnchor1(-1.0f, b + L ); // the string of
    the cord is tightened at (-1.0f, b+L)
b2Vec2 groundAnchor2(1.5f, b + L); // the string of
    the cord is tightened at (1.5f, b+L)
// the last float is the ratio
pulleyDef.Initialize(m_boxl, m_boxr, groundAnchor1,
    groundAnchor2, anchor1, anchor2, 1.0f);

```

```

        m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&
            pulleyDef);
    }
}

b2Body* m_boxl;
b2Body* m_boxr;
float F1 = 50.0f;
int theta = 30;
b2PulleyJoint* m_joint1;
void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_D:
            m_boxr->ApplyForceToCenter(b2Vec2(-13500.0f, 0.0f),
                true);
            break;
        case GLFW_KEY_F:
            m_boxr->SetLinearVelocity(b2Vec2(-14.0f, 0.0f));
            break;
    }
}
void Step(Settings& settings) override
{
    Test::Step(settings);
    b2MassData massData1 = m_boxl->GetMassData();
    b2Vec2 position1 = m_boxl->GetPosition();
    b2Vec2 velocity1 = m_boxl->GetLinearVelocity();
    b2MassData massData2 = m_boxr->GetMassData();
    b2Vec2 position2 = m_boxr->GetPosition();
    b2Vec2 velocity2 = m_boxr->GetLinearVelocity();
    float m1 = massData1.mass;
    float m2 = massData2.mass;
    float g = 9.8f;

    float ke1 = 0.5*m1*((velocity1.x)*(velocity1.x));
    float ke2 = 0.5*m2*((velocity2.x)*(velocity2.x));
    float T = 25.0f;

    g_debugDraw.DrawString(5, m_textLine, "Press D to push the
        cart to negative x axis with force of 13,500 N");
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Press F to push the
        cart to negative x axis with linear velocity of 14 m/s");
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Left Box position =
        (%4.1f, %4.1f)", position1.x, position1.y);
}

```

```

        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Left Box velocity = "
            "(%4.1f, %4.1f)", velocity1.x, velocity1.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Left Box Mass = %.6f",
            massData1.mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Left Box Kinetic
            Energy = %.6f", ke1);
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Right Cart position = "
            "(%4.1f, %4.1f)", position2.x, position2.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Right Cart velocity = "
            "(%4.1f, %4.1f)", velocity2.x, velocity2.y);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Right Cart Mass = %.6f
            ", massData2.mass);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Right Cart Kinetic
            Energy = %.6f", ke2);
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "The tension of the
            cord = %.6f", T);
        m_textLine += m_textIncrement;

        float ratio = m_joint1->GetRatio();
        float L = m_joint1->GetCurrentLengthA() + ratio * m_joint1->
            GetCurrentLengthB();
        g_debugDraw.DrawString(5, m_textLine, "L1 + %4.2f * L2 = %4.2
            f", (float) ratio, (float) L);
        m_textLine += m_textIncrement;

        printf("%4.2f %4.2f %4.2f %4.2f \n", position1.y, position2.x
            , ke1, ke2);
    }

    static Test* Create()
    {
        return new WorkGeneralvariable;
    }
};

static int testIndex = RegisterTest("Kinetic Energy and Work", "General
Variable Force", WorkGeneralvariable::Create);

```

C++ Code 53: tests/work_generalvariable.cpp "Work Done by a General Variable Box2D"

In this Box2D simulation, we set the friction coefficient for the box on the left to be 0.3 so when we push the cart to the left it will eventually stop, while the floor for the cart that is moving is frictionless. The force to push the cart is 13,500 N, and it stops after displacement of $\Delta x = 2 \text{ m}$, the value of x_1 and x_2 will still be the same because it represents the distance to the pulley, the pulley shall be at $x = 0$, even if the starting x position and ending position at the simulation is not at $x = 1$ or $x = 3$. With the displacement is the same of 2 m, the resulting kinetic energy will stay the same.

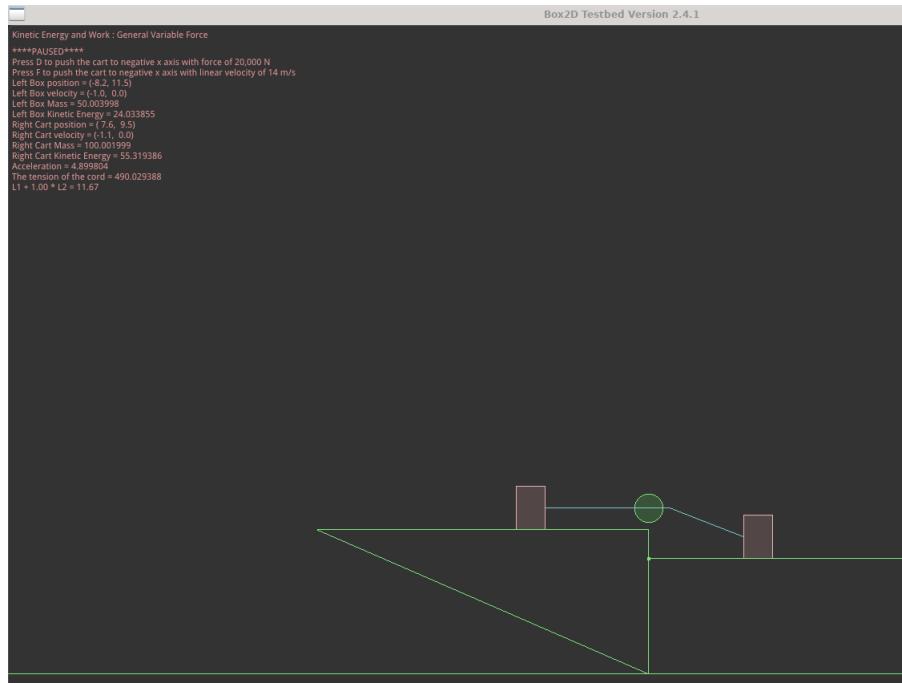


Figure 9.9: The cart on the right is being pushed with a force $F = 13,500 \text{ N}$ on the frictionless floor from $x_1 = 3.00 \text{ m}$ to $x_2 = 1.00 \text{ m}$ (the current simulation code can be located in: [DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/work_generalvariable.cpp](#)).

Some explanations for the codes:

-
-
-

XI. POWER

[DF*] The power due to a force is the rate at which that force does work on an object.

[DF*] If the force does work W during a time interval Δt , the average power due to the force over that time interval is

$$P_{avg} = \frac{W}{\Delta t} \quad (9.11)$$

[DF*] Instantaneous power is the instantaneous rate of doing work

$$P = \frac{dW}{dt} \quad (9.12)$$

[DF*] For a force \vec{F} at an angle ϕ to the direction of travel of the instantaneous velocity \vec{v} , the instantaneous power is

$$P = Fv \cos \phi = \vec{F} \cdot \vec{v} \quad (9.13)$$

[DF*] In the British system, the unit of power is the foot-pound per second. Often the horsepower is used. These are related by

$$1 \text{ watt} = 1W = 1 \text{ J/s} = 0.738 \text{ ft} \cdot \text{lb/s} \quad (9.14)$$

$$1 \text{ horsepower} = 1 \text{ hp} = 550 \text{ ft} \cdot \text{lb/s} = 746 \text{ W} \quad (9.15)$$

[DF*] Work can be expressed as power multiplied by time

$$\begin{aligned} 1 \text{ kilowatt-hour} &= 1kW \cdot h \\ &= (10^3 \text{ W})(3600 \text{ s}) \\ &= 3.60 \times 10^6 \text{ J} \\ &= 3.60 \text{ MJ} \end{aligned} \quad (9.16)$$

The watt and the kilowatt-hour have become identified as electrical units.

[DF*] Example:

- (a) At a certain instant, a particle-like object is acted on by a force $\vec{F} = (4)\hat{i} - (2)\hat{j} + (9)\hat{k}$ while the object's velocity is $\vec{v} = -(2)\hat{i} + (4)\hat{k}$. What is the instantaneous rate at which the force does work on the object?
- (b) At some other time, the velocity consists of only a y component. If the force is unchanged and the instantaneous power is -12 W , what is the velocity of the object?

Solution:

- (a) We obtain

$$\begin{aligned} P &= \vec{F} \cdot \vec{v} \\ &= (4)(-2) + (-2)(0) + (9)(4) \\ &= 28 \end{aligned}$$

the force does work on the object for 28 W

- (b) Focusing with a one-component velocity

$$\vec{v} = v\hat{j}$$

thus

$$\begin{aligned} P &= \vec{F} \cdot \vec{v} \\ -12 &= (-2)v \\ v &= 6 \end{aligned}$$

the velocity is 6 m/s.

[DF*] Example:

A force $\vec{F} = (3)\hat{i} + (7)\hat{j} + (7)\hat{k}$ acts on a 2 kg mobile object that moves from an initial position of $\vec{d}_i = (3)\hat{i} - (2)\hat{j} + (5)\hat{k}$ to a final position of $\vec{d}_f = -(5)\hat{i} + (4)\hat{j} + (7)\hat{k}$ in 4 seconds.

- (a) The work done on the object by the force in the 4 s interval.
- (b) The average power due to the force during that interval.
- (c) The angle between vectors \vec{d}_i and \vec{d}_f

Solution:

- (a) The object displacement is

$$\begin{aligned} \vec{d} &= \vec{d}_f - \vec{d}_i \\ &= (-8)\hat{i} + (6)\hat{j} + (2)\hat{k} \end{aligned}$$

Thus,

$$\begin{aligned} W &= \vec{F} \cdot \vec{d} \\ &= (3)(-8) + (7)(6) + (7)(2) \\ &= 32 \end{aligned}$$

- (b) The average power is

$$P_{avg} = \frac{W}{t} = \frac{32}{4} = 8$$

- (c) The distance from the coordinate origin to the initial position is

$$||\vec{d}_i|| = \sqrt{(3)^2 + (-2)^2 + (5)^2} = 6.16$$

and the magnitude of the distance from the coordinate origin to the final position is

$$||\vec{d}_f|| = \sqrt{(-5)^2 + (4)^2 + (7)^2} = 9.49$$

Their scalar (dot) product is

$$\begin{aligned} \vec{d}_i \cdot \vec{d}_f &= (3)(-5) + (-2)(4) + (5)(7) \\ &= 12 \end{aligned}$$

Thus, the angle between the two vectors is

$$\begin{aligned} \phi &= \cos^{-1} \left(\frac{\vec{d}_i \cdot \vec{d}_f}{||\vec{d}_i|| ||\vec{d}_f||} \right) \\ &= \cos^{-1} \left(\frac{12.0}{(6.16)(9.49)} \right) \\ &= 78.2^\circ \end{aligned}$$

[DF*] Example:

A funny car accelerates from rest through a measured track distance in time T with the engine operating at a constant power P . If the track crew can increase the engine power by a differential amount dP , what is the change in the time required for the run?

Solution:

According to the problem statement, the power of the car is

$$P = \frac{dW}{dt} = \frac{d}{dt} \left(\frac{1}{2} mv^2 \right) = mv \frac{dv}{dt}$$

with $mv \frac{dv}{dt}$ is a constant. The condition implies $dt = mv \frac{dv}{P}$, which can be integrated afterwards

$$\begin{aligned} P &= mv \frac{dv}{dt} \\ dt &= \frac{mv}{P} dv \\ \int_0^T dt &= \int_0^{v_T} \frac{mv}{P} dv \\ T &= \frac{mv_T^2}{2P} \end{aligned}$$

where v_T is the speed of the car at $t = T$. On the other hand, the total distance traveled can be written as

$$\begin{aligned} L &= \int_0^T v dt \\ &= \int_0^{v_T} v \frac{mv}{P} dv \\ &= \frac{m}{P} \int_0^{v_T} v^2 dv \\ &= \frac{mv_T^3}{3P} \end{aligned}$$

By squaring the expression for L and substituting the expression for T , we obtain

$$\begin{aligned} L^2 &= \left(\frac{mv_T^3}{3P} \right)^2 \\ &= \frac{8P}{9m} \left(\frac{mv_T^2}{2P} \right)^3 \\ &= \frac{8PT^3}{9m} \end{aligned}$$

which implies that

$$PT^3 = \frac{9}{8} mL^2$$

with $\frac{9}{8} mL^2$ is a constant. Differentiating the above equation gives

$$\begin{aligned} dPT^3 + 3PT^2 dt &= 0 \\ dt &= -\frac{T}{3P} dP \end{aligned}$$

XII. SIMULATION FOR POWER: MOVING ELEVATOR WITH Box2D

Taken from problem number 49 chapter 7 of [6].

A fully loaded, slow-moving freight elevator has a cab with a total mass of 1200 kg, which is required to travel upward 54 meters in 3 minutes, starting and ending at rest. The elevator's counterweight has a mass of only 950 kg, and so the elevator motor must help. What average power is required of the force the motor exerts on the cab via the cable?

Solution:

We have a loaded elevator moving upward at a constant speed. The forces involved are: gravitational force on the elevator, gravitational force on the counterweight, and the force by the motor via cable. The total work is the sum of the work done by gravity on the elevator, the work done by gravity on the counterweight, and the work done by the motor on the system:

$$W = W_e + W_c + W_m$$

Since the elevator moves at constant velocity, its kinetic energy does not change and according to the work-kinetic energy theorem the total work done is zero,

$$W = \Delta K = 0$$

The elevator moves upward through 54 m, so the work done by gravity on it is

$$\begin{aligned} W_e &= -m_e g d \\ &= -(1200)(9.8)(54) \\ &= -6.35 \times 10^5 \end{aligned}$$

The counterweight moves downward the same distance, so the work done by gravity on it is

$$\begin{aligned} W_c &= -m_c g d \\ &= -(950)(9.8)(54) \\ &= 5.03 \times 10^5 \end{aligned}$$

Since $W = 0$, the work done by the motor on the system is

$$\begin{aligned} W &= 0 \\ W_e + W_c + W_m &= 0 \\ W_m &= -W_e - W_c \\ &= 6.35 \times 10^5 - 5.03 \times 10^5 \\ &= 1.32 \times 10^5 \end{aligned}$$

The work is done in a time interval of $\Delta t = 3 \text{ min} = 180 \text{ s}$, so the power supplied by the motor to lift the elevator is

$$\begin{aligned} P &= \frac{W_m}{\Delta t} \\ &= \frac{1.32 \times 10^5}{180} \\ &= 7.4 \times 10^2 \end{aligned}$$

The power needed is 740 Watt.

```
#include "test.h"
#include "imgui/imgui.h"
class WorkPulleyelevator : public Test
{
public:
WorkPulleyelevator()
{
    float y = 16.0f;
    float z = -28.0f;
    float L = 12.0f;
    float a = 1.0f;
    float b = 2.0f;

    m_world->SetGravity(b2Vec2(0.0f,-9.8f));
    b2Body* ground = NULL;
    {
        b2BodyDef bd;
        ground = m_world->CreateBody(&bd);

        b2CircleShape circle;
        circle.m_radius = 2.0f;

        circle.m_p.Set(-10.0f, y + b + L);
        ground->CreateFixture(&circle, 0.0f);

        circle.m_p.Set(10.0f, y + b + L);
        ground->CreateFixture(&circle, 0.0f);
    }

    // Create the box hanging on the left
    b2PolygonShape boxShape1;
    boxShape1.SetAsBox(a, b); // width and length of the
                            // left box / left elevator

    b2FixtureDef boxFixtureDef1;
    boxFixtureDef1.restitution = 0.75f;
    boxFixtureDef1.density = 950.0f/8.0f; // this will
                                         // affect the left elevator mass
    boxFixtureDef1.friction = 0.3f;
    boxFixtureDef1.shape = &boxShape1;

    b2BodyDef boxBodyDef1;
    boxBodyDef1.type = b2_dynamicBody;
    boxBodyDef1.position.Set(-10.0f, z);
    // boxBodyDef2.fixedRotation = true;
}
```

```

m_boxl = m_world->CreateBody(&boxBodyDef1);
b2Fixture *boxFixture1 = m_boxl->CreateFixture(&
    boxFixtureDef1);

// Create the box hanging on the right
b2PolygonShape boxShape2;
boxShape2.SetAsBox(a, b); // width and length of the
    right box / right elevator

b2FixtureDef boxFixtureDef2;
boxFixtureDef2.restitution = 0.75f;
boxFixtureDef2.density = 150.0f; // this will affect
    the right elevator mass
boxFixtureDef2.friction = 0.3f;
boxFixtureDef2.shape = &boxShape2;

b2BodyDef boxBodyDef2;
boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(10.0f, z); // the distance
    traveled by the right elevator is 54 meter
// boxBodyDef2.fixedRotation = true;

m_boxr = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
    boxFixtureDef2);
m_boxr->SetLinearVelocity(b2Vec2(0.0f, 5.0f)); //
    Apply the right elevator motor to set linear
    velocity upward

b2PulleyJointDef pulleyDef;
b2Vec2 anchor1(-10.0f, z + b);
b2Vec2 anchor2(10.0f, z + b );
b2Vec2 groundAnchor1(-10.0f, y + b + L);
b2Vec2 groundAnchor2(10.0f, y + b + L);
pulleyDef.Initialize(m_boxl, m_boxr, groundAnchor1,
    groundAnchor2, anchor1, anchor2, 1.5f);

m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&
    pulleyDef);

m_time = 0.0f;
}

}

b2Body* m_boxl;
b2Body* m_boxr;
float m_vel1;
float m_vel2;

```

```

        float l_mass;
        float r_mass;
        float m_time;
        bool m_fixed_camera;

    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_C:
                m_fixed_camera = !m_fixed_camera;
                if(m_fixed_camera)
                {
                    g_camera.m_center = b2Vec2(2.0f, 10.0f);
                    g_camera.m_zoom = 3.3f; // zoom out camera
                }
                break;
        }
    }
    void UpdateUI() override
    {
        ImGui::SetNextWindowPos(ImVec2(10.0f, 200.0f));
        ImGui::SetNextWindowSize(ImVec2(420.0f, 150.0f));
        ImGui::Begin("Elevator Controls", nullptr,
                    ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

        if (ImGui::SliderFloat("Left Elevator velocity", &m_vel1,
                               -50.0f, 100.0f, "%.\0f"))
        {
            m_boxl->SetLinearVelocity(b2Vec2(0.0f, m_vel1));
        }
        if (ImGui::SliderFloat("Right Elevator velocity", &m_vel2,
                               -50.0f, 100.0f, "%.\0f"))
        {
            m_boxr->SetLinearVelocity(b2Vec2(0.0f, m_vel2));
        }
        if (ImGui::SliderFloat("Left Elevator mass", &l_mass, 0.0f,
                               5000.0f, "%.\0f"))
        {
            b2MassData boxMassData1;
            boxMassData1.mass = l_mass;
            boxMassData1.I = 30.0f;
            m_boxl->SetMassData(&boxMassData1);
        }
        if (ImGui::SliderFloat("Right Elevator mass", &r_mass, 0.0f,
                               5000.0f, "%.\0f"))
        {
            b2MassData boxMassData2;

```

```

        boxMassData2.mass = r_mass;
        boxMassData2.I = 30.0f;
        m_boxr->SetMassData(&boxMassData2);
    }
    ImGui::End();
}
void Step(Settings& settings) override
{
    Test::Step(settings);
    m_time += 1.0f / 60.0f; // assuming we are using frequency of
                           // 60 Hertz
    float m = m_boxl->GetMass(); // same result as GetMassData()

    b2MassData massData1 = m_boxl->GetMassData();
    b2MassData massData2 = m_boxr->GetMassData();
    b2Vec2 position1 = m_boxl->GetPosition();
    b2Vec2 velocity1 = m_boxl->GetLinearVelocity();
    b2Vec2 position2 = m_boxr->GetPosition();
    b2Vec2 velocity2 = m_boxr->GetLinearVelocity();
    float ratio = m_joint1->GetRatio();
    float L = m_joint1->GetCurrentLengthA() + ratio * m_joint1->
        GetCurrentLengthB();

    g_debugDraw.DrawString(5, m_textLine, "Press C = Camera fixed
        /tracking");
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
        f", m_time);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Left Elevator Mass =
        %4.2f", massData1.mass);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Left Elevator Mass
        Center = %4.2f", massData1.center);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Left Elevator Inertia
        = %4.2f", massData1.I);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Left Elevator position
        = (%4.1f, %4.1f)", position1.x, position1.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Left Elevator velocity
        = (%4.1f, %4.1f)", velocity1.x, velocity1.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Right Elevator Mass =
        %4.2f", massData2.mass);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Right Elevator

```

```

        position = (%4.1f, %4.1f)", position2.x, position2.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Right Elevator
velocity = (%4.1f, %4.1f)", velocity2.x, velocity2.y);
m_textLine += m_textIncrement;

g_debugDraw.DrawString(5, m_textLine, "L1 + %4.2f * L2 = %4.2
f", (float) ratio, (float) L);
m_textLine += m_textIncrement;
if(!m_fixed_camera)
{
    g_camera.m_center = m_boxr->GetPosition();
    g_camera.m_zoom = 3.3f; // zoom out camera
}
}

static Test* Create()
{
    return new WorkPulleylelevator;
}

b2PulleyJoint* m_joint1;
};

static int testIndex = RegisterTest("Kinetic Energy and Work", "Freight
Elevator", WorkPulleylelevator::Create);

```

C++ Code 54: tests/work_pulleylelevator.cpp "Work Pulley Elevator Box2D"

Some explanations for the codes:

- This is the elevator controls GUI. We can change the left and right elevator' velocity and mass to see their dynamic interaction. We put **SetMassData(&boxMassData1)** because we need to use **const b2MassData**, when you read b2Body Class reference in the Public Member functions for **SetMassData** you will have to follow the syntax rule, it is more or less like creating a fixture for a body.

```

void UpdateUI() override
{
    ImGui::SetNextWindowPos(ImVec2(10.0f, 200.0f));
    ImGui::SetNextWindowSize(ImVec2(420.0f, 150.0f));
    ImGui::Begin("Elevator Controls", nullptr,
                ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);
    ;

    if (ImGui::SliderFloat("Left Elevator velocity", &m_vel1
                           , -50.0f, 100.0f, "%0f"))
    {
        m_boxl->SetLinearVelocity(b2Vec2(0.0f, m_vel1));
    }
}

```

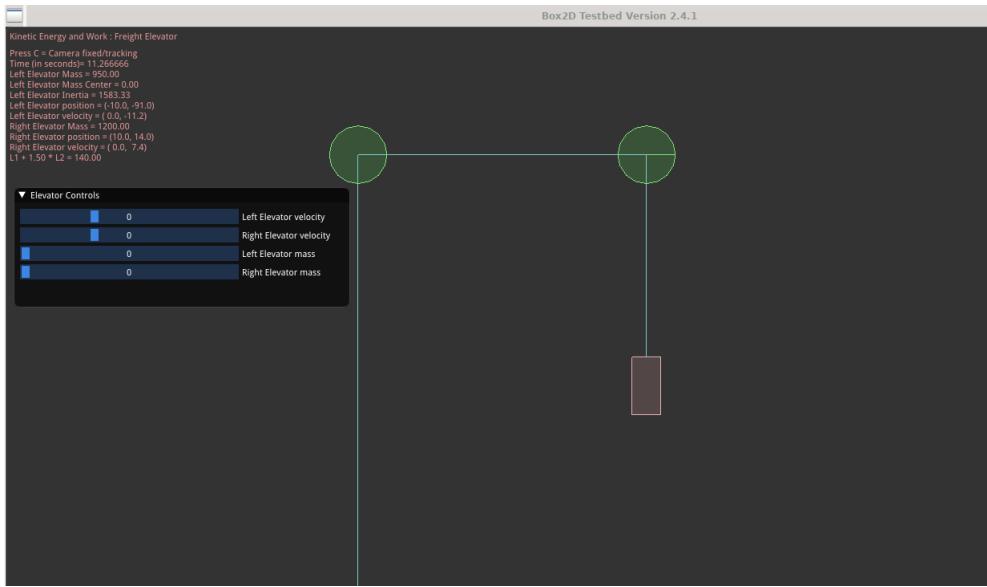


Figure 9.10: The simulation of a freight elevator on the right with mass of 1200 kg travel upward 54 meters (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/work_pulleylelevator.cpp`).

```

if (ImGui::SliderFloat("Right Elevator velocity", &
    m_vel2, -50.0f, 100.0f, "%0f"))
{
    m_boxr->SetLinearVelocity(b2Vec2(0.0f, m_vel2));
}
if (ImGui::SliderFloat("Left Elevator mass", &l_mass,
    0.0f, 5000.0f, "%0f"))
{
    b2MassData boxMassData1;
    boxMassData1.mass = l_mass;
    boxMassData1.I = 30.0f;
    m_boxl->SetMassData(&boxMassData1);
}
if (ImGui::SliderFloat("Right Elevator mass", &r_mass,
    0.0f, 5000.0f, "%0f"))
{
    b2MassData boxMassData2;
    boxMassData2.mass = r_mass;
    boxMassData2.I = 30.0f;
    m_boxr->SetMassData(&boxMassData2);
}
//ImGui::SliderFloat("Mass", &l_mass, 0.0f, 360.0f, "%0
f");
ImGui::End();
}

```

```
b2Body Class Reference

A rigid body. These are created via b2World::CreateBody. More...
#include <b2_body.h>

Public Member Functions

    b2Fixture * CreateFixture (const b2FixtureDef *def)
    b2Fixture * CreateFixture (const b2Shape *shape, float density)
    void DestroyFixture (b2Fixture *fixture)
    void SetTransform (const b2Vec2 &position, float angle)
const b2Transform & GetTransform () const
const b2Vec2 & GetPosition () const
float GetAngle () const
const b2Vec2 & GetWorldCenter () const
Get the world position of the center of mass.
const b2Vec2 & GetLocalCenter () const
Get the local position of the center of mass.
void SetLinearVelocity (const b2Vec2 &v)
const b2Vec2 & GetLinearVelocity () const
void SetAngularVelocity (float omega)
float GetAngularVelocity () const
void ApplyForce (const b2Vec2 &force, const b2Vec2 &point, bool wake)
void ApplyForceToCenter (const b2Vec2 &force, bool wake)
void ApplyTorque (float torque, bool wake)
void ApplyLinearImpulse (const b2Vec2 &impulse, const b2Vec2 &point, bool wake)
void ApplyLinearImpulseToCenter (const b2Vec2 &impulse, bool wake)
```

Figure 9.11: The `b2Body` class reference with its public member functions, you can learn what you can modify and get data from `b2Body`.

From the official Box2D source code you can see the mass data from `../box2d/include/box2d/b2_shape.h`, they are in the form of struct, the mass data are: **mass**, **center**, **I**.

```
b2Body Class Reference

A rigid body. These are created via b2World::CreateBody. More...
#include <b2_body.h>

Public Member Functions

    b2Fixture * CreateFixture (const b2FixtureDef *def)
    b2Fixture * CreateFixture (const b2Shape *shape, float density)
    void DestroyFixture (b2Fixture *fixture)
    void SetTransform (const b2Vec2 &position, float angle)
const b2Transform & GetTransform () const
const b2Vec2 & GetPosition () const
float GetAngle () const
const b2Vec2 & GetWorldCenter () const
Get the world position of the center of mass.
const b2Vec2 & GetLocalCenter () const
Get the local position of the center of mass.
void SetLinearVelocity (const b2Vec2 &v)
const b2Vec2 & GetLinearVelocity () const
void SetAngularVelocity (float omega)
float GetAngularVelocity () const
void ApplyForce (const b2Vec2 &force, const b2Vec2 &point, bool wake)
void ApplyForceToCenter (const b2Vec2 &force, bool wake)
void ApplyTorque (float torque, bool wake)
void ApplyLinearImpulse (const b2Vec2 &impulse, const b2Vec2 &point, bool wake)
void ApplyLinearImpulseToCenter (const b2Vec2 &impulse, bool wake)
```

Figure 9.12: The `SetMassData` public member functions has the same properties as `CreateFixture`, thus the syntax are more or less quite the same.

- It is quite hard to make the elevator goes up in 180 seconds since by default (without addition of linear velocity) from Box2D simulation the right elevator will travel upward 54 meters and arrive at the top in 13 seconds, with linear velocity of 5 m/s toward positive y axis the time to travel upward 54 meters is around 10 seconds. Thus we are going to use the simulation with linear mapping, so we can determine the force the motor exerts in real world condition from the simulation. By setting

```
m_boxr = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_boxr->CreateFixture(&boxFixtureDef2
);
m_boxr->SetLinearVelocity(b2Vec2(0.0f, 5.0f)); // Apply the
right elevator motor to set linear velocity upward
```

We create a motor that exerts a power to push the elevator with linear velocity of 5 m/s toward positive y axis, hence it will make

$$t_{real} \approx 18t_{simulation}$$

and from equation (7.1) we have

$$v = v_0 + at$$

since the relation between v and t is linear, thus

$$v_{real} \approx 18v_{simulation}$$

with this we know the required power of the force the motor exerts in real life.

XIII. SIMULATION FOR POWER: PENDULUM CRATE WITH Box2D

Taken from problem number 57' chapter 7 from [6].

A 230 kg crate hangs from the end of a rope of length $L = 12.0 \text{ m}$. You push horizontally on the crate with a varying force \vec{F} to move it distance $d = 4.00 \text{ m}$ to the side.

- (a) What is the magnitude of \vec{F} when the crate is in this final position?
- (b) During the crate's displacement, what is the total work done on it?
- (c) During the crate's displacement, what is the work done by the gravitational force on the crate?
- (d) During the crate's displacement, what is the work done by the pull on the crate from the rope?
- (e) Knowing that the crate is motionless before and after its displacement, use the answers to (b), (c) and (d) to find the work your force \vec{F} does on the crate.
- (f) Why is the work of your force not equal to the product of the horizontal displacement and the answer to (a)?

Solution:

- (a) To hold the crate at equilibrium in the final situation, \vec{F} must have the same magnitude as the horizontal component of the rope's tension $T \sin \theta$, where θ is the angle between the rope (in the final position) and vertical:

$$\theta = \sin^{-1} \left(\frac{d}{L} \right) = \sin^{-1} \left(\frac{4.00}{12.0} \right) = 19.5^\circ$$

But the vertical component of the tension supports against the weight:

$$T \cos \theta = mg$$

Thus,

$$\begin{aligned} T &= \frac{mg}{\cos \theta} \\ &= \frac{(230)(9.8)}{\cos 19.5^\circ} \\ &= 2390.728 \\ &\approx 2391 \end{aligned}$$

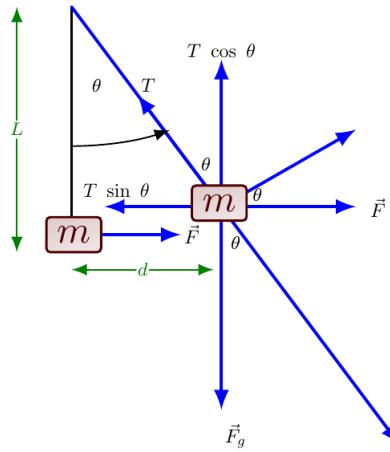


Figure 9.13: The body diagram and the illustration of the crate hangs from the end of a rope of length $L = 12\text{ m}$, then pushed horizontally with a force \vec{F} to move it distance d .

and

$$F = T \sin \theta = (2391) \sin 19.5^\circ = 797$$

the tension is 2391 N and the force is 797 N.

(b) Since there is no change in kinetic energy, the net work on it is zero.

(c) The work done by gravity is

$$W_g = \vec{F}_g \cdot \vec{d} = -mgh$$

The ratio of the vertical displacement toward the rope length is

$$\frac{h}{L} = 1 - \cos \theta$$

when the angle $\theta = 90^\circ$ the vertical displacement will be the same as the rope length, L . Thus, for this problem the vertical component of the displacement is $h = L(1 - \cos \theta)$. With $L = 12\text{ m}$, we will obtain

$$W_g = -mgh = -(230)(9.8)(12(1 - \cos 19.5^\circ)) = -1547$$

the work done by the gravitational force on the crate is 1547 J.

(d) The tension vector is everywhere perpendicular to the direction of motion, so its work is zero (since $\cos 90^\circ = 0$).

(e) The implication of the previous three parts is that the work due to \vec{F} is $-W_g$ (so the net work turns out to be zero). Thus,

$$W_F = -W_g = 1547$$

(f) Since \vec{F} does not have constant magnitude, we cannot expect equation

$$W = \vec{F} \cdot \vec{d}$$

to apply, since that equation is a work done by a constant force.

```

#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#include "test.h"
#include <fstream>

class CratePendulum : public Test
{
public:
    CratePendulum()
    {
        b2Body* b1;
        {
            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
                (40.0f, 0.0f));

            b2BodyDef bd;
            b1 = m_world->CreateBody(&bd);
            b1->CreateFixture(&shape, 0.0f);
        }
        // the two blocks below for creating boxes are only
        // for sweetener.
        {
            b2PolygonShape shape;
            shape.SetAsBox(7.0f, 0.25f, b2Vec2_zero, 0.7f);
            // Gradient is 0.7

            b2BodyDef bd;
            bd.position.Set(6.0f, 6.0f);
            b2Body* ground = m_world->CreateBody(&bd);
            ground->CreateFixture(&shape, 0.0f);
        }
        {
            b2PolygonShape shape;
            shape.SetAsBox(7.0f, 0.25f, b2Vec2_zero, -0.7f)
                ; // Gradient is -0.7

            b2BodyDef bd;
            bd.position.Set(-6.0f, 6.0f);
            b2Body* ground = m_world->CreateBody(&bd);
            ground->CreateFixture(&shape, 0.0f);
        }

        b2Body* b2; // the hanging bar for the pendulum
    }
}

```

```

        b2PolygonShape shape;
        shape.SetAsBox(7.25f, 0.25f);

        b2BodyDef bd;
        bd.position.Set(0.0f, 37.0f);
        b2 = m_world->CreateBody(&bd);
        b2->CreateFixture(&shape, 0.0f);
    }

    b2RevoluteJointDef jd;
    b2Vec2 anchor;

    // Create the pendulum ball
    b2PolygonShape boxShape1;
    boxShape1.SetAsBox(1.0f, 0.5f);

    b2FixtureDef boxFixtureDef1;
    boxFixtureDef1.restitution = 0.75f;
    boxFixtureDef1.density = 115.0f; // this will affect
        the left elevator mass
    boxFixtureDef1.friction = 0.3f;
    boxFixtureDef1.shape = &boxShape1;

    b2BodyDef boxBodyDef1;
    boxBodyDef1.type = b2_dynamicBody;
    boxBodyDef1.position.Set(0.0f, 24.0f);
    // boxBodyDef2.fixedRotation = true;
    // boxBodyDef1.angularDamping = 0.2f;

    m_box = m_world->CreateBody(&boxBodyDef1);
    b2Fixture *boxFixture1 = m_box->CreateFixture(&
        boxFixtureDef1);

    // Create the anchor and connect it to the crate
    anchor.Set(0.0f, 36.0f); // x and y axis position for
        the Pendulum anchor
    jd.Initialize(b2, m_box, anchor);
    //jd.collideConnected = true;
    m_world->CreateJoint(&jd); // Create the Pendulum
        anchor

    m_time = 0.0f;
}
b2Body* m_box;
float m_time;

void Keyboard(int key) override
{

```

```

        switch (key)
    {
        case GLFW_KEY_S:
            m_box->SetTransform(b2Vec2(4.0f, 24.0f), 0); // warp or teleport it to move with displacement of 4
            break;
        case GLFW_KEY_T:
            m_time = 0.0f;
            break;
    }
}

void Step(Settings& settings) override
{
    m_time += 1.0f / 60.0f; // assuming we are using frequency of 60 Hertz
    b2Vec2 v = m_box->GetLinearVelocity();
    float omega = m_box->GetAngularVelocity();
    float angle = m_box->GetAngle();
    b2MassData massData = m_box->GetMassData();
    b2Vec2 position = m_box->GetPosition();
    float m = massData.mass;
    float g = 9.8f;
    float L = 12.0f;
    float d = 4.0f;
    float theta = asin(d/L);
    float T = (m*g)/(cos(theta));
    float F = T*sin(theta);
    float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f * massData.I * omega * omega;

    g_debugDraw.DrawString(5, m_textLine, "Press S to apply force 10,000 N to positive x axis");
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Press T to reset time");
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6f", m_time);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Ball position, x,y =(%4.1f, %4.1f)", position.x, position.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Mass = %4.1f", massData.mass);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Theta = %4.2f", theta*RADTODEG);
}

```

```

        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Rope tension =
            %4.2f", T);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Pushing Force =
            %4.2f", F);
        m_textLine += m_textIncrement;

        g_debugDraw.DrawString(5, m_textLine, "Kinetic energy
            = %4.1f", ke);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Linear velocity
            = %4.1f", v);
        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Angle (in
            degrees) = %4.1f", angle*RADTODEG);
        m_textLine += m_textIncrement;
        // Print the result in every time step then plot it
        // into graph with either gnuplot or anything

        printf("%4.2f %4.2f %4.2f\n", position.x, position.y,
            angle*RADTODEG);
        //std::ofstream MyFile("/root/output.txt"); ;
        //MyFile << angle << " " << position.x << " " <<
        position.y;

        Test::Step(settings);
    }

    static Test* Create()
    {
        return new CratePendulum;
    }

};

static int testIndex = RegisterTest("Kinetic Energy and Work", "
    Crate Pendulum", CratePendulum::Create);

```

C++ Code 55: *tests/work_cratependulum.cpp* "Crate Pendulum Box2D"

From the Box2D simulation I got $\theta = 18.35^\circ$ for making displacement of $d = 4$ for the crate, it is probably because I set the $y = 24$ for that displacement.

Some explanations for the codes:

- We are using **SetTransform** to make the displacement, there is a bug when we click "S" the crate does not move, but when we click the crate it will move normally like pendulum crate.

