Functions & Graphing

Lecture 19

Hartmut Kaiser

https://teaching.hkaiser.org/spring2024/csc3380/

Software Development Notes

Test Driven Development

- Add requirements/stories
- Write automated tests for each
- Write code until it passes
- Test pass % acts as progress bar to next release

Manual Testing

- The most basic form of testing
  - Run the program yourself
  - When an error occurs, write it down
  - Ensure that the error can be reproduced

- Create issue on github
  - How to run? What input used?
  - What system run on?
  - Everything needed to reproduce
  - What result is expected? What result is seen?

Issues with Manual Testing

- To be effective, bugs must be tracked
  - Bug reproduction steps must be carefully retained
  - Have to re-test after changes
- Expensive and error-prone

Automated Testing

- Manual testing is still necessary for sanity checks
- But we can create automated tests to provide immediate feedback
- Tests are automatically run for each change set (commit)
- We receive a checklist afterwards seeing how many, and which, tests succeeded and failed

Unit Testing

- The most common kind of automated testing is Unit Testing
- Unit testing is a form of automated testing where you test a single class, module, or method
- A unit is the smallest testable portion of an application:
  - Each unit test tests one thing
  - We can test every function of the unit, and every meaningful case of the function
  - Pay particular attention to testing boundary cases
    - Zero length strings
    - Check at minimum/maximum index bounds

Regressions

- It is extremely common for a new update to fix old bugs, but introduce new ones
- If a test used to pass, but after applying a change set it fails, this is called a regression
- Automated testing provides a mechanism to quickly discover regressions
- When fixing regressions always add a test verifying it has been fixed

Integration Tests

- An integration test crosses unit boundaries
  - When more than one class is involved
  - Interactions with external systems
- Simple examples:
  - Testing that a database works with a software system
  - Testing that a frontend works with a backend
  - Testing a system component works with another system component
  - Testing that the interface for an external data source works with a software system

Mutation Testing

- Basic idea: test the tests
- Take some code that passes all the tests
- Mutate that code (an operation, a constant, etc.)
- One of the tests should now fail
- If not, you need more tests!

Test Coverage

- Automated testing raises an interesting problem:
  - How do we know we've tested everything
- How many tests do we actually need?
  - Tests take time to execute
  - There is no benefit to redundant tests
  - We might be missing an important test case

Test Coverage

- Function coverage: has every function/method been called at least once?
- Statement coverage: has every statement been executed once?
- Branch coverage: has every branch been executed?
- Condition coverage: has every Boolean expression been evaluated as both true and false.

Function Coverage

- Every function called at least once
- This is the absolute bare minimum level of coverage
- Even still, it is surprisingly difficult

Statement Coverage

- The most common “bare-minimum” level of testing observed in real software
- Ensure that each statement is executed at least once

Statement Coverage: How to Calculate

- (#tested statements) / (#statements)
  - That's it
  - Shoot for 100%
- Note: we care about the innermost statements
  - the ones inside the body of ifs and loops

Branch Coverage

- Every branch must be covered
- Often similar to statement coverage

Limits on Testing

- Testing can find the presence of faults, not absence
- Testing is difficult for certain domains (games, graphics, …)
- Testing assumes that methods terminate (the halting problem)
  - If they don't, testing must freeze or be occasionally wrong
- Testing is not a substitute for code review

Zero-Defects Philosophy

- Introduced by Philip Crosby at the Pershing Missile Program
- The philosophy was introduced to reduce the failure rate of the Pershing Missile
- Major failures happen when engineers tolerate mistakes because they know inspectors will detect them later
- Getting it right the first time dramatically reduces errors

Zero-defects Software Development

- Don’t write code you know to be bad, because your friends will catch the error in testing
- Don’t build new code on defective code
- Don’t add features if your tests aren’t passing
  - Create new features only if all existing tests pass successfully
  - Create new tests with each new feature

Functions & Graphing

Abstract

- Here we present ways of graphing functions and data and some of the programming techniques needed to do so, notably scaling.

Note

- This course is about programming
  - The examples – such as graphics – are simply examples of
    - Useful programming techniques
    - Useful tools for constructing real programs
  - Look for the way the examples are constructed
    - How are "big problems" broken down into little ones and solved separately?
    - How are classes defined and used
      - Do they have sensible data members?
      - Do they have useful member functions?
    - Use of variables
      - Are there too few?
      - Too many?
      - How would you have named them better?

