**Department of Computing**

**CS-361: Computer Graphics  
  
Class: BSCS-12ABC & SE12AB**

**Lab 12: Curves and surfaces**

**CLO 2 -** Apply mathematical and algorithmic principles to implement basic computer graphics techniques, such as line drawing and shading.

**CLO 3-** Develop interactive graphics applications using modern graphics APIs such as OpenGL or DirectX.

**CLO 4 -** Design and implement 2D and 3D graphical solutions for real-world problems.

**Date: 29th April 2025**

**Time: 2:00 PM – 4:50 PM**

# Instructor: Dr. Sidra Sutana

# Lab Engineer: Mr. Aftab Farooq

# **Name:** Ahmed Mohiuddin Shah

# **CMS ID:** 415216

# **Section:** BSCS-12-A

# **Lab:** 12

**Lab 12: Curves and surfaces**

**Introduction:**Curves and surfaces are foundational elements in computer graphics used for modeling smooth and complex shapes. Understanding their mathematical representations allows for precise control over shape and flow in 2D and 3D space. This lab focuses on the basics of curve modeling, the construction of parametric curves, and the concept of continuity that ensures smooth transitions between connected curves.

### **Lab Objective:**

### The objective of this lab is to introduce the fundamental principles of curve and surface modeling. Students will implement parametric curves such as Bezier curves and circular arcs, explore parametric (Cⁿ) and geometric (Gⁿ) continuity, and construct multi-segment curves ensuring smooth transitions between connected segments.

### **Tools/Software Requirement:**

* **Operating System:**
  + Windows / macOS / Linux (Ubuntu recommended)
* **Development Environment:**
  + **Windows:** [Code::Blocks](http://www.codeblocks.org/) or [Visual Studio](https://visualstudio.microsoft.com/)
  + **macOS:** [Xcode](https://developer.apple.com/xcode/)
  + **Linux:** GCC and g++ compilers
* **Graphics Libraries:**
  + **OpenGL** (built-in on macOS and Linux, available in Windows IDEs)
  + **GLUT** (OpenGL Utility Toolkit)
  + **GLEW** (OpenGL Extension Wrangler Library)
* **Package Manager (for macOS/Linux):**
  + **Homebrew** (macOS): brew install freeglut glew
  + **APT** (Linux): sudo apt-get install freeglut3-dev glew-utils
* **Compilers:**
  + **Windows:** MinGW (for Code::Blocks) or Microsoft C++ Compiler (for Visual Studio)
  + **macOS/Linux:** GCC/G++

### **Prerequisites:**

* Understanding of 2D and 3D coordinate systems.
* Familiarity with vector operations and basic linear algebra.
* Prior exposure to interpolation techniques and mathematical functions.

### 

### **Task Description:**

## Task 1: Basics of Curves

**Objective:** Understand how curves differ from polylines and how control points influence their shape.

**Steps:**

* Create a visual comparison of a polyline vs. a smooth curve (e.g., quadratic Bezier).
* Show how adjusting control points changes curve behavior.
* Render the control polygon..

## 

## Code:

#include <raylib.h>

#include <cmath>

#include <vector>

#include <algorithm>

// Function to calculate a point on a quadratic Bezier curve

Vector2 QuadraticBezier(Vector2 *p0*, Vector2 *p1*, Vector2 *p2*, *float* *t*)

{

*float* u = 1.0f - *t*;

*float* tt = *t* \* *t*;

*float* uu = u \* u;

Vector2 point = {

uu \* *p0*.x + 2 \* u \* *t* \* *p1*.x + tt \* *p2*.x,

uu \* *p0*.y + 2 \* u \* *t* \* *p1*.y + tt \* *p2*.y};

return point;

}

*void* drawBezierCurve(Vector2 *p0*, Vector2 *p1*, Vector2 *p2*, *int* *segments*)

{

for (*int* i = 0; i < *segments*; ++i)

{

*float* t1 = (*float*)i / (*float*)*segments*;

*float* t2 = (*float*)(i + 1) / (*float*)*segments*;

Vector2 point1 = QuadraticBezier(*p0*, *p1*, *p2*, t1);

// Vector2 point2 = QuadraticBezier(p0, p1, p2, t2);

DrawCircleV(point1, 2, BLUE);

}

}

*struct* Curve

{

Vector2 p[3];

*void* draw()