```

void Keyboard(int key) override
{

```

```

switch (key)
{
    case GLFW_KEY_S:
        m_box->SetTransform(b2Vec2(4.0f, 24.0f), 0); // 
            warp or teleport it to move with displacement
            of 4
        break;
    case GLFW_KEY_T:
        m_time = 0.0f;
        break;
}
}

```

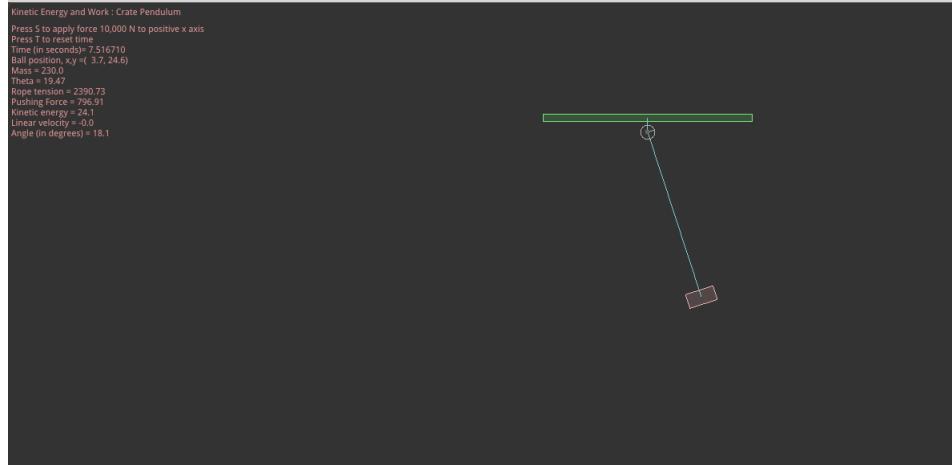


Figure 9.14: The simulation a crate being pushed with force \vec{F} to move a distance of $d = 4$ m (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/work_cratependulum.cpp`).

XIV. SIMULATION FOR POWER: VERTICAL FACING UPWARD SPRING WITH Box2D

A 250 g block is dropped onto a relaxed vertical spring that has a spring constant of $k = 2.5 \text{ N/cm}$. The block becomes attached to the spring and compresses the spring 12 cm before momentarily stopping. While the spring is being compressed, what work is done on the block by

- (a) The gravitational force on it
- (b) The spring force
- (c) What is the speed of the block just before it hits the spring? (Assume that friction is negligible)
- (d) If the speed at impact is doubled, what is the maximum compression of the spring?

Solution:

- (a) The compression of the spring is $d = 0.12 \text{ m}$. The work done by the force of gravity (acting on the block) is,

$$\begin{aligned} W_1 &= mgd \\ &= (0.25)(9.8)(0.12) \\ &= 0.29 \end{aligned}$$

the work done by the force of gravity is 0.29 J.

- (b) For the spring constant we will convert it into MKS system, $k = 2.5 \text{ N/cm} = 250 \text{ N/m}$. Thus, the work done by the spring is,

$$\begin{aligned} W_2 &= -\frac{1}{2}kd^2 \\ &= -\frac{1}{2}(250)(0.12)^2 \\ &= -1.8 \end{aligned}$$

- (c) The speed v_i of the block just before it hits the spring is found from the work-kinetic energy theorem:

$$\begin{aligned} \Delta K &= W_1 + W_2 \\ 0 - \frac{1}{2}mv_i^2 &= W_1 + W_2 \\ v_i &= \sqrt{\frac{(-2)(W_1 + W_2)}{m}} \\ &= \sqrt{\frac{(-2)(0.29 - 1.8)}{0.25}} \\ &= 3.5 \end{aligned}$$

the speed $v_i = 3.5 \text{ m/s}$.

- (d) If we instead had $v'_i = 7 \text{ m/s}$, we reverse the above steps and solve for d' . Recalling the theorem used in part (c), we have

$$\begin{aligned} \Delta K' &= W'_1 + W'_2 \\ 0 - \frac{1}{2}mv'_i^2 &= W'_1 + W'_2 \\ -\frac{1}{2}mv_i'^2 &= mgd' - \frac{1}{2}kd'^2 \\ -\frac{1}{2}kd'^2 + mgd' + \frac{1}{2}mv_i'^2 &= 0 \\ d' &= \pm \frac{mg + \sqrt{m^2g^2 + mkv_i'^2}}{k} \end{aligned}$$

now by choosing the positive root we will have

$$d' = \frac{mg + \sqrt{m^2g^2 + mkv_i'^2}}{k} = \frac{(0.25)(9.8) + \sqrt{(0.25)^2(9.8)^2 + (0.25)(250)(7)^2}}{250} = 0.23$$

the maximum compression of the spring for speed of $v'_i = 7 \text{ m/s}$ is $d' = 0.23 \text{ m}$.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class WorkVerticalspringupward : public Test
{
public:
    WorkVerticalspringupward()
    {
        b2Body* ground = NULL;
        m_world->SetGravity(b2Vec2(0.0f, -9.8f));
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                0.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        // Create the wall surrounding the spring
        b2BodyDef bd2;
        bd2.type = b2_staticBody;
        bd2.angularDamping = 0.1f;

        bd2.position.Set(-0.05f, 5.5f);
        b2Body* leftwall = m_world->CreateBody(&bd2);

        b2PolygonShape shape2;
        shape2.SetAsBox(0.5f, 5.5f);
        leftwall->CreateFixture(&shape2, 5.0f);

        b2BodyDef bd3;
        bd3.type = b2_staticBody;
        bd3.angularDamping = 0.1f;

        bd3.position.Set(2.05f, 5.5f);
        b2Body* rightwall = m_world->CreateBody(&bd3);

        b2PolygonShape shape3;
        shape3.SetAsBox(0.5f, 5.5f);
        rightwall->CreateFixture(&shape3, 5.0f);

        // Create a static body as the box for the spring
        b2BodyDef bd1;
```

```

bd1.type = b2_staticBody;
bd1.angularDamping = 0.1f;

bd1.position.Set(1.0f, 0.5f);
b2Body* springbox = m_world->CreateBody(&bd1);

b2PolygonShape shape1;
shape1.SetAsBox(0.5f, 0.5f);
springbox->CreateFixture(&shape1, 5.0f);

// Create the box as the movable object
b2PolygonShape boxShape;
boxShape.SetAsBox(0.5f, 0.1f);

b2FixtureDef boxFixtureDef;
boxFixtureDef.restitution = 0.0f;
boxFixtureDef.density = 1.5f; // this will affect the spring
mass
boxFixtureDef.friction = 0.1f;
boxFixtureDef.shape = &boxShape;

b2BodyDef boxBodyDef;
boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(1.0f, 11.0f); // the box will be
located in (1,11.0)

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
;
//m_box->SetGravityScale(-7); // negative means it will goes
upward, positive it will goes downward
// Make a distance joint for the box / ball with the static
box above
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f),
boxBodyDef.position);
jd.collideConnected = true; // In this case we decide to
allow the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f; // the relaxed length of the spring:
m_minLength
m_maxLength = 12.0f;
b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
m_dampingRatio, jd.bodyA, jd.bodyB);

```

```

m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
m_joint->SetMinLength(m_minLength);
m_joint->SetMaxLength(m_maxLength);

// Create the falling block as the movable object
b2PolygonShape blockShape;
blockShape.SetAsBox(0.5f, 0.5f);

b2FixtureDef blockFixtureDef;
blockFixtureDef.restitution = 0.0f; // bounciness
blockFixtureDef.density = 0.25f; // this will affect the
                               falling block mass
blockFixtureDef.friction = 0.1f;
blockFixtureDef.shape = &blockShape;

b2BodyDef blockBodyDef;
blockBodyDef.type = b2_dynamicBody;
blockBodyDef.position.Set(1.0f, 12.0f);

m_block = m_world->CreateBody(&blockBodyDef);
b2Fixture *blockFixture = m_block->CreateFixture(&
                                                 blockFixtureDef);

m_time = 0.0f;
}

b2Body* m_box;
b2Body* m_block;
b2DistanceJoint* m_joint;
float m_length;
float m_time;
float m_minLength;
float m_maxLength;
float m_hertz;
float m_dampingRatio;

void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_W:
            m_box->ApplyForceToCenter(b2Vec2(0.0f, 10000.0f), true
                                      );
            break;
        case GLFW_KEY_S:
            m_box->ApplyForceToCenter(b2Vec2(0.0f, -9500.0f),
                                      true);
            break;
        case GLFW_KEY_T:
    }
}

```

```

        m_time = 0.0f;
        break;
    }
}

void UpdateUI() override
{
    ImGui::SetNextWindowPos(ImVec2(10.0f, 200.0f));
    ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
    ImGui::Begin("Joint Controls", nullptr,
        ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

    if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0f"))
    {
        m_length = m_joint->SetLength(m_length);
    }

    if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"))
    {
        float stiffness;
        float damping;
        b2LinearStiffness(stiffness, damping, m_hertz,
            m_dampingRatio, m_joint->GetBodyA(), m_joint->
            GetBodyB());
        m_joint->SetStiffness(stiffness);
        m_joint->SetDamping(damping);
    }

    if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
        , 2.0f, "%.1f"))
    {
        float stiffness;
        float damping;
        b2LinearStiffness(stiffness, damping, m_hertz,
            m_dampingRatio, m_joint->GetBodyA(), m_joint->
            GetBodyB());
        m_joint->SetStiffness(stiffness);
        m_joint->SetDamping(damping);
    }

    ImGui::End();
}

void Step(Settings& settings) override
{
    b2MassData massData = m_box->GetMassData();
    b2MassData massData2 = m_block->GetMassData();
    b2Vec2 position = m_box->GetPosition();
}

```

```

b2Vec2 velocity = m_box->GetLinearVelocity();
b2Vec2 velocity2 = m_block->GetLinearVelocity();
m_time += 1.0f / 60.0f; // assuming we are using frequency of
// 60 Hertz
float m = massData.mass;
float g = 9.8f;
float y = 11.0f;
// y = the position at which when we place the mass it would
// not move / equilibrium position
// y = y position of the ceiling - m_minLength - initial y
// position of the mass
float k = 250.0f;
float y_eq = 11.0f;

g_debugDraw.DrawString(5, m_textLine, "Press W to apply force
    10,000 N upward / S to apply force 9,500 N downward");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Press T to reset time"
);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
    f", m_time);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Falling Block Mass =
    %4.2f", massData2.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Falling Block velocity
    = (%4.1f, %4.1f)", velocity2.x, velocity2.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Spring position =
    (%4.1f, %4.1f)", position.x, position.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Spring velocity =
    (%4.1f, %4.1f)", velocity.x, velocity.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Spring Mass = %4.2f",
    massData.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "The spring constant, k
    = %4.1f", k);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Equilibrium position
    for the spring, y = %4.1f", y_eq);
m_textLine += m_textIncrement;
// Print the result in every time step then plot it into
// graph with either gnuplot or anything

printf("%4.2f %4.2f\n", velocity2.y, position.y);

```

```

        Test::Step(settings);
    }
    static Test* Create()
    {
        return new WorkVerticalspringupward;
    }

};

static int testIndex = RegisterTest("Kinetic Energy and Work", "Vertical
Spring Upward", WorkVerticalspringupward::Create);

```

C++ Code 56: *tests/work_verticalspringupward.cpp* "Block Dropped onto Relaxed Spring Box2D"

Some explanations for the codes:

- The important part of this simulation is to set the height where we drop the block onto the spring since it will affect the speed before it hits the spring. With mass of 250 g, the height to drop the block is very near with the spring, not too high.

```

// Create the falling block as the movable object
b2PolygonShape blockShape;
blockShape.SetAsBox(0.5f, 0.5f);

b2FixtureDef blockFixtureDef;
blockFixtureDef.restitution = 0.0f; // bounciness
blockFixtureDef.density = 0.25f; // this will affect the
                               falling block mass
blockFixtureDef.friction = 0.1f;
blockFixtureDef.shape = &blockShape;

b2BodyDef blockBodyDef;
blockBodyDef.type = b2_dynamicBody;
blockBodyDef.position.Set(1.0f, 12.0f);

m_block = m_world->CreateBody(&blockBodyDef);
b2Fixture *blockFixture = m_block->CreateFixture(&
                                                 blockFixtureDef);

```

```

-----  

-1.88 10.86  

-1.96 10.84  

-2.12 10.82  

-2.29 10.80  

-2.45 10.78  

-2.61 10.76  

-2.78 10.74  

-2.94 10.72  

-3.10 10.70  

-3.27 10.68  

-3.43 10.67  

-3.59 10.65  

-3.76 10.64  

-0.70 10.63  

-0.60 10.62  

-0.50 10.61  

-0.40 10.60  

-0.29 10.60  

-0.19 10.60  

-0.09 10.59

```

Figure 9.15: The velocity of the falling block is around 3.5 m/s just before it hits the spring, the negative sign means it is going toward negative y axis.

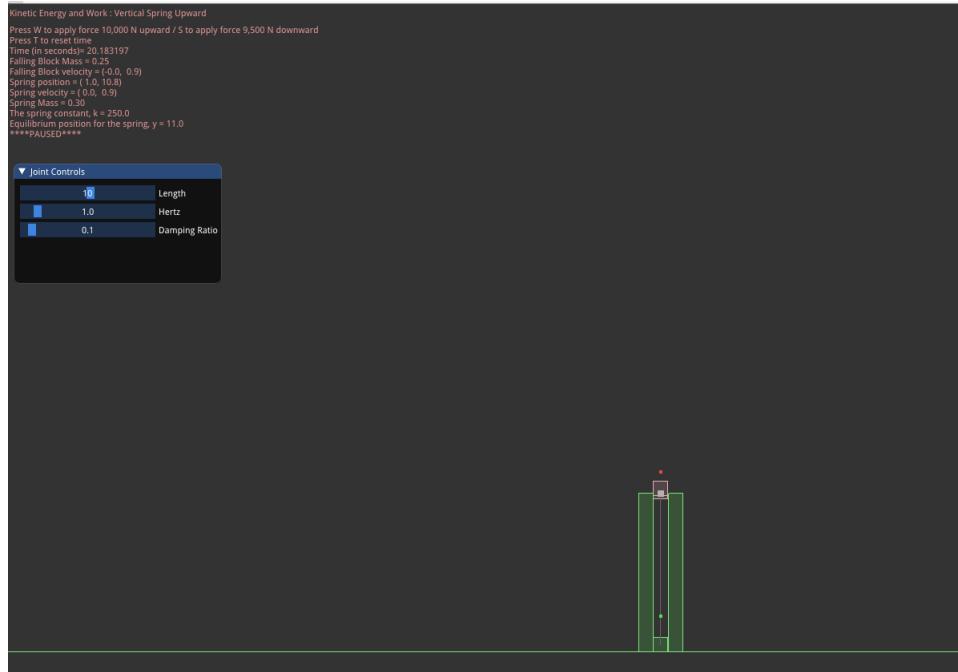


Figure 9.16: The simulation of a 250 g block dropped onto a relaxed vertical spring that has a spring constant $k = 250 \text{ N/m}$ (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/work_verticalspringupward.cpp`).

Chapter 10

DFSimulatorC++ IV: Potential Energy and Conservation of Energy

"Je vais et je vient, entre tes reins.." - Je t'aime Moi Non Plus song

M^{omentum}

I. SIMULATION FOR WITH Box2D

C++ Code 57: *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- ---
- ---
- ---

Chapter 11

DFSimulatorC++ V: Center of Mass and Linear Momentum

"Yes bla, thank you bla.." - 1 week old Browni' puppies when I bring them to Puncak Bintang then put them in small gazebo to be breastfeeded by Browni

M^{omentum}

I. SIMULATION FOR MOMENTUM WITH Box2D

You need to copy from my repository' directory `../Source Codes/C++/DianFreya-box2d-testbed`, then go inside the directory and open the terminal then type:

```
mkdir build  
cd build  
cmake ..  
make  
./testbed
```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

C++ Code 58: `tests/projectile_motion.cpp` "Projectile Motion Box2D"

Some explanations for the codes:

- ---

- ---

Chapter 12

DFSimulatorC++ VI: Rotation

*"Sudo, oh honey honey.. You are my Cambridge girl.. Honey.. oh Sudo Sudo.. you are my pretty girl" -
Sugar song changed to Sudo Sudo by Mischkra and DS Glanzsche*

M^{omentum}

I. SIMULATION FOR MOMENTUM WITH Box2D

You need to copy from my repository' directory `../Source Codes/C++/DianFreya-box2d-testbed`, then go inside the directory and open the terminal then type:

```
mkdir build  
cd build  
cmake ..  
make  
./testbed
```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

C++ Code 59: `tests/projectile_motion.cpp` "Projectile Motion Box2D"

Some explanations for the codes:

- ---
- ---
- ---

Chapter 13

DFSimulatorC++ VII: Rolling, Torque, and Angular Momentum

"Reality flows from cause to effect like a mathematical equation. Mortals can't comprehend this. They're pathetic" - Albert Silverberg

M^{omentum}

I. SIMULATION FOR MOMENTUM WITH Box2D

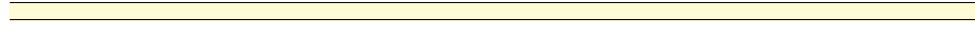
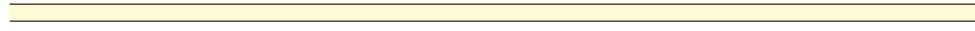
You need to copy from my repository' directory `../Source Codes/C++/DianFreya-box2d-testbed`, then go inside the directory and open the terminal then type:

```
mkdir build  
cd build  
cmake ..  
make  
./testbed
```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

C++ Code 60: `tests/projectile_motion.cpp` "Projectile Motion Box2D"

Some explanations for the codes:

- 
- 
- 

Chapter 14

DFSimulatorC++ VIII: Equilibrium and Elasticity

"Tantum Ergo Sacromentum, veneremur cernui, et antiquum documentum, novo cedat ritui" - Pange Lingua

M^{omentum}

I. SIMULATION FOR MOMENTUM WITH Box2D

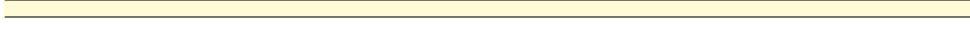
You need to copy from my repository' directory `../Source Codes/C++/DianFreya-box2d-testbed`, then go inside the directory and open the terminal then type:

```
mkdir build  
cd build  
cmake ..  
make  
./testbed
```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

C++ Code 61: `tests/projectile_motion.cpp` "Projectile Motion Box2D"

Some explanations for the codes:

- 
- 
- 

Chapter 15

DFSimulatorC++ IX: Gravity

"Nous allons voir" - DS Glanzsche

We are bounded by gravity, we walk, do activity, run, eat, all affected by gravity, that's why we step on this land of this planet earth. That's why rocket is designed that way with specific machine in order to go against gravity and able to get out of this planet. It should be understood, not only with theory in Physics formula and Mathematics differential equation, but why things fall that way, and for how long till it reach the ground from certain height?

This first simulation is asked by the Elder at Valhalla Projection. I thought I was asked to do some push and pulling box simulation first. But this comes first, let see then. Our favorite phrase in French, "Nous allons voir."

I. MATHEMATICAL PHYSICS FORMULA FOR GRAVITY

We are going to remember how Newton' formula changes our lives, remember that:

$$F = ma$$

means that force that is applied to a body is proportional with the mass and acceleration that is ongoing with the body. Now for an object that is falling we will have:

$$F = mg$$

The different is that we regard a as the acceleration towards horizontal axis, while g is the acceleration towards the vertical axis.

Based on elementary differential equation [1], we will have:

$$\begin{aligned} F &= mg \\ m \frac{dv}{dt} &= mg \\ (\text{without air drag}) \quad F &= mg - \gamma v \\ m \frac{dv}{dt} &= mg - \gamma v(t) \end{aligned}$$

with v is the speed or velocity of the object that is falling, a variable that depends on t , time variable, thus we will write $v(t)$ instead of just v . We need to solve the differential equation above to obtain:

$$v(t), x(t), T$$

with $v(t)$ is the solution that represents the velocity of the falling object at time t , and $x(t)$ is the distance that the objects fall at time t , and time T is the time required for the object to fall for certain height, say h meters. We need to solve this so then we can input the numerical computation into C++ code so the animation or simulation of the falling object due to gravity with (or without) air drag is realistic.

Now, since computer is really helpful for computation in symbolic and numerical terms, I choose to use JULIA to solve the differential equation above. Thus,

$$\begin{aligned} m \frac{dv}{dt} &= mg - \gamma v(t) \\ \frac{dv}{dt} &= g - \frac{\gamma v(t)}{m} \\ v(t) &= \frac{mg}{\gamma} - \frac{mge^{-\frac{\gamma t}{m}}}{\gamma} \end{aligned}$$

Now if we substitute the value for $m = 10$, an object with mass that is 10 kg, $g = 9.8 \text{ m/s}^2$, which is the gravity on earth, and the air drag coefficient is $\gamma = 0.3$, we will have:

$$v(t) = 326.667 - 326.667e^{-0.03t}$$

If we alter the input parameters, such as different mass for the object or on different planet with different gravity, we will obtain different numerical solution. Now what we need to input to C++ code is this:

$$v(t) = \frac{mg}{\gamma} - \frac{mge^{-\frac{\gamma t}{m}}}{\gamma}$$

Then we define m, g, γ , and eventually h , the height from which the object falls, later on. There should be impact as well if the speed is too fast and the height is too low, like meteor hitting on this planet on Dinosaur era, but we will talk about it later on.

```
julia> include("diffeq.jl")
The differential equation:
d
m·—(v(t)) = g·m - γ·v(t)
dt
Initial Value problem solution:
-γ·t
m
v(t) = ————— - —————·e
γ γ
The solution for the differential equation with certain parameter inputs:
-0.03·t
v(t) = 326.666666666667 - 326.666666666667·e
-
```

Figure 15.1: The symbolical computation for finding the solution of an object falling with JULIA and the SymPy package (ch5-1-gravitydiffeq.jl).

It is very important and necessary, as we are having a lot of air flights and air sports currently, knowing how to jump with parachute from an airplane and estimate time to land with certain speed can be of help one day.

We are going to compare the gravity / falling bodies simulation between Bullet physics engine and ReactPhysics3D.

II. SIMULATION FOR GRAVITY WITH BULLET3, GLEW, GLFW AND OPENGL

Since this is the very first simulation asked by the elder, by coincidence a book has a nice scope on this with Bullet3, thus after modifying I am going to show how to simulate 3 dynamic bodies with shape of a sphere and different masses fall from the same height. Inspired by Chapter 7 from [10], I add two other bodies. You can directly copy the files for this example or type it manually.

We can compile all examples from Bullet, but I prefer to create one like this (per project based), then compile, compiling all examples for Bullet took so long, since it already expanded. With only using the necessary library for our test simulation purpose, we won't need to compile all other examples, that in my opinion is more time saving and can help to focus more on current physics problem and its' simulation.

In a new directory, create a file by opening a terminal and type:
vim main.cpp

```
// GLEW needs to be included first
#include <GL/glew.h>

// GLFW is included next
#include <GLFW/glfw3.h>
#include "ShaderLoader.h"
#include "Camera.h"
#include "LightRenderer.h"

#include "MeshRenderer.h"
#include "TextureLoader.h"
#include <btBulletDynamicsCommon.h>
#include<chrono>

void initGame();
void renderScene();

Camera* camera;
LightRenderer* light;

MeshRenderer* sphere;
MeshRenderer* sphere2;
MeshRenderer* sphere3;
MeshRenderer* ground;

//physics
```

```

btDiscreteDynamicsWorld* dynamicsWorld;

void initGame() {

    // Enable the depth testing
    glEnable(GL_DEPTH_TEST);

    ShaderLoader shader;

    GLuint flatShaderProgram = shader.createProgram("/root/
        SourceCodes/CPP/Assets/Shaders/FlatModel.vs", "/root/
        SourceCodes/CPP/Assets/Shaders/FlatModel.fs");
    GLuint texturedShaderProgram = shader.createProgram("/root/
        SourceCodes/CPP/Assets/Shaders/TexturedModel.vs", "/root/
        SourceCodes/CPP/Assets/Shaders/TexturedModel.fs");

    camera = new Camera(45.0f, 800, 600, 0.1f, 100.0f, glm::vec3
        (0.0f, 8.0f, 30.0f)); //camera position at y=+8 and z=+30

    light = new LightRenderer(MeshType::kTriangle, camera);
    light->setProgram(flatShaderProgram);
    light->setPosition(glm::vec3(0.0f, 3.0f, 0.0f));

    TextureLoader tLoader;

    GLuint sphereTexture = tLoader.getTextureID("/root/
        SourceCodes/CPP/Assets/Textures/globe.jpg");
    GLuint groundTexture = tLoader.getTextureID("/root/
        SourceCodes/CPP/Assets/Textures/ground.jpg");

    //init physics
    btBroadphaseInterface* broadphase = new btDbvtBroadphase();
    btDefaultCollisionConfiguration* collisionConfiguration = new
        btDefaultCollisionConfiguration();
    btCollisionDispatcher* dispatcher = new btCollisionDispatcher
        (collisionConfiguration);
    btSequentialImpulseConstraintSolver* solver = new
        btSequentialImpulseConstraintSolver();

    dynamicsWorld = new btDiscreteDynamicsWorld(dispatcher,
        broadphase, solver, collisionConfiguration);
    dynamicsWorld->setGravity(btVector3(0, -0.8f, 0));

    // Sphere Rigid Body
    btCollisionShape* sphereShape = new btSphereShape(1); //
        sphere with radius 1
    btCollisionShape* sphereShape2 = new btSphereShape(1);
    btCollisionShape* sphereShape3 = new btSphereShape(1);

```

```

// Set the initial location where the spheres fall
btDefaultMotionState* sphereMotionState = new
    btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1)
        , btVector3(-3, 13, 0)));
btDefaultMotionState* sphereMotionState2 = new
    btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1)
        , btVector3(0, 13, 0)));
btDefaultMotionState* sphereMotionState3 = new
    btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1)
        , btVector3(3, 13, 0)));

btScalar mass = 5.0;
btScalar mass2 = 10.0;
btScalar mass3 = 15.0;
btVector3 sphereInertia(0, 0, 0);
btVector3 sphereInertia2(0, 0, 0);
btVector3 sphereInertia3(0, 0, 0);
sphereShape->calculateLocalInertia(mass, sphereInertia);
sphereShape2->calculateLocalInertia(mass2, sphereInertia2);
sphereShape3->calculateLocalInertia(mass3, sphereInertia3);

btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI(
    mass, sphereMotionState, sphereShape, sphereInertia);
btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI2(
    mass2, sphereMotionState2, sphereShape2, sphereInertia2);
btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI3(
    mass3, sphereMotionState3, sphereShape3, sphereInertia3);

btRigidBody* sphereRigidBody = new btRigidBody(
    sphereRigidBodyCI);
btRigidBody* sphereRigidBody2 = new btRigidBody(
    sphereRigidBodyCI2);
btRigidBody* sphereRigidBody3 = new btRigidBody(
    sphereRigidBodyCI3);
sphereRigidBody->setRestitution(1.0f);
sphereRigidBody->setFriction(1.0f);
sphereRigidBody2->setRestitution(1.0f);
sphereRigidBody2->setFriction(1.0f);
sphereRigidBody3->setRestitution(1.0f);
sphereRigidBody3->setFriction(1.0f);

dynamicsWorld->addRigidBody(sphereRigidBody);
dynamicsWorld->addRigidBody(sphereRigidBody2);
dynamicsWorld->addRigidBody(sphereRigidBody3);

// Sphere 1 Mesh
sphere = new MeshRenderer(MeshType::kSphere, camera,

```

```

        sphereRigidBody);
sphere->setProgram(texturedShaderProgram);
sphere->setTexture(sphereTexture);
sphere->setScale(glm::vec3(1.0f));

//Sphere 2 Mesh
sphere2 = new MeshRenderer(MeshType::kSphere, camera,
    sphereRigidBody2);
sphere2->setProgram(texturedShaderProgram);
sphere2->setTexture(sphereTexture);
sphere2->setScale(glm::vec3(1.0f));

// Sphere 3 Mesh
sphere3 = new MeshRenderer(MeshType::kSphere, camera,
    sphereRigidBody3);
sphere3->setProgram(texturedShaderProgram);
sphere3->setTexture(sphereTexture);
sphere3->setScale(glm::vec3(1.0f));

// Ground Rigid body
btCollisionShape* groundShape = new btBoxShape(btVector3(4.0f
    , 0.5f, 4.0f));
btDefaultMotionState* groundMotionState = new
    btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1)
    , btVector3(0, -2.0f, 0)));
    btRigidBody::btRigidBodyConstructionInfo groundRigidBodyCI
    (0.0f, new btDefaultMotionState(), groundShape, btVector3
    (0, 0, 0));
    btRigidBody* groundRigidBody = new btRigidBody(
        groundRigidBodyCI);

groundRigidBody->setFriction(1.0);
groundRigidBody->setRestitution(0.5);

groundRigidBody->setCollisionFlags(btCollisionObject::
    CF_STATIC_OBJECT);

dynamicsWorld->addRigidBody(groundRigidBody);

// Ground Mesh
ground = new MeshRenderer(MeshType::kCube, camera,
    groundRigidBody);
ground->setProgram(texturedShaderProgram);
ground->setTexture(groundTexture);
ground->setScale(glm::vec3(4.0f, 0.5f, 4.0f));
}

```

```

void renderScene(){
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(1.0, 1.0, 0.0, 1.0);

    //light->draw();
    sphere->draw();
    sphere2->draw();
    sphere3->draw();
    ground->draw();
}

int main(int argc, char **argv)
{
    glfwInit();
    GLFWwindow* window = glfwCreateWindow(800, 600, " Bullet and
        OpenGL ", NULL, NULL);
    glfwMakeContextCurrent(window);
    glewInit();

    initGame();
    auto previousTime = std::chrono::high_resolution_clock::now()
        ;
    while (!glfwWindowShouldClose(window)){

        auto currentTime = std::chrono::high_resolution_clock
            ::now();
        float dt = std::chrono::duration<float, std::chrono::
            seconds::period>(currentTime - previousTime).
            count();

        dynamicsWorld->stepSimulation(dt);

        renderScene();

        glfwSwapBuffers(window);
        glfwPollEvents();

        previousTime = currentTime;
    }
    glfwTerminate();

    delete camera;
    delete light;

    return 0;
}

```

C++ Code 62: *main.cpp "Gravity with 3 Bodies"*

Do not follow 100 %, read slowly and try to debug and tinker by yourself, if there are warnings and errors find the solutions by yourself if possible, otherwise try to ask. Another thing is do kindly adjust the path, for example at these lines:

```
GLuint flatShaderProgram = shader.createProgram("/root/SourceCodes/CPP/
    Assets/Shaders/FlatModel.vs", "/root/SourceCodes/CPP/Assets/Shaders/
    FlatModel.fs");
GLuint texturedShaderProgram = shader.createProgram("/root/SourceCodes/CPP/
    Assets/Shaders/TexturedModel.vs", "/root/SourceCodes/CPP/Assets/Shaders
    /TexturedModel.fs");
```

Your path shall not be the same as mine: `/root/SourceCodes/CPP/Assets/Shaders/FlatModel.fs` for **FlatModel.fs**, a fragment shader. All the shader and texture and images related are inside repository as well, you can suit yourself if you like it use that or use your own.

Some explanations for the codes:

- To add physics we use:
`#include <btBulletDynamicsCommon.h>`
- The Bullet libraries we are using are: **BulletCollision**, **BulletDynamics**, **LinearMath**. We will need to link to this library when compiling either with CMake or manual compiling.
- To create a Physics world we use this:
`btDiscreteDynamicsWorld* dynamicsWorld;`
- After create the Physics world, then inside the **void initGame()** we will have:

```
btBroadphaseInterface* broadphase = new btDbvtBroadphase();
btDefaultCollisionConfiguration* collisionConfiguration = new
    btDefaultCollisionConfiguration();
btCollisionDispatcher* dispatcher = new btCollisionDispatcher(
    collisionConfiguration);
btSequentialImpulseConstraintSolver* solver = new
    btSequentialImpulseConstraintSolver();
```

Collision detection is done in two phase: broadphase (the physics engine eliminates all the objects that are unlikely to collide) and narrowphase (the actual shape of the object is used to check the likelihood of a collision). Pairs of object are created with a strong likelihood of collision. For **btCollisionDispatcher**, a pair of objects that have a strong likelihood for colliding are tested for collision using their actual shapes. For **btSequentialImpulseConstraintSolver**, you can create constraints, such as a hinge constraint or slider constraint which can restrict motion or rotation of one object about another object. The calculation is repeated a number of times to get the optimal solution.

- Still in **void initGame()**:

```
dynamicsWorld = new btDiscreteDynamicsWorld(dispatcher,
    broadphase, solver, collisionConfiguration);
dynamicsWorld->setGravity(btVector3(0, -9.8f, 0));
```

to create a new **dynamicsWorld** by passing the **dispatcher**, **broadphase**, **solver**, and **collisionConfiguration** as parameters to the **btDiscreteDynamicsWorld** function. After that we can set the parameters for our physics with gravity of $-9.8f$ in the y -axis.