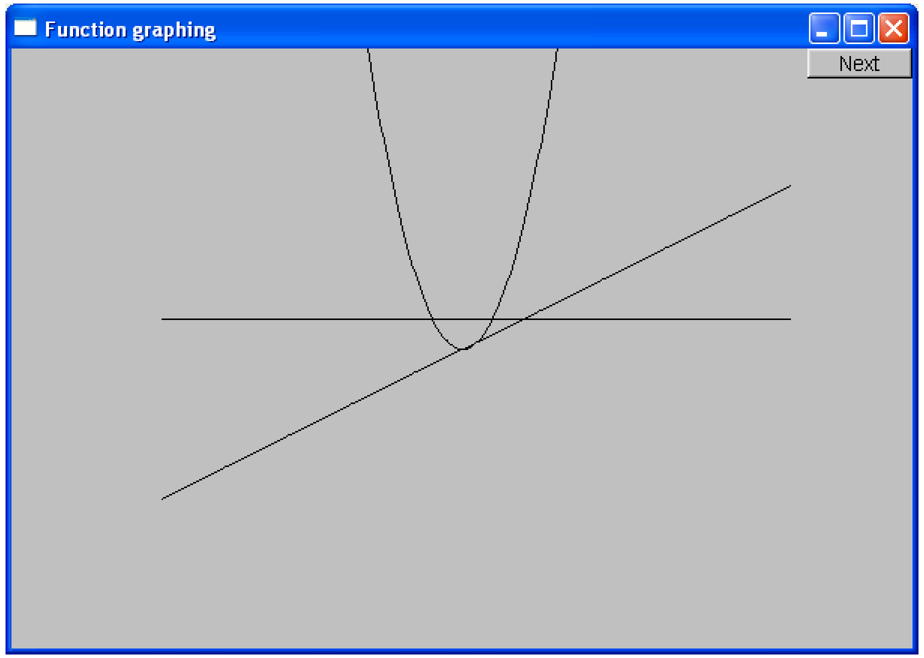
Graphing functions

- Start with something really simple
  - Always remember "Hello, World!"
- We graph functions of one argument yielding a value
  - Plot (x, f(x)) for values of x in some range [r1,r2)
- Let's graph three simple functions

```
// y == 1
double one(double) { return 1; }

// y == x/2
double slope(double x) { return x / 2; }

// y == x*x
double square(double x) { return x * x; }
```

Functions

```
// y == 1
double one(double) { return 1; }

// y == x/2
double slope(double x) { return x / 2; }

// y == x*x
double square(double x) { return x * x; }
```



How do we write code to do this?

Function to be graphed

```
// make a window
Simple_window win(Point(100, 100), xmax, ymax, "Function graphing");

Function s(one, -10, 11, orig, n_points, x_scale, y_scale);
Function s2(slope, -10, 11, orig, n_points, x_scale, y_scale);
Function s3(square, -10, 11, orig, n_points, x_scale, y_scale);

win.attach(s);     // attach Lines x to Window win
win.attach(s2);    // attach Lines x to Window win
win.attach(s3);    // attach Lines x to Window win

win.wait_for_button();    // Draw!
```