{

DrawLine(p[0].x, p[0].y, p[1].x, p[1].y, BLUE);

DrawLine(p[1].x, p[1].y, p[2].x, p[2].y, BLUE);

DrawCircle(p[0].x, p[0].y, 5, RED);

DrawCircle(p[1].x, p[1].y, 5, RED);

DrawCircle(p[2].x, p[2].y, 5, RED);

}

*void* drawBezier(*int* *segments*)

{

for (*int* i = 0; i < *segments*; ++i)

{

*float* t1 = (*float*)i / (*float*)*segments*;

Vector2 point1 = QuadraticBezier({p[0].x, p[0].y}, {p[1].x, p[1].y}, {p[2].x, p[2].y}, t1);

DrawCircleV(point1, 2, BLUE);

}

}

};

*struct* ContinousCurve

{

std::vector<Curve> curves;

*void* addCurve(Curve *c*) {

if (curves.empty())

{

curves.push\_back(*c*);

}

else

{

Curve &lastCurve = curves.back();

lastCurve.p[2] = *c*.p[0]; // Connect the end of the last curve to the start of the new curve

curves.push\_back(*c*);

}

}

*void* draw()

{

for (Curve &c : curves)

{

c.draw();

c.drawBezier(100);

}

}

*void* drawBezier(*int* *segments*)

{

for (Curve &c : curves)

{

c.drawBezier(*segments*);

}

}

};

*int* main()

{

InitWindow(800, 600, "Task 1 - Curve Visualization - Polyline vs Bezier");

// SetTargetFPS(60);

ContinousCurve continousCurve;

// Create individual curves

Curve curve1 = {{{100, 300}, {200, 100}, {300, 300}}};

Curve curve2 = {{{300, 300}, {400, 500}, {500, 300}}};

Curve curve3 = {{{500, 300}, {600, 100}, {700, 300}}};

// Add curves to the continuous curve

continousCurve.addCurve(curve1);

continousCurve.addCurve(curve2);

continousCurve.addCurve(curve3);

while (!WindowShouldClose())

{

if (IsMouseButtonDown(MOUSE\_LEFT\_BUTTON))

{

for (*int* i = 0; i < continousCurve.curves.size(); i++)

{

for (*int* j = 0; j < 3; j++)

{

if (CheckCollisionPointCircle(GetMousePosition(), continousCurve.curves[i].p[j],40))

{

continousCurve.curves[i].p[j] = GetMousePosition();

}

}

}

}

// add a new curve

if (IsMouseButtonPressed(MOUSE\_RIGHT\_BUTTON))

{

Vector2 p0 = GetMousePosition();

Vector2 p1 = {p0.x + 100, p0.y - 100};

Vector2 p2 = {p0.x + 200, p0.y};

Curve newCurve = {{p0, p1, p2}};

continousCurve.addCurve(newCurve);

}

BeginDrawing();

ClearBackground(RAYWHITE);

// Draw the continuous curve

continousCurve.draw();

EndDrawing();

}

CloseWindow();

}

## Output:

## 

## 

## Task 2: Drawing Parametric Curves

**Objective:** Implement and render basic parametric curves.

**Steps:**

* Plot a circle, ellipse, and spiral using parametric equations.
* Use a slider or animation to move a point along the curve.
* Allow real-time interaction to change parameters (e.g., radius, angle).

Code:

#include <raylib.h>

#include <cmath>

#define RAYGUI\_IMPLEMENTATION

#include "raygui.h"

*int* main()

{

InitWindow(1280, 720, "Task 2 - Curve Visualization - Parametric Curves");

SetTargetFPS(60);

*float* t = 0.0f; // Parameter for animation

*float* spiralT = 0.0f; // Parameter for spiral

*float* circleradius = 100.0f; // Radius for circle

*float* ellipseRadiusX = 100.0f; // X radius for ellipse

*float* ellipseRadiusY = 50.0f; // Y radius for ellipse

*float* angleIncrement = 0.01f; // Increment for animation

*int* sign = 1; // Sign for direction

*float* spiralSpacing = 5.0f; // Spacing for spiral

*float* length = 4.0f; // Length of spiral

while (!WindowShouldClose())

{

// Update

t += angleIncrement;

spiralT += angleIncrement \* sign;

// Draw

BeginDrawing();

ClearBackground(RAYWHITE);

// Draw Circle

for (*float* theta = 0; theta < 2 \* PI; theta += 0.01f)