- After finish with the Physics world, we can create rigid bodies or soft bodies, still in **void initGame()**:

```

btCollisionShape* sphereShape = new btSphereShape(1);

btDefaultMotionState* sphereMotionState = new
    btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1),
        btVector3(0, 15, 0)));

btScalar mass = 10.0;
btVector3 sphereInertia(0, 0, 0);
sphereShape->calculateLocalInertia(mass, sphereInertia);

btRigidBody::btRigidBodyConstructionInfo sphereRigidBodyCI(mass
    , sphereMotionState, sphereShape, sphereInertia);

btRigidBody* sphereRigidBody = new btRigidBody(
    sphereRigidBodyCI);
sphereRigidBody->setRestitution(1.0f);
sphereRigidBody->setFriction(1.0f);

dynamicsWorld->addRigidBody(sphereRigidBody);

```

new btSphereShape(1); means create a sphere shape with radius of 1. The **btDefaultMotionState(btTransform(btQuaternion(0, 0, 0, 1), btVector3(0, 15, 0)))**; specifies the rotation and position of the sphere, it will be located on $x = 0, y = 15, z = 0$. To test it you may change the **btQuaternion(0, 0, 0, 1)** into **btQuaternion(0.3, 0, 0.5, 0.7)** to see how the sphere rotates while falling. After that, we set the mass to be 10, the inertia, and calculate inertia of the **sphereShape**.

Then, to create the rigid body, we first create **btRigidBodyConstructionInfo** and pass the rigid body' variables / parameters. We then set physical properties of the rigid body, such as the restitution and the friction. For restitution 0, means the rigid body will have no bounciness, while 1 means the rigid body is very bouncy, like a basketball. While for friction, 0 means a smooth rigid body, while 1 means a very rough rigid body.

Finally, we add the rigid body to the Physics world.

- In **Mesh.cpp**, we define the vertices for Triangle, Quad, Cube, and Sphere. Thus in **MeshRenderer.h** and **MeshRenderer.cpp** we are not only render the sphere, but to make it behave like a sphere that obeying the law of physics, we pass the rigid body to the sphere mesh.

We use **btTransform** variable to get the transformation from the rigid body's **getMotionState** function and then get the **WorldTransform** variable and set it to our **btTransform** variable **t**.

We create **btQuaternion** type to store rotation and **btvector3** to store translation values using the **getRotation** and **getOrigin** functions of the **btTransform** class.

We create three **glm::mat4** variables, called **RotationMatrix**, **TranslationMatrix**, and **ScaleMatrix**. We set the values of rotation and translation using the **glm::rotate** and **glm::translation**

functions.

The rest of the files (.cpp and .h) can be obtained at the repository, all will be (don't forget the texture, vertex and fragment shaders):

- Camera.cpp
- Camera.h
- CMakeLists.txt
- LightRenderer.cpp
- LightRenderer.h
- main.cpp
- Mesh.cpp
- Mesh.h
- MeshRenderer.cpp
- MeshRenderer.h
- ShaderLoader.cpp
- ShaderLoader.h
- TextureLoader.cpp
- TextureLoader.h

After having all of them then open the terminal at this working directory and type:

```
mkdir build  
cd build  
cmake ..  
make  
./result
```

If you want to compile manually without CMake, type:

```
g++ *.cpp -o result -lBulletCollision -lBulletDynamics -lLinearMath -lGlew -lglfw -lGL -  
I/usr/include/stb-master  
./result
```

This example has no air drag. If we want an air drag we need to damp the velocity when the bodies are falling. Air drag depends on the geometry shape and material of the object. On October 26th, 2023 I drop down from the same height a small dog doll and 1 Euro cent coin, they fall at the same time. But, when I do that with the Euro coin and a piece of paper, the paper still flying when the coin hits the ground. That's Physics, why that happens? Back to the geometry shape and material of the paper.

Calculate the time it hits the ground. Is mass not affecting time to hit the ground? Check the formula, etc.

III. SIMULATION FOR GRAVITY WITH Box2D

i. Build DianFreya Modified Box2D Testbed

We have modified all the codes for Box2D testbed by removing the original tests codes into our own codes to learn Physics and Math from beginning, with Box2D library and imgui, learning C++, Physics and Math become more fun. You are welcome to copy from my repository' directory `../Source Codes/C++/DianFreya-box2d-testbed`, then go inside the directory and open the terminal then type:

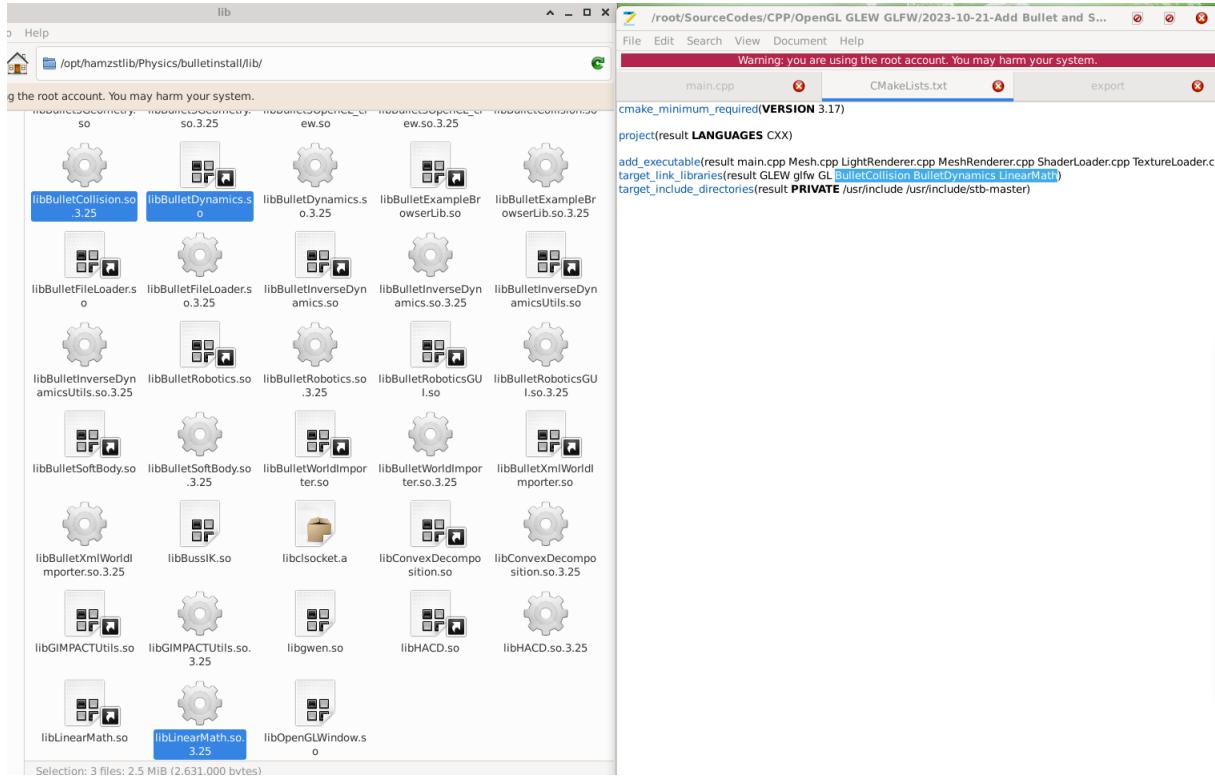


Figure 15.2: Linking the libraries from Bullet for this simulation, make sure that you have set the right LIBRARY_PATH for the Bullet' libraries that were installed and compiled.

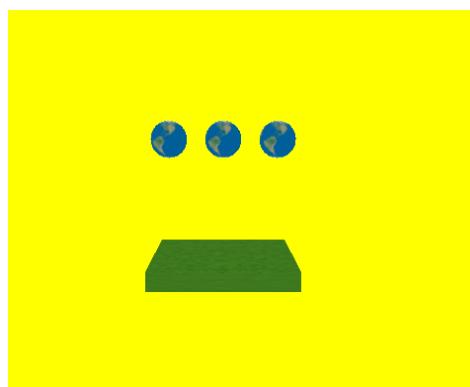


Figure 15.3: Three falling bodies with shape of sphere, from left to right have masses of: 5, 10, then 15. (files for this example in folder: DFSimulatorC/Source Codes/C++/DianFreya-Bullet/Gravity and 3 Bodies).

```
mkdir build
cd build
cmake ..
make
./testbed
```

If you are having a difficulty or error, read again chapter 5 on how to download, compile and install Box2D from raw to become library, along with copying the headers of imgui and sajson. If you read again, besides the source codes inside `../tests/` that I replaced, the `CMakeLists.txt` is being modified to include the source codes from the newly modified source codes in `../tests/`, thus when the testbed is being build it will only show the new tests for this book simulation, not the original default Box2D testbed tests.

Look for the related simulation under the **Tests** tab on the right panel, then choose **Gravitation/Gravity Check**.

```
#include "test.h"
#include <iostream>

class GravityCheck: public Test
{
public:

    GravityCheck()
    {
        m_world->SetGravity(b2Vec2(0.0f, -9.8f));
        b2Timer timer;
        // Perimeter Ground body
        {
            b2BodyDef bd;
            b2Body* ground = m_world->CreateBody(&bd);

            b2EdgeShape shapeGround;
            shapeGround.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
                (30.0f, 0.0f));
            ground->CreateFixture(&shapeGround, 0.0f);

            b2EdgeShape shapeTop;
            shapeTop.SetTwoSided(b2Vec2(-40.0f, 30.0f), b2Vec2
                (30.0f, 30.0f));
            ground->CreateFixture(&shapeTop, 0.0f);

            b2EdgeShape shapeLeft;
            shapeLeft.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
                (-40.0f, 30.0f));
            ground->CreateFixture(&shapeLeft, 0.0f);

            b2EdgeShape shapeRight;
            shapeRight.SetTwoSided(b2Vec2(30.0f, 0.0f), b2Vec2
```

```

        (30.0f, 30.0f));
ground->CreateFixture(&shapeRight, 0.0f);
}

// Create the left ball
b2CircleShape ballShape2;
ballShape2.m_p.SetZero();
ballShape2.m_radius = 0.5f;

b2FixtureDef ballFixtureDef2;
ballFixtureDef2.restitution = 0.75f; // the bounciness
ballFixtureDef2.density = 7.3f; // this will affect the ball
mass
ballFixtureDef2.friction = 0.1f;
ballFixtureDef2.shape = &ballShape2;

b2BodyDef ballBodyDef2;
ballBodyDef2.type = b2_dynamicBody;
ballBodyDef2.position.Set(-10.0f, 25.0f);
// ballBodyDef2.angularDamping = 0.2f;

m_ball2 = m_world->CreateBody(&ballBodyDef2);
b2Fixture *ballFixture2 = m_ball2->CreateFixture(&
ballFixtureDef2);

// Create the right ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.75f; // the bounciness
ballFixtureDef.density = 3.3f; // this will affect the ball
mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(0.0f, 25.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&
ballFixtureDef);
m_createTime = timer.GetMilliseconds();
}

b2Body* m_ball;

```

```

b2Body* m_ball2;
void Step(Settings& settings) override
{
    b2MassData massData2 = m_ball2->GetMassData();
    g_debugDraw.DrawString(5, m_textLine, "Left Ball Mass = %.6f"
        , massData2.mass);
    m_textLine += m_textIncrement;

    b2Vec2 position2 = m_ball2->GetPosition();
    g_debugDraw.DrawString(5, m_textLine, "Left Ball Position, x"
        = %.6f", position2.x);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Left Ball Position, y"
        = %.6f", position2.y);
    m_textLine += m_textIncrement;

    b2MassData massData = m_ball->GetMassData();
    g_debugDraw.DrawString(5, m_textLine, "Right Ball Mass = %.6f"
        , massData.mass);
    m_textLine += m_textIncrement;

    b2Vec2 position = m_ball->GetPosition();
    g_debugDraw.DrawString(5, m_textLine, "Right Ball Position, x"
        = %.6f", position.x);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Right Ball Position, y"
        = %.6f", position.y);
    m_textLine += m_textIncrement;

    g_debugDraw.DrawString(5, m_textLine, "create time = %6.2f ms"
        ,
        m_createTime);
    m_textLine += m_textIncrement;

    printf("%4.2f %4.2f \n", position2.y, position.y);

    Test::Step(settings);
}

static Test* Create()
{
    return new GravityCheck;

}

float m_createTime;
};

```

```
static int testIndex = RegisterTest("Gravitation", "Gravity Check",
    GravityCheck::Create);
```

C++ Code 63: *tests/gravity_check.cpp* "Gravity Check Box2D"

Some explanations for the codes:

- To create the green perimeter:

```
// Perimeter Ground body
{
    b2BodyDef bd;
    b2Body* ground = m_world->CreateBody(&bd);

    b2EdgeShape shapeGround;
    shapeGround.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
        (30.0f, 0.0f));
    ground->CreateFixture(&shapeGround, 0.0f);

    b2EdgeShape shapeTop;
    shapeTop.SetTwoSided(b2Vec2(-40.0f, 30.0f), b2Vec2(30.0f
        , 30.0f));
    ground->CreateFixture(&shapeTop, 0.0f);

    b2EdgeShape shapeLeft;
    shapeLeft.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2
        (-40.0f, 30.0f));
    ground->CreateFixture(&shapeLeft, 0.0f);

    b2EdgeShape shapeRight;
    shapeRight.SetTwoSided(b2Vec2(30.0f, 0.0f), b2Vec2(30.0f
        , 30.0f));
    ground->CreateFixture(&shapeRight, 0.0f);
}
```

- Create two balls objects inside **GravityCheck()**

```
// Create the left ball
b2CircleShape ballShape2;
ballShape2.m_p.SetZero();
ballShape2.m_radius = 0.5f;

b2FixtureDef ballFixtureDef2;
ballFixtureDef2.restitution = 0.75f; // the bounciness
ballFixtureDef2.density = 7.3f; // this will affect the ball
mass
ballFixtureDef2.friction = 0.1f;
ballFixtureDef2.shape = &ballShape2;

b2BodyDef ballBodyDef2;
ballBodyDef2.type = b2_dynamicBody;
```

```

ballBodyDef2.position.Set(-10.0f, 25.0f);
// ballBodyDef2.angularDamping = 0.2f;

m_ball2 = m_world->CreateBody(&ballBodyDef2);
b2Fixture *ballFixture2 = m_ball2->CreateFixture(&
    ballFixtureDef2);

// Create the right ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.75f; // the bounciness
ballFixtureDef.density = 3.3f; // this will affect the ball
mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(0.0f, 25.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
);
m_createTime = timer.GetMilliseconds();

```

- Declare each of the ball as Box2D body.

```

b2Body* m_ball;
b2Body* m_ball2;

```

- To show on the top left of the screen for the x and y position of each ball (left and right), along with its mass. The y position for each ball will be printed out on the terminal / xterm

```

void Step(Settings& settings) override
{
    b2MassData massData2 = m_ball2->GetMassData();
    g_debugDraw.DrawString(5, m_textLine, "Left Ball Mass =
        %.6f", massData2.mass);
    m_textLine += m_textIncrement;

    b2Vec2 position2 = m_ball2->GetPosition();
    g_debugDraw.DrawString(5, m_textLine, "Left Ball
        Position, x = %.6f", position2.x);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Left Ball
        Position, y = %.6f", position2.y);
}

```

```

    m_textLine += m_textIncrement;

    b2MassData massData = m_ball->GetMassData();
    g_debugDraw.DrawString(5, m_textLine, "Right Ball Mass =
        %.6f", massData.mass);
    m_textLine += m_textIncrement;

    b2Vec2 position = m_ball->GetPosition();
    g_debugDraw.DrawString(5, m_textLine, "Right Ball
        Position, x = %.6f", position.x);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Right Ball
        Position, y = %.6f", position.y);
    m_textLine += m_textIncrement;

    g_debugDraw.DrawString(5, m_textLine, "create time =
        %6.2f ms",
    m_createTime);
    m_textLine += m_textIncrement;

    printf("%4.2f %4.2f \n", position2.y, position.y);

    Test::Step(settings);
}

```

Now, if you want to save the output into textfile (.txt), you can type this before running testbed from terminal:

./testbed > ballgravity.txt

it will save the textfile named **ballgravity.txt** inside the directory containing the **testbed**, watch out that the first two lines usually contain strings, for my case it is the data of my OpenGL, GLSL and Mesa version, just delete that strings and leave the two columns of numerical data only.

Now, we can use **Gnuplot** to help plot the *y* position data (the height) of each ball with respect to time, now open terminal at the directory containing the **ballgravity.txt** and type:

```

gnuplot
set xlabel "time"
set ylabel "y position"
plot "ballgravity.txt" using 1 title "Ball mass 5.73" with linespoints, "ballgravity.txt" using 2 title
"Ball mass 2.59" with linespoints

```

IV. SIMULATION FOR GRAVITY WITH REACTPHYSICS3D

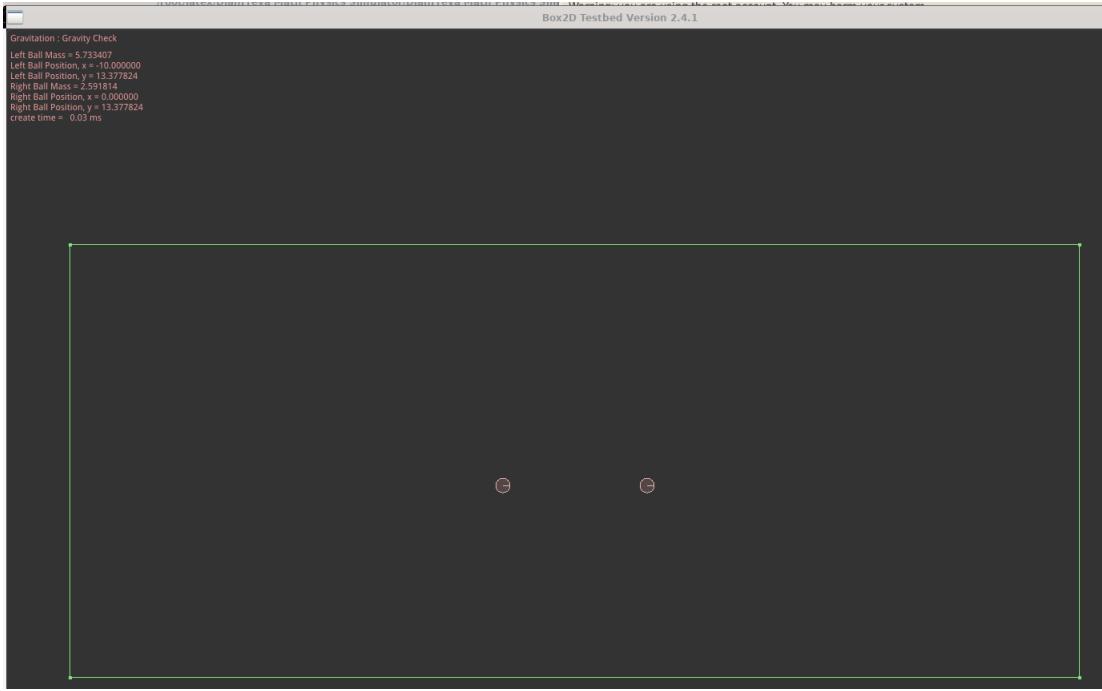


Figure 15.4: The falling object due to gravity simulation with Box2D, it can be played when you build the testbed. (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/gravity_check.cpp`).

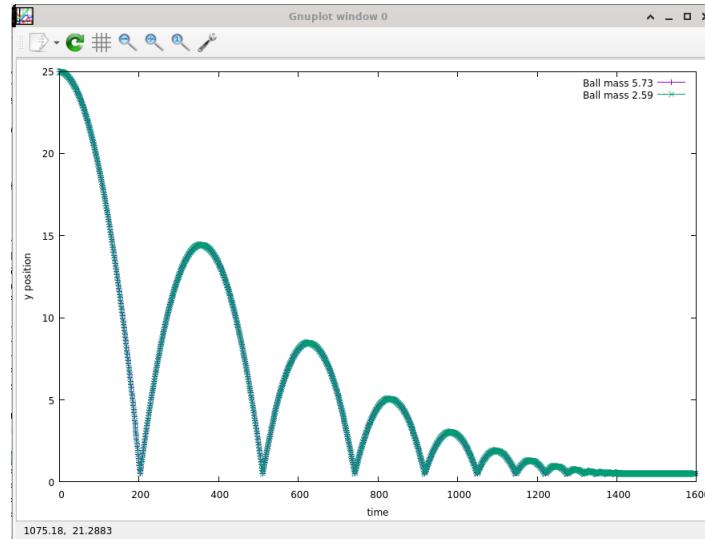


Figure 15.5: The Gnuplot of the `ballgravity.txt`. We want to see the behavior of same shape objects fall from same height, same restitution / bounciness and friction. The only different is the mass. Turns out they have the exact same behavior while falling.

Chapter 16

DFSimulatorC++ X: Fluids

"Reality flows from cause to effect like a mathematical equation. Mortals can't comprehend this. They're pathetic" - Albert Silverberg

M^{omentum}

I. SIMULATION FOR MOMENTUM WITH Box2D

You need to copy from my repository' directory `../Source Codes/C++/DianFreya-box2d-testbed`, then go inside the directory and open the terminal then type:

```
mkdir build  
cd build  
cmake ..  
make  
./testbed
```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

C++ Code 64: `tests/projectile_motion.cpp` "Projectile Motion Box2D"

Some explanations for the codes:

- ---

- ---

Chapter 17

DFSimulatorC++ XI: Oscillations

"You don't need a reason to study science and engineering, it is better than wasting time partying or gossiping. Great minds always discuss idea not discuss people." - DS Glanzsche

*"Physical problems cannot be analyzed by mathematics alone" - from Mathematical Models book
(Richard Haberman)*

When objects move back and forth repeatedly or periodically then that objects are oscillating [6]. If you play PlayStation with the vibrating joystick, when it vibrates, it is oscillating. I believe we don't need a lot of reasons to learn science, learn it as a means to spend time worthwhile.

Breakthrough for real are made by people like Leonardo Da Vinci and Isaac Newton (never marry and die as virgin, after hundreds of years their inventions still used and for hundreds of years more), transmuting sexual energy and desire into productive energy worth more than exponential power of compound interest. Real scientists and engineers don't do orgy or sex party, their party or celebration granted by the divine, as an example one of their creations bought by royalty for USD 400 million for a painting of "Salvator Mundi", or named as one of the great airport that sell a nice Insalata Salmon and Croissant.

I. SIMPLE HARMONIC MOTION

The frequency of the oscillation, denoted by f , is the number of times per second that it completes a full oscillation (a cycle) and has the unit of hertz (Hz), where

$$1 \text{ Hz} = 1 \text{ s}^{-1}$$

The time for one full cycle is the period T of the oscillation, which is

$$T = \frac{1}{f} \quad (17.1)$$

A simple harmonic motion (SHM) is a sinusoidal function of time t . That is, it can be written as a sine or a cosine of time t . Therefore

$$x(t) = x_m \cos(\omega t + \phi) \quad (17.2)$$

with $x(t)$ is the displacement at time t , x_m is the amplitude, t is time, ϕ is the phase constant or phase angle, ω is the angular frequency of the motion, together $(\omega t + \phi)$ is the phase.

To relate the angular frequency to the frequency f and period T , note that the position $x(t)$ of the object / particle must return to its initial value at the end of a period. That is, if $x(t)$ is the position at some chose time t , then the particle must return to that same position at time $t + T$. Returning to the same position can be written as

$$x_m \cos \omega t = x_m \cos \omega(t + T) \quad (17.3)$$

with

$$\begin{aligned} \omega(t + T) &= \omega t + 2\pi \\ \omega T &= 2\pi \\ \omega &= \frac{2\pi}{T} = 2\pi f \end{aligned}$$

the SI unit of angular frequency is the radian per second.

Now to find the velocity of the simple harmonic motion, we use the derivative with respect to time.

$$\begin{aligned} v(t) &= \frac{dx(t)}{dt} \\ &= \frac{d}{dt}[x_m \cos(\omega t + \phi)] \\ &= -\omega x_m \sin(\omega t + \phi) \end{aligned}$$

The velocity depends on time because the sine function varies with time, with the range of value for sine is $[-1, 1]$.

Now, if we differentiate the velocity of the simple harmonic motion, we will obtain the acceleration function of the particle in simple harmonic motion:

$$\begin{aligned} a(t) &= \frac{dv(t)}{dt} \\ &= \frac{d}{dt}[-\omega x_m \sin(\omega t + \phi)] \\ &= -\omega^2 x_m \cos(\omega t + \phi) \end{aligned}$$

The acceleration varies as well, because cosine function varies with time. Thus, we will see that

$$a(t) = -\omega^2 x(t)$$

- The particle's acceleration is always opposite its displacement
- To tell that a physical phenomena is in the category of simple harmonic motion, its acceleration and displacement will be related by a constant ω^2 , with ω is the angular frequency of the motion.

i. The Force Law for Simple Harmonic Motion

We can apply Newton's second law to describe the force responsible for simple harmonic motion:

$$F = ma = m(-\omega^2 x) = -(m\omega^2)x$$

The minus sign means that the direction of the force on the particle is opposite the direction of the displacement. In SHM, the force is a restoring force that it fights against the displacement, attempting to restore the particle to the center point at $x = 0$.

If we relate it with Hooke's law that stated

$$F = -kx$$

we can relate the spring constant k (a measure of the stiffness of the spring) to the mass of the block and the resulting angular frequency of the simple harmonic motion, we will obtain:

$$k = m\omega^2$$

The block-spring system is called a linear simple harmonic oscillator, since F is proportional to x to the first power. If you ever see a situation in which the force in an oscillation is always proportional to the displacement but in the opposite direction, you can say that the oscillation is simple harmonic motion. If you know the oscillating mass, you can determine the angular frequency of the motion as

$$\omega = \sqrt{\frac{k}{m}}$$

you can also determine the period of the motion with

$$T = 2\pi\sqrt{\frac{m}{k}}$$

Every oscillating system, like a violin string, has some element of "springiness" and some element of "inertia" or mass.

II. ENERGY IN SIMPLE HARMONIC MOTION

A particle in simple harmonic motion has, at any time, kinetic energy of

$$K = \frac{1}{2}mv^2$$

and potential energy of

$$U = \frac{1}{2}kx^2$$

If no friction is present, the mechanical energy of the oscillator will be

$$E = K + U \tag{17.4}$$

remains constant, even though K and U change over time.

Now, if we see the potential energy as the function of time

$$U(t) = \frac{1}{2}kx^2 = \frac{1}{2}kx_m^2 \cos^2(\omega t + \phi)$$

we can also write the kinetic energy as the function of time

$$K(t) = \frac{1}{2}mv^2 = \frac{1}{2}kx_m^2 \sin^2(\omega t + \phi)$$

combining both, we will get

$$\begin{aligned} E &= U(t) + K(t) \\ &= \frac{1}{2}kx_m^2 \cos^2(\omega t + \phi) + \frac{1}{2}kx_m^2 \sin^2(\omega t + \phi) \\ &= \frac{1}{2}kx_m^2 [\cos^2(\omega t + \phi) + \sin^2(\omega t + \phi)] \\ E &= \frac{1}{2}kx_m^2 \end{aligned}$$

it is stated that the mechanical energy of a linear oscillator is indeed constant and independent of time.

III. AN ANGULAR SIMPLE HARMONIC OSCILLATOR

A torsion pendulum is an angular version of a linear simple harmonic oscillator. The disk oscillates in a horizontal plane between the angular amplitude $-\theta_m$ and θ_m . The element of springiness or elasticity is associated with the twisting of a suspension wire rather than the extension and compression of a spring.

If we rotate the disk by some angular displacement θ from its rest position ($\theta = 0$) and release it, it will oscillate about that position in angular simple harmonic motion. Rotating the disk through an angle θ introduces a restoring torque given by

$$\tau = -\kappa\theta$$

with κ is the torsion constant, that depends on the length, diameter and material of the suspension wire. If we think of it as the angular form of Hooke's law, the period for the angular simple harmonic motion will be

$$T = 2\pi\sqrt{\frac{I}{\kappa}}$$

we replace the mass from simple harmonic motion with its equivalent, I , the rotational inertia of the oscillating disk. The same as we replace k , the spring constant, with κ , the torsion constant.

IV. PENDULUMS, CIRCULAR MOTION

Consider a pendulum of length L , at one end is attached to a fixed point and free to rotate about it. A mass of m is attached to the other end of the pendulum. From observations, a pendulum oscillates in a manner at least qualitatively similar to a spring-mass system.

We assume the mass M is large enough so that, as an approximation, we state that all the mass is contained at the bob of the pendulum (the mass of the rigid shaft of the pendulum is assumed negligible).

- The pendulum moves in two dimensions, while the spring-mass system was constrained to move in one dimension.
- A pendulum also involves only one degree of freedom as it is constrained to move along the circumference of a circle of radius L .

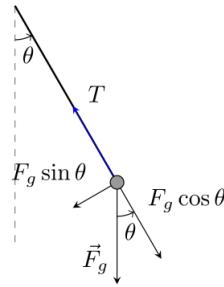


Figure 17.1: A simple pendulum with the forces acting on the pendulum' particle (called bob of the pendulum) with mass m are the gravitational force \vec{F}_g and the force \vec{T} from the string. The tangential component $F_g \sin \theta$ of the gravitational force is a restoring force that tends to bring the pendulum back to its central position.

By applying Newton's second law of motion then develop it into polar coordinates.

$$\vec{F} = m\vec{a}$$

$$m \frac{d^2\vec{x}}{dt^2} = \vec{F}$$

where \vec{x} is the position vector of the mass (the vector from the origin to the mass). In 3 dimensional rectangular coordinates, $\vec{x} = x\hat{i} + y\hat{j} + z\hat{k}$, thus the acceleration is given by

$$\frac{d^2\vec{x}}{dt^2} = \frac{d^2\vec{x}}{dt^2}\hat{i} + \frac{d^2\vec{y}}{dt^2}\hat{j} + \frac{d^2\vec{z}}{dt^2}\hat{k}$$

with $\hat{i}, \hat{j}, \hat{k}$ are unit vectors with fixed magnitude and fixed directions.

In polar coordinates (centered at the fixed vertex of the pendulum), the position vector is pointed outward with length L .

$$\vec{x} = L\vec{r} \quad (17.5)$$

where \vec{r} is the radial unit vector. The polar angle is introduced such that $\theta = 0$ corresponds to the pendulum in its "natural" position. L is constant since the pendulum does not vary in length. Thus

$$\frac{d^2\vec{x}}{dt^2} = L \frac{d^2\vec{r}}{dt^2} \quad (17.6)$$

Although the magnitude \vec{r} is constant with $|\vec{r}| = 1$ (the pendulum is always moving circling), its direction varies in space. To determine the change in the radial unit vector \vec{r} , we express it in terms of the cartesian unit vectors,

$$\vec{r} = \sin \theta \hat{i} - \cos \theta \hat{j} \quad (17.7)$$

The θ -unit vector is perpendicular to \vec{r} and of unit length (and is in the direction of increasing θ), thus

$$\hat{\theta} = \cos \theta \hat{i} + \sin \theta \hat{j} \quad (17.8)$$

In order to calculate the acceleration vector $\frac{d^2\vec{x}}{dt^2}$, the velocity vector $\frac{d\vec{x}}{dt}$ must first be calculated

$$\begin{aligned}\vec{x} &= L\vec{r} \\ \frac{d\vec{x}}{dt} &= L\frac{d\vec{r}}{dt} + \frac{dL}{dt}\vec{r} \\ \frac{d\vec{x}}{dt} &= L\frac{d\vec{r}}{dt}\end{aligned}$$

since L is a constant for a pendulum and $\frac{dL}{dt} = 0$.

With the chain rule we will obtain

$$\begin{aligned}\frac{d\hat{r}}{dt} &= \frac{d\hat{r}}{d\theta} \frac{d\theta}{dt} \\ &= \frac{d}{d\theta} (\sin \theta \hat{i} - \cos \theta \hat{j}) \frac{d\theta}{dt} \\ &= \frac{d\theta}{dt} (\cos \theta \hat{i} + \sin \theta \hat{j}) \\ \frac{d\hat{r}}{dt} &= \frac{d\theta}{dt} \hat{\theta}\end{aligned}\tag{17.9}$$

Now, for the change in θ with respect to time, we will use chain rule again to obtain

$$\begin{aligned}\frac{d\hat{\theta}}{dt} &= \frac{d\hat{\theta}}{d\theta} \frac{d\theta}{dt} \\ &= \frac{d}{d\theta} (\cos \theta \hat{i} + \sin \theta \hat{j}) \frac{d\theta}{dt} \\ &= \frac{d\theta}{dt} (-\sin \theta \hat{i} + \cos \theta \hat{j}) \\ \frac{d\hat{\theta}}{dt} &= -\frac{d\theta}{dt} \hat{\theta}\end{aligned}\tag{17.10}$$

Thus, the velocity vector is in the direction of $\hat{\theta}$.

$$\frac{d\vec{x}}{dt} = L\frac{d\theta}{dt}\hat{\theta}$$

The magnitude of the velocity is $L\frac{d\theta}{dt}$, if motion lies along the circumference of a circle. If L is constant, then in a short length of time the position vector has changed only a little in the θ direction.

$$\frac{d\vec{x}}{dt} \approx \frac{\vec{x}(t + \Delta t) - \vec{x}(t)}{\Delta t} \approx \frac{L\Delta\theta \hat{\theta}}{\Delta t} \approx L\frac{d\theta}{dt}\hat{\theta}$$

The θ component of the velocity is the distance L times the angular velocity $d\theta/dt$. The acceleration vector is obtained as the derivative of the velocity vector:

$$\begin{aligned}\frac{d^2\vec{x}}{dt^2} &= L\frac{d}{dt} \left(\frac{d\theta}{dt}\hat{\theta} \right) \\ &= L \left[\frac{d^2\theta}{dt^2}\hat{\theta} - \left(\frac{d\theta}{dt} \right)^2 \hat{r} \right]\end{aligned}$$

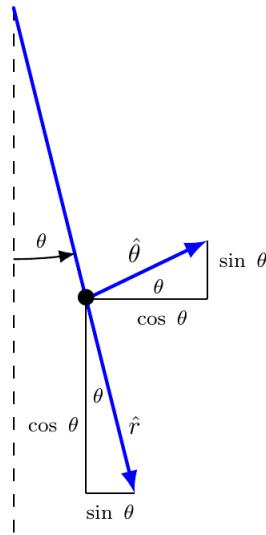


Figure 17.2: The direction of \hat{r} and $\hat{\theta}$ vary in space, both has the magnitude of the same unit vectors.

The angular component of the acceleration is $L \frac{d^2\theta}{dt^2}$. It exists only if the angle is accelerating. If L is constant, the radial component of the acceleration, $L \left(\frac{d\theta}{dt} \right)^2$, is always directed inwards. In physics, it is called the centripetal acceleration and will occur even if the angle is only steadily increasing (if the angular velocity $\frac{d\theta}{dt}$ is constant).

For any forces \vec{F} , when a mass is constrained to move in a circle, Newton's law implies

$$mL \frac{d^2\theta}{dt^2} \hat{\theta} - mL \left(\frac{d\theta}{dt} \right)^2 \hat{r} = \vec{F} \quad (17.11)$$

Now, we will examine the forces for a pendulum:

- Gravitational force, $-mg \hat{j}$, which can be expressed in terms of polar coordinates

$$\begin{aligned} \hat{i} &= \hat{r} \sin \theta + \hat{\theta} \cos \theta \\ \hat{j} &= -\hat{r} \cos \theta + \hat{\theta} \sin \theta \end{aligned} \quad (17.12)$$

Thus, the gravitational force

$$m\vec{g} = -mg \hat{j} = mg \cos \theta \hat{r} - mg \sin \theta \hat{\theta}$$

- The forces on the bob of the pendulum that make it to be in motion along the circle. If there were no these forces, then the mass would not move along the circle. The mass is held by the rigid shaft of the pendulum, which exerts a force $T\hat{r}$ towards the origin.