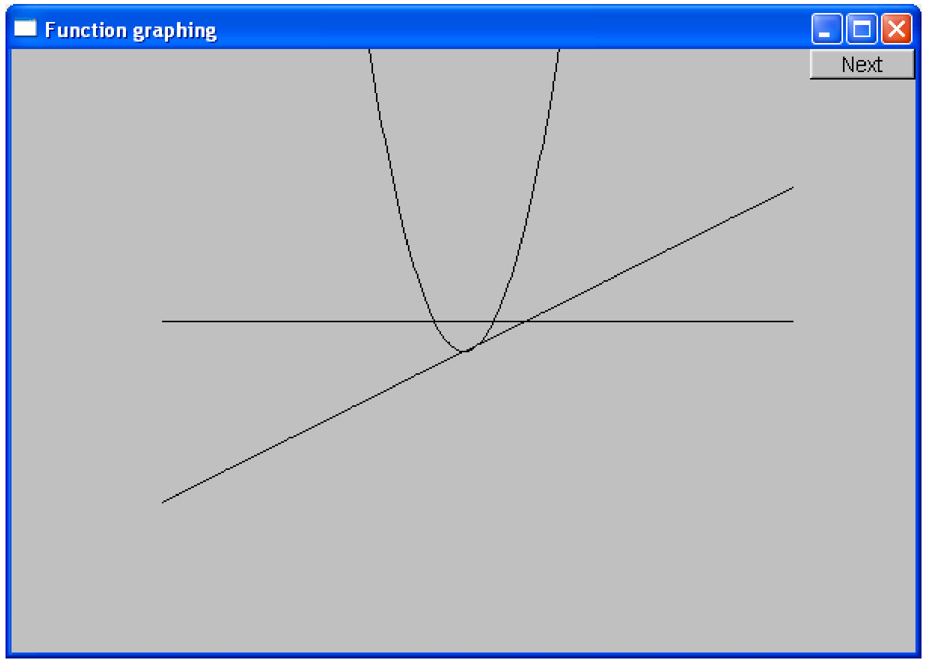
“stuff” to make the graph fit into the window

Range in which to graph

We need some Constants

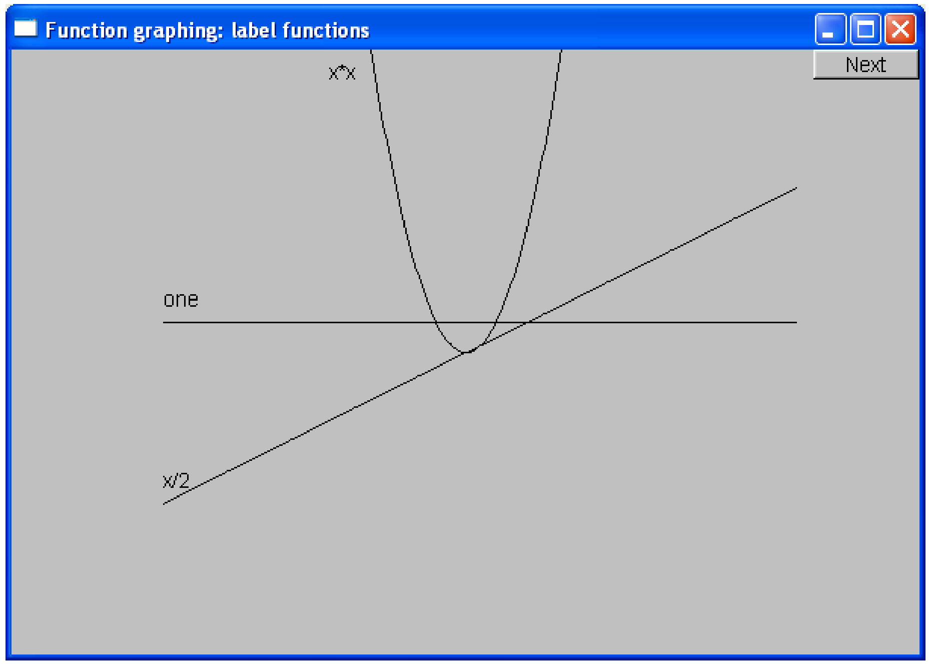
```
const int xmax = 600;        // window size
const int ymax = 400;

const int x_orig = xmax / 2;
const int y_orig = ymax / 2;
const Point orig(x_orig, y_orig);    // position of (0, 0) in window

const int r_min = -10;       // range [-10:11) == [-10:10] of x
const int r_max = 11;

const int n_points = 400;    // number of points used in range

const int x_scale = 20;      // scaling factors (one unit == 20 pixels)
const int y_scale = 20;

// Choosing a center (0, 0), scales, and number of points can be fiddly
// The range usually comes from the definition of what you are doing
```

Functions – but what does it mean?