{

*float* x = 400 + circleradius \* cos(theta);

*float* y = 300 + circleradius \* sin(theta);

DrawPixel(x, y, RED);

}

// Draw Ellipse

for (*float* theta = 0; theta < 2 \* PI; theta += 0.01f)

{

*float* x = 600 + ellipseRadiusX \* cos(theta);

*float* y = 400 + (ellipseRadiusY / 2) \* sin(theta);

DrawPixel(x, y, BLUE);

}

// Draw Spiral

for (*float* theta = 0; theta < length \* PI; theta += 0.01f)

{

*float* x = 400 + (spiralSpacing \* theta) \* cos(theta);

*float* y = 300 + (spiralSpacing \* theta) \* sin(theta);

DrawPixel(x, y, GREEN);

}

// Animate a point along the circle

*float* pointX = 400 + circleradius \* cos(t);

*float* pointY = 300 + circleradius \* sin(t);

DrawCircle(pointX, pointY, 5, BLACK);

// Animate a point along the ellipse

*float* ellipseX = 600 + ellipseRadiusX \* cos(t);

*float* ellipseY = 400 + (ellipseRadiusY / 2) \* sin(t);

DrawCircle(ellipseX, ellipseY, 5, BLACK);

// Animate a point along the spiral according to length

if (spiralT > length \* PI)

{

// spiralT = 0.0f;

sign \*= -1; // Reverse direction

}

else if (spiralT < 0)

{

// spiralT = 0.0f;

sign \*= -1; // Reverse direction

}

*float* spiralX = 400 + (spiralSpacing \* spiralT) \* cos(spiralT);

*float* spiralY = 300 + (spiralSpacing \* spiralT) \* sin(spiralT);

DrawCircle(spiralX, spiralY, 5, BLACK);

// Draw UI

GuiLabel((Rectangle){150, 10, 200, 20}, "Circle Animation");

// Sliders for Circle

GuiSlider((Rectangle){150, 40, 200, 20}, "Circle Radius", NULL, &circleradius, 50.0f, 200.0f);

// Sliders for Ellipse

GuiSlider((Rectangle){150, 70, 200, 20}, "Ellipse Radius X", NULL, &ellipseRadiusX, 50.0f, 200.0f);

GuiSlider((Rectangle){150, 100, 200, 20}, "Ellipse Radius Y", NULL, &ellipseRadiusY, 25.0f, 100.0f);

// Sliders for Spiral

GuiSlider((Rectangle){150, 130, 200, 20}, "Spiral Spacing", NULL, &spiralSpacing, 1.0f, 20.0f);

GuiSlider((Rectangle){150, 160, 200, 20}, "Spiral Length", NULL, &length, 1.0f, 20.0f);

GuiSlider((Rectangle){150, 190, 200, 20}, "Angle Increment", NULL, &angleIncrement, 0.01f, 0.1f);

GuiLabel((Rectangle){150, 220, 200, 20}, "Press ESC to exit");

// printf("Circle Radius: %.2f\n", spiralT);

EndDrawing();