The forces on the bob of the pendulum that result in motion along the circle is

$$mL \frac{d^2\theta}{dt^2} \hat{\theta} - mL \left(\frac{d\theta}{dt} \right)^2 \hat{r} = mg \cos \theta \hat{r} - mg \sin \theta \hat{\theta} - T\hat{r}$$

Each component of this vector force equation yields an ordinary differential equation:

$$mL \frac{d^2\theta}{dt^2} = -mg \sin \theta \quad (17.13)$$

$$-mL \left(\frac{d\theta}{dt} \right)^2 = mg \cos \theta - T \quad (17.14)$$

A two-dimensional vector equation is equivalent to two scalar equations. T could be obtained from the second equation, equation (17.14), after determining θ from the first equation, equation (17.13). The first equation implies that the mass times the θ component of the acceleration must balance the θ component of the gravitational force.

The mass m can be cancelled from both sides of the first equation. Thus, the motion of the pendulum does not depend on the magnitude of the mass m attached to the pendulum. Only varying the length, L , or the gravity g , will affect the motion.

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L} \sin \theta \quad (17.15)$$

The pendulum is governed by a nonlinear differential equation, equation (17.15), and hence is called a nonlinear pendulum. The restoring force, $mg \sin \theta$ does not depend linearly on θ . Nonlinear problems are usually considerably more difficult to solve than linear ones.

Now, recall from calculus for small θ ,

$$\sin \theta \approx \theta$$

Geometrically, near the origin the functions $\sin \theta$ and θ are nearly identical. By using this approximation, the differential equation becomes

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L} \theta \quad (17.16)$$

this is called the linearized pendulum. This is the same type of differential equation as the one governing a linearized spring-mass system without friction. Hence, a linearized pendulum also executes simple harmonic motion. The pendulum oscillates with circular frequency

$$\omega = \sqrt{\frac{g}{L}} \quad (17.17)$$

and the period

$$T = \frac{2\pi}{\omega} = 2\pi \sqrt{\frac{L}{g}} \quad (17.18)$$

The three equations (17.16), (17.17), (17.18) valid only as long as θ is small. The period of oscillation is independent of amplitude. As a reminder, when we say θ it means in radians (in pi terms), not in degree.

V. NONLINEAR PENDULUM

Finish Potential Energy of Physics, then we finish this along with the phase plane and direction field plot with C++. Pendulum is harder than spring-mass system, consider put it below -Sentinel

VI. (OPTIONAL?) THE EQUATION OF MOTION FOR SIMPLE PENDULUM

A simple pendulum [3], which consists of a string of length l with a point mass m on the end will have the position of the pendulum specified by the angle θ between the string and the vertical line. The equation of motion of the pendulum is given by:

$$\tau = I\theta^{(2)} = I \frac{d^2\theta}{dt^2} \quad (17.19)$$

where I is the moment of inertia and is given by

$$I = ml^2$$

The torque τ is given by

$$\tau = -mgl \sin \theta$$

thus

$$\frac{d^2\theta}{dt^2} + \frac{g}{l} \sin \theta = 0 \quad (17.20)$$

VII. SIMULATION OF SIMPLE PENDULUM WITH Box2D

i. Build DianFreya Modified Box2D Testbed

You need to copy from my repository' directory `../Source Codes/C++/DianFreya-box2d-testbed`, then go inside the directory and open the terminal then type:

```
mkdir build
cd build
cmake ..
make
./testbed
```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Oscillations/Simple Pendulum**.

```
#define DEGTORAD 0.0174532925199432957f
#define RADTODEG 57.295779513082320876f
#include "test.h"
#include <fstream>

class SimplePendulum : public Test
{
public:

    SimplePendulum()
    {
        b2Body* b1;
        {
            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
                0.0f));
    }
}
```

```
b2BodyDef bd;
b1 = m_world->CreateBody(&bd);
b1->CreateFixture(&shape, 0.0f);
}
// the two blocks below for creating boxes are only for
sweetener.
{
    b2PolygonShape shape;
    shape.SetAsBox(7.0f, 0.25f, b2Vec2_zero, 0.7f); // Gradient is 0.7

    b2BodyDef bd;
    bd.position.Set(6.0f, 6.0f);
    b2Body* ground = m_world->CreateBody(&bd);
    ground->CreateFixture(&shape, 0.0f);
}
{
    b2PolygonShape shape;
    shape.SetAsBox(7.0f, 0.25f, b2Vec2_zero, -0.7f); // Gradient is -0.7

    b2BodyDef bd;
    bd.position.Set(-6.0f, 6.0f);
    b2Body* ground = m_world->CreateBody(&bd);
    ground->CreateFixture(&shape, 0.0f);
}

b2Body* b2; // the hanging bar for the pendulum
{
    b2PolygonShape shape;
    shape.SetAsBox(7.25f, 0.25f);

    b2BodyDef bd;
    bd.position.Set(0.0f, 37.0f);
    b2 = m_world->CreateBody(&bd);
    b2->CreateFixture(&shape, 0.0f);
}

b2RevoluteJointDef jd;
b2Vec2 anchor;

// Create the pendulum ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

b2FixtureDef ballFixtureDef;
```

```

ballFixtureDef.restitution = 0.75f;
ballFixtureDef.density = 3.3f; // this will affect the ball
mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(0.0f, 25.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&
    ballFixtureDef);

// Create the anchor and connect it to the ball
anchor.Set(0.0f, 36.0f); // x and y axis position for the
Pendulum anchor
jd.Initialize(b2, m_ball, anchor);
//jd.collideConnected = true;
m_world->CreateJoint(&jd); // Create the Pendulum anchor
}

void Step(Settings& settings) override
{
    b2Vec2 v = m_ball->GetLinearVelocity();
    float omega = m_ball->GetAngularVelocity();
    float angle = m_ball->GetAngle();
    b2MassData massData = m_ball->GetMassData();
    b2Vec2 position = m_ball->GetPosition();

    float ke = 0.5f * massData.mass * b2Dot(v, v) + 0.5f *
        massData.I * omega * omega;

    g_debugDraw.DrawString(5, m_textLine, "Ball position, x = %.6
        f", position.x);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Ball position, y = %.6
        f", position.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
        .mass);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Kinetic energy = %.6f"
        , ke);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Linear velocity = %.6f
        ", v);
}

```

```

        m_textLine += m_textIncrement;
        g_debugDraw.DrawString(5, m_textLine, "Angle (in degrees) =
            %.6f", angle*RADTODEG);
        m_textLine += m_textIncrement;
        // Print the result in every time step then plot it into
        // graph with either gnuplot or anything
        printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle*
            RADTODEG);
        Test::Step(settings);
    }

    static Test* Create()
    {
        return new SimplePendulum;
    }
    b2Body* m_ball;
};

static int testIndex = RegisterTest("Oscillations", "Simple Pendulum",
    SimplePendulum::Create);

```

C++ Code 65: *tests/simple_pendulum.cpp* "Simple Pendulum Box2D"

Some explanations for the codes:

- To create a line below:

```

b2Body* b1;
{
    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(-40.0f, 0.0f), b2Vec2(40.0f,
        0.0f));

    b2BodyDef bd;
    b1 = m_world->CreateBody(&bd);
    b1->CreateFixture(&shape, 0.0f);
}

```

- Create a Revolute Joint named **jd** that can revolve

```

b2RevoluteJointDef jd;
b2Vec2 anchor;

```

- Create the pendulum ball, set its' position, friction, restitution, radius, density, angular damping (so the pendulum will eventually stop instead of forever oscillating) with shape of a circle, but we call it **ballShape**.

```

// Create the pendulum ball
b2CircleShape ballShape;
ballShape.m_p.SetZero();
ballShape.m_radius = 0.5f;

```

```

b2FixtureDef ballFixtureDef;
ballFixtureDef.restitution = 0.75f;
ballFixtureDef.density = 3.3f; // this will affect the ball
mass
ballFixtureDef.friction = 0.1f;
ballFixtureDef.shape = &ballShape;

b2BodyDef ballBodyDef;
ballBodyDef.type = b2_dynamicBody;
ballBodyDef.position.Set(0.0f, 25.0f);
// ballBodyDef.angularDamping = 0.2f;

m_ball = m_world->CreateBody(&ballBodyDef);
b2Fixture *ballFixture = m_ball->CreateFixture(&ballFixtureDef
);

```

- Now after the ball for the pendulum is created we can connect it to the anchor with `CreateJoint()`;

```

// Create the anchor and connect it to the ball
anchor.Set(0.0f, 36.0f); // x and y axis position for the
Pendulum anchor
jd.Initialize(b2, m_ball, anchor);
//jd.collideConnected = true;
m_world->CreateJoint(&jd); // Create the Pendulum anchor

```

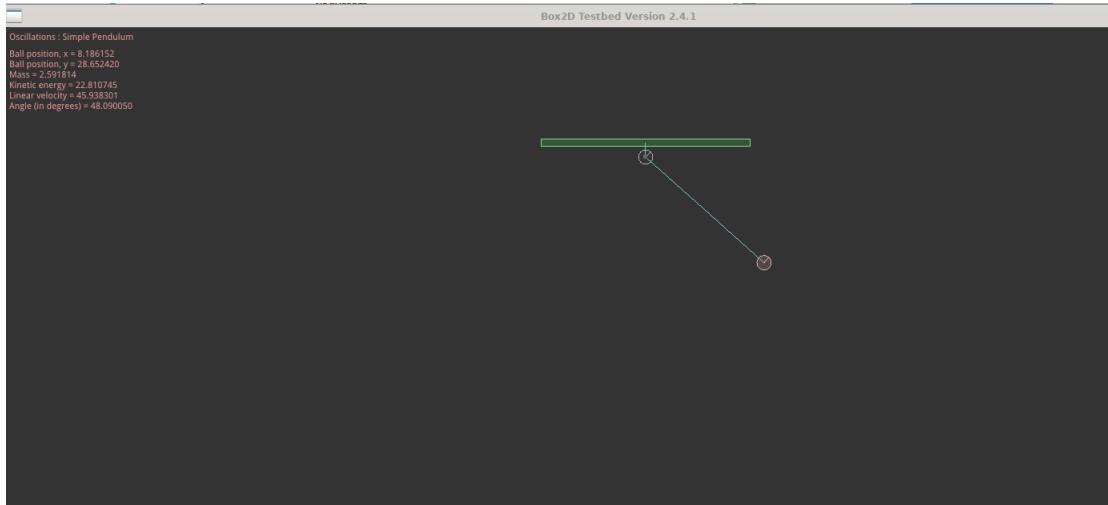


Figure 17.3: The simple pendulum simulation with Box2D, it can be played when you build the testbed. (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/simple_pendulum.cpp`).

- To print the output of the x position, y position and the angle (in degrees) of the pendulum into the terminal / xterm

```
printf("%4.2f %4.2f %4.2f\n", position.x, position.y, angle*
RADTODEG);
```

After recompiling this, you can save it into textfile by opening the testbed with this command:
./testbed > /root/output.txt

you may drag the pendulum to certain degree and let it swing for 5 periods or around that, afterwards close the testbed to record the result, then see it at **/root/output.txt**. You need to clean up a bit and delete the strings at the beginning of the textfile so it will only left you with numbers only in 3 columns.

To plot it you can use gnuplot, as it is very handy when we have raw data, then need to plot it fast, easy, and powerful even for 3-dimensional surface plot. Open terminal from the directory that contain the "output.txt" and type:

```
gnuplot
plot "output.txt" using 1 title "x_{m}" with lines
```

you can also try:

```
plot "output.txt" using 1 title "x_{m}" with lines, "output.txt" using 3 title "angle"
```

```
File Edit Search View Document Help
Warning: you are using the root account. You may harm your system.

0.13 25.00 0.67
0.18 25.00 0.94
0.23 25.00 1.21
0.28 25.00 1.48
0.33 25.01 1.74
0.39 25.01 2.01
0.44 25.01 2.28
0.49 25.01 2.55
0.54 25.01 2.81
0.59 25.02 3.08
0.64 25.02 3.35
0.69 25.02 3.61
0.74 25.03 3.88
0.79 25.03 4.14
0.85 25.03 4.41
0.90 25.04 4.67
0.95 25.04 4.94
1.00 25.05 5.20
1.05 25.05 5.46
1.10 25.05 5.72
1.15 25.06 5.99
1.20 25.07 6.25
1.25 25.07 6.51
1.30 25.08 6.77
1.35 25.08 7.02
1.39 25.09 7.28
1.44 25.10 7.54
1.49 25.10 7.79
1.54 25.11 8.05
1.59 25.12 8.30
1.64 25.12 8.56
1.68 25.13 8.81
1.73 25.14 9.06
1.78 25.14 9.31
1.83 25.15 9.56
*** *** ***
```

Figure 17.4: The "output.txt" shall only contains 3 columns of numerical data.

Box2D is an amazing Physics Engine library that can be used not only to create game but for learning all kinds of science, since we are all bounds to Physics even atoms and quantum are still kneel to the Physics. It won't be too long to gain mastery over learning this for anyone who want to invest their time, interest and disk space of their brain.

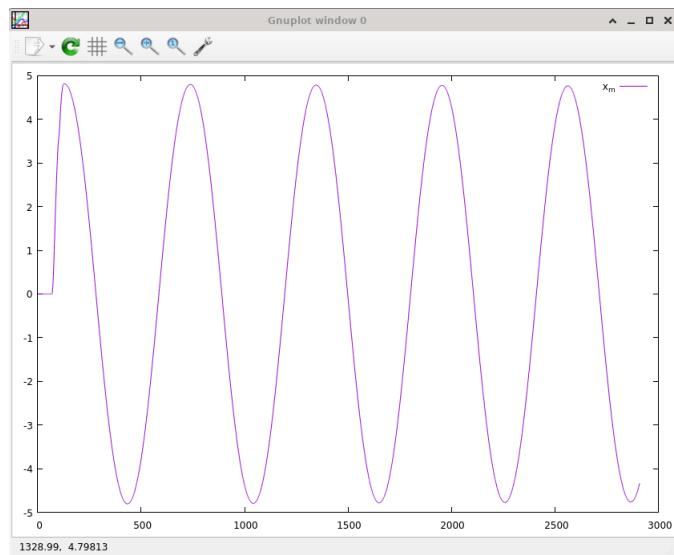


Figure 17.5: The "output.txt" is being plotted by using gnuplot for the data of the x_m (the displacement) with respect to time.

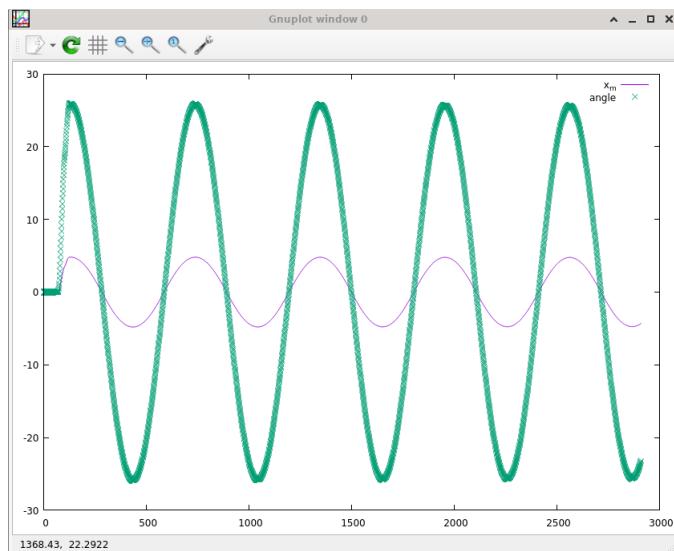


Figure 17.6: The "output.txt" is being plotted by using gnuplot for the data of the x_m (the displacement) and the angle of the pendulum with respect to time. We may see that the angle and the displacement of the pendulum have a linear relationship.

VIII. HORIZONTAL SPRING-MASS SYSTEM

When we observe a spring-mass system, once it is set in motion the object with mass moves back and forth (oscillates) [5]. This simple spring-mass system exhibits behavior of complex system resemble the motions of clock-like mechanisms and aid in the understanding of the up-and-down motion of the ocean surface. Remember the Newton's second law of motion that describing how a

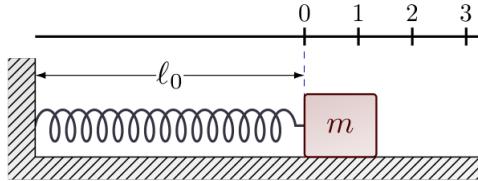


Figure 17.7: The spring-mass system at rest with $x = 0$ as the position of the mass on the x axis, and l_0 is the original length of the spring, it is called the equilibrium or unstretched position of the spring.

particle reacts to a force

$$\vec{F} = m\vec{a} = m \frac{d\vec{v}}{dt} = m \frac{d^2\vec{x}}{dt^2}$$

Easily remembered as "F equals ma ."

The distance x is then referred to as the displacement from equilibrium or the amount of stretching of the spring.

- If we stretch the spring (let $x > 0$), then the spring exerts a force pulling the mass back towards the equilibrium position (that is $F < 0$). As we increase the stretching of the spring, the force exerted by the spring would increase. For stretched spring, we will have this relation: $x_1 < x_2$, hence $F_2 < F_1$.
- If the spring is contracted ($x < 0$), then the spring pushes the mass again towards the equilibrium position ($F > 0$). For contracted spring, we will have this relation: $x_1 < x_2$, hence $F_1 < F_2$.

Such a force, F , is called the restoring force. We have assumed that the force only depends on the amount of stretching of the spring, not on any other quantities.

On the seventeenth century, physicist Hooke approximate the relationship between the force and the stretching of the spring with a straight line, thus Hooke's law is

$$F = -kx \tag{17.21}$$

with k as the spring constant. Higher positive force is required to get the mass to its equilibrium position after being contracted deeper. Bigger negative force is required to get the mass to its equilibrium position after being stretched longer.

Using Hooke's law, Newton's second law of motion yields

$$m \frac{d^2x}{dt^2} = -kx \tag{17.22}$$

as the simplest mathematical model of a spring-mass system.

IX. SIMULATION OF HORIZONTAL SPRING-MASS SYSTEM WITH Box2D

We are going to simulate the horizontal spring-mass system at rest then be given force toward positive x axis, and toward negative x axis.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class SpringTest : public Test
{
public:
    SpringTest()
    {
        b2Body* ground = NULL;
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                0.0f));
            ground->CreateFixture(&shape, 0.0f);
        }
        // Create a static body as the box for the spring
        b2BodyDef bd1;
        bd1.type = b2_staticBody;
        bd1.angularDamping = 0.1f;

        bd1.position.Set(1.0f, 0.5f);
        b2Body* springbox = m_world->CreateBody(&bd1);

        b2PolygonShape shape1;
        shape1.SetAsBox(0.5f, 0.5f);
        springbox->CreateFixture(&shape1, 5.0f);

        // Create the box as the movable object
        b2PolygonShape boxShape;
        boxShape.SetAsBox(0.5f, 0.5f);

        b2FixtureDef boxFixtureDef;
        boxFixtureDef.restitution = 0.75f;
        boxFixtureDef.density = 7.3f; // this will affect the box
        mass
        boxFixtureDef.friction = 0.1f;
        boxFixtureDef.shape = &boxShape;

        b2BodyDef boxBodyDef;
```

```

boxBodyDef.type = b2_dynamicBody;
boxBodyDef.position.Set(5.0f, 0.5f);

m_box = m_world->CreateBody(&boxBodyDef);
b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
;
//m_box->SetGravityScale(-7); // negative means it will goes
// upward, positive it will goes downward
// Make a distance joint for the box / ball with the static
// box above
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f),
    boxBodyDef.position);
jd.collideConnected = true; // In this case we decide to
// allow the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f;
m_maxLength = 10.0f;
b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
    m_dampingRatio, jd.bodyA, jd.bodyB);

m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
m_joint->SetMinLength(m_minLength);
m_joint->SetMaxLength(m_maxLength);

m_time = 0.0f;
}
b2Body* m_box;
b2DistanceJoint* m_joint;
float m_length;
float m_time;
float m_minLength;
float m_maxLength;
float m_hertz;
float m_dampingRatio;

void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_A:
            //m_box->SetLinearVelocity(b2Vec2(30.0f, 0.0f));
            m_box->ApplyForceToCenter(b2Vec2(-8000.0f, 0.0f),
                true);
            break;
    }
}

```

```

        case GLFW_KEY_S:
            //m_box->SetLinearVelocity(b2Vec2(-30.0f, 0.0f));
            m_box->ApplyForceToCenter(b2Vec2(-5000.0f, 0.0f),
                true);
            break;
        case GLFW_KEY_D:
            m_box->ApplyForceToCenter(b2Vec2(5000.0f, 0.0f), true)
                ;
            break;
        case GLFW_KEY_F:
            m_box->ApplyForceToCenter(b2Vec2(8000.0f, 0.0f), true)
                ;
            break;
        case GLFW_KEY_G:
            m_box->ApplyForceToCenter(b2Vec2(12000.0f, 0.0f), true)
                );
            break;
    }
}

void UpdateUI() override
{
    ImGui::SetNextWindowPos(ImVec2(10.0f, 100.0f));
    ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
    ImGui::Begin("Joint Controls", nullptr,
        ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

    if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0f"))
    {
        m_length = m_joint->SetLength(m_length);
    }

    if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"))
    {
        float stiffness;
        float damping;
        b2LinearStiffness(stiffness, damping, m_hertz,
            m_dampingRatio, m_joint->GetBodyA(), m_joint->
            GetBodyB());
        m_joint->SetStiffness(stiffness);
        m_joint->SetDamping(damping);
    }

    if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
        , 2.0f, "%.1f"))
    {
        float stiffness;

```

```

        float damping;
        b2LinearStiffness(stiffness, damping, m_hertz,
                           m_dampingRatio, m_joint->GetBodyA(), m_joint->
                           GetBodyB());
        m_joint->SetStiffness(stiffness);
        m_joint->SetDamping(damping);
    }

    ImGui::End();
}
void Step(Settings& settings) override
{
    b2MassData massData = m_box->GetMassData();
    b2Vec2 position = m_box->GetPosition();
    b2Vec2 velocity = m_box->GetLinearVelocity();
    m_time += 1.0f / 60.0f; // assuming we are using frequency of
                           60 Hertz

    g_debugDraw.DrawString(5, m_textLine, "Press A/S/D/F/G to
                                         apply different force to the box");
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
                                         f", m_time);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Mass position = (%4.1f
                                         , %4.1f)", position.x, position.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Mass velocity = (%4.1f
                                         , %4.1f)", velocity.x, velocity.y);
    m_textLine += m_textIncrement;
    g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
                           .mass);
    m_textLine += m_textIncrement;
    // Print the result in every time step then plot it into
    // graph with either gnuplot or anything

    printf("%4.2f\n", position.x);

    Test::Step(settings);
}
static Test* Create()
{
    return new SpringTest;
}

static int testIndex = RegisterTest("Oscillations", "Spring Test",

```

```
SpringTest::Create);
```

C++ Code 66: *tests/spring_test.cpp "Horizontal Spring Mass System Box2D"*

Some explanations for the codes:

- To create the static body as the wall block that will be attached to the spring

```
b2BodyDef bd1;
bd1.type = b2_staticBody;
bd1.angularDamping = 0.1f;

bd1.position.Set(1.0f, 0.5f);
b2Body* springbox = m_world->CreateBody(&bd1);

b2PolygonShape shape1;
shape1.SetAsBox(0.5f, 0.5f);
springbox->CreateFixture(&shape1, 5.0f);
```

- To create the "hallucination" spring in Box2D we can use **DistanceJoint** that is joining the static body as the wall block and the dynamic body as the mass / a movable object.

```
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f), boxBodyDef.
    position);
jd.collideConnected = true; // In this case we decide to allow
    the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f;
m_maxLength = 10.0f;
b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
    m_dampingRatio, jd.bodyA, jd.bodyB);

m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
m_joint->SetMinLength(m_minLength);
m_joint->SetMaxLength(m_maxLength);
```

Distance joint is one of the simplest joint, we can make it work like a spring, it makes the distance between two bodies must be constant.

- The keyboard press events, the function **ApplyForceToCenter** with negative force means it will direct the force toward the negative x axis, while positive force means it will direct the force toward the positive x axis. The coding in Box2D is quite the opposite of the theory when we stretch the spring ($x > 0$) thus the force to pull the mass back to equilibrium position will be negative, that is $F < 0$. It should be understood, so all the confusions shall be cleared up, as in Box2D, **ApplyForceToCenter(b2Vec2(-8000.0f, 0.0f), true);** means applying force of 8000 N to the box toward the negative x axis, it has nothing to do with the force to bring the mass to equilibrium position.

```

void Keyboard(int key) override
{
    switch (key)
    {
        case GLFW_KEY_A:
            //m_box->SetLinearVelocity(b2Vec2(30.0f, 0.0f));
            m_box->ApplyForceToCenter(b2Vec2(-8000.0f, 0.0f)
                , true);
            break;
        case GLFW_KEY_S:
            //m_box->SetLinearVelocity(b2Vec2(-30.0f, 0.0f))
            ;
            m_box->ApplyForceToCenter(b2Vec2(-5000.0f, 0.0f)
                , true);
            break;
        case GLFW_KEY_D:
            m_box->ApplyForceToCenter(b2Vec2(5000.0f, 0.0f),
                true);
            break;
        case GLFW_KEY_F:
            m_box->ApplyForceToCenter(b2Vec2(8000.0f, 0.0f),
                true);
            break;
        case GLFW_KEY_G:
            m_box->ApplyForceToCenter(b2Vec2(12000.0f, 0.0f),
                true);
            break;
    }
}

```

- A feature from ImGui at the top left, we can change the length for the joint / the "hallucination" spring, the damping ratio that will determine the speed of convergence to the equilibrium position, and the frequency / Hertz parameter, the higher the Hertz, the faster the movement that our eyes see, it will look like it suddenly finish.

```

void UpdateUI() override
{
    ImGui::SetNextWindowPos(ImVec2(10.0f, 100.0f));
    ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
    ImGui::Begin("Joint Controls", nullptr,
        ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize)
        ;

    if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f,
        "%f"))
    {
        m_length = m_joint->SetLength(m_length);
    }
}

```

```

        if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%1f"))
    {
        float stiffness;
        float damping;
        b2LinearStiffness(stiffness, damping, m_hertz,
                           m_dampingRatio, m_joint->GetBodyA(), m_joint
                           ->GetBodyB());
        m_joint->SetStiffness(stiffness);
        m_joint->SetDamping(damping);
    }

    if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio,
                           0.0f, 2.0f, "%1f"))
    {
        float stiffness;
        float damping;
        b2LinearStiffness(stiffness, damping, m_hertz,
                           m_dampingRatio, m_joint->GetBodyA(), m_joint
                           ->GetBodyB());
        m_joint->SetStiffness(stiffness);
        m_joint->SetDamping(damping);
    }

    ImGui::End();
}

```

After recompiling this, you can save it into textfile by opening the testbed with this command:
./testbed > spring.txt
you may press "G" or "F" or any other key or do mouse click to drag the box and let it oscillates, afterwards close the testbed to record the result, then see it at **../(current working directory)/spring.txt**. You need to clean up a bit and delete the strings at the beginning of the textfile so it will only left you with numbers only in 1 column.

To plot it you can use gnuplot. Open terminal from the directory that contain the "spring.txt" and type:

```

gnuplot
set xlabel "t"
plot "spring.txt" using 1 title "x (t)" with lines

```

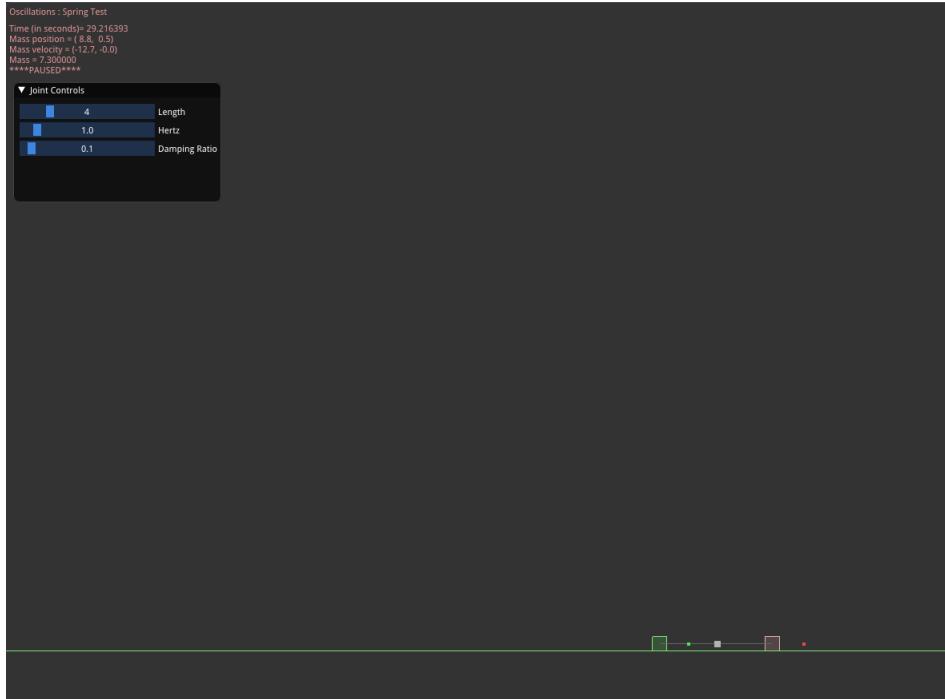


Figure 17.8: The horizontal spring-mass system simulation , besides keyboard press events you can click the box with mouse and drag it toward positive or negative x axis to see it oscillating till it rests at equilibrium position ($x = 5, y = 0.5$) again (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/spring_test.cpp`).

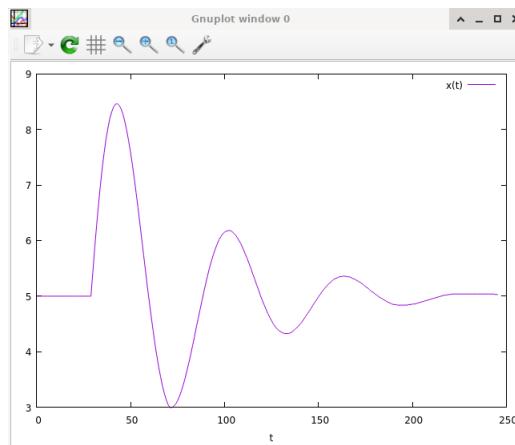


Figure 17.9: The gnuplot of a horizontal spring-mass system, with friction 0.1 and damping ratio of 0.1, mass of $m = 7.3$, and the rest position at $x = 5, y = 0.5$ after being pushed toward positive x axis with a force of $F = 10,000$.

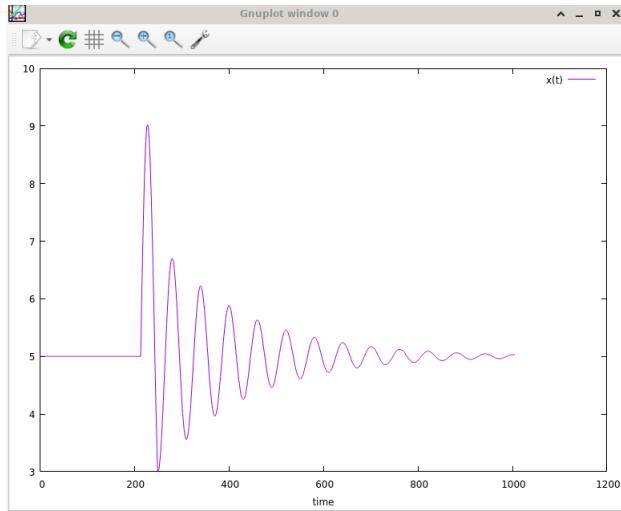


Figure 17.10: The gnuplot of a horizontal spring-mass system, friction 0 and damping ratio of 0, mass of $m = 7.3$, and the rest position at $x = 5, y = 0.5$ after being pushed toward positive x axis with a force of $F = 10,000$.

X. VERTICAL SPRING-MASS SYSTEM

Now, consider a vertical spring-mass system, which the derivation of the equation governing the horizontal spring-mass system does not apply to the vertical system. There is another force, gravity, that will be approximated as a constant mg , or the weight. Hence the Newton's law for the vertical system becomes

$$m \frac{d^2y}{dt^2} = -ky - mg \quad (17.23)$$

where y is the vertical coordinate. $y = 0$ is the position at which the spring exerts no force. The

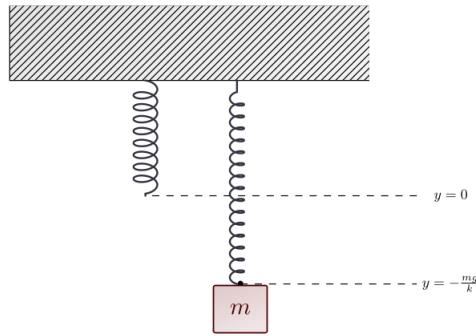


Figure 17.11: Gravitational effect on spring-mass equilibrium for vertical spring. The spring sags downwards a distance $\frac{mg}{k}$ when the mass is added. The stiffer the spring (k larger), the smaller the sag of the spring.

force at which we could place the mass and it would not move will be called an equilibrium

position. It follows that

$$\begin{aligned}\sum F_y &= ma_y \\ -ky - mg &= m(0) \\ -ky &= mg \\ y &= -\frac{m}{k}g\end{aligned}$$

where $y = -\frac{m}{k}g$ is the equilibrium position of this spring-mass-gravity system. Only at that position will the force due to gravity balance the upward force of the spring. The negative sign is to represent the opposite direction between the spring' force and the gravity force toward the mass. In some cases, we can also use Δy instead of negative sign, since it means displacement from the equilibrium position

$$k\Delta y = mg$$

To translate coordinate systems from one with an origin at $y = 0$ (the position of the unstretched spring) to one with an origin at $y = -\frac{mg}{k}$ (the equilibrium position with the mass). Let Z to be the variable representing the displacement from this equilibrium position:

$$\begin{aligned}Z &= y - \left(-\frac{mg}{k}\right) \\ &= y + \frac{mg}{k}\end{aligned}$$

Z here for the vertical spring-mass system is replacing x for the horizontal spring-mass system, thus

$$m \frac{d^2Z}{dt^2} = -kZ \quad (17.24)$$

XI. SIMULATION OF VERTICAL SPRING-MASS SYSTEM WITH Box2D

Taken from problem no 88 from [6]. I modify some parameters a bit.

A block weighing 200 N oscillates at one end of a vertical spring for which $k = 100 \text{ N/m}$; the other end of the spring is attached to a ceiling. At a certain instant the spring is stretched 3.0 m beyond its relaxed length (the length when no object is attached) and the block has zero velocity.

- (a) What is the net force on the block at this instant?
- (b) What is the amplitude?
- (c) What is the period of the resulting simple harmonic motion?
- (d) What is the maximum kinetic energy of the block as it oscillates?

Solution:

- (a) The Hooke's law force will have opposite direction with the weight' force, we choose the positive direction to be upward thus

$$\begin{aligned}\sum F &= kx - mg \\ &= 100(3.0) - 200 \\ &= 100\end{aligned}$$

the net force is 100 N upward.

- (b) The equilibrium position is where the upward Hooke's law force balances the weight, which corresponds to the spring being stretched (from unstretched length) by

$$y = \frac{mg}{k} = \frac{200}{100} = 2$$

Thus, relative to the equilibrium position, the block(at the instant) is the bottom turning point (when $v = 0$) at $x = -x_m$, where the amplitude is

$$x_m = 3.0 - 2.0 = 1.0$$

the amplitude is 1.0 m. How to plot this amplitude is explained below along with the C++ code to create the simulation for this problem.

- (c) We know that

$$m = \frac{W}{g} = \frac{200}{9.8} \approx 20$$

the mass is 20 kg, thus

$$T = 2\pi\sqrt{\frac{m}{k}} = 2\pi\sqrt{\frac{20}{100}} = 2.81$$

the period is 2.81 s.

- (d) The maximum kinetic energy is equal to the maximum potential energy $\frac{1}{2}kx_m^2$. Thus,

$$\begin{aligned} K_m &= U_m \\ &= \frac{1}{2}(100)(1.0) \\ &= 50 \end{aligned}$$

the maximum kinetic energy is 50 J.

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class VerticalspringTest : public Test
{
public:
    VerticalspringTest()
    {
        b2Body* ground = NULL;
        m_world->SetGravity(b2Vec2(0.0f, -9.8f));
        {
            b2BodyDef bd;
            ground = m_world->CreateBody(&bd);

            b2EdgeShape shape;
            shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(46.0f,
                0.0f));
        }
    }
};
```

```

        ground->CreateFixture(&shape, 0.0f);
    }
    // Create a static body as the box for the spring
    b2BodyDef bd1;
    bd1.type = b2_staticBody;
    bd1.angularDamping = 0.1f;

    bd1.position.Set(1.0f, 23.5f);
    b2Body* springbox = m_world->CreateBody(&bd1);

    b2PolygonShape shape1;
    shape1.SetAsBox(0.5f, 0.5f);
    springbox->CreateFixture(&shape1, 5.0f);

    // Create the box as the movable object
    b2PolygonShape boxShape;
    boxShape.SetAsBox(0.5f, 0.5f);

    b2FixtureDef boxFixtureDef;
    boxFixtureDef.restitution = 0.75f;
    boxFixtureDef.density = 200.0f/9.8f; // this will affect the
        box mass
    boxFixtureDef.friction = 0.1f;
    boxFixtureDef.shape = &boxShape;

    b2BodyDef boxBodyDef;
    boxBodyDef.type = b2_dynamicBody;
    boxBodyDef.position.Set(1.0f, 19.5f); // the box will be
        located in (1,19.5), 2.0 m beyond the spring relaxed
        length. It is the equilibrium position where ky = mg

    m_box = m_world->CreateBody(&boxBodyDef);
    b2Fixture *boxFixture = m_box->CreateFixture(&boxFixtureDef)
        ;
    //m_box->SetGravityScale(-7); // negative means it will goes
        upward, positive it will goes downward
    // Make a distance joint for the box / ball with the static
        box above
    m_hertz = 1.0f;
    m_dampingRatio = 0.1f;

    b2DistanceJointDef jd;
    jd.Initialize(springbox, m_box, b2Vec2(1.0f, 23.5f),
        boxBodyDef.position);
    jd.collideConnected = true; // In this case we decide to
        allow the bodies to collide.
    m_length = jd.length;
    m_minLength = 2.0f; // the relaxed length of the spring:

```

```

        m_minLength
        m_maxLength = 14.0f;
        b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
                           m_dampingRatio, jd.bodyA, jd.bodyB);

        m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
        m_joint->SetMinLength(m_minLength);
        m_joint->SetMaxLength(m_maxLength);

        m_time = 0.0f;
    }
    b2Body* m_box;
    b2DistanceJoint* m_joint;
    float m_length;
    float m_time;
    float m_minLength;
    float m_maxLength;
    float m_hertz;
    float m_dampingRatio;

    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_W:
                m_box->ApplyForceToCenter(b2Vec2(0.0f, 10000.0f), true
                                         );
                break;
            case GLFW_KEY_S:
                m_box->ApplyForceToCenter(b2Vec2(0.0f, -9500.0f),
                                         true);
                break;
            case GLFW_KEY_T:
                m_time = 0.0f;
                break;
        }
    }
    void UpdateUI() override
    {
        ImGui::SetNextWindowPos(ImVec2(10.0f, 200.0f));
        ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
        ImGui::Begin("Joint Controls", nullptr,
                    ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

        if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0
                               f"))
        {
            m_length = m_joint->SetLength(m_length);
        }
    }
}

```

```

        }

        if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"))
        {
            float stiffness;
            float damping;
            b2LinearStiffness(stiffness, damping, m_hertz,
                m_dampingRatio, m_joint->GetBodyA(), m_joint->
                GetBodyB());
            m_joint->SetStiffness(stiffness);
            m_joint->SetDamping(damping);
        }

        if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
            , 2.0f, "%.1f"))
        {
            float stiffness;
            float damping;
            b2LinearStiffness(stiffness, damping, m_hertz,
                m_dampingRatio, m_joint->GetBodyA(), m_joint->
                GetBodyB());
            m_joint->SetStiffness(stiffness);
            m_joint->SetDamping(damping);
        }

        ImGui::End();
    }
    void Step(Settings& settings) override
    {
        b2MassData massData = m_box->GetMassData();
        b2Vec2 position = m_box->GetPosition();
        b2Vec2 velocity = m_box->GetLinearVelocity();
        m_time += 1.0f / 60.0f; // assuming we are using frequency of
        60 Hertz
        float k;
        float m = massData.mass;
        float g = 9.8;
        float y = 23.5 - m_minLength - 19.5;
        // y = the position at which when we place the mass it would
        // not move / equilibrium position
        // y = y position of the ceiling - m_minLength - initial y
        // position of the mass
        k = (m*g)/y;
        float y_eq;
        y_eq = -m*g/k;

        g_debugDraw.DrawString(5, m_textLine, "Press W to apply force"
    }
}

```

```

        10N upward / S to apply force 10 N downward");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Press T to reset time"
);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
f", m_time);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Mass position = (%4.1f
, %4.1f)", position.x, position.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Mass velocity = (%4.1f
, %4.1f)", velocity.x, velocity.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Mass = %.6f", massData
.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "The spring constant, k
= %4.1f", k);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "The position at which
the spring exerts no force / mass hanging, y = 0");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Equilibrium position
for the mass, y = -mg/k= %4.1f", y_eq);
m_textLine += m_textIncrement;
// Print the result in every time step then plot it into
graph with either gnuplot or anything

printf("%4.2f\n", position.y);

Test::Step(settings);
}
static Test* Create()
{
    return new VerticalspringTest;
}

static int testIndex = RegisterTest("Oscillations", "Vertical Spring Test",
VerticalspringTest::Create);

```

C++ Code 67: *tests/verticalspring_test.cpp "Simple Vertical Spring Box2D"*

Some explanations for the codes:

- It is said that distance joint can be used as a spring. Thus to create a distance joint that can act like a spring / "hallucination" spring, which does not look like a spring, but acts like a

spring. We set the **m_minLength** to be 2, since the problem stated that the relaxed length is 2 m for the spring (the length when nothing is attached on the spring).

```
// Make a distance joint for the box / ball with the static box
// above
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 23.5f), boxBodyDef
    .position);
jd.collideConnected = true; // In this case we decide to allow
    the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f; // the relaxed length of the spring:
    m_minLength
m_maxLength = 14.0f;
b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
    m_dampingRatio, jd.bodyA, jd.bodyB);

m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
m_joint->SetMinLength(m_minLength);
m_joint->SetMaxLength(m_maxLength);
```

The distance joint can also be made soft, like a spring-damper connection. Softness is achieved by tuning two constants in the definition: frequency and damping ratio. Think of the frequency as the frequency of a harmonic oscillator (like a guitar string). The frequency is specified in Hertz. Typically the frequency should be less than a half the frequency of the time step. So if you are using a 60Hz time step, the frequency of the distance joint should be less than 30Hz. The reason is related to the Nyquist frequency.

The damping ratio is non-dimensional and is typically between 0 and 1, but can be larger. At 1, the damping is critical (all oscillations should vanish).

```
jointDef.frequencyHz = 4.0f;
jointDef.dampingRatio = 0.5f;
b2LinearStiffness(jointDef.stiffness, jointDef.damping,
    frequencyHz, dampingRatio, jointDef.bodyA, jointDef.bodyB);
```

To plot the $x(t)$ with respect to time and see the amplitude, we need to plot the y position after we push the mass downward with a force, I choose a force of 9500 N, that you can apply when you press "S".

Now, at the working directory, open the testbed with this command:

./testbed > verticalspring.txt

then see it at **(current working directory)/verticalspring.txt**. You need to clean up a bit and delete the strings at the beginning of the textfile so it will only left you with numbers only in column format.

To plot it you can use gnuplot. Open terminal from the directory that contain the "vertical-spring.txt" and type:

```
gnuplot
set xlabel "time"
plot "verticalspring.txt" using 1 title "x(t)" with lines
```

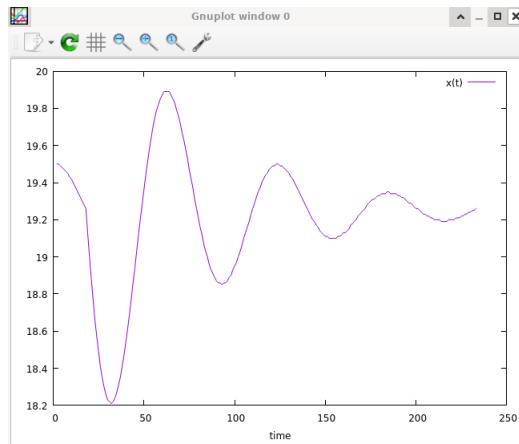


Figure 17.12: The amplitude of this problem of vertical spring-mass system is 1 m, the equilibrium position is at $y = 19.5$ but the mass fell a bit lower to 19.2 due to gravity, the relaxed length of the spring is at $y = 21.5$. When we stretched the mass downward for 1 meter then it will oscillates till it comes back to its equilibrium position again. The x axis represents the time in 1/60 seconds

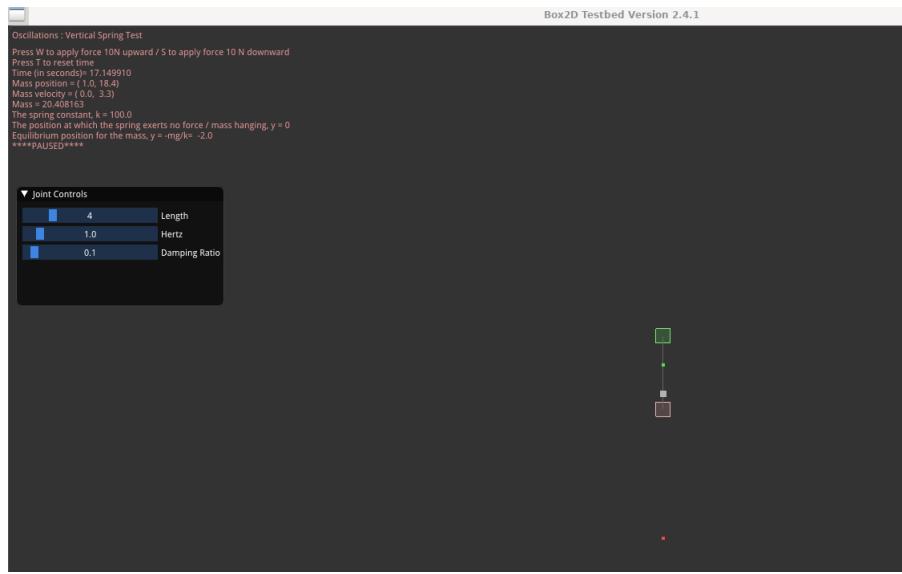


Figure 17.13: The vertical spring-mass system simulation for this current problem (the current simulation code can be located in: `DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/verticalspring_test.cpp`).

XII. OSCILLATION OF A SPRING-MASS SYSTEM

Oscillations resides as chapter 15 for Fundamental of Physics book [6], this explains why we are going to examine the second order differential equation that governs this spring-mass system. The differential equation describing a spring mass system is

$$m \frac{d^2x}{dt^2} = -kx$$

with the restoring force is proportional to the stretching of the spring. A differential equation that differentiating the dependent variable twice is called second-order differential equation(x is the dependent variable, while t is an independent variable), it is homogeneous and linear differential equation, you can read [1] to know how to solve that differential equation. The general solution for the second order differential equation is

$$x = c_1 \cos \omega t + c_2 \sin \omega t \quad (17.25)$$

with c_1 and c_2 are constants that can be determined with initial values or conditions, where

$$\omega^2 = \frac{k}{m}$$

This second-order ordinary differential equation has the general form

$$\frac{d^2x}{dt^2} = f \left(t, x, \frac{dx}{dt} \right)$$

where f is some given function, in spring-mass system case $f = -\frac{kx}{m}$.

The initial conditions for second-order differential equations of the spring-mass system are usually in the form of these

$$\begin{aligned}x(0) &= x_0 \\x'(0) &= x'_0\end{aligned}$$

where x_0 and x'_0 are constants that represent the value of x at time 0, and the value of $x' = \frac{dx}{dt} = v$ at time 0, respectively.

The general solution of a second-order linear homogeneous differential equation is a linear combination of two homogeneous solutions. For constant coefficient differential equations, the homogeneous solutions are usually in the form of simple exponentials, e^{rt} . We will explore about this, now let's go back to the differential equation:

$$\frac{d^2x}{dt^2} = -\frac{k}{m}x$$

we see that $\frac{k}{m}$ can be regarded as constant, where $\omega^2 = \frac{k}{m}$, now what is the function that the second derivative is itself? None other than exponential function, we derive e^x with respect to x for thousand of times it will still be e^x , it is amazing as it is a transcendental function and I spend more than 1 month just to summarize that chapter in Calculus. The application for exponential function is enormous as compound interest uses exponential as well.

Back to the differential equation, with $x = x(t)$, $\frac{dx}{dt} = x'(t)$, $\frac{d^2x}{dt^2} = x''(t)$ we will get

$$\begin{aligned}x'' &= -\omega^2x \\x'' + \omega^2x &= 0\end{aligned}$$

we can see it in the form of $ax'' + bx' + cx = 0$ with $a = 1, b = 0, c = \omega^2$, now if we seek solutions of the form $x(t) = e^{rt}$, based on [1], then r must be a root of the characteristic equation, the characteristic equation for that is

$$\begin{aligned}r^2 + \omega^2 &= 0 \\r^2 &= -\omega^2 \\r &= \sqrt{-\omega^2} \\r &= \pm i\omega\end{aligned}$$

so its roots are

$$r_1 = i\omega, \quad r_2 = -i\omega$$

if the roots r_1 and r_2 are conjugate complex numbers $r_{1,2} = \pm i\omega$, which occurs whenever the discriminant $b^2 - 4ac$ of the $ax'' + bx' + cx = 0$ is negative ($0 - ((4)(1)\omega^2) < 0$), then the general solution is

$$\begin{aligned}x(t) &= c_1 e^{r_1 t} + c_2 e^{r_2 t} \\x(t) &= c_1 e^{i\omega t} + c_2 e^{-i\omega t}\end{aligned}$$

with c_1 and c_2 are constants. We know that from the Euler 'formula,

$$\begin{aligned}e^{i\omega t} &= \cos(\omega t) + i \sin(\omega t) \\e^{-i\omega t} &= \cos(\omega t) - i \sin(\omega t)\end{aligned}$$

thus

$$\begin{aligned}x(t) &= c_1 e^{i\omega t} + c_2 e^{-i\omega t} \\&= c_1 (\cos(\omega t) + i \sin(\omega t)) + c_2 (\cos(\omega t) - i \sin(\omega t)) \\&= (c_1 + c_2) \cos(\omega t) + i(c_1 - c_2) \sin(\omega t) \\x(t) &= d_1 \cos(\omega t) + d_2 \sin(\omega t)\end{aligned}$$

with the constants d_1 and d_2 defined by

$$\begin{aligned}d_1 &= c_1 + c_2 \\d_2 &= i(c_1 - c_2)\end{aligned}$$

The constants d_1 and d_2 are arbitrary since given any value of d_1 and d_2 , there exists values of c_1 and c_2 , namely

$$\begin{aligned}c_1 &= \frac{1}{2}(d_1 - id_2) \\c_2 &= \frac{1}{2}(d_1 + id_2)\end{aligned}$$

Definition 17.1: Equivalent of Linear Combination for $\sin(\omega t)$ and $\cos(\omega t)$

An arbitrary linear combination of $e^{i\omega t}$ and $e^{-i\omega t}$,

$$x(t) = c_1 e^{i\omega t} + c_2 e^{-i\omega t}$$

is equivalent to an arbitrary linear combination of $\cos \omega t$ and $\sin \omega t$,

$$x(t) = d_1 \cos(\omega t) + d_2 \sin(\omega t)$$

with

$$\begin{aligned}d_1 &= c_1 + c_2 \\d_2 &= i(c_1 - c_2)\end{aligned}$$

$$\begin{aligned}c_1 &= \frac{1}{2}(d_1 - id_2) \\c_2 &= \frac{1}{2}(d_1 + id_2)\end{aligned}$$

and the Euler 'formula,

$$\begin{aligned}e^{i\omega t} &= \cos(\omega t) + i \sin(\omega t) \\e^{-i\omega t} &= \cos(\omega t) - i \sin(\omega t)\end{aligned}$$

After long deriving the formula, we should now be able to state that the general solution of

$$m \frac{d^2 x}{dt^2} = -kx$$

is

$$x(t) = d_1 \cos(\omega t) + d_2 \sin(\omega t)$$

where $\omega = \sqrt{\frac{k}{m}}$ and d_1 and d_2 are arbitrary constants. The general solution is a linear combination of two oscillatory functions, a cosine and a sine. An equivalent expression for the solution is

$$x(t) = A \sin(\omega t + \phi_0) \quad (17.26)$$

This is shown by noting

$$\sin(\omega t + \phi_0) = \sin \omega t \cos \phi_0 + \cos \omega t \sin \phi_0$$

in which case This is shown by noting

$$\begin{aligned} d_1 &= A \sin \phi_0 \\ d_2 &= A \cos \phi_0 \end{aligned}$$

If you are given d_1 and d_2 , you can determine both A and ϕ_0 , and using some trigonometric properties $\sin^2 \phi_0 + \cos^2 \phi_0 = 1$ results in an equation for A^2

$$\begin{aligned} A &= (d_1^2 + d_2^2)^{1/2} \\ \phi_0 &= \tan^{-1} \left(\frac{d_1}{d_2} \right) \end{aligned}$$

The expression, $x(t) = A \sin(\omega t + \phi_0)$, is especially convenient for sketching the displacement as a function of time. It shows that the sum of any multiple of $\cos \omega t$ plus any multitude of $\sin \omega t$ is itself a sinusoidal function.

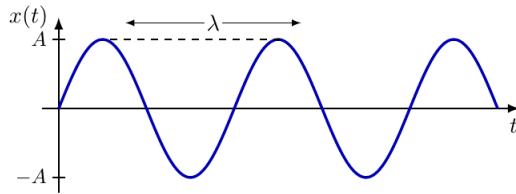


Figure 17.14: The period and amplitude of spring-mass system oscillation.

- A is called the amplitude of the oscillation, it can be easily computed from the above equation if d_1 and d_2 are known.
- The phase of oscillation is $\omega t + \phi_0$, with ϕ_0 being the phase at $t = 0$.
- The mass after reaching its maximum displacement (x largest), returns to the same position T units of time later.
- The entire oscillation repeats itself every T units of time, T is called the period of oscillation.
- Larger k will make shorter period of T .

This motion is referred to as simple harmonic motion. The mass oscillates sinusoidally around the equilibrium position $x = 0$. The solution is periodic in time.

Mathematically, a function $f(t)$ is said to be periodic with period T if

$$f(t + T) = f(t)$$

To determine the period T , we recall that the trigonometric functions are periodic with period 2π . Thus, for a complete oscillation, as t increases to $t + T$

$$[(\omega t + T) + \phi_0] - [(\omega t + \phi_0)] = 2\pi$$

Consequently the period is

$$T = \frac{2\pi}{\omega} = 2\pi\sqrt{\frac{m}{k}} \quad (17.27)$$

with ω as the circular frequency, or the number of periods in 2π units of time:

$$\omega = \frac{2\pi}{T} = \sqrt{\frac{k}{m}} \quad (17.28)$$

The number of oscillations in one unit of time is the frequency f ,

$$f = \frac{1}{T} = \frac{\omega}{2\pi} = \frac{1}{2\pi\sqrt{\frac{k}{m}}}$$

with f is measured in cycles per second (Hertz). Since a spring-mass system normally oscillates with frequency $\frac{1}{2\pi}\sqrt{\frac{k}{m}}$, this value is referred to as the natural frequency of a spring-mass system of mass m and spring constant k . Other physical systems have natural frequencies of oscillation, like a vibrating string, you might love to learn about wave equation, it requires knowledge in partial differential equation.

Now, we are going to dig onto the initial value problem, remember that

$$x(t) = d_1 \cos(\omega t) + d_2 \sin(\omega t) \quad (17.29)$$

is the general solution of the differential equation describing a spring-mass system.

$$m \frac{d^2x}{dt^2} = -kx \quad (17.30)$$

where d_1 and d_2 are arbitrary constants and $\omega = \sqrt{\frac{k}{m}}$. We can determine the constants d_1 and d_2 from the initial conditions of the spring-mass system. In fact for a spring-mass system, the two parameters k and m are not important, it is their ratio k/m that is important.

One way to initiate motion in a spring-mass system is to strike the mass, or to pull/push the mass to some position x_0 and then let go, thus

$$x(0) = x_0$$

and at $t = 0$, the velocity of the mass, $\frac{dx}{dt}$, is zero,

$$\frac{dx}{dt}(0) = 0$$

Thus, we have two initial conditions:

$$\begin{aligned} x(0) &= x_0 \\ x'(0) &= 0 \end{aligned}$$

Two initial conditions are necessary since the differential equation involves the second derivative in time. To solve this initial value problem, the arbitrary constants d_1 and d_2 are determined so the equation $x(t) = d_1 \cos(\omega t) + d_2 \sin(\omega t)$ satisfies the initial conditions.

To solve the initial value problem above:

$$\begin{aligned}x(t) &= d_1 \cos(\omega t) + d_2 \sin(\omega t) \\x(0) &= x_0 \\d_1 \cos(\omega(0)) + d_2 \sin(\omega(0)) &= x_0 \\d_1 &= x_0 \\ \frac{dx(t)}{dt} &= -d_1\omega \sin(\omega t) + d_2\omega \cos(\omega t) \\x'(t) &= -d_1\omega \sin(\omega t) + d_2\omega \cos(\omega t) \\x'(0) &= -d_1\omega \sin(\omega(0)) + d_2\omega \cos(\omega(0)) \\d_2\omega &= 0 \\d_2 &= 0\end{aligned}$$

We will obtain the solution from the initial value problem:

$$x(t) = x_0 \cos \omega t \quad (17.31)$$

we can also use another initial conditions, where $v(0) = v_0 \neq 0$, where there is an initial velocity when the mass is being let go after stretched / contracted.

XIII. A Two-MASSES OSCILLATOR

The force of a spring-mass system can be approximated as being simply proportional to the stretching of the spring. Now, instead of attaching a spring and a mass to a rigid wall, we attach a spring and a mass to another mass, which is also free to move. Let us assume that the two masses are m_1 and m_2 , while the connecting spring is known to have an unstretched length l and spring constant k .

We must formulate Newton's law of motion for each mass. The force on each mass equals its mass times its acceleration. To obtain acceleration we will need the position of each mass, x_1 and x_2 are the distances each mass is from a fixed origin.

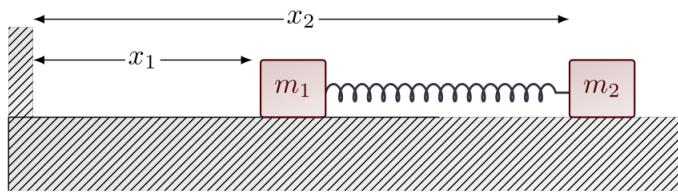


Figure 17.15: The illustration for two-masses-spring system.

Although the unstretched length of the spring is l , it is not necessary that $x_2 - x_1 = l$, for in many circumstances the spring may be stretched or compressed.

We may impose initial conditions such that initially

$$x_2 - x_1 \neq l$$

Now, from Newton's law of motion, it follows that if F_1 is the force on mass m_1 and if F_2 is the force on mass m_2 , then

$$\begin{aligned} m_1 \frac{d^2 x_1}{dt^2} &= F_1 \\ m_2 \frac{d^2 x_2}{dt^2} &= F_2 \end{aligned}$$

To complete the derivation of the equations of motion, we must determine the two forces, F_1 and F_2 .

The only force on each mass is due to the spring. Each force is an application of Hooke's law; the force is proportional to the stretching of the spring. The stretching of the spring is the length of the spring $x_2 - x_1$ minus the unstretched length l , hence

$$x_2 - x_1 - l$$

The magnitude of the force is just the spring constant k times the stretching, and the direction of the force should be carefully analyzed.

- If the spring is stretched ($x_2 - x_1 - l > 0$), then the mass m_1 is being pulled to the right.
- If the spring is contracted ($x_2 - x_1 - l < 0$), then the mass m_1 is being pushed to the left.

The force on mass m_1 :

$$m_1 \frac{d^2 x_1}{dt^2} = k(x_2 - x_1 - l) \quad (17.32)$$

$$m_2 \frac{d^2 x_2}{dt^2} = -k(x_2 - x_1 - l) \quad (17.33)$$

Although the magnitude of the force on m_2 is the same as m_1 the spring force acts in the opposite direction.

Now, we are going to combine two coupled second order ordinary differential equations involving two unknowns x_1 and x_2 .

$$\begin{aligned} m_1 \frac{d^2 x_1}{dt^2} + m_2 \frac{d^2 x_2}{dt^2} &= 0 \\ \frac{d^2}{dt^2} (m_1 x_1 + m_2 x_2) & \end{aligned}$$

thus, we will have the center of mass of the system:

$$\frac{m_1 x_1 + m_2 x_2}{m_1 + m_2}$$

the center of mass does not accelerate, but moves at a constant velocity (determined from initial conditions). If we view the system of two masses and a spring as a single entity, then there are no external forces on it, the system obeys Newton's first law and will not accelerate. When there is no acceleration, the system of two masses moves at a constant velocity.

Now, let z be the stretching of the spring between two masses,

$$\begin{aligned}\frac{d^2}{dt^2}(x_2 - x_1) &= -\frac{k}{m_2}(x_2 - x_1 - l) - \frac{k}{m_1}(x_2 - x_1 - l) \\ z &= x_2 - x_1 - l \\ \frac{d^2z}{dt^2} &= x_2 - x_1\end{aligned}$$

thus

$$\frac{d^2z}{dt^2}(x_2 - x_1) = -k \left(\frac{1}{m_1} + \frac{1}{m_2} \right) z$$

We see that the stretching of the spring executes simple harmonic motion. The circular frequency is $\sqrt{k \left(\frac{1}{m_1} + \frac{1}{m_2} \right)}$.

The two-masses spring system is the same type of oscillation if certain mass m were placed on the same spring

$$\frac{1}{m} = \frac{1}{m_1} + \frac{1}{m_2} \quad (17.34)$$

This mass m is less than either m_1 or m_2 , and can be called reduced mass with

$$m = \frac{m_1 m_2}{m_1 + m_2} \quad (17.35)$$

Attaching a spring-mass system to a movable mass reduces the effective mass. The stretching of the spring executes simple harmonic motion as though a smaller mass was attached, the entire system may move, i.e., the center of mass moves at a constant velocity.

XIV. SIMULATION OF TWO MASSES SPRING SYSTEM WITH Box2D

This example is taken from exercise 9.7 "Mechanical Vibrations" chapter from [5].

Consider a mass m_1 attached to a spring (of unstretched length d) and pulled by a constant force $F_2 = m_2 g$.

- (a) Suppose that the system is in equilibrium when $x = L$. Is $L > d$ or is $L < d$? If L and d are known, what is the spring constant k ?
- (b) If the system is at rest in the position $x = L$ and the mass m_2 is suddenly removed (for example, by cutting the string connecting m_1 and m_2), then what is the period and amplitude of oscillation of m_1 ?

Solution:

- (a) First, we need to use Hooke's law, that is

$$F = -kx$$

with F is the force of the string pulling / pushing the mass back to its' rest position. This force is always have opposite direction with the force that pull or push the mass away from

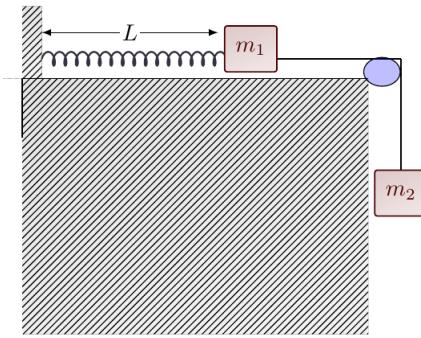


Figure 17.16: The two masses spring-mass system illustration, when there is no mass m_2 , the length of unstretched spring connected to mass m_1 is d .

its' rest / equilibrium position.

The equilibrium position for this system is where the leftward Hooke's law balances the force from the mass m_2 , which is $F = m_2g$, which corresponds to the spring being stretched to L distance.

For m_1 when at rest, we will have

$$F_1 = -k(\Delta x) = -k(x_1 - d)$$

with x_1 is the displacement of the mass m_1 from d , the unstretched length of the spring that is attached to m_1 . We can assign the number such as $d = 0$, hence a pull toward m_1 will make $x_1 > 0$, a push toward m_1 will make $x_1 < 0$

Adding another mass, m_2 that pull the mass m_1 , then we know that there will be a pull of mass m_1 from mass m_2 that makes the spring being stretched to $x = L$, by using Newton's second law for equilibrium system

$$\begin{aligned} \sum F_1 &= m_1a \\ T - kx &= m_1a \\ T &= m_1a + kx \end{aligned}$$

The cord tension T is positive, since it is having the same direction as the acceleration, a , while $-kx$ the spring force is negative since it is a force that will pull the mass m_1 back to its' equilibrium position, and has opposite direction of the acceleration, a .

$$\begin{aligned} \sum F_2 &= m_2a \\ m_2g - T &= m_2a \\ T &= m_2g - m_2a \end{aligned}$$

The weight force due to gravity is positive due to the same direction with the acceleration, a , while the tension on the cord T has the opposite direction toward the acceleration, a . In summary, F_1 is the force working on mass m_1 and F_2 is the force working on mass m_2 , and T

is the cord tension that connecting m_1 and m_2 , hence

$$\begin{aligned} m_1a + kx &= m_2g - m_2a \\ kx &= -m_2a - m_1a + m_2g \\ k &= \frac{-(m_1 + m_2)a + m_2g}{x} \end{aligned}$$

when the system is in equilibrium, the acceleration should be $a = 0$, thus

$$\begin{aligned} k &= \frac{-(m_1 + m_2)(0) + m_2g}{x} \\ k &= \frac{m_2g}{x} \\ k &= \frac{m_2g}{L} \end{aligned}$$

because the system is in equilibrium when $x = L$. Thus, $L > d$.