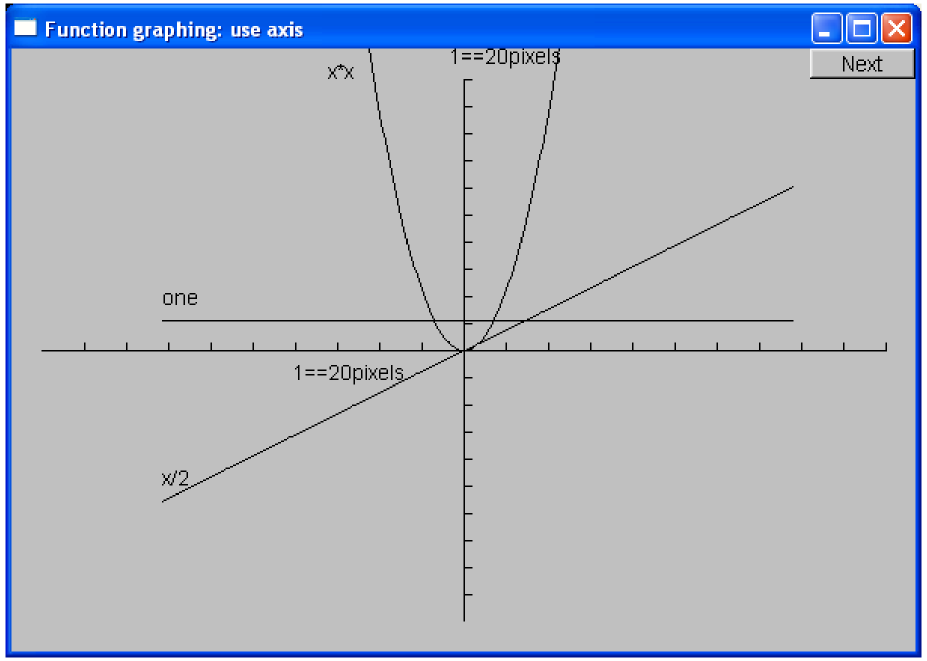
- What's wrong with this?
  - No axes (no scale)
  - No labels

Label the Functions



```
Text ts(Point(100, y_orig - 30), "one");
Text ts2(Point(100, y_orig + y_orig / 2 - 10), "x/2");
Text ts3(Point(x_orig - 90, 20), "x*x");
```

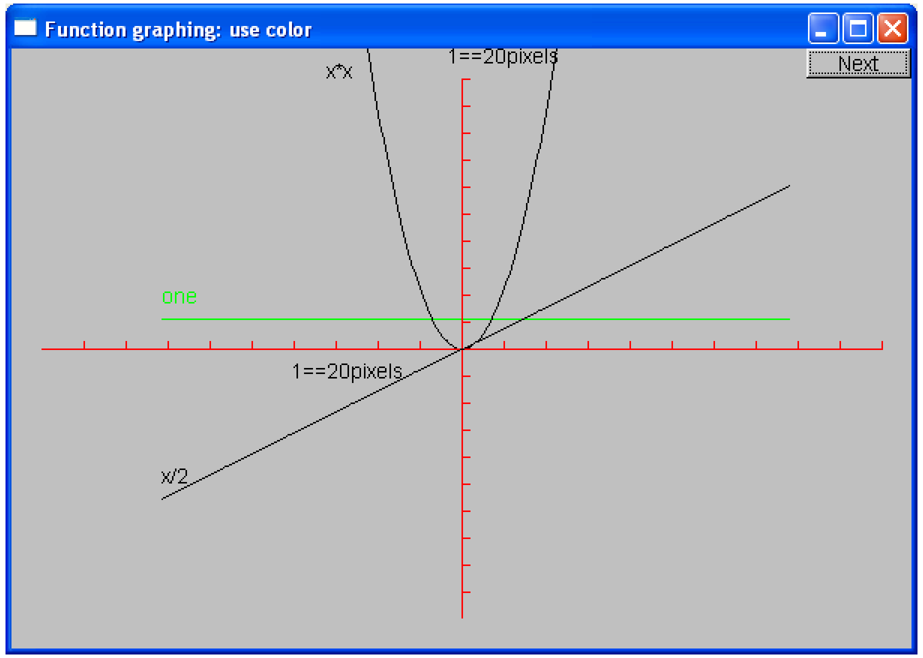
Add x-axis and y-axis



- We can use axes to show (0,0) and the scale:

```
Axis x(Axis::x, Point(20, y_orig),
    xlength, xlength / x_scale, "1 == 20 pixels");
Axis y(Axis::y, Point(x_orig, ymax - 20),
    ylength, ylength / y_scale, "1 == 20 pixels");
```