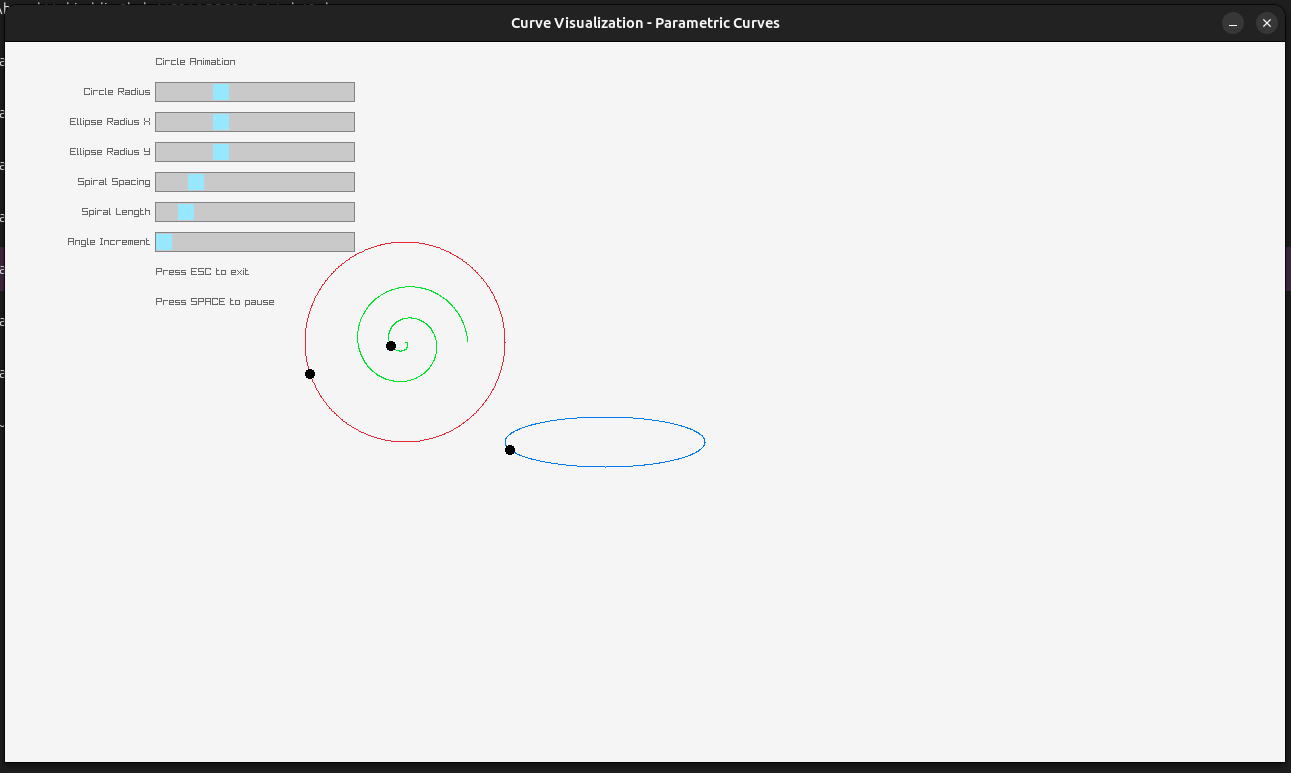
}

CloseWindow();

return 0;

}

Output:



## 

## Task 3: Geometric and Parametric Continuity

**Objective:** Explore the difference between **G¹/G²** and **C¹/C²** continuity.

**Steps:**

* Create a multi-segment Bezier curve.
* Implement continuity conditions:
  + For **C¹**: ensure shared control points and tangents match.
  + For **G¹**: ensure only tangent direction continuity.
* You need to adjust control points and visualize effects on continuity.

Code:

#include <raylib.h>

#include <cmath>

#include <vector>

#include <algorithm>

// Function to calculate a point on a quadratic Bezier curve

Vector2 QuadraticBezier(Vector2 *p0*, Vector2 *p1*, Vector2 *p2*, *float* *t*)

{

*float* u = 1.0f - *t*;

*float* tt = *t* \* *t*;

*float* uu = u \* u;

Vector2 point = {

uu \* *p0*.x + 2 \* u \* *t* \* *p1*.x + tt \* *p2*.x,

uu \* *p0*.y + 2 \* u \* *t* \* *p1*.y + tt \* *p2*.y};

return point;

}

*struct* Curve

{

Vector2 p[3];

*void* draw()

{

DrawLine(p[0].x, p[0].y, p[1].x, p[1].y, BLUE);

DrawLine(p[1].x, p[1].y, p[2].x, p[2].y, BLUE);

DrawCircle(p[0].x, p[0].y, 5, RED);

DrawCircle(p[1].x, p[1].y, 5, RED);

DrawCircle(p[2].x, p[2].y, 5, RED);

}

*void* drawBezier(*int* *segments*)

{

for (*int* i = 0; i < *segments*; ++i)

{

*float* t1 = (*float*)i / (*float*)*segments*;

Vector2 point1 = QuadraticBezier(p[0], p[1], p[2], t1);

DrawCircleV(point1, 2, BLUE);

}

}

};

*struct* ContinousCurve

{

std::vector<Curve> curves;

*void* addCurve(Curve *c*)

{

if (curves.empty())

{

curves.push\_back(*c*);

}

else

{

Curve &lastCurve = curves.back();

lastCurve.p[2] = *c*.p[0]; // Connect the end of the last curve to the start of the new curve

curves.push\_back(*c*);

}

}

*void* enforceC1Continuity()

{

for (size\_t i = 1; i < curves.size(); ++i)