- (b) Destroy the pulley joint, see what happens to box 1

```
#include "settings.h"
#include "test.h"
#include "imgui/imgui.h"
#include <iostream>

class SpringTwomasses : public Test
{
public:
    SpringTwomasses()
    {
        b2Body* ground = NULL;
        // Create the pulley
        b2BodyDef bdp;
        ground = m_world->CreateBody(&bdp);

        b2CircleShape circle;
        circle.m_radius = 0.5f;

        circle.m_p.Set(12.0f, 0.5f); // circle with center of
                                    (12, 0.5)
        ground->CreateFixture(&circle, 0.0f);
    }

    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(-46.0f, 0.0f), b2Vec2(12.0f,
                                                0.0f));
}
```

```

        ground->CreateFixture(&shape, 0.0f);
    }
{
    b2BodyDef bd;
    ground = m_world->CreateBody(&bd);

    b2EdgeShape shape;
    shape.SetTwoSided(b2Vec2(12.0f, 0.0f), b2Vec2(12.0f
        ,-20.0f));
    ground->CreateFixture(&shape, 0.0f);
}
// Create a static body as the box for the spring
b2BodyDef bd1;
bd1.type = b2_staticBody;
bd1.angularDamping = 0.1f;

bd1.position.Set(1.0f, 0.5f);
b2Body* springbox = m_world->CreateBody(&bd1);

b2PolygonShape shape1;
shape1.SetAsBox(0.5f, 5.5f);
springbox->CreateFixture(&shape1, 5.0f);

// Create the left box connected to the spring as the movable
// object
b2PolygonShape leftboxShape;
leftboxShape.SetAsBox(0.5f, 0.5f);

b2FixtureDef leftboxFixtureDef;
leftboxFixtureDef.restitution = 0.75f;
leftboxFixtureDef.density = 7.3f; // this will affect the box
// mass
leftboxFixtureDef.friction = 0.1f;
leftboxFixtureDef.shape = &leftboxShape;

b2BodyDef leftboxBodyDef;
leftboxBodyDef.type = b2_dynamicBody;
leftboxBodyDef.position.Set(5.0f, 0.5f);

m_box = m_world->CreateBody(&leftboxBodyDef);
b2Fixture *leftboxFixture = m_box->CreateFixture(&
    leftboxFixtureDef);
//m_box->SetGravityScale(-7); // negative means it will goes
// upward, positive it will goes downward

// Create the box hanging on the right
b2PolygonShape boxShape2;

```

```

boxShape2.SetAsBox(0.5f, 0.5f); // width and length of the
                                box

b2FixtureDef boxFixtureDef2;
boxFixtureDef2.restitution = 0.75f;
boxFixtureDef2.density = 3.3835f; // this will affect the box
                                mass, mass = density*5.9969
boxFixtureDef2.friction = 0.3f;
boxFixtureDef2.shape = &boxShape2;

b2BodyDef boxBodyDef2;
boxBodyDef2.type = b2_dynamicBody;
boxBodyDef2.position.Set(12.8f, -1.5f);

m_boxr = m_world->CreateBody(&boxBodyDef2);
b2Fixture *boxFixture2 = m_boxr->CreateFixture(&
                                                boxFixtureDef2);

// Make a distance joint for the box / ball with the static
// box above
m_hertz = 1.0f;
m_dampingRatio = 0.1f;

b2DistanceJointDef jd;
jd.Initialize(springbox, m_box, b2Vec2(1.0f, 0.5f),
              leftboxBodyDef.position);
jd.collideConnected = true; // In this case we decide to
                            allow the bodies to collide.
m_length = jd.length;
m_minLength = 2.0f;
m_maxLength = 10.0f;
b2LinearStiffness(jd.stiffness, jd.damping, m_hertz,
                   m_dampingRatio, jd.bodyA, jd.bodyB);

m_joint = (b2DistanceJoint*)m_world->CreateJoint(&jd);
m_joint->SetMinLength(m_minLength);
m_joint->SetMaxLength(m_maxLength);

// Create the Pulley
b2PulleyJointDef pulleyDef;
b2Vec2 anchor1(5.5f, 0.5f); // the position of the end string
                            of the left cord is (5.5,0.5) connecting to the left box
b2Vec2 anchor2(12.8f, -1.0f); // the position of the end
                                string of the right cord is (14.0f,-1) connecting to the
                                right hanging box
b2Vec2 groundAnchor1(11.5f, 0.5f ); // the string of the cord
                                is tightened at (11.5, 0.5)
b2Vec2 groundAnchor2(12.5f, 0.5f); // the string of the cord

```

```

        is tightened at (12.5, 0.5)
        // the last float is the ratio
        pulleyDef.Initialize(m_box, m_boxr, groundAnchor1,
            groundAnchor2, anchor1, anchor2, 1.0f);

        m_joint1 = (b2PulleyJoint*)m_world->CreateJoint(&pulleyDef);

        m_time = 0.0f;
    }
    b2Body* m_box;
    b2Body* m_boxr;
    b2DistanceJoint* m_joint;
    b2PulleyJoint* m_joint1;
    float m_length;
    float m_time;
    float m_minLength;
    float m_maxLength;
    float m_hertz;
    float m_dampingRatio;

    void Keyboard(int key) override
    {
        switch (key)
        {
            case GLFW_KEY_A:
                //m_box->SetLinearVelocity(b2Vec2(30.0f, 0.0f));
                m_box->ApplyForceToCenter(b2Vec2(-8000.0f, 0.0f),
                    true);
                break;
            case GLFW_KEY_S:
                //m_box->SetLinearVelocity(b2Vec2(-30.0f, 0.0f));
                m_box->ApplyForceToCenter(b2Vec2(-5000.0f, 0.0f),
                    true);
                break;
            case GLFW_KEY_D:
                m_box->ApplyForceToCenter(b2Vec2(5000.0f, 0.0f), true)
                    ;
                break;
            case GLFW_KEY_F:
                m_box->ApplyForceToCenter(b2Vec2(8000.0f, 0.0f), true)
                    ;
                break;
            case GLFW_KEY_G:
                m_box->ApplyForceToCenter(b2Vec2(12000.0f, 0.0f), true)
                    );
                break;
        }
    }
}

```

```

void UpdateUI() override
{
    ImGui::SetNextWindowPos(ImVec2(10.0f, 300.0f));
    ImGui::SetNextWindowSize(ImVec2(260.0f, 150.0f));
    ImGui::Begin("Joint Controls", nullptr,
                 ImGuiWindowFlags_NoMove | ImGuiWindowFlags_NoResize);

    if (ImGui::SliderFloat("Length", &m_length, 0.0f, 20.0f, "%.0f"))
    {
        m_length = m_joint->SetLength(m_length);
    }

    if (ImGui::SliderFloat("Hertz", &m_hertz, 0.0f, 10.0f, "%.1f"))
    {
        float stiffness;
        float damping;
        b2LinearStiffness(stiffness, damping, m_hertz,
                           m_dampingRatio, m_joint->GetBodyA(), m_joint->
                           GetBodyB());
        m_joint->SetStiffness(stiffness);
        m_joint->SetDamping(damping);
    }

    if (ImGui::SliderFloat("Damping Ratio", &m_dampingRatio, 0.0f
                           , 2.0f, "%.1f"))
    {
        float stiffness;
        float damping;
        b2LinearStiffness(stiffness, damping, m_hertz,
                           m_dampingRatio, m_joint->GetBodyA(), m_joint->
                           GetBodyB());
        m_joint->SetStiffness(stiffness);
        m_joint->SetDamping(damping);
    }

    ImGui::End();
}

void Step(Settings& settings) override
{
    b2MassData massData = m_box->GetMassData();
    b2Vec2 position = m_box->GetPosition();
    b2Vec2 velocity = m_box->GetLinearVelocity();
    b2MassData massDatar = m_boxr->GetMassData();
    b2Vec2 positionr = m_boxr->GetPosition();
    b2Vec2 velocityr = m_boxr->GetLinearVelocity();
    m_time += 1.0f / 60.0f; // assuming we are using frequency of
}

```

```

60 Hertz

g_debugDraw.DrawString(5, m_textLine, "Press A/S/D/F/G to
    apply different force to the box");
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Time (in seconds)= %.6
    f", m_time);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Left Mass position =
    (%4.1f, %4.1f)", position.x, position.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Left Mass velocity =
    (%4.1f, %4.1f)", velocity.x, velocity.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Right Mass position =
    (%4.1f, %4.1f)", positionr.x, positionr.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Right Mass velocity =
    (%4.1f, %4.1f)", velocityr.x, velocityr.y);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Left Mass = %.6f",
    massData.mass);
m_textLine += m_textIncrement;
g_debugDraw.DrawString(5, m_textLine, "Right Mass = %.6f",
    massDataar.mass);
m_textLine += m_textIncrement;
// Print the result in every time step then plot it into
graph with either gnuplot or anything

printf("%4.2f\n", position.x);

Test::Step(settings);
}
static Test* Create()
{
    return new SpringTwomasses;
}

static int testIndex = RegisterTest("Oscillations", "Spring Two Masses",
    SpringTwomasses::Create);

```

C++ Code 68: *tests/spring_twomasses.cpp* "Two Masses Spring-Mass System Box2D"

Some explanations for the codes:

-

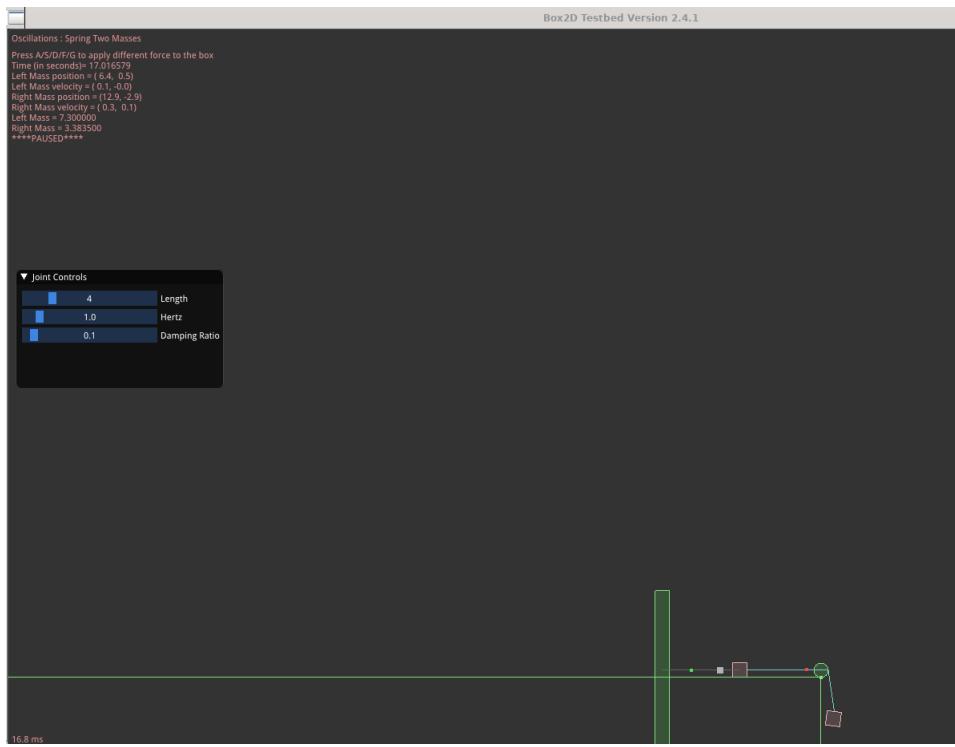


Figure 17.17: The two masses spring-mass system simulation , besides keyboard press events you can click the box with mouse and drag it toward positive or negative x axis to see it oscillating and affecting the other body that is hanging on the right (the current simulation code can be located in: [DFSimulatorC/Source Codes/C++/DianFreya-box2d-testbed/tests/spring_twomasses.cpp](#)).

XV. FRICTION AND OSCILLATIONS

The amplitude of a real-life spring-mass system when oscillating will decrease over time, the mass will oscillates around its' equilibrium position with smaller and smaller magnitude until it stops.

For a spring-mass system, the friction between the mass and the surrounding air media cause the amplitude of the oscillation to diminish. When the spring is moving to the left, the friction force is moving to the right, and when the spring is moving to the right, the friction force is moving to the left.

- When the velocity is positive, $\frac{dx}{dt} > 0$, then the frictional force f_k must be negative, $f_k < 0$.
- When the velocity is negative, $\frac{dx}{dt} < 0$, then the frictional force f_k must be positive, $f_k > 0$.

To assume that the frictional force is linearly dependent on the velocity:

$$f_k = -c \frac{dx}{dt} \quad (17.36)$$

where c is a positive constant referred to as the friction coefficient. This force-velocity relationship is called a linear damping force; a damped oscillation is the same as an oscillations that decays.

Now, we will have the differential equation describing spring-mass system with a linear damping force and a linear restoring force as

$$m \frac{d^2x}{dt^2} + c \frac{dx}{dt} + kx = 0 \quad (17.37)$$

It must be verified that solutions to the second order differential equation above behave in a manner consistent with the real-life observations. There are a lot of forces that can make the spring-mass system decays, other than a linear damping force.

To solve the second order differential equation, we are going to use exponential again, e^{rt} , as two linearly independent solutions can almost be written in the form of exponential. Where r is the characteristic equation obtained by direct substitution.

$$mr^2 + cr + k = 0 \quad (17.38)$$

is the characteristic equation of $m \frac{d^2x}{dt^2} + c \frac{dx}{dt} + kx = 0$. The two roots of the characteristic equation are

$$r = \frac{-c \pm \sqrt{c^2 - 4mk}}{2m} \quad (17.39)$$

with c and mk must have the same dimension.

There are three cases:

- **Case I: Underdamped Oscillations ($c^2 < 4mk$)**

If $c^2 < 4mk$, then the coefficient of friction is small, thus the damping force is not particularly large. In this case, the roots of the characteristic equation are complex conjugates of each other(complex number)

$$r = -\frac{c}{2m} \pm i\omega$$

where

$$\omega = \frac{\sqrt{4mk - c^2}}{2m} = \sqrt{\frac{k}{m} - \frac{c^2}{4m^2}}$$

The general solution for this case is

$$\begin{aligned} x(t) &= ae^{(-\frac{c}{2m} + i\omega)t} + be^{(-\frac{c}{2m} - i\omega)t} \\ &= e^{-\frac{ct}{2m}} \left(ae^{i\omega t} + be^{-i\omega t} \right) \end{aligned} \quad (17.40)$$

An arbitrary linear combination of $e^{i\omega t}$ and $e^{-i\omega t}$ is equivalent to an arbitrary linear combination of $\cos \omega t$ and $\sin \omega t$, thus the motion of a linearly damped spring-mass system in the underdamped case is described by

$$\begin{aligned} x(t) &= e^{-\frac{ct}{2m}} (c_1 \cos \omega t + c_2 \sin \omega t) \\ &= Ae^{-\frac{ct}{2m}} \sin(\omega t + \phi_0) \end{aligned} \quad (17.41)$$

The solution is the product of an exponential and a sinusoidal function.

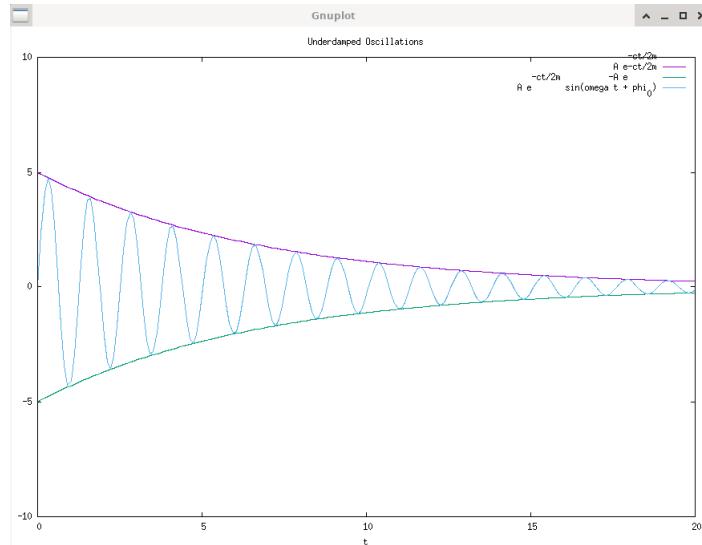


Figure 17.18: The exponential function $Ae^{-\frac{ct}{2m}}$ and its negative $-Ae^{-\frac{ct}{2m}}$ along with the solution $x(t) = Ae^{-\frac{ct}{2m}} \sin(\omega t + \phi_0)$ for the underdamped case (the code can be located in: `DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++/C++ Gnuplot/ch15-underdampedoscillation/main.cpp`).

- At the maximum value of the sinusoidal function, $x(t)$ equals the exponential alone.
- At the minimum value of the sinusoidal function, x equals minus the exponential.
- The exponential $Ae^{-\frac{ct}{2m}}$ is called the amplitude of the oscillation. Thus, the amplitude is exponentially decays.
- **Case II: Critically Damped Oscillations ($c^2 = 4mk$)**
When $c^2 = 4mk$, the spring-mass system is said to be critically damped. Mathematically, it makes both the exponential solutions become the same. But, from the physical point of view, this case is insignificant. This is because the quantities c , m , and k are all experimentally measured quantities. Any small deviation can shift the system's condition into overdamped or underdamped oscillation.

The solution for critically damped oscillation is

$$x(t) = e^{-\frac{ct}{2m}} (At + B) \quad (17.42)$$

The solution will returns to its equilibrium position as $t \rightarrow \infty$, since the exponential decay is much stronger than an algebraic growth.

- **Case III: Overdamped Oscillations ($c^2 > 4mk$)**

If the friction is sufficiently large, then

$$c^2 > 4mk$$

this is called as an overdamped system. The motion of the mass is no longer a decaying oscillation. Thus, the solution is

$$x(t) = c_1 e^{r_1 t} + c_2 e^{r_2 t} \quad (17.43)$$

where r_1 and r_2 are real and both negative,

$$r_1 = \frac{-c + \sqrt{c^2 - 4mk}}{2m}$$

$$r_2 = \frac{-c - \sqrt{c^2 - 4mk}}{2m}$$

If the friction is sufficiently large, we should expect that the mass decays to its equilibrium position quite quickly.

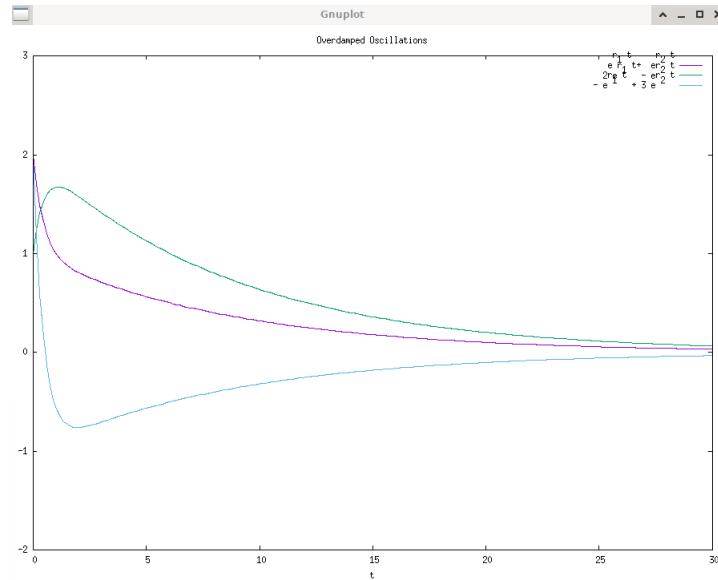


Figure 17.19: The solution for the motion of the mass $x(t) = c_1 e^{r_1 t} + c_2 e^{r_2 t}$ with different values of c_1 and c_2 for the overdamped case (the code can be located in: `DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++/C++ Gnuplot/ch15-overdampedoscillation/main.cpp`).

Chapter 18

DFSimulatorC++ XII: Waves

"Reality flows from cause to effect like a mathematical equation. Mortals can't comprehend this. They're pathetic" - Albert Silverberg

M^{omentum}

I. SIMULATION FOR MOMENTUM WITH Box2D

You need to copy from my repository' directory `../Source Codes/C++/DianFreya-box2d-testbed`, then go inside the directory and open the terminal then type:

```
mkdir build  
cd build  
cmake ..  
make  
./testbed
```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

C++ Code 69: `tests/projectile_motion.cpp` "Projectile Motion Box2D"

Some explanations for the codes:

- ---
- ---
- ---

Chapter 19

DFSimulatorC++ XIII: Temperature, Heat, and the First Law of Thermodynamics

"Reality flows from cause to effect like a mathematical equation. Mortals can't comprehend this. They're pathetic" - Albert Silverberg

M^{omentum}

I. SIMULATION FOR MOMENTUM WITH Box2D

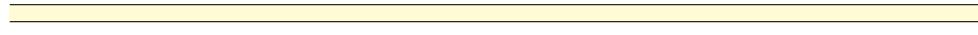
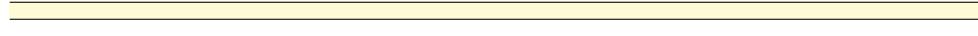
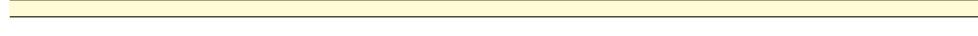
You need to copy from my repository' directory `../Source Codes/C++/DianFreya-box2d-testbed`, then go inside the directory and open the terminal then type:

```
mkdir build  
cd build  
cmake ..  
make  
./testbed
```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

C++ Code 70: `tests/projectile_motion.cpp` "Projectile Motion Box2D"

Some explanations for the codes:

- 
- 
- 

Chapter 20

DFSimulatorC++ XIV: The Kinetic Theory of Gases

"Reality flows from cause to effect like a mathematical equation. Mortals can't comprehend this. They're pathetic" - Albert Silverberg

M^{omentum}

I. SIMULATION FOR MOMENTUM WITH Box2D

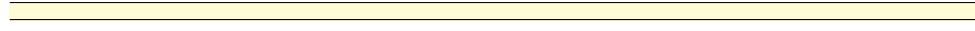
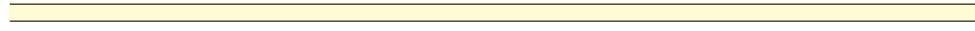
You need to copy from my repository' directory `../Source Codes/C++/DianFreya-box2d-testbed`, then go inside the directory and open the terminal then type:

```
mkdir build  
cd build  
cmake ..  
make  
./testbed
```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

C++ Code 71: `tests/projectile_motion.cpp` "Projectile Motion Box2D"

Some explanations for the codes:

- 
- 
- 

Chapter 21

DFSimulatorC++ XV: Entropy and the Second Law of Thermodynamics

"Reality flows from cause to effect like a mathematical equation. Mortals can't comprehend this. They're pathetic" - Albert Silverberg

M^{omentum}

I. SIMULATION FOR MOMENTUM WITH Box2D

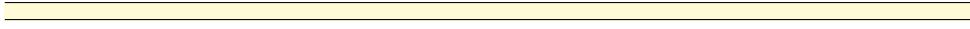
You need to copy from my repository' directory `../Source Codes/C++/DianFreya-box2d-testbed`, then go inside the directory and open the terminal then type:

```
mkdir build  
cd build  
cmake ..  
make  
./testbed
```

Look for the related simulation under the **Tests** tab on the right panel, then choose **Motion in 2D/Projectile Motion**.

C++ Code 72: `tests/projectile_motion.cpp` "Projectile Motion Box2D"

Some explanations for the codes:

- 
- 
- 

Chapter 22

DFSimulatorC++ XVI: Probability and Data Analysis

*"My only place is by your side. If you want to carry out your wish, I will follow you. That is my Nature." -
Sarah (Suikoden III)*

When we have movement and we

Chapter 23

DFSimulatorC++ XVII: Numerical Linear Algebra

"Plaisir d'amour, ne dure qu'un moment. Chagrin d'amour dure toute la vie." - Plaisir D'amour (Nana Mouskori)

When I read book "Engineering Circuit Analysis" [8] there are nonlinear circuits we encounter every day, for examples they capture and decode signals for TVs and radios, perform calculations hundreds of billions of times a second inside microprocessors, convert speech into electrical signals for transmission over fibre-optic cables as well as cellular networks. The problem is nonlinear systems are not easy to solve that is why we often use linearization for the nonlinear systems so it will be easy to solve that problem, even if no physical system (including electrical circuits) is ever perfectly linear. It is the linear approximations to physical electrical circuits that are used for the computation. This is why we need to learn some techniques in Numerical Linear Algebra, since every mathematical scientist needs to work effectively with vectors and matrices, that will expand more to functions and operators. Let the fun begins.

All the files and data used in this chapter can be found in this book repository:

[DFSimulatorC/Source Codes/C++/C++ Gnuplot SymbolicC++/ch23-Numerical Linear Algebra/](#)

I. MATRIX-VECTOR MULTIPLICATION

[DF*] Let x be an n -dimensional column vector and let A be an $m \times n$ matrix (m rows, n columns). Then the matrix-vector product b is a m -dimensional column vector such that

$$b = Ax$$

then b is a linear combination of the columns of A , with

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

we can then defined b as follows:

$$b_i = \sum_{j=1}^n a_{ij}x_j, \quad i = 1, \dots, m \quad (23.1)$$

Here b_i denotes the i th entry of b , a_{ij} denotes the i, j entry of A , and x_j denotes the j th entry of x .

[DF*] Few assumptions:

- All quantities belong to \mathbb{C} , the field of complex numbers.

[DF*] The map $x \mapsto Ax$ is linear, which means, for any $x, y \in \mathbb{C}^n$ and any $\alpha \in \mathbb{C}$,

$$\begin{aligned} A(x + y) &= Ax + Ay \\ A(\alpha x) &= \alpha Ax \end{aligned}$$

II. C++ COMPUTATION: A MATRIX TIMES A VECTOR

Let a_j denote the j th column of A , an m -vector, then

$$b = Ax = \sum_{j=1}^n x_j a_j \quad (23.2)$$

we see that b is expressed as a linear combination of the column a_j .

Now for the computation we can use this simple code from **bottomscience.com**, you can create this c++ file with name of **multiplicationofmatrices.cpp** or just copy the related code from the repository of this book.

```
#include <iostream>
using namespace std;

int main()
{
    int a[10][10], b[10][10], mult[10][10], r1, c1, r2, c2, i, j, k;

    cout << "Enter rows and columns for first matrix: ";
    cin >> r1 >> c1;
    cout << "Enter rows and columns for second matrix: ";
    cin >> r2 >> c2;
```

```
// If column of first matrix is not equal to row of second matrix,  
// ask the user to enter the size of matrix again.  
while (c1!=r2)  
{  
    cout << "Error! column of first matrix not equal to row of  
    second.";  
  
    cout << "Enter rows and columns for first matrix: ";  
    cin >> r1 >> c1;  
  
    cout << "Enter rows and columns for second matrix: ";  
    cin >> r2 >> c2;  
}  
  
// Storing elements of first matrix.  
cout << endl << "Enter elements of matrix 1:" << endl;  
for(i = 0; i < r1; ++i)  
for(j = 0; j < c1; ++j)  
{  
    cout << "Enter element a" << i + 1 << j + 1 << " : ";  
    cin >> a[i][j];  
}  
  
// Storing elements of second matrix.  
cout << endl << "Enter elements of matrix 2:" << endl;  
for(i = 0; i < r2; ++i)  
for(j = 0; j < c2; ++j)  
{  
    cout << "Enter element b" << i + 1 << j + 1 << " : ";  
    cin >> b[i][j];  
}  
  
// Initializing elements of matrix mult to 0.  
for(i = 0; i < r1; ++i)  
for(j = 0; j < c2; ++j)  
{  
    mult[i][j]=0;  
}  
  
// Multiplying matrix a and b and storing in array mult.  
for(i = 0; i < r1; ++i)  
for(j = 0; j < c2; ++j)  
for(k = 0; k < c1; ++k)  
{  
    mult[i][j] += a[i][k] * b[k][j];  
}
```

```

// Displaying the multiplication of two matrix.
cout << endl << "Output Matrix: " << endl;
for(i = 0; i < r1; ++i)
    for(j = 0; j < c2; ++j)
    {
        cout << " " << mult[i][j];
        if(j == c2-1)
            cout << endl;
    }

    return 0;
}

```

C++ Code 73: *multiplicationofmatrices.cpp* "Computation: Multiplication of matrices"

To compile it, type:

```

g++ -o main multiplicationofmatrices.cpp
./main

```

Now, we can try to have a matrix A of size 3×3 with vector x a column vector with size of 3.

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

To compute for b we just use the code above and we will obtain

$$b = \begin{bmatrix} 14 \\ 32 \\ 50 \end{bmatrix}$$

```

Enter rows and columns for first matrix: 3
3
Enter rows and columns for second matrix: 3
1

Enter elements of matrix 1:
Enter element a11 : 1
Enter element a12 : 2
Enter element a13 : 3
Enter element a21 : 4
Enter element a22 : 5
Enter element a23 : 6
Enter element a31 : 7
Enter element a32 : 8
Enter element a33 : 9

Enter elements of matrix 2:
Enter element b11 : 1
Enter element b21 : 2
Enter element b31 : 3

Output Matrix:
14
32
50

```

Figure 23.1: The computation for matrix A times a vector x , to obtain $b = Ax$.

III. C++ COMPUTATION: A MATRIX TIMES A VECTOR WITH DATA FROM TEXTFILE

The difference from the previous section is that we don't have to type one by one the entries of the matrix and the vector, if we have a textbox filled with columns of data, and we know the number of elements for the vector and the matrix size, we can just put it under the working directory along with the C++ code. We are also using **struct** and **pointers** here to take the vector elements and matrix indices from textbox that can then be processed or computed at the **int main()** / the main driver for C++.

In a working directory, create 2 textfiles, one for storing vector elements (**vector1.txt**), the other for storing matrix (**matrix1.txt**). Then, create a C++ file name **main.cpp**, the code is below, and it is also available at the repository.

```
#include <iostream>
#include <vector>
#include <cmath>

#include "gnuplot-iostream.h"

const int N = 4;

using std::cout;
using std::endl;
using namespace std;

void printArray(int** arr) {

    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
        {
            cout << arr[i][j] << ' ';
        }
        cout << endl;
    }
}

int* vec() {
    static int x[N];
    std::ifstream in("vector1.txt");
    float vectortiles[N];
    for (int i = 0; i < N; ++i)
    {
        in >> vectortiles[i];
        x[i] = vectortiles[i];
    }
    return x;
}
```

```
struct matrix
{
    int arr1[N][N];
};

struct matrix func(int N)
{
    struct matrix matrix_mem;
    std::fstream in("matrix1.txt");
    float matrxtiles[N][N];
    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
        {
            in >> matrxtiles[i][j];
            matrix_mem.arr1[i][j] = matrxtiles[i][j] ;
        }
    }
    return matrix_mem;
}

int main() {

    int* ptrvec;
    ptrvec = vec();
    cout << "Vector x is: " << endl;
    for (int i = 0; i < N; ++i)
    {
        cout <<ptrvec[i]<< ' ' << endl;
    }

    cout << "Matrix A is: " << endl;

    struct matrix a;
    a = func(N);
    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
        {
            cout << a.arr1[i][j]<< ' ';
        }
        cout << endl;
    }

    float b[N][N];
    float B[N];
```

```

        for (int i = 0; i < N; ++i)
        {
            B[i] = 0;
        }

        for (int i = 0; i < N; ++i)
        {
            for (int j = 0; j < N; ++j)
            {
                b[i][j] = 0;
            }
        }

        for (int i = 0; i < N; ++i)
        {
            for (int j = 0; j < N; ++j)
            {
                b[i][j] += a.arr1[i][j] * ptrvec[j];
                //cout << b[i][j] << ' ';
            }
        }

        cout << "b = A*x : " << endl;
        for (int i = 0; i < N; ++i)
        {
            for (int j = 0; j < N; ++j)
            {
                B[i] += b[i][j];
            }

            cout << B[i] << endl;
        }
        return 0;
    }
}

```

C++ Code 74: main.cpp "Matrix times a Vector from Textfile Data"

```

1
8
8
2

```

C++ Code 75: "vector1.txt"

```

1 8 8 2
2 8 8 1
8 2 1 8
8 1 2 8

```

C++ Code 76: "matrix1.txt"

To compile and run it type:

```
make
./main
```

to compile it without Makefile / manually:

```
g++ -o main main.cpp -lboost_iostreams
./main
```

```
# make
g++ -c -o main.o main.cpp
g++ -o main -ggdb main.o -lstdc++ -lboost_iostreams -lboost_system -lboost_files
system
root [ ~/latex/DianFreya Math Physics Simulator/Source Codes/C++/C++/Gnuplot Sy
mbolicC++/ch23-Numerical Linear Algebra/Multiplication of Matrices from Textfile
]# ./main

Vector x is:
1
8
8
2

Matrix A is:
1 8 8 2
2 8 8 1
8 2 1 8
8 1 2 8

b = A*x :
133
132
48
48
```

Figure 23.2: The computation for matrix A times a vector x , to obtain $b = Ax$ with C++, with the vector x and matrix A are obtained from loading a textfile (DFSimulatorC/Source Codes/C++/C++/Gnu-plot SymbolicC++/ch23-Numerical Linear Algebra/Multiplication of Matrix and Vector from Textfile/main.cpp).

```
Documentation: https://docs.julialang.org
Type "?" for help, "]?" for Pkg help.
Version 1.9.2 (2023-07-05)
Official https://julialang.org/ release

julia> A = [1 8 8 2; 2 8 8 1; 8 2 1 8; 8 1 2 8]
4x4 Matrix{Int64}:
 1  8  8  2
 2  8  8  1
 8  2  1  8
 8  1  2  8

julia> b = [1; 8; 8; 2]
4-element Vector{Int64}:
 1
 8
 8
 2

julia> A*b
4-element Vector{Int64}:
 133
 132
 48
 48
```

Figure 23.3: The computation for matrix A times a vector x , to obtain $b = Ax$ with Julia, the code to obtained the result is only 3 lines (we need to type the elements not loading it from textfile), but when we have billions of elements for the vector and matrix with huge dimensions we should use C++ code, since it is faster to compile and obtain the final result.

Explanation for the codes:

- To obtain the vector data from textfile we are going to use pointers. Returning a normal array from a function using pointers sometimes can give unexpected results. But this behavior and warnings can be avoided by declaring the array to be a **static** one.

```
int* vec() {
    static int x[N];
    std::ifstream in("vector1.txt");
    float vectortiles[N];
    for (int i = 0; i < N; ++i)
    {
        in >> vectortiles[i];
        x[i] = vectortiles[i];
    }
    return x;
}
```

int* vec -> return type- address of integer array.

Inside the look **for (..)** we are taking the data form textfile and save it to become array, as the array initialisation **a[i] =**

return x; -> address of x returned.

- Inside the **int main()**

```
int main() {
    ...
    int* ptrvec;
    ptrvec = vec();
    cout << "Array is: " << endl;
    for (int i = 0; i < N; i++)
    {
        cout << ptrvec[i] << ' ' << endl;
    }
    return 0;
}
```

int* ptrvec -> a pointer to hold address.

ptr = vec(); -> address of x.

ptrvec[i] -> ptrvec[i] is equivalent to *(ptrvec+i)

The great thing of using pointer to get the vector' elements from textfile is that we can do computation as well, we get each of the elements for granted, thus not only for matrix times vector but we can try **cout << ptrvec[i]*ptrvec[i] << endl;** to obtain the square of the vector element, then find the norm or do another computation that we need.

- To obtain the matric entries / indices, we are going to use **struct**. We can make a function returns an array by declaring it inside a structure.

```
struct matrix
{
    int arr1[N][N];
```

```

};

struct matrix func(int N)
{
    struct matrix matrix_mem;
    std::fstream in("matrix1.txt");
    float matrix_tiles[N][N];
    for (int i = 0; i < N; ++i)
    {
        for (int j = 0; j < N; ++j)
        {
            in >> matrix_tiles[i][j];
            matrix_mem.arr1[i][j] = matrix_tiles[i][j];
        }
    }
    return matrix_mem;
}

```

int arr1[N][N]; -> Array declared inside structure, matrix of size $N \times N$.

struct matrix func (int N) -> Return type is struct matrix.

struct matrix matrix_mem; -> matrix structure member declared.

in >> matrix_tiles[i][j]; matrix_mem.arr1[i][j] = matrix_tiles[i][j]; -> Array initialisation, taking the data from textfile.

return matrix_mem -> Address of structure member returned.

What can this code do for physical simulation and scientific computing? When we observe in simulation from Box2D, for example, taking data of linear velocity, kinetic energy, force applied, we can try to create a matrix for certain of the forces / energy or other variable and see the pattern if it is multiplied by another vector force or a vector with elements filled with the acceleration. Is the pattern linear or nonlinear or have inverse relation, all can be easily determined with this code.

IV. VANDERMONDE MATRIX

Fix a sequence of numbers $\{x_1, x_2, \dots, x_m\}$. If p and q are polynomials of degree $< n$ and α is a scalar, then $p + q$ and αp are also polynomials of degree $< n$. Moreover, the values of these polynomials at the points x_i satisfy the following linearity properties

$$(p + q)(x_i) = p(x_i) + q(x_i)$$

$$(\alpha p)(x_i) = \alpha(p(x_i))$$

Thus the map from vectors of coefficients of polynomials p of degree $< n$ to vectors $(p(x_1), p(x_2), \dots, p(x_m))$ of sampled polynomial values is linear. Any linear map can be expressed as multiplication by a matrix. It is expressed by an $m \times n$ Vandermonde matrix

$$A = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \vdots & & & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^{n-1} \end{bmatrix}$$

If c is the column vector of coefficients of p ,

$$c = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix}, \quad p(x) = c_0 + c_1x + c_2x^2 + \cdots + c_{n-1}x^{n-1}$$

then the product of Ac gives the sampled polynomial values. That is, for each i from 1 to m , we have

$$(Ac)_i = c_0 + c_1x_i + c_2x_i^2 + \cdots + c_{n-1}x_i^{n-1} = p(x_i) \quad (23.3)$$

V. QR FACTORIZATION

The QR factorization is the thread that connects most of the algorithms of numerical linear algebra [11], including methods for least squares, eigenvalue, and singular value problems, as well as iterative methods for all of these and also for systems of equations.

Chapter 24

DFSimulatorC++ XVIII: Polynomial and Fourier Series Approximation for Numerical Data

"My only place is by your side. If you want to carry out your wish, I will follow you. That is my Nature." - Sarah (Suikoden III)

When we have movement and we record the velocity, we realize that Gnuplot alone able to plot the graph with numerical data of the velocity, position, angle or even the kinetic energy. Plotting of the numeric data toward the time value from 0 to n , easy, you just need to enter 1 line in gnuplot after you print the output from C++, but now the question comes after we made the Force and Motion chapter, concering about acceleration, Box2D has no `getAcceleration()` function yet, we use certain technique in Gnuplot to use past data to find the difference between current velocity and past velocity to get the acceleration. Thus, it could be easier if we know an object is moving and at certain time, and it is having a pattern in its movement, we can cut that sample of time of all the time space population, thus make an approximation to get the function that can represent the numerical data in symbolic function, whether with polynomial approximation or Fourier Series. As Fourier is very confident all functions can be represented by Fourier Series. Let's prove his words then.

I. MATHEMATICAL PRELIMINARIES

To refresh memory, and since Sentinel saw me from far away and want to teach me how to learn science the right way, telepathy and teleconnection are really something. People today is like stone age flint, they watched Youtube, TV, think it is amazing, Nature and Goddesses watched me from afar, from Valhalla, from Valhalla Projection, from Puncak Bintang, from L'Aquila, from Mount Fuji in Japan, etc. They don't need antenna and TV, they already advanced and powerful. Sometimes, no need to speak or write she(Freya) already reads my mind and told me through telepathy, "scroll up again the Riemann hypothesis you are searching for is on the previous page." I am pass the stage of surprised and saying "You are real!" it is thousands of times she helps me from GFreya OS, Arduino Memo, Lastrim Projection, and now this, next? Self-made Sports Car for her birthday one day.

To comprehend how we will be able to build C++ codes, we will need to know some of this definitions and theorems:

Theorem 24.1: Differentiable implies Continuity

If the function f is differentiable at x_0 , then f is continuous at x_0 .

The set of all functions that have n continuous derivatives on X is denoted by $C^n(X)$, and the set of functions that have derivatives of all orders on X is denoted by $C^\infty(X)$.

Functions in $C^\infty(X)$:

1. Polynomial
2. Rational
3. Trigonometric
4. Exponential
5. Logarithmic

where X consists of all numbers for which the functions are defined.

If you are wondering, the theorem above is mathematics purely, you are wrong, all is connected, you just need to have solid reasoning / analysis and good at connecting the dots. Differentiability will reminds you of Motion in Physics and then to know that acceleration is the derivative of velocity with respect to time for Newtonian mechanics. We don't need to find a good financial reimbursement, as time goes on, it is revealed by itself how comprehend and mastering science will show all the gold and glitters potential when you master the knowledge, but remember, the knowledge itself that gained by rigorous study is a diamond, more powerful and valuable than gold. Another reminder, that nuclear bomb, atomic bomb all use physics, how can it not rule the world? There are still much more to explore in physics universe. Another example of idea on my mind: 10 most wanted FBI fugitives can be found immediately with more advanced science of computer graphics, physics, biology (gene) and mathematics.