Use color (in moderation)



```
s.set_color(Graph_lib::Color::green);
ts.set_color(Graph_lib::Color::green);

x.set_color(Graph_lib::Color::red);
y.set_color(Graph_lib::Color::red);
```

The Implementation of Function

- We need a type for the argument specifying the function to graph
  - `using` can be used to declare a new name for a type

```
using color = int;              // now color means int
```

  - Define the type of our desired argument, Fct

```
using Fct = std::function<double(double)>;    // now Fct means function
                                              // taking a double argument
                                              // and returning a double
```

  - Examples of functions of type `double(double)`:

```
double one(double x) { return 1; }          // y==1
double slope(double x) { return x / 2; }   // y==x/2
double square(double x) { return x * x; }  // y==x*x
```

Now Define “Function”

```
struct Function : Shape    // Function is derived from Shape
{
    // all it needs is a constructor:
    Function(Fct f,        // f is a Fct  (takes a double, returns a double)
        double r1,         // the range of x values (arguments to f) [r1:r2)
        double r2,
        Point orig,        // the screen location of (0, 0)
        int count,         // number of points used to draw the function
                           // (number of line segments used is count-1)
        double xscale,     // the location (x, f(x)) is (xscale*x, yscale*f(x))
        double yscale);
};
```

Implementation of Function

```cpp
Function::Function(Fct f, double r1, double r2,
    Point xy, int count, double xscale, double yscale)
{
    if (r2 - r1 <= 0)
        throw std::runtime_error("bad graphing range");
    if (count <= 0)
        throw std::runtime_error("non-positive graphing count");

    double dist = (r2 - r1) / count;
    double r = r1;
    for (int i = 0; i < count; ++i)
    {
        add(Point(xy.x + int(r * xscale), xy.y - int(f(r) * yscale)));
        r += dist;
    }
}
```

Default Arguments

- Seven arguments are too many!
  - Many too many
  - We're just asking for confusion and errors
  - Provide defaults for some (trailing) arguments
    - Default arguments are often useful for constructors

```
struct Function : Shape
{
    Function(Fct f, double r1, double r2, Point xy, int count = 100,
        double xscale = 25, double yscale = 25);
};

Function f1(sqrt, -10, 11, Point(0, 0), 100, 25, 25);    // ok (obviously)
Function f2(sqrt, -10, 11, Point(0, 0), 100, 25);        // ok: exactly the same as f1
Function f3(sqrt, -10, 11, Point(0, 0), 100);            // ok: exactly the same as f1
Function f4(sqrt, -10, 11, Point(0, 0));                 // ok: exactly the same as f1
```