{

curves[i].p[0] = curves[i - 1].p[2]; // Shared control point

curves[i].p[1] = {

2 \* curves[i].p[0].x - curves[i - 1].p[1].x,

2 \* curves[i].p[0].y - curves[i - 1].p[1].y}; // Adjust tangent

}

}

*void* enforceG1Continuity()

{

for (size\_t i = 1; i < curves.size(); ++i)

{

curves[i].p[0] = curves[i - 1].p[2]; // Shared control point

Vector2 tangent = {

curves[i].p[0].x - curves[i - 1].p[1].x,

curves[i].p[0].y - curves[i - 1].p[1].y}; // Tangent direction

*float* length = sqrt(tangent.x \* tangent.x + tangent.y \* tangent.y);

tangent.x /= length;

tangent.y /= length;

*float* magnitude = sqrt(pow(curves[i].p[1].x - curves[i].p[0].x, 2) +

pow(curves[i].p[1].y - curves[i].p[0].y, 2));

curves[i].p[1] = {

curves[i].p[0].x + tangent.x \* magnitude,

curves[i].p[0].y + tangent.y \* magnitude}; // Adjust tangent direction

}

}

*void* draw()

{

for (Curve &c : curves)

{

c.draw();

c.drawBezier(100);

}

}

};

*int* main()

{

InitWindow(800, 600, "Task 3 - Curve Visualization - G1 vs C1 Continuity");

SetTargetFPS(60);

ContinousCurve continousCurve;

// Create individual curves

Curve curve1 = {{{100, 300}, {200, 100}, {300, 300}}};

Curve curve2 = {{{300, 300}, {400, 500}, {500, 300}}};

Curve curve3 = {{{500, 300}, {600, 100}, {700, 300}}};

// Add curves to the continuous curve

continousCurve.addCurve(curve1);

continousCurve.addCurve(curve2);

continousCurve.addCurve(curve3);

*bool* enforceC1 = false;

*bool* enforceG1 = false;

while (!WindowShouldClose())

{

if (IsKeyPressed(KEY\_ONE))

{

enforceC1 = true;

enforceG1 = false;

continousCurve.enforceC1Continuity();

}

if (IsKeyPressed(KEY\_TWO))

{

enforceC1 = false;

enforceG1 = true;

continousCurve.enforceG1Continuity();

}

if (IsMouseButtonDown(MOUSE\_LEFT\_BUTTON))

{

for (*int* i = 0; i < continousCurve.curves.size(); i++)

{

for (*int* j = 0; j < 3; j++)

{

if (CheckCollisionPointCircle(GetMousePosition(), continousCurve.curves[i].p[j], 40))

{

continousCurve.curves[i].p[j] = GetMousePosition();

}

}

}

}

BeginDrawing();

ClearBackground(RAYWHITE);

DrawText("Press 1 for C1 Continuity", 10, 10, 20, DARKGRAY);

DrawText("Press 2 for G1 Continuity", 10, 40, 20, DARKGRAY);

continousCurve.draw();