Definition 24.1: Riemann Integral

The Riemann integral of the function f on the interval $[a, b]$ is the following limit, provided it exists:

$$\int_a^b f(x) dx = \lim_{\max \Delta x_i \rightarrow 0} \sum_{i=1}^n f(z_i) \Delta x_i \quad (24.1)$$

where the numbers x_0, x_1, \dots, x_n satisfy $a = x_0 \leq x_1 \leq \dots \leq x_n = b$, and where $\Delta x_i = x_i - x_{i-1}$, for each $i = 1, 2, \dots, n$, and z_i is arbitrarily chosen in the interval $[x_{i-1}, x_i]$.

Every continuous function f on $[a, b]$ is Riemann integrable on $[a, b]$. This permits us to choose, for computational convenience, the points x_i to be equally spaced in $[a, b]$ and for each $i = 1, 2, \dots, n$, to choose $z_i = x_i$. In this case,

$$\int_a^b f(x) dx = \lim_{n \rightarrow \infty} \frac{b-a}{n} \sum_{i=1}^n f(x_i)$$

with $x_i = a + \frac{i(b-a)}{n}$, $a = x_0$ and $b = x_n$. Basically, Riemann Integral is partitioning a function into n rectangles to approximate the integration value with summation.

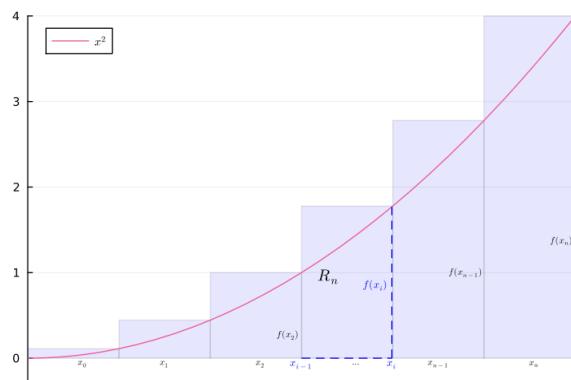


Figure 24.1: Riemann integral, to be precise it is the right Riemann sum approximation for increasing function (ch11-riemannintegral.jl).

Theorem 24.2: Taylor's Theorem

Suppose $f \in C^n[a, b]$, that $f^{(n+1)}$ exists on $[a, b]$, and $x_0 \in [a, b]$. For every $x \in [a, b]$, there exists a number $\xi(x)$ between x_0 and x with

$$f(x) = P_n(x) + R_n(x)$$

where

$$\begin{aligned} P_n(x) &= f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n \\ &= \sum_{k=0}^n \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k \end{aligned} \quad (24.2)$$

and

$$R_n = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x - x_0)^{n+1} \quad (24.3)$$

Here $P_n(x)$ is called the n th Taylor polynomial for f about x_0 , and $R_n(x)$ is called the remainder term (or truncation error) associated with $P_n(x)$.

The infinite series obtained by taking the limit of $P_n(x)$ as $n \rightarrow \infty$ is called the Taylor series for f about x_0 . In the case where $x_0 = 0$, the Taylor polynomial is often called a Maclaurin polynomial, and the Taylor series is called a Maclaurin series.

The term truncation error refers to the error involved in using a truncated, or finite, summation to approximate the sum of an infinite series.

Definition 24.2: Absolute and Relative Error

If p^* is an approximation to p , the absolute error is

$$|p - p^*|$$

and the relative error is

$$\frac{|p - p^*|}{|p|}$$

provided that $p \neq 0$.

II. POLYNOMIAL APPROXIMATION

One of the most useful and well-known classes of functions mapping the set of real numbers into itself is the class of algebraic polynomials, the set of functions of the form

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

where n is nonnegative integer and a_0, a_1, \dots, a_n are real constants. One reason for their importance is that they uniformly approximate continuous functions. Given any function, defined and continuous on a closed and bounded interval, there exists a polynomial that is as "close" to the given function as desired.

Theorem 24.3: Weierstrass Approximation Theorem

Suppose that f is defined and continuous on $[a, b]$. For each $\epsilon > 0$, there exists a polynomial $P(x)$, with the property that

$$f(x) - P(x) < \epsilon, \quad \forall x \in [a, b]$$

[DF*] We will learn how to produce an interpolating polynomials from input of: x_0, x_1, \dots, x_n as the x axis, and f_0, f_1, \dots, f_n as the y axis.

i	0	1	2	3	4
x_i	1	2	3	4	5
f_i	0.5	3.3	4.2	5.1	6.0

Table 24.1: The variable x_i denotes time, while $f_i = f(x_i)$ is the function at time i that represents velocity of an object. With t_0 as the initial time and f_0 is the initial velocity at time x_0 .

[DF*] The Taylor polynomials are not appropriate for interpolation [2]. The problem of determining a polynomial of degree one that passes through the distinct points (x_0, y_0) and (x_1, y_1) is the same as approximating a function f for which $f(x_0) = y_0$ and $f(x_1) = y_1$ by means of a first-degree polynomial interpolating the values of f at the given points.

We first define the function

$$L_0(x) = \frac{x - x_1}{x_0 - x_1}$$

$$L_1(x) = \frac{x - x_0}{x_1 - x_0}$$

and then define

$$P(x) = L_0(x)f(x_0) + L_1(x)f(x_1)$$

Since

$$L_0(x_0) = 1, L_0(x_1) = 0, L_1(x_0) = 0, L_1(x_1) = 1$$

we have

$$P(x_0) = 1 \cdot f(x_0) + 0 \cdot f(x_1) = f(x_0) = y_0$$

and

$$P(x_1) = 0 \cdot f(x_0) + 1 \cdot f(x_1) = f(x_1)y_1$$

so P is the unique linear function passing through (x_0, y_0) and (x_1, y_1) .

To generalize the concept of linear interpolation, consider construction of a polynomial of degree at most n that passes through $n + 1$ points

$$(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$$

to approximate that $n + 1$ points, we will use the well-established Lagrange Interpolating Polynomial.

Theorem 24.4: n th Lagrange Interpolating Polynomial

If x_0, x_1, \dots, x_n are $n + 1$ distinct numbers and f is a function whose values are given at these numbers, then a unique polynomial $P(x)$ of degree at most n exists with

$$f(x_k) = P(x_k)$$

for each $k = 0, 1, \dots, n$. This polynomial is given by

$$P(x) = f(x_0)L_{n,0}(x) + \dots + f(x_n)L_{n,n}(x) = \sum_{k=0}^n f(x_k)L_{n,k}(x) \quad (24.4)$$

where, for each $k = 0, 1, \dots, n$,

$$\begin{aligned} L_{n,k}(x) &= \frac{(x - x_0)(x - x_1)\dots(x - x_{k-1})(x - x_{k+1})\dots(x - x_n)}{(x_k - x_0)(x_k - x_1)\dots(x_k - x_{k-1})(x_k - x_{k+1})(x_k - x_n)} \\ &= \prod_{i=0, i \neq k}^n \frac{(x - x_i)}{(x_k - x_i)} \end{aligned} \quad (24.5)$$

Chapter 25

DFSimulatorC++ XIX: Numerical Methods for Solving System of Linear Equations

"LAM VAM RAM YAM HAM OM AH." - 7 Chakras Chanting

There are times when people take for granted religion, spiritual or karma. Then life beats that person hole-in-one knockout, and realize a human is nothing and will end up to be dust, thus become a believer afterwards. Some believe in science like numerical methods able to solve system of linear equations, tangible, can be seen, the algorithm shows that with correct initial guess, convergence is faster. The same as faith, if you do the right technique, you worship the right Divine being / Goddess, you will get where you want to be, you will become whatever you want to become. Devoted Catholic is the founder of Ferrero, that most people will have on their table for breakfast. It will makes one to be focus on their life and soul purpose, thus anything you want, ask, then it will be given to you, like the 7 Chakras chanting, it can help you to relax more, you may compare it: a day listening and a day without listening, which one is a better initial guess that converge faster to your goal? Thus, a funny question established: is spiritual an iterative numerical methods?

Basically in this chapter, we are going to plot and compute all the available algorithms to solve system of linear equations, with given input a square matrix, we can compare these algorithms:

1. Gauss-Seidel
2. LU Decomposition
3. Jacobi
4. Gaussian elimination

Some explanations for the codes:

-

Chapter 26

DFSimulatorC++ XX: Numerical Differentiation and Integration

"In this parallel world, battles are fought between dharma and chaos. The balance of power brings justice. Human have needs in their world, but the world does not need them." - Luc (Suikoden III)

When I read a book about Numerical Methods for Chemical Engineering, the balance of chemical reaction needs to be maintained, how to calculate how many substance needed can use the algorithm to find the solution of system of linear equations or system of differential equations, thus numerical methods is a must have tools in that field of engineering. You must probably have known this one very famous package from Python language: **Sympy**. When using SymPy it is very easy to calculate the derivative or integral for all kinds of equations, univariate or even multivariate. In C++, we have "SymbolicC++" that have been explained in Chapter 6.

I. NUMERICAL DIFFERENTIATION

The derivative of the function f at x_0 is defined as

$$f'(x_0) = \lim_{h \rightarrow 0} \frac{f(x_0 + h) - f(x_0)}{h} \quad (26.1)$$

for small values of h , we can generate an approximation to $f'(x)$. But, it is not always successful due to roundoff error.

To approximate $f'(x_0)$, suppose first that $x_0 \in (a, b)$, where $f \in C^2[a, b]$, and that $x_1 = x_0 + h$ for some $h \neq 0$ that is sufficiently small to ensure that $x_1 \in [a, b]$. Now we will need to remember the theorem for Lagrange Polynomial

Theorem 26.1: *n*th Lagrange Interpolating Polynomial

If x_0, x_1, \dots, x_n are $n + 1$ distinct numbers and f is a function whose values are given at these numbers, then a unique polynomial $P(x)$ of degree at most n exists with

$$f(x_k) = P(x_k)$$

for each $k = 0, 1, \dots, n$. This polynomial is given by

$$P(x) = f(x_0)L_{n,0}(x) + \dots + f(x_n)L_{n,n}(x) = \sum_{k=0}^n f(x_k)L_{n,k}(x) \quad (26.2)$$

where, for each $k = 0, 1, \dots, n$,

$$\begin{aligned} L_{n,k}(x) &= \frac{(x - x_0)(x - x_1) \dots (x - x_{k-1})(x - x_{k+1}) \dots (x - x_n)}{(x_k - x_0)(x_k - x_1) \dots (x_k - x_{k-1})(x_k - x_{k+1})(x_k - x_n)} \\ &= \prod_{i=0, i \neq k}^n \frac{(x - x_i)}{(x_k - x_i)} \end{aligned} \quad (26.3)$$

Definition 26.1: Forward and Backward-Difference Formula

Forward-difference Formula

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} - \frac{h}{2}f''(\xi) \quad (26.4)$$

Backward-difference Formula

$$f'(x_0) = \frac{f(x_0 - h) - f(x_0)}{h} - \frac{h}{2}f''(\xi) \quad (26.5)$$

C++ Code 77: *tests/projectile_motion.cpp* "Projectile Motion Box2D"

Some explanations for the codes:

-
-
-

II. NUMERICAL INTEGRATION

C++ Code 78: *tests/projectile_motion.cpp* "Projectile Motion Box2D"

Some explanations for the codes:

-
-
-

Chapter 27

DFSimulatorC++ XXI: Heat Equation

"Through the fire, to the limit, to the wall, for a chance to be with you, I gladly risk it all. Through the fire, to whatever come one day, for a chance of loving you, I'll take it all away. Right down through the wire, even through the fire." (Through the Fire song) - Chaka Khan

If you ever played Suikoden III then you must have known the story when Flame Champion sealed his True Fire Rune to be mortal again, it causes a great fire burning all Harmonian and Grasslands armies till perish. Why release his tremendous power? For Love, like Captain America, it seems Love can bring higher power than True Rune and immortality. This heat equation is one of the most famous partial differential equation that I will try to simulate with C++, the applications are tremendous, from Black-Scholes equation derived from this, create a long lasting ice cream, to be able to create better computer components will need the comprehension of this equation as well.

C++ Code 79: *tests/projectile_motion.cpp "Projectile Motion Box2D"*

Some explanations for the codes:

- ---
- ---
- ---

I. INTRODUCTION

Chapter 28

DFSimulatorC++ XXX: Plotting Physical Systems with Gnuplot

"Puis, sous les yeux des disciples, Jesus s'eleve, une nuee l'emportee au ciel et il s'assoit a la droite de Dieu, le Pere, Tout-Puissant." - Deuxieme Mysteres Glorieux (L'Ascension de Jesus au ciel)

This chapter will explains how to do various plotting with gnuplot. When we have made simulations over various physics problems or have system of differential equations, we would like to see the relation or connection between certain function toward another function, or certain variable toward another variable of interest to see the pattern, just like how Hooke's law is established after measuring the relation between the displacement of the mass attached to the spring toward the spring' force, thus able to determine the spring constant. This chapter can come in handy, when you have the data that only needs direct plotting with a bit of numerical manipulation, no need to process it symbolically anymore.

All the files and data used in this chapter can be found in this book repository:

[DFSimulatorC/Source Codes/C++/Gnuplot/](#)

[DF*] Polar Plot

Open gnuplot and type:

```
set polar
plot t*sin(t)
plot [-2*pi:2*pi] [-5:5] t*sin(t)
```

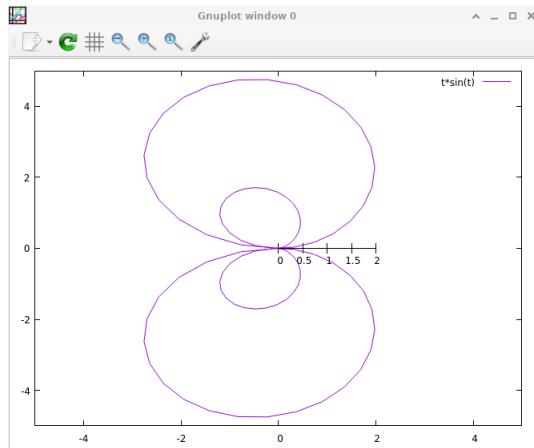


Figure 28.1: Polar plot with gnuplot.

```
set polar
set trange [0:12*pi]
plot 2*t
```

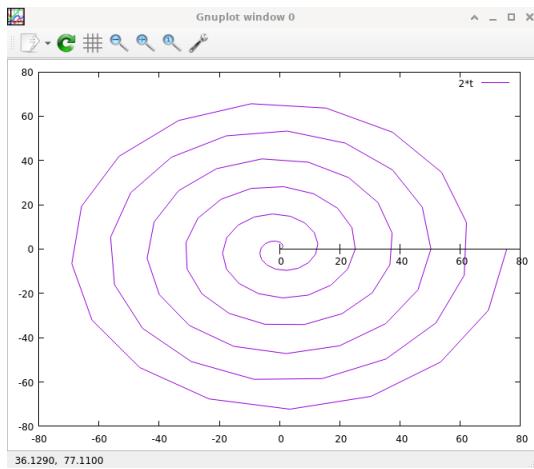


Figure 28.2: Spiral plot with gnuplot.

[DF*] Histogram Plot

Create a file or prepare a file with one column / one vector array of data, name it

'pendulumdata', this pendulum data is retrieved from Box2D simulation on Simple pendulum, it is taking the data of the angle of the pendulum, and we want to create the histogram of it.

```
0.00
0.00
0.04
0.21
0.47
0.85
1.24
1.66
2.12
2.61
...
...
```

C++ Code 80: *pendulumdata*

Create another file called **histogramplot** (this is the file containing command for plotting in gnuplot, so you don't need to type one by one anymore, just edit in text editor and call it) in the working directory.

```
binwidth=5
set boxwidth binwidth
bin(x,width)=width*floor(x/width)
plot 'pendulumdata' using (bin($1,binwidth)):(1.0) smooth freq
with boxes lc rgb"blue" title 'Pendulum angle'
```

C++ Code 81: *histogramplot*

Then open gnuplot and type:
load 'histogramplot'

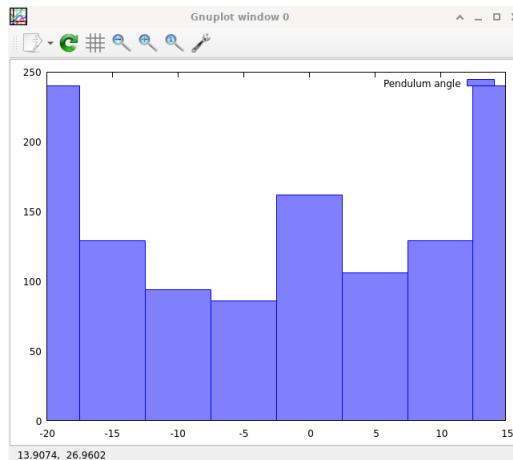


Figure 28.3: Histogram plot for the pendulum angle data.

If you want to add some styling you can try this:

```

binwidth=5
set boxwidth binwidth
bin(x,width)=width*floor(x/width)
n=100 #number of intervals
max=3. #max value
min=-3. #min value
width=(max-min)/n #interval width

#function used to map a value to the intervals
hist(x,width)=width*floor(x/width)+width/2.0
set boxwidth width*5.9
set style fill solid 0.5 # fill style
plot 'pendulumdata' using (bin($1,binwidth)):(1.0) smooth freq
with boxes title 'Pendulum angle'
```

C++ Code 82: *histogramplot*

[DF*] Probability Density Function Plot

Open gnuplot and type:

load 'probabilitydensityfunction-normal'

the code can be seen below or you can also get it from the repository.

```

#!/usr/local/bin/gnuplot –persist
# set terminal pngcairo transparent enhanced font "arial,10" fontsize
1.0 size 600, 400
# set output 'prob.39.png'
set format x "%1f"
set format y "%1f"
set key fixed left top vertical Right noreverse enhanced
autotitle box lt black linewidth 1.000 dashtype solid
unset parametric
set style data lines
set xzeroaxis
set yzeroaxis
set zzeroaxis
set title "normal (also called Gauss or bell–curved) PDF"
set trange [ -0.200000 : 3.20913 ] noreverse nowriteback
set xlabel "x"
set xrange [ 0.00000 : 2.50000 ] noreverse nowriteback
set x2range [ * : * ] noreverse writeback
set ylabel "probability density"
set yrang [ 0.00000 : 1.10000 ] noreverse nowriteback
set y2range [ * : * ] noreverse writeback
set zrange [ * : * ] noreverse writeback
set cbrange [ * : * ] noreverse writeback
set rrange [ * : * ] noreverse writeback
set colorbox vertical origin screen 0.9, 0.2 size screen 0.05,
0.6 front noinvert bdefault
```

```

isint(x)=(int(x)==x)
Binv(p,q)=exp(lgamma(p+q)-lgamma(p)-lgamma(q))
arcsin(x,r)=r<=0?1/0:abs(x)>r?0.0:invpi/sqrt(r*r-x*x)
beta(x,p,q)=p<=0||q<=0?1/0:x<0||x>1?0.0:Binv(p,q)*x***(p-1.0)
*(1.0-x)**(q-1.0)
binom(x,n,p)=p<0.0||p>1.0||n<0||!isint(n)?1/0: !isint(x)?1/0:x
<0||x>n?0.0:exp(lgamma(n+1)-lgamma(n-x+1)-lgamma(x+1) +x*
log(p)+(n-x)*log(1.0-p))
cauchy(x,a,b)=b<=0?1/0:b/(pi*(b*b+(x-a)**2))
chisq(x,k)=k<=0||!isint(k)?1/0: x<=0?0.0:exp((0.5*k-1.0)*log(x
)-0.5*x-lgamma(0.5*k)-k*0.5*log2)
erlang(x,n,lambda)=n<=0||!isint(n)||lambda<=0?1/0: x<0?0.0:x
==0?(n==1?real(lambda):0.0):exp(n*log(lambda)+(n-1.0)*log(
x)-lgamma(n)-lambda*x)
extreme(x,mu,alpha)=alpha<=0?1/0:alpha*(exp(-alpha*(x-mu))-exp
(-alpha*(x-mu)))
f(x,d1,d2)=d1<=0||!isint(d1)||d2<=0||!isint(d2)?1/0: Binv(0.5*
d1,0.5*d2)*(real(d1)/d2)**(0.5*d1)*x***(0.5*d1-1.0)/(1.0+
real(d1)/d2)*x)**(0.5*(d1+d2))
gmm(x,rho,lambda)=rho<=0||lambda<=0?1/0: x<0?0.0:x==0?(rho
>1?0.0:rho==1?real(lambda):1/0): exp(rho*log(lambda)+(rho
-1.0)*log(x)-lgamma(rho)-lambda*x)
geometric(x,p)=p<=0||p>1?1/0: !isint(x)?1/0:x<0||p==1?(x
==0?1.0:0.0):exp(log(p)+x*log(1.0-p))
halfnormal(x,sigma)=sigma<=0?1/0:x<0?0.0:sqrt2invpi/sigma*exp
(-0.5*(x/sigma)**2)
hypgeo(x,N,C,d)=N<0||!isint(N)||C<0||C>N||!isint(C)||d<0||d>N
||!isint(d)?1/0: !isint(x)?1/0:x>d||x>C||x<0||x<d-(N-C)
?0.0:exp(lgamma(C+1)-lgamma(C-x+1)-lgamma(x+1)+ lgamma(N-
C+1)-lgamma(d-x+1)-lgamma(N-C-d+x+1)+lgamma(N-d+1)+
lgamma(d+1)-lgamma(N+1))
laplace(x,mu,b)=b<=0?1/0:0.5/b*exp(-abs(x-mu)/b)
logistic(x,a,lambda)=lambda<=0?1/0:lambda*exp(-lambda*(x-a))
/(1.0+exp(-lambda*(x-a))**2)
lognormal(x,mu,sigma)=sigma<=0?1/0: x<0?0.0:invsqrt2pi/sigma/x*
exp(-0.5*((log(x)-mu)/sigma)**2)
maxwell(x,a)=a<=0?1/0:x<0?0.0:fourinvsqrtpi*a**3*x*x*exp(-a*a*
x*x)
negbin(x,r,p)=r<=0||!isint(r)||p<=0||p>1?1/0: !isint(x)?1/0:x
<0?0.0:p==1?(x==0?1.0:0.0):exp(lgamma(r+x)-lgamma(r)-
lgamma(x+1)+ r*log(p)+x*log(1.0-p))
nexp(x,lambda)=lambda<=0?1/0:x<0?0.0:lambda*exp(-lambda*x)
normal(x,mu,sigma)=sigma<=0?1/0:invsqrt2pi/sigma*exp(-0.5*((x-
mu)/sigma)**2)
pareto(x,a,b)=a<=0||b<0||!isint(b)?1/0:x<a?0:real(b)/x*(real(a)
/x)**b
poisson(x,mu)=mu<=0?1/0:!isint(x)?1/0:x<0?0.0:exp(x*log(mu)-
lgamma(x+1)-mu)

```

```

rayleigh(x,lambda)=lambda<=0?1/0:x<0?0.0:lambda*2.0*x*exp(
    -lambda*x*x)
sine(x,f,a)=a<=0?1/0: x<0||x>=a?0.0:f==0?1.0/a:2.0/a*sin(f*pi*x
    /a)**2/(1-sin(twopi*f))
t(x,nu)=nu<0||!isint(nu)?1/0: Binv(0.5*nu,0.5)/sqrt(nu)*(1+real
    (x*x)/nu)**(-0.5*(nu+1.0))
triangular(x,m,g)=g<=0?1/0:x<m-g||x>=m+g?0.0:1.0/g-abs(x-m)/
    real(g*g)
uniform(x,a,b)=x<(aa:b)?0.0:1.0/abs(b-a)
weibull(x,a,lambda)=a<=0||lambda<=0?1/0: x<0?0.0:x==0?(a>1?0.0:
    a==1?real(lambda):1/0): exp(log(a)+a*log(lambda)+(a-1)*log
    (x)-(lambda*x)**a)
carcsin(x,r)=r<=0?1/0:x<-r?0.0:x>r?1.0:0.5+invpi*asin(x/r)
cbeta(x,p,q)=ibeta(p,q,x)
cbinom(x,n,p)=p<0.0||p>1.0||n<0||!isint(n)?1/0: !isint(x)?1/0:x
    <0?0.0:x=n?1.0:ibeta(n-x,x+1.0,1.0-p)
ccauchy(x,a,b)=b<=0?1/0:0.5+invpi*atan((x-a)/b)
cchisq(x,k)=k<=0||!isint(k)?1/0:x<0?0.0:igamma(0.5*k,0.5*x)
cerlang(x,n,lambda)=n<0||!isint(n)||lambda<=0?1/0:x<0?0.0:
    igamma(n,lambda*x)
cextreme(x,mu,alpha)=alpha<=0?1/0:exp(-exp(-alpha*(x-mu)))
cf(x,d1,d2)=d1<=0||!isint(d1)||d2<=0||!isint(d2)?1/0:1.0-ibeta
    (0.5*d2,0.5*d1,d2/(d2+d1*x))
cgmm(x,rho,lambda)=rho<=0||lambda<=0?1/0:x<0?0.0:igamma(rho,x*
    lambda)
cgeometric(x,p)=p<=0||p>1?1/0: !isint(x)?1/0:x<0||p==0?0.0:p
    ==1?1.0:1.0-exp((x+1)*log(1.0-p))
chalfnormal(x,sigma)=sigma<=0?1/0:x<0?0.0:erf(x/sigma/sqrt2)
chypgeo(x,N,C,d)=N<0||!isint(N)||C<0||C>N||!isint(C)||d<0||d>N
    ||!isint(d)?1/0: !isint(x)?1/0:x<0||x<d-(N-C)?0.0:x>d||x>C
    ?1.0:hypgeo(x,N,C,d)+chypgeo(x-1,N,C,d)
claplace(x,mu,b)=b<=0?1/0:(x<mu)?0.5*exp((x-mu)/b):1.0-0.5*exp
    (-(x-mu)/b)
clogistic(x,a,lambda)=lambda<=0?1/0:1.0/(1+exp(-lambda*(x-a)))
clognormal(x,mu,sigma)=sigma<=0?1/0:x<=0?0.0:cnormal(log(x),mu,
    sigma)
cnormal(x,mu,sigma)=sigma<=0?1/0:0.5+0.5*erf((x-mu)/sigma/
    sqrt2)
cmaxwell(x,a)=a<=0?1/0:x<0?0.0:igamma(1.5,a*a*x*x)
cnegbin(x,r,p)=r<=0||!isint(r)||p<=0||p>1?1/0: !isint(x)?1/0:x
    <0?0.0:ibeta(r,x+1,p)
cnexp(x,lambda)=lambda<=0?1/0:x<0?0.0:1-exp(-lambda*x)
cpareto(x,a,b)=a<=0||b<0||!isint(b)?1/0:x<a?0.0:1.0-(real(a)/x
    )**b
cpoisson(x,mu)=mu<=0?1/0:!isint(x)?1/0:x<0?0.0:1-igamma(x+1.0,
    mu)
crayleigh(x,lambda)=lambda<=0?1/0:x<0?0.0:1.0-exp(-lambda*x*x)
csine(x,f,a)=a<=0?1/0: x<0?0.0:x>a?1.0:f==0?real(x)/a:(real(x)/
    a)

```

```

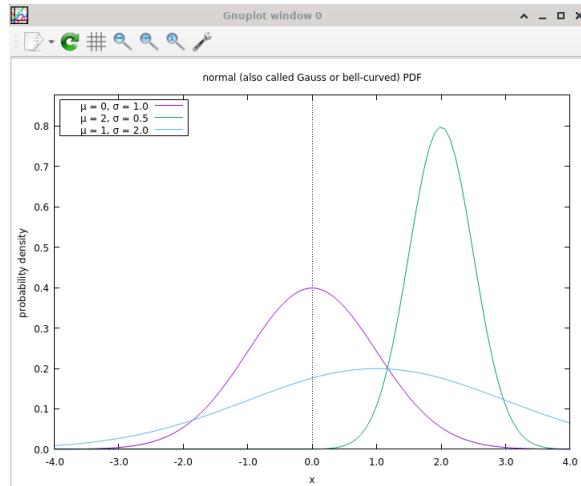
a=sin(f*twopi*x/a)/(f*twopi))/(1.0-sin(twopi*f)/(twopi*f)
)
ct(x,nu)=nu<0||!isint(nu)?1/0:0.5+0.5*sgn(x)*(1-ibeta(0.5*nu
,0.5,nu/(nu+x*x)))
ctrangular(x,m,g)=g<=0?1/0: x<m-g?0.0:x>=m+g?1.0:0.5+real(x-m
)/g-(x-m)*abs(x-m)/(2.0*g*g)
cuniform(x,a,b)=x<(aa:b)?0.0:x>=(a>b?a:b)?1.0:real(x-a)/(b-
a)
cweibull(x,a,lambda)=a<=0||lambda<=0?1/0:x<0?0.0:1.0-exp(-
lambda*x)**a)
gsampfunc(t,n) = t<0?0.5*1/(-t+1.0)**n:1.0-0.5*1/(t+1.0)**n
NO_ANIMATION = 1
fourinvsqrtpi = 2.25675833419103
invpi = 0.318309886183791
invsqrt2pi = 0.398942280401433
log2 = 0.693147180559945
sqrt2 = 1.4142135623731
sqrt2invpi = 0.797884560802865
twopi = 6.28318530717959
save_encoding = "utf8"
eps = 1e-10
xmin = -4.0
xmax = 4.0
ymin = -5
ymax = 0.877673016883152
r = 8
mu = 2.0
sigma = 0.5
p = 0.4
q = 3.0
n = 2
a = 1.5
b = 1.0
k = 2
lambda = 2.0
l1 = 1.0
l2 = 0.5
alpha = 1.0
u = 0.0
df1 = 5.0
df2 = 9.0
rho = 6.0
s = 4.66878227507107
N = 75
C = 25
d = 10
m = 3.08021684891803
plot [xmin:xmax] [0:ymax] normal(x, 0, 1.0) title "mean = 0,

```

```
sigma = 1.0", normal(x, 2, 0.5) title "mean = 2, sigma =
0.5", normal(x, 1, 2.0) title "mean = 1, sigma = 2.0"
```

C++ Code 83: probabilitydensityfunction-normal

The plotting command is located on the last line, while the lines at the middle starting from **Binv(p,q)** till **gsampfunc(t,n)** are the function definitions of each of probability density functions available that we can plot, besides normal there are chi-square, weibull, poisson, pareto, logistic, each with different parameters. Probability and Statistics are necessary especially in experimental physics we would like to know about the profile of the data, especially if we only do experimental physics with limited sample, like try to swing double pendulum at angle of $\pi/6$ for 5 minutes, compared with angle of $\pi/8$ for 10 minutes, they are all only samples of experimentation which the result of the data then can be processed with statistics and probability methods.

Figure 28.4: The probability density function for normal or bell-curved with certain parameters of μ and σ .