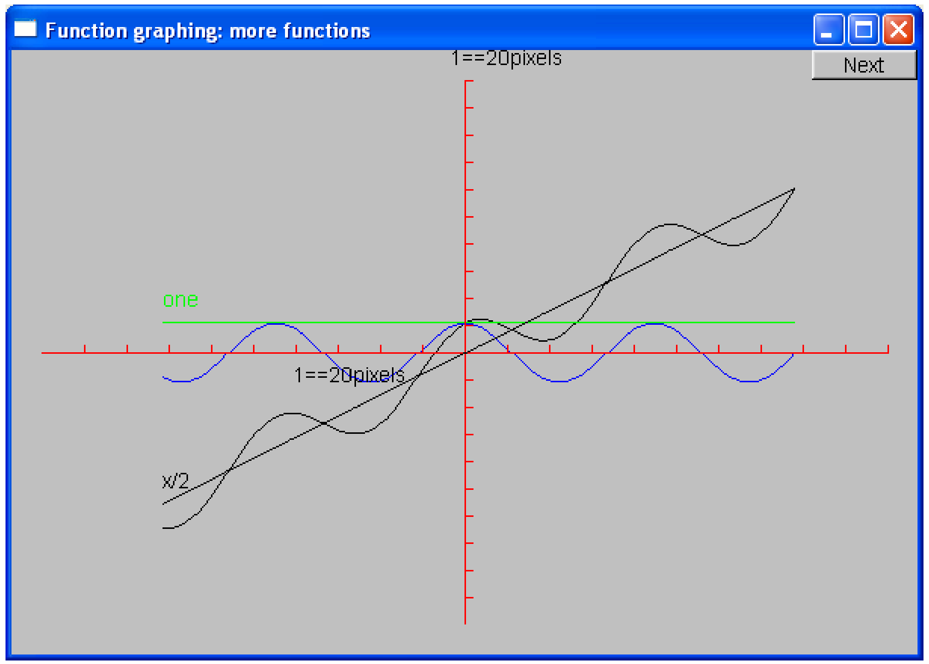
Function

- Is Function a “pretty class”?
  - No
    - Why not?
  - What could you do with all of those position and scaling arguments?
  - If you can’t do something genuinely clever, do something simple, so that the user can do anything needed
    - Such as adding parameters so that the caller can have control

Some more Functions

```
#include <cmath>    // standard mathematical functions

// You can combine functions (e.g., by addition):
double sloping_cos(double x)
{
    return std::cos(x) + slope(x);
}

Graph_lib::Function s4(std::cos, -10, 11, orig, 400, 20, 20);
s4.set_color(Graph_lib::Color::blue);

Graph_lib::Function s5(sloping_cos, -10, 11, orig, 400, 20, 20);
```

Cos and sloping-cos



Standard mathematical Functions (header <cmath>)

```
double abs(double);     // absolute value

double ceil(double d);  // smallest integer >= d
double floor(double d); // largest integer <= d

double sqrt(double d);  // d must be non-negative

double cos(double);
double sin(double);
double tan(double);
double acos(double);    // result is non-negative; “a” for “arc”
double asin(double);    // result nearest to 0 returned
double atan(double);
double sinh(double);    // “h” for “hyperbolic”
double cosh(double);
double tanh(double);
```

Standard mathematical functions (header <cmath>)

```
double exp(double);         // base e
double log(double d);       // natural logarithm (base e)
                            // d must be positive
double log10(double);       // base 10 logarithm

double pow(double x, double y);     // x to the power of y
double pow(double x, int y);        // x to the power of y
double atan2(double x, double y);   // atan(x/y)
// floating-point remainder, same sign as d % m
double fmod(double d, double m);
double ldexp(double d, int i);      // d*pow(2,i)
```

Code for Axis

```
struct Axis : Shape {
    enum Orientation {  x, y, z  };

    Axis(Orientation d, Point xy, int length,
        int number_of_notches = 0,    // default: no notches
        string label = "");           // default: no label

    void draw_lines() const;
    void move(int dx, int dy);

    void set_color(Color);    // in case we want to change the color
                              // of all parts at once

    // line stored in (base) Shape
    // orientation not stored (can be deduced from line)
    Text label;
    Lines notches;
};
```

Axis Implementation

```
Axis::Axis(Orientation d, Point xy, int length, int n, string lab) : label(Point(0, 0), lab)
{
    if (length < 0) throw std::runtime_error("bad axis length");
    switch (d) {
    case Axis::x:
    {
        Shape::add(xy);                                  // axis line begin
        Shape::add(Point(xy.x + length, xy.y));    // axis line end
        if (n > 1) {
            int dist = length / n;
            int x = xy.x + dist;
            for (int i = 0; i < n; ++i) {
                notches.add(Point(x, xy.y), Point(x, xy.y - 5));
                x += dist;
            }
        }
        label.move(length / 3, xy.y + 20);    // put label under the line
        break;
    }
```

Axis Implementation

```
void Axis::draw_lines() const
{
    Shape::draw_lines();     // the line
    notches.draw();          // the notches may have a difference color from the line
    label.draw();            // the label may have a different color from the line
}

void Axis::move(int dx, int dy)
{
    Shape::move(dx, dy);    // the line
    notches.move(dx, dy);
    label.move(dx, dy);
}

void Axis::set_color(Color c)
{
    // ... the obvious three lines
}
```

Why Graphing?