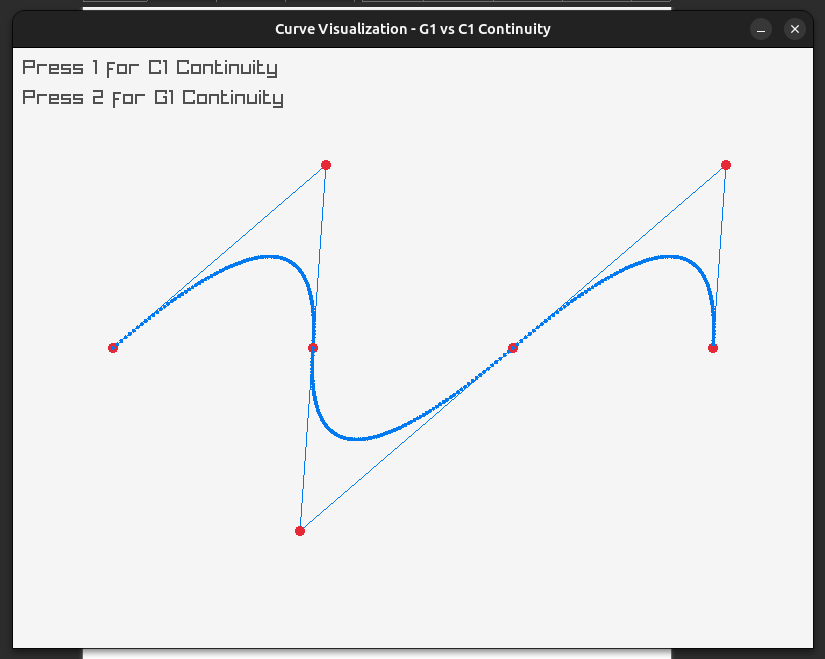
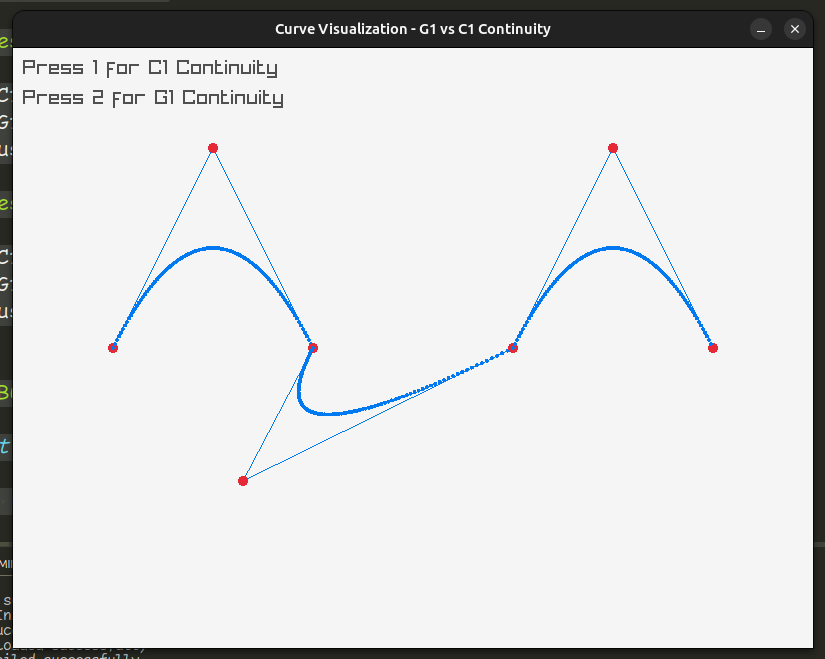
EndDrawing();

}

CloseWindow();

}

Output:



### 

### **Deliverables:**

 Compile a single word document by filling in the solution part and submit this Word file on LMS

 Include screenshots of the program outputs.

 Submit your Lab Word File and code files seperately on submission link.

# Lab Rubrics

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Lab Rubrics for (Lab-12:Curves and surfaces) | | | | | |
|  | | | | | |
| **Sr.**  **No.** | **Assessment** | **Unacceptable (0 Marks)** | **Does Not Meet Expectations (1/2 Marks)** | **Meets Expectations (3/4 Marks)** | **Exceeds Expectations (5 Marks)** |
| **1** | **Illustrating the basic understanding of semantics and syntax**  **(CLO3, PLO5)** | The student did not submit any work.  OR  The student plagiarized the solution and/or used unfair means. | The student is unable to demonstrate the understanding of syntax of C language and is unable to write an executable code.  The student is not able to understand the structure of a program at all. | The student demonstrates some understanding of syntax of C language and is able to write a code with few errors.  The student is able to understand the structure but still learning the syntax. | The student demonstrates good understanding of syntax of C language and is able to write executable code without help  The student is able to understand the structure and is able to identify problems in the code  when introduced |
| **2** | **Software Tool Usage**  **(CLO4-PLO3)** | The student demonstrates a lack of understanding of tool usage.  Implementation has syntax/semantic/runtime errors, and the student is unable to debug and correct the errors.  The code has inadequate comments and variable names and does not adhere to the coding standards.  No Error handling has been performed.  Documentation is poorly structured. | The student demonstrates some understanding of tool usage.  The codes are correct in terms of their syntax, however, the program output is not always correct in all test cases.  The code has limited comments and inconsistent variable names and may not adhere to the coding standards.  Some Error handling has been performed.  Documentation is adequately structured. | The student demonstrates a good understanding of tool usage.  Furthermore, his/her coding is complete and functional, and the program output is correct in all test cases.  The code has sufficient comments and consistent variable names and reasonably adhere to the coding standards.  Adequate Error handling has been performed.  Documentation is well structured. |