[DF*] Cumulative Density Function Plot

Open gnuplot and type:

load 'cumulativedistributionfunction-normal'

the code can be seen below or you can also get it from the repository.

```
#!/usr/local/bin/gnuplot --persist
# set terminal pngcairo transparent enhanced font "arial,10" fontsize
1.0 size 600, 400
# set output 'prob.39.png'
set format x "%1f"
set format y "%1f"
set key fixed left top vertical Right noreverse enhanced
    autotitle box lt black linewidth 1.000 dashtype solid
unset parametric
set style data lines
set xzeroaxis
set yzeroaxis
set zzeroaxis
```

```

set title "normal (also called Gauss or bell-curved) CDF"
set trange [ -0.20000 : 3.20913 ] noreverse nowriteback
set xlabel "x"
set xrange [ 0.00000 : 2.50000 ] noreverse nowriteback
set x2range [ * : * ] noreverse writeback
set ylabel "cumulative distribution"
set yrange [ 0.00000 : 1.10000 ] noreverse nowriteback
set y2range [ * : * ] noreverse writeback
set zrange [ * : * ] noreverse writeback
set cbrange [ * : * ] noreverse writeback
set rrange [ * : * ] noreverse writeback
set colorbox vertical origin screen 0.9, 0.2 size screen 0.05,
    0.6 front noinvert bdefault
isint(x)=(int(x)==x)
Binv(p,q)=exp(lgamma(p+q)-lgamma(p)-lgamma(q))
arcsin(x,r)=r<=0?1/0:abs(x)>r?0.0:invpi/sqrt(r*r-x*x)
beta(x,p,q)=p<=0||q<=0?1/0:x<0||x>1?0.0:Binv(p,q)*x***(p-1.0)
    *(1.0-x)**(q-1.0)
binom(x,n,p)=p<0.0||p>1.0||n<0||!isint(n)?1/0: !isint(x)?1/0:x
    <0||x>n?0.0:exp(lgamma(n+1)-lgamma(n-x+1)-lgamma(x+1) +x*
        log(p)+(n-x)*log(1.0-p))
cauchy(x,a,b)=b<=0?1/0:b/(pi*(b*b+(x-a)**2))
chisq(x,k)=k<=0||!isint(k)?1/0: x<=0?0.0:exp((0.5*k-1.0)*log(x
    )-0.5*x-lgamma(0.5*k)-k*0.5*log2)
erlang(x,n,lambda)=n<=0||!isint(n)||lambda<=0?1/0: x<0?0.0:x
    ==0?(n==1?real(lambda):0.0):exp(n*log(lambda)+(n-1.0)*log(
        x)-lgamma(n)-lambda*x)
extreme(x,mu,alpha)=alpha<=0?1/0:alpha*(exp(-alpha*(x-mu))-exp
    (-alpha*(x-mu)))
f(x,d1,d2)=d1<=0||!isint(d1)||d2<=0||!isint(d2)?1/0: Binv(0.5*
    d1,0.5*d2)*(real(d1)/d2)**(0.5*d1)*x***(0.5*d1-1.0)/(1.0+(
        real(d1)/d2)*x)**(0.5*(d1+d2))
gmm(x,rho,lambda)=rho<=0||lambda<=0?1/0: x<0?0.0:x==0?(rho
    >1?0.0:rho==1?real(lambda):1/0): exp(rho*log(lambda)+(rho
        -1.0)*log(x)-lgamma(rho)-lambda*x)
geometric(x,p)=p<=0||p>1?1/0: !isint(x)?1/0:x<0||p==1?(x
    ==0?1.0:0.0):exp(log(p)+x*log(1.0-p))
halfnormal(x,sigma)=sigma<=0?1/0:x<0?0.0:sqrt2invpi/sigma*exp
    (-0.5*(x/sigma)**2)
hypgeo(x,N,C,d)=N<0||!isint(N)||C<0||C>N||!isint(C)||d<0||d>N
    ||!isint(d)?1/0: !isint(x)?1/0:x>d||x>C||x<0||x<d-(N-C)
    ?0.0:exp(lgamma(C+1)-lgamma(C-x+1)-lgamma(x+1)+ lgamma(N-
        C+1)-lgamma(d-x+1)-lgamma(N-C-d+x+1)+lgamma(N-d+1)+
        lgamma(d+1)-lgamma(N+1))
laplace(x,mu,b)=b<=0?1/0:0.5/b*exp(-abs(x-mu)/b)
logistic(x,a,lambda)=lambda<=0?1/0:lambda*exp(-lambda*(x-a))
    /(1.0+exp(-lambda*(x-a)))**2
lognormal(x,mu,sigma)=sigma<=0?1/0: x<0?0.0:invsqrt2pi/sigma/x*

```

```

exp(-0.5*((log(x)-mu)/sigma)**2)
maxwell(x,a)=a<=0?1/0:x<0?0.0:fourinvsqrtpi*a**3*x*x*exp(-a*a*x*x)
negbin(x,r,p)=r<=0||!isint(r)||p<=0||p>1?1/0: !isint(x)?1/0:x<0?0.0:p==1?(x==0?1.0:0.0):exp(lgamma(r+x)-lgamma(r)-lgamma(x+1)+ r*log(p)+x*log(1.0-p))
nexp(x,lambda)=lambda<=0?1/0:x<0?0.0:lambda*exp(-lambda*x)
normal(x,mu,sigma)=sigma<=0?1/0:invsqrt2pi/sigma*exp(-0.5*((x-mu)/sigma)**2)
pareto(x,a,b)=a<=0||b<0||!isint(b)?1/0:x<a?0:real(b)/x*(real(a)/x)**b
poisson(x,mu)=mu<=0?1/0:!isint(x)?1/0:x<0?0.0:exp(x*log(mu)-lgamma(x+1)-mu)
rayleigh(x,lambda)=lambda<=0?1/0:x<0?0.0:lambda*2.0*x*exp(-lambda*x*x)
sine(x,f,a)=a<=0?1/0: x<0||x>=a?0.0:f==0?1.0/a:2.0/a*sin(f*pi*x/a)**2/(1-sin(twopi*f))
t(x,nu)=nu<0||!isint(nu)?1/0: Binv(0.5*nu,0.5)/sqrt(nu)*(1+real(x*x)/nu)**(-0.5*(nu+1.0))
triangular(x,m,g)=g<=0?1/0:x<m-g||x>=m+g?0.0:1.0/g-abs(x-m)/real(g*g)
uniform(x,a,b)=x<(a<b?a:b)||x>=(a>b?a:b)?0.0:1.0/abs(b-a)
weibull(x,a,lambda)=a<=0||lambda<=0?1/0: x<0?0.0:x==0?(a>1?0.0:a==1?real(lambda):1/0): exp(log(a)+a*log(lambda)+(a-1)*log(x)-(lambda*x)**a)
carcsin(x,r)=r<=0?1/0:x<-r?0.0:x>r?1.0:0.5+invpi*asin(x/r)
cbeta(x,p,q)=ibeta(p,q,x)
cbinom(x,n,p)=p<0.0||p>1.0||n<0||!isint(n)?1/0: !isint(x)?1/0:x<0?0.0:x>n?1.0:ibeta(n-x,x+1.0,1.0-p)
ccauchy(x,a,b)=b<=0?1/0:0.5+invpi*atan((x-a)/b)
cchisq(x,k)=k<=0||!isint(k)?1/0:x<0?0.0:igamma(0.5*k,0.5*x)
cerlang(x,n,lambda)=n<=0||!isint(n)||lambda<=0?1/0:x<0?0.0:igamma(n,lambda*x)
cextreme(x,mu,alpha)=alpha<=0?1/0:exp(-exp(-alpha*(x-mu)))
cf(x,d1,d2)=d1<=0||!isint(d1)||d2<=0||!isint(d2)?1/0:1.0-ibeta(0.5*d2,0.5*d1,d2/(d2+d1*x))
cgmm(x,rho,lambda)=rho<=0||lambda<=0?1/0:x<0?0.0:igamma(rho,x*lambda)
cgeometric(x,p)=p<=0||p>1?1/0: !isint(x)?1/0:x<0||p==0?0.0:p==1?1.0-exp((x+1)*log(1.0-p))
chalfnormal(x,sigma)=sigma<=0?1/0:x<0?0.0:erf(x/sigma/sqrt2)
chypgeo(x,N,C,d)=N<0||!isint(N)||C<0||C>N||!isint(C)||d<0||d>N||!isint(d)?1/0: !isint(x)?1/0:x<0||x<d-(N-C)?0.0:x>d||x>C?1.0:hypgeo(x,N,C,d)+chypgeo(x-1,N,C,d)
claplace(x,mu,b)=b<=0?1/0:(x<mu)?0.5*exp((x-mu)/b):1.0-0.5*exp(-(x-mu)/b)
clogistic(x,a,lambda)=lambda<=0?1/0:1.0/(1+exp(-lambda*(x-a)))
clognormal(x,mu,sigma)=sigma<=0?1/0:x<=0?0.0:cnormal(log(x),mu,

```

```

sigma)
cnormal(x,mu,sigma)=sigma<=0?1/0:0.5+0.5*erf((x-mu)/sigma/
sqrt2)
cmaxwell(x,a)=a<=0?1/0:x<0?0.0:igamma(1.5,a*a*x*x)
cnegbin(x,r,p)=r<=0||!isint(r)||p<=0||p>1?1/0: !isint(x)?1/0:x
<0?0.0:ibeta(r,x+1,p)
cnexp(x,lambda)=lambda<=0?1/0:x<0?0.0:1-exp(-lambda*x)
cpareto(x,a,b)=a<=0||b<0||!isint(b)?1/0:x<a?0.0:1.0-(real(a)/x
)**b
cpoisson(x,mu)=mu<=0?1/0:!isint(x)?1/0:x<0?0.0:1-igamma(x+1.0,
mu)
crayleigh(x,lambda)=lambda<=0?1/0:x<0?0.0:1.0-exp(-lambda*x*x)
csine(x,f,a)=a<=0?1/0: x<0?0.0:x>a?1.0:f==0?real(x)/a:(real(x)/
a-sin(f*twopi*x/a)/(f*twopi))/(1.0-sin(twopi*f)/(twopi*f)
)
ct(x,nu)=nu<0||!isint(nu)?1/0:0.5+0.5*sgn(x)*(1-ibeta(0.5*nu
,0.5,nu/(nu+x*x)))
ctriangular(x,m,g)=g<=0?1/0: x<m-g?0.0:x>=m+g?1.0:0.5+real(x-m
)/g-(x-m)*abs(x-m)/(2.0*g*g)
cuniform(x,a,b)=x<(aa:b)?0.0:x>=(a>b?a:b)?1.0:real(x-a)/(b-
a)
cweibull(x,a,lambda)=a<=0||lambda<=0?1/0:x<0?0.0:1.0-exp(-(lambda*x)**a)
gsampfunc(t,n) = t<0?0.5*1/(-t+1.0)**n:1.0-0.5*1/(t+1.0)**n
NO_ANIMATION = 1
fourinvsqrtpi = 2.25675833419103
invpi = 0.318309886183791
invsqrt2pi = 0.398942280401433
log2 = 0.693147180559945
sqrt2 = 1.4142135623731
sqrt2invpi = 0.797884560802865
twopi = 6.28318530717959
save_encoding = "utf8"
eps = 1e-10
xmin = -4.0
xmax = 4.0
ymin = -5
ymax = 0.877673016883152
r = 8
mu = 2.0
sigma = 0.5
p = 0.4
q = 3.0
n = 2
a = 1.5
b = 1.0
k = 2
lambda = 2.0

```

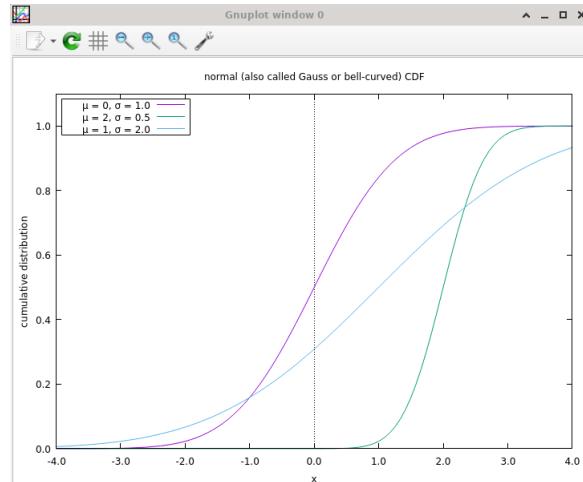
```

11 = 1.0
12 = 0.5
alpha = 1.0
u = 0.0
df1 = 5.0
df2 = 9.0
rho = 6.0
s = 4.66878227507107
N = 75
C = 25
d = 10
m = 3.08021684891803
plot [xmin:xmax] [0:1.1] mu = 0, sigma = 1.0, cnormal(x, mu,
    sigma) title "mean = 0, sigma = 1.0", \
mu = 2, sigma = 0.5, cnormal(x, mu, sigma) title "mean = 2,
    sigma = 0.5", \
mu = 1, sigma = 2.0, cnormal(x, mu, sigma) title "mean = 1,
    sigma = 2.0"

```

C++ Code 84: *cumulativedistributionfunction-normal*

The code for the beginning and middle part are the same as the probability density function code, the only difference are the title, the label, and the function we use here is **cnormal** not **normal**.

**Figure 28.5:** The cumulative distribution function for normal or bell-curved with certain parameters of μ and σ .

[DF*] Vector Field Plot

When the quantity that depends on x and y has both a magnitude and a direction it can be represented by an arrow whose length is proportional to the magnitude. If we divide the $x - y$ plane into a grid and draw an arrow at each grid point representing the direction and magnitude at that point, we have a vector plot. As we are associating two quantities, the magnitude and direction(or, equivalently, Δx and Δy) for each value of x and y , we can think of this type of visualization as a 4D plot. Vector plots are used for representing fluid

flows, electric fields, and many other physical systems.

Create a file that store the data that we want to plot as vector field, it should be 2 columns of data. Name it **data**, the one provided is obtained from Box2D simulation of a kinetic work example for moving a cutting tool along x axis for 46 meters. Then at the same working directory create a file name **vectorfieldplot**.

```
set xrange [-23:23]
set yrange [0:2]
# only integer x-coordinates
set samples 11
# only integer y-coordinates
set isosamples 11
# we need data, so we use the special filename "data", which
# produces x,y-pairs
set title "Vector Field"
plot "data" using 1:2:1:(2.*$2) with vectors title 'F(x,y) = <x
, 2y> Kinetic Work '
```

C++ Code 85: *vectorfieldplot*

Open gnuplot and type:
load 'vectorfieldplot'

Another alternative to plot the vector field is

```
set xrange [-5:5]
set yrange [-5:5]
dx(x) = x
dy(x) = 2*x
set title 'Vector field F(x,y) = <x, 2y>'
plot "data" using 1:2:(dx($1)):(dy($2)) w vec title 'F(x,y) = <
x, 2y>'
```

C++ Code 86: *vectorfieldplot1*

The third alternative to plot vector field on a plane is Another alternative to plot the vector field is

```
set xrange [-3*pi:3*pi]
set yrange [-pi:pi]
set iso 20
set samp 20
unset key
a = 2
plot "data" using 1:2:(-a*sin($1)*cos($2)): (a*cos($1)*sin($2))
\ 
with vectors size .06, 15 filled
```

C++ Code 87: *vectorfieldplot2*

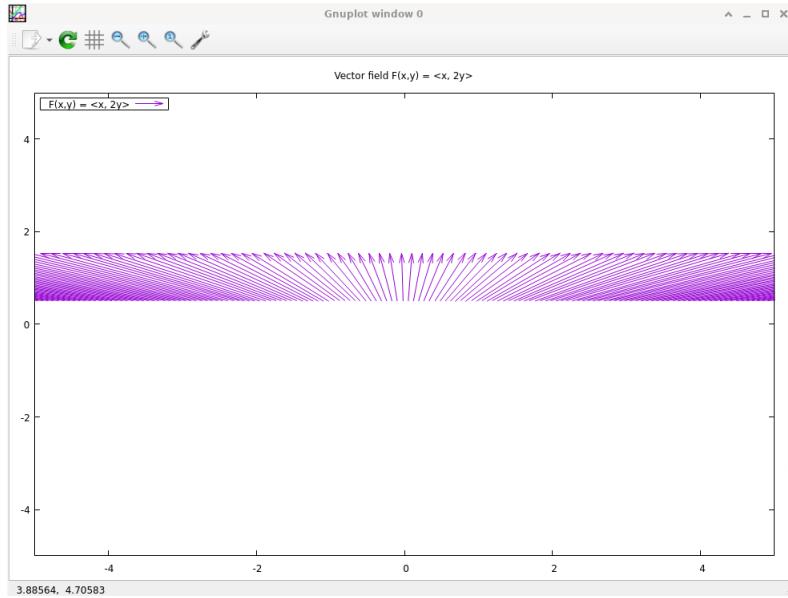


Figure 28.6: Vector field plot of $F(x,y) = \langle x, 2y \rangle$ from Kinetic Work data of x and y position of a cutting tool that is moved 46 meters.

How to figure out the arrows' location, magnitude and direction:

plot "data" using 1:2:(-a*sin(\$1)*cos(\$2)):(a*cos(\$1)*sin(\$2)) with vectors size .06, 15 filled

with the general syntax should look like this:

plot "data" using (first variable / x location of the arrow):(second variable / y location of the arrow):(function or constant to determine direction of each arrow):(function or constant to determine magnitude of each arrow)

We are going to examine that command for gnuplot, here are the explanations:

- **using 1:2**

Means the initial location / the tail of each arrow is (x, y) with x is the data from the first column(1) and y is the data from the second column(2). If the initial location is like this **using 2:1**, then the tail will be (y, x) .

- **:(-a*sin(\$1)*cos(\$2)):**

The direction of the arrow, the slope of the arrow. It is located in the middle after we determine the tail of the arrow, we determine the direction, it is a sine cosine function of the data, with \$1 represents column 1 or x , and \$2 represents column 2 or y , the direction for the i -th arrow will be toward $-a * \sin(x_i) * \cos(y_i)$, this explains why each arrow can have different direction.

- **:(a*cos(\$1)*sin(\$2))**

The magnitude for each arrow, again it is a sine cosine function toward x and y , this explains why each arrow may have different magnitude.

Basically, gnuplot can only plot vector fields when reading data from a file. In standard, your file will have to have 4 columns, x , y , delta x and delta y , and gnuplot will then plot a vector from (x,y) to $(x+\text{delta } x, y+\text{delta } y)$ for each line in the file, then type in gnuplot:

```
plot "file.dat" using 1:2:3:4 with vectors head filled lt 2
```

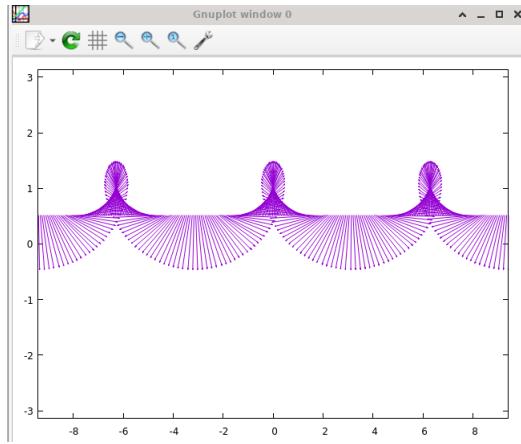


Figure 28.7: Vector field plot of $F(x, y) = \langle -2\sin(x) \cos(y), 2\cos(x) \sin(y) \rangle$ from Kinetic Work data of x and y position of a cutting tool that is moved 46 meters.

But with modification like the examples above, we can also use 2 columns of data.

[DF*] Parametric Plot

Parametric plot is a plot where the x and y values each depend on a third variable, called a parameter. The set parametric command changes the meaning of plot (splot) from normal functions to parametric functions. For 3-d plotting, the surface is described as $x = f(u, v)$, $y = g(u, v)$, $z = h(u, v)$. Therefore a triplet of functions is required. Create a file named **parametricplot**:

```
unset key
set parametric
set samp 1000
set view 60, 45
set urange [0: 10]
splot u*cos(10*u), u*sin(10*u), u lw 3
```

C++ Code 88: 3dparametricplot-invertedcone

Open gnuplot and type:

load '3dparametricplot-invertedcone'

Another example for plotting a sphere with parametric plot:

```
unset key
set parametric
set samp 1000
set view 60, 45
set xrange [-pi/2:pi/2]
set yrange [-pi/2:pi/2]
set xlabel 'x'
set ylabel 'y'
set zlabel 'z'
```

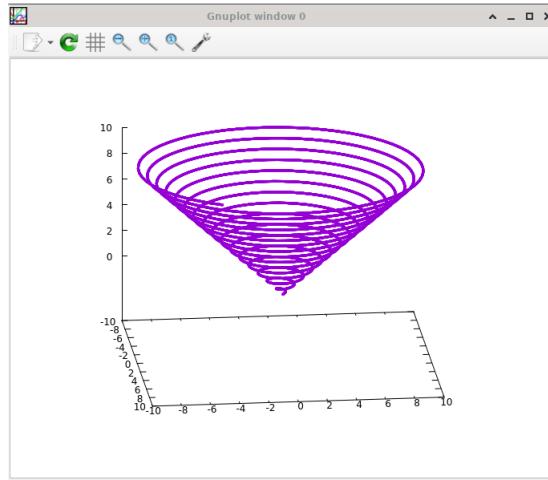


Figure 28.8: Parametric plot of function $(x, y) = (u * \cos(10 * u), u * \sin(10 * u), u)$ with $u = [0, 10]$ (DFSimulatorC/Source Codes/C++/Gnuplot/3dparametricplot-invertedcone).

```
set urange [-10: 10]
set vrangle [-10: 10]
splot cos(u)*cos(v),cos(u)*sin(v),sin(u) lw 3
```

C++ Code 89: parametricplotsphere

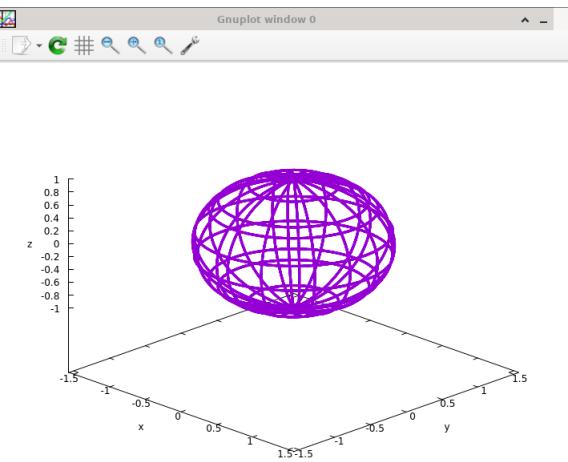


Figure 28.9: Parametric plot of function $(x, y, z) = (\cos(u) \cos(v), \cos(u) \sin(v), \sin(u))$ with $u = [-10, 10]$, $v = [-10, 10]$ (DFSimulatorC/Source Codes/C++/Gnuplot/3dparametricplot-sphere).

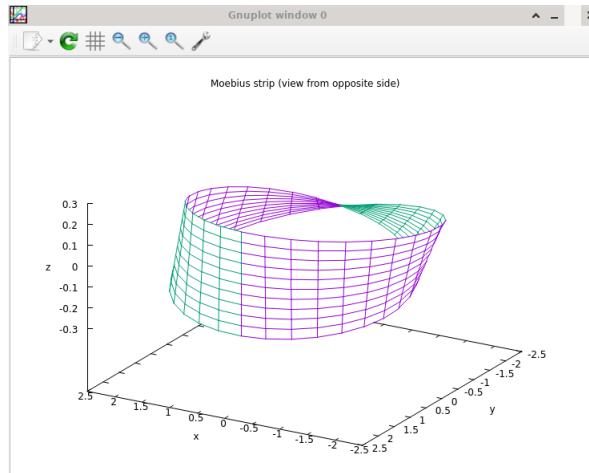


Figure 28.10: Parametric plot of function $(x,y,z) = 2 - v \sin(u/2) \sin(u), 2 - v \sin(u/2) \cos(u), v \cos(u/2)$, this function is known as Moebius strip (DFSimulatorC/Source Codes/C++/Gnuplot/3dparametricplot-moebiusstrip).

[DF*] Contour Plot

A contour plot traces value by drawing curves, just as in the topographical maps for the hikers. Create a file named **contourplot**.

```

set xrange [-4:4]
set yrang e [-4:4]
set iso 200
set samp 200
set key rmargin
unset surf
set contour base
set cntrparam levels auto 9
set view map
splot sin(y**2+x**2) - cos(x**2) title "sin(x^{2}) + y^{2}) - cos(x^{2})"

```

C++ Code 90: *contourplot*

Open gnuplot and type:
load 'contourplot'

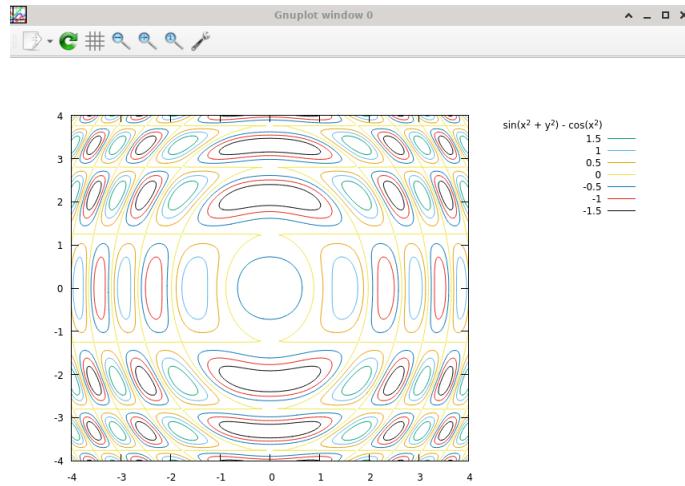


Figure 28.11: Contour plot of function $\sin(x^2 + y^2) - \cos(x^2)$.

[DF*] Complex Plot

When we observe the physical phenomena we can plot them and see the relation thus create a formula out of it, there are more beyond the conservative relation, when you have learn modern physics and more advanced physics, there are relation between each variable that can only be explained with complex functions.

Create a file named **3dcomplexfunctionplot**, or you can retrieve it from the repository, the code can be seen below.

```

#!/usr/local/bin/gnuplot --persist
# set terminal pngcairo background "#ffffff" enhanced font "times"
#          fontscale 1.0 size 640, 480
# set output 'complex_trig.11.png'
set border -1 front lt black linewidth 1.000 dashtype solid
set grid nopolar
set grid xtics nomxtics ytics nomytics noztics nomztics nortics
    nomrtics \
nox2tics nomx2tics noy2tics nomy2tics nocbtics nomcbtics
set grid layerdefault lt 0 linecolor 0 linewidth 0.500, lt 0
    linecolor 0 linewidth 0.500
unset key
unset parametric
set view 66, 336, 1.2, 1.2
set view equal xyz
set isosamples 100, 100
set size ratio 1 1,1
set style data lines
set xyplane at 0
set xtics norangelimit

```

```

set xtics ("−pi/2" −1.57080, "−pi/4" −0.785398, "0" 0.00000,
           "pi/4" 0.785398, "pi/2" 1.57080)
set ytics norangelimit
set ytics ("−pi/2" −1.57080, "−pi/4" −0.785398, "0" 0.00000,
           "pi/4" 0.785398, "pi/2" 1.57080)
unset ztics
set cbtics norangelimit
set cbtics ("0" −3.14159, "2pi" 3.14159)
set title "atanh( x + iy )"
set urange [ −1.57080 : 1.57080 ] noreverse nowriteback
set vrangle [ −1.57080 : 1.57080 ] noreverse nowriteback
set xlabel "Real"
set xlabel offset character 0, −2, 0 font "" textcolor lt −1
    rotate parallel
set xrange [ −1.57080 : 1.57080 ] noreverse nowriteback
set x2range [ * : * ] noreverse writeback
set ylabel "Imaginary"
set ylabel offset character 0, −2, 0 font "" textcolor lt −1
    rotate parallel
set yrange [ −1.57080 : 1.57080 ] noreverse nowriteback
set y2range [ * : * ] noreverse writeback
set zlabel "magnitude"
set zlabel offset character 3, 0, 0 font "" textcolor lt −1
    rotate
set zrange [ * : * ] noreverse writeback
set cblabel "Phase Angle"
set cblabel offset character −2, 0, 0 font "" textcolor lt −1
    rotate
set cbrange [ −3.14159 : 3.14159 ] noreverse nowriteback
set rrangle [ * : * ] noreverse writeback
set palette positive nops_allcF maxcolors 0 gamma 1.5 color
    model HSV
set palette defined ( 0 0 1 1, 1 1 1 1 )
set colorbox user
set colorbox vertical origin screen 0.85, 0.2 size screen 0.05,
    0.6 front noinvert bdefault
Hue(x,y) = (pi + atan2(−y,−x)) / (2*pi)
phase(x,y) = hsv2rgb( Hue(x,y), sqrt(x**2+y**2), 1. )
rp(x,y) = real(f(x,y))
f(x,y) = atanh(x + y*{0,1})
ip(x,y) = imag(f(x,y))
color(x,y) = hsv2rgb( Hue( rp(x,y), ip(x,y) ), abs(f(x,y)), 1.
    )
NO_ANIMATION = 1
save_encoding = "utf8"
## Last datafile plotted: "++"
splot '++' using 1:2:(abs(f($1,$2))):(color($1,$2)) with pm3d
    lc rgb variable

```

C++ Code 91: *3dcomplexfunctionplot*

Open gnuplot and type:
load '3dcomplexfunctionplot'

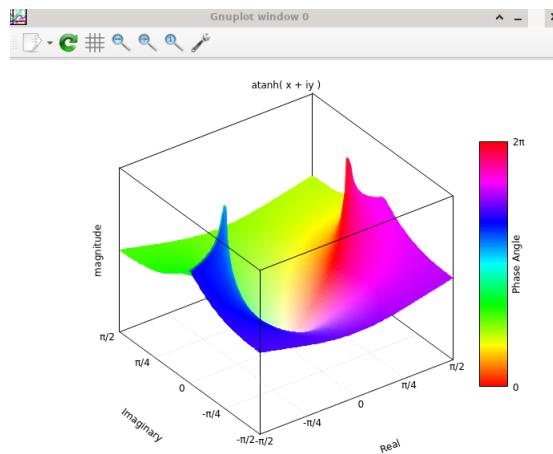


Figure 28.12: 3D plot of complex function $\text{atan}(x + iy)$ (DFSimulatorC/Source Codes/C++/Gnuplot/3dcomplexfunctionplot).

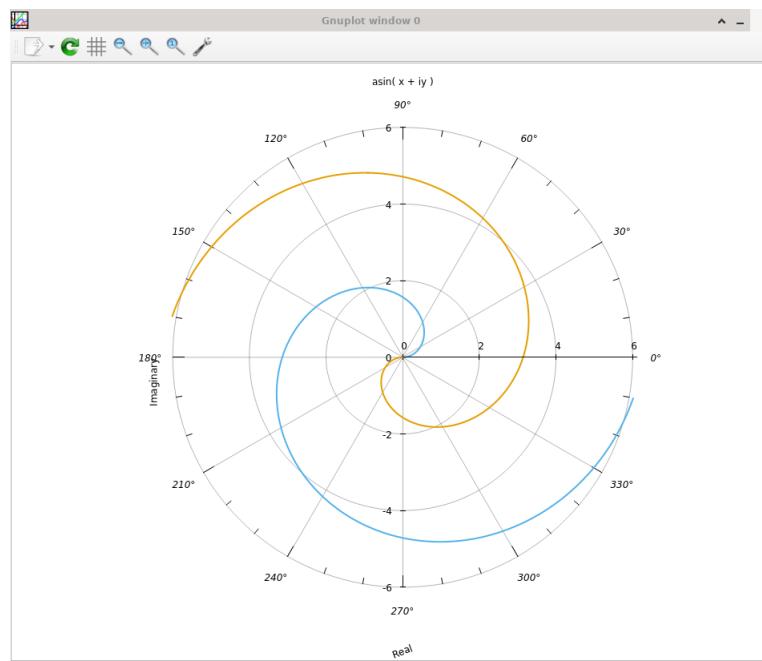


Figure 28.13: 2D polar plot of complex function $\text{asinh}(x + iy)$ (DFSimulatorC/Source Codes/C++/Gnuplot/2dpolarcomplexfunctionplot).

Bibliography

- [1] Boyce, William E., DiPrima, Richard C. (2010) Elementary Differential Equations and Boundary Value Problems 9th Edition, John Wiley & Sons, Hoboken, New Jersey, United States.
- [2] Burden, Richard L., Faires, J. Douglas (2001) Numerical Analysis 7th Edition, Brooks/Cole, Pacific Grove, California, United States.
- [3] Fletcher, Glenna. (1994) Mathematical Methods in Physics, Wm. C. Brown Publishers, Dubuque, VA, United States.
- [4] Gordon, Scott V., Clevenger, John (2019) Computer Graphics Programming in OpenGL with C++, Mercury Learning and Information, Dulles, VA, United States.
- [5] Haberman, Richard. (1977) Mathematical Models: Mechanical Vibrations, Population Dynamics, and Traffic Flow, Prentice-Hall, Englewood Cliffs, New Jersey, United States.
- [6] Walker, Jearl, Halliday, David, Resnick, Robert. (2014) Fundamental of Physics, John Wiley & Sons, Hoboken, New Jersey, United States.
- [7] Hardy, Y, Tan Kiat Shi and Steeb, W.-H. (2008). Computer Algebra with SymbolicC++, World Scientific Publishing, Singapore.
- [8] Hayt, William H. (2019). Engineering Circuit Analysis 9th Edition, McGraw-Hill Education, New York, United States.
- [9] Phillips, Lee (2020) Gnuplot 5 Second Edition, Alogus Publishing.
- [10] Shekar, Siddharth (2019) C++ Game Development By Example, Packt, Birmingham, United Kingdom.
- [11] Trefethen, Lloyd N., III, David Bau (1997) Numerical Linear Algebra, SIAM, 3600 University City Science Center, Philadelphia, United States.
- [12] Vries, Joey de. (2020) Learn OpenGL - Graphics Programming, Kendall & Welling.
- [13] Verzani, J., (2023) Calculus With Julia

Listings

1	test.cpp "Hey Beautiful Goddess."	16
2	main.cpp "Green Screen"	17
3	main.cpp "SOIL GLM Rotating Images"	20
4	main.cpp "SFML Keyboard Event"	28
5	main.cpp "Render Mesh with Lighting"	32
6	Camera.cpp "Render Mesh with Lighting"	35
7	Camera.h "Render Mesh with Lighting"	35
8	CMakeLists.txt "Render Mesh with Lighting"	36
9	LightRenderer.cpp "Render Mesh with Lighting"	36
10	LightRenderer.h "Render Mesh with Lighting"	38
11	Mesh.cpp "Render Mesh with Lighting"	39
12	Mesh.h "Render Mesh with Lighting"	43
13	ShaderLoader.cpp "Render Mesh with Lighting"	44
14	ShaderLoader.h "Render Mesh with Lighting"	46
15	main.cpp "Cube and Moving Camera"	49
16	camera.fs "Cube and Moving Camera"	58
17	camera.vs "Cube and Moving Camera"	58
18	main.cpp "Yaw Pitch Roll for 3D Cube"	60
19	fragShader.gsl "Yaw Pitch Roll for 3D Cube"	66
20	Utils.cpp "Yaw Pitch Roll for 3D Cube"	67
21	Utils.h "Yaw Pitch Roll for 3D Cube"	71
22	vertShader.gsl "Yaw Pitch Roll for 3D Cube"	72
23	main.cpp "Hello Box2D"	96
24	main.cpp "Bullet Example"	104
25	main.cpp "Slope plotting Gnuplot C++"	109
26	gnuplot_i.hpp	110
27	Makefile "Gnuplot Slope Makefile"	112
28	main.cpp "Function plotting Gnuplot C++"	113
29	main.cpp "3D Surface plotting Gnuplot C++"	115
30	main.cpp "3D Curve plotting Gnuplot C++"	117
31	matrix.txt "2D Plot from Textfile with Gnuplot through C++"	120
32	main.cpp "2D Plot from Textfile with Gnuplot through C++"	120
33	Makefile "2D Plot from Textfile with Gnuplot through C++"	122
34	main.cpp "2D Plot and Fit Curve from Textfile with Gnuplot through C++"	123
35	Makefile "2D Plot and Fit Curve from Textfile with Gnuplot through C++"	124
36	derivation.cpp "Symbolic Derivation for Function in One Variable C++"	128
37	integral.cpp "Symbolic Integration for Function in One Variable C++"	129

38	Makefile "SymbolicC++ Example"	129
39	tests/projectile_motion.cpp "Projectile Motion Box2D"	134
40	tests/projectile_dropped.cpp "Projectile Motion Box2D"	142
41	tests/circular_motion.cpp "Uniform Circular Motion Box2D"	156
42	tests/newton_firstlaw.cpp "Newton's First Law Box2D"	166
43	tests/newton_firstlaw2dforces.cpp "Newton's First Law and 2D Forces Box2D"	176
44	tests/pulley_jointtriangle.cpp "Applying Newton's Law Box2D"	186
45	tests/frictional_forces.cpp "Static Frictional Forces Box2D"	196
46	tests/pulley_kineticfriction.cpp "Pulley Kinetic Friction Box2D"	203
47	tests/dragforce_skier.cpp "Drag Force Skier Box2D"	211
48	tests/circular_motionairplane.cpp "Circular Motion Airplane Box2D"	221
49	tests/kineticenergy_collision.cpp "Kinetic Energy Train Crash Box2D"	228
50	tests/kineticenergy_work.cpp "Kinetic Energy Work Box2D"	237
51	tests/work_gravitationalforce.cpp "Push and Pull along a Ramp Box2D"	245
52	tests/spring_work.cpp "Work Done by A Spring Force Box2D"	253
53	tests/work_generalvariable.cpp "Work Done by a General Variable Box2D"	260
54	tests/work_pulleylevelator.cpp "Work Pulley Elevator Box2D"	270
55	tests/work_cratependulum.cpp "Crate Pendulum Box2D"	279
56	tests/work_verticalspringupward.cpp "Block Dropped onto Relaxed Spring Box2D"	285
57	tests/projectile_motion.cpp "Projectile Motion Box2D"	287
58	tests/projectile_motion.cpp "Projectile Motion Box2D"	289
59	tests/projectile_motion.cpp "Projectile Motion Box2D"	291
60	tests/projectile_motion.cpp "Projectile Motion Box2D"	293
61	tests/projectile_motion.cpp "Projectile Motion Box2D"	295
62	main.cpp "Gravity with 3 Bodies"	299
63	tests/gravity_check.cpp "Gravity Check Box2D"	308
64	tests/projectile_motion.cpp "Projectile Motion Box2D"	315
65	tests/simple_pendulum.cpp "Simple Pendulum Box2D"	325
66	tests/spring_test.cpp "Horizontal Spring Mass System Box2D"	333
67	tests/verticalspring_test.cpp "Simple Vertical Spring Box2D"	343
68	tests/spring_twomasses.cpp "Two Masses Spring-Mass System Box2D"	359
69	tests/projectile_motion.cpp "Projectile Motion Box2D"	369
70	tests/projectile_motion.cpp "Projectile Motion Box2D"	371
71	tests/projectile_motion.cpp "Projectile Motion Box2D"	373
72	tests/projectile_motion.cpp "Projectile Motion Box2D"	375
73	multiplicationofmatrices.cpp "Computation: Multiplication of matrices"	380
74	main.cpp "Matrix times a Vector from Textfile Data"	383
75	"vector1.txt"	385
76	"matrix1.txt"	385
77	tests/projectile_motion.cpp "Projectile Motion Box2D"	400
78	tests/projectile_motion.cpp "Projectile Motion Box2D"	400
79	tests/projectile_motion.cpp "Projectile Motion Box2D"	401
80	pendulumdata	405
81	histogramplot	405
82	histogramplot	406
83	probabilitydensityfunction-normal	406
84	cumulativedistributionfunction-normal	410
85	vectorfieldplot	415

86	vectorfieldplot1	415
87	vectorfieldplot2	415
88	3dparametricplot-invertedcone	417
89	parametricplotsphere	417
90	contourplot	419
91	3dcomplexfunctionplot	420