- Because you can see things in a graph that are not obvious from a set of numbers
  - How would you understand a sine curve if you couldn’t (ever) see one?
- Visualization is
  - Key to understanding in many fields
  - Used in most research and business areas
    - Science, medicine, business, telecommunications, control of large systems

An example: e^x

$$e^x == 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!} + \frac{x^7}{7!} + \ldots$$

- Where '!' Means factorial (e.g. 4! == 4*3*2*1)

Simple algorithm to approximate e^x

```
double fac(int n) {                    // factorial
    if (n == 1) return 1;
    return fac(n - 1) * n;
}

double term(double x, int n) {    // x^n/n!
    return std::pow(x, n) / fac(n);
}

double expe(double x, int n) {    // sum of n terms of x
    double sum = 0;
    for (int i = 0; i < n; ++i)
        sum += term(x, i);
    return sum;
}
```

Simple algorithm to approximate e^x

- But we can only graph functions of one arguments, so how can we get graph `expr(x, n)` for various `n`?

```
auto expN = [n](double x) { return expe(x, n); };
```

- Equivalent to:

```
int expN_number_of_terms = 6;   // nasty sneaky global argument to expN

double expN(double x)
{
    return expe(x, expN_number_of_terms);
}
```

“Animate” approximations to e^x

```
Simple_window win(Point(100, 100), xmax, ymax,
    "Function graphing");    // make a window

Axis x(Axis::x, Point(20, y_orig), xlength,
    xlength / x_scale, "1 == 20 pixels");
x.set_color(Color::red);

Axis y(Axis::y, Point(x_orig, ymax - 20),
    ylength, ylength / y_scale, "1 == 20 pixels");
y.set_color(Color::red);

Function real_exp(std::exp, -15, 15, orig, 400, 20, 20);
real_exp.set_color(Color::blue);
```

"Animate" Approximations to e^x

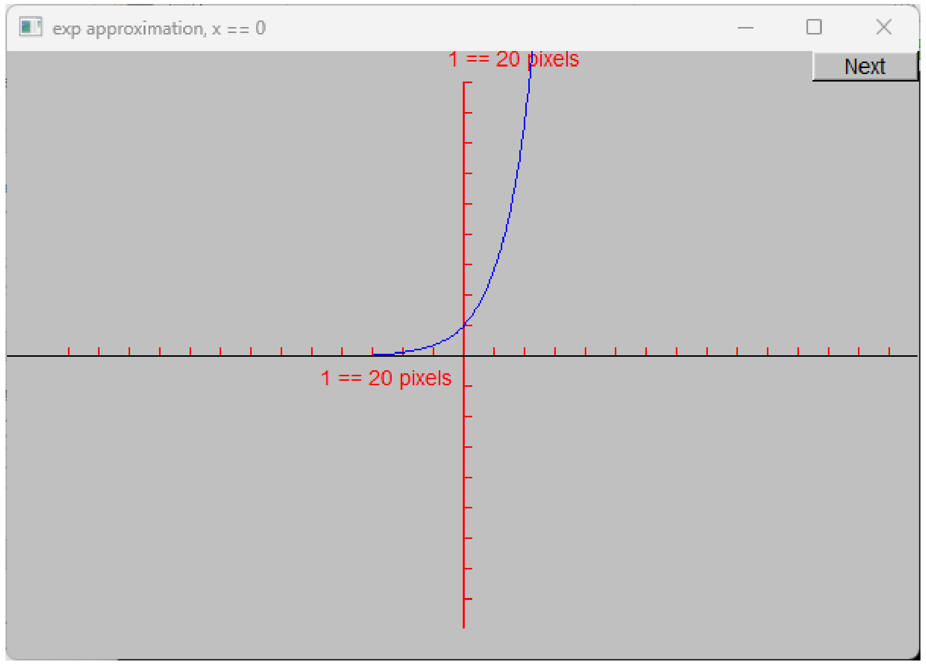
```
for (int n = 0; n < 40; ++n)
{
    auto expN = [n](double x) { return expe(x, n); };

    Graph_lib::Function f(expN, -15, 15, orig, 400, 20, 20);
    f.set_color(Graph_lib::Color::black);

    win.set_label("exp approximation, x == " + std::to_string(n));
    win.attach(f);

    win.wait_for_button();    // Draw!

    win.detach(f);
}
```

Demo

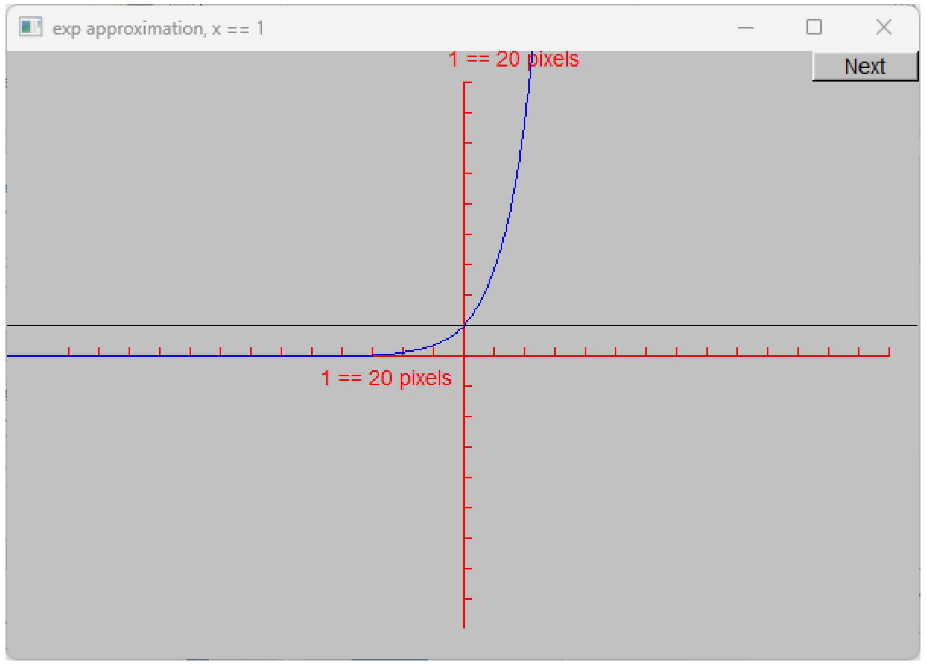
- The following screenshots are of the successive approximations of exp(x) using

```
auto expN = [n](double x) { return expe(x, n); };
```

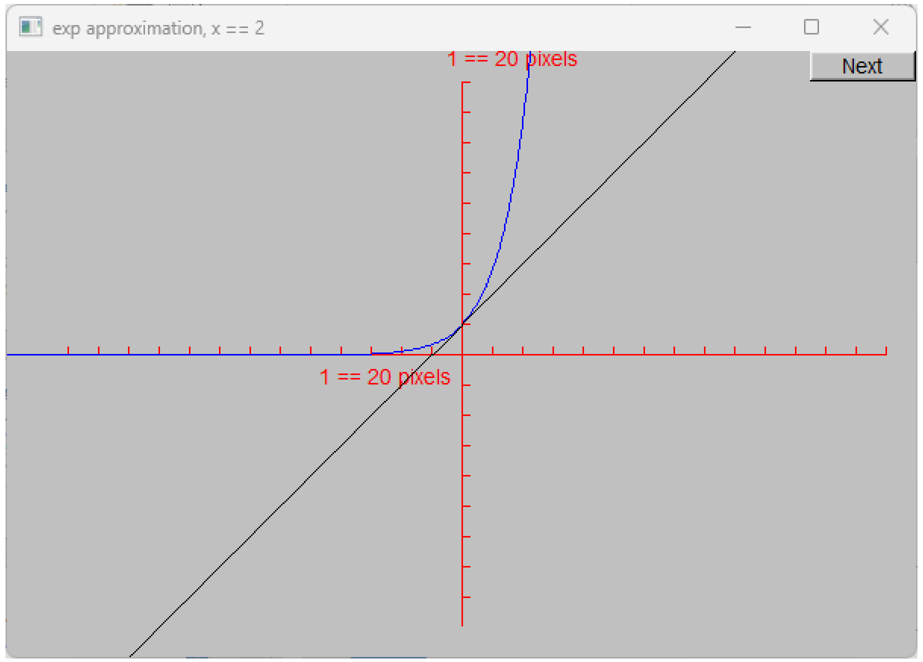
Demo n = 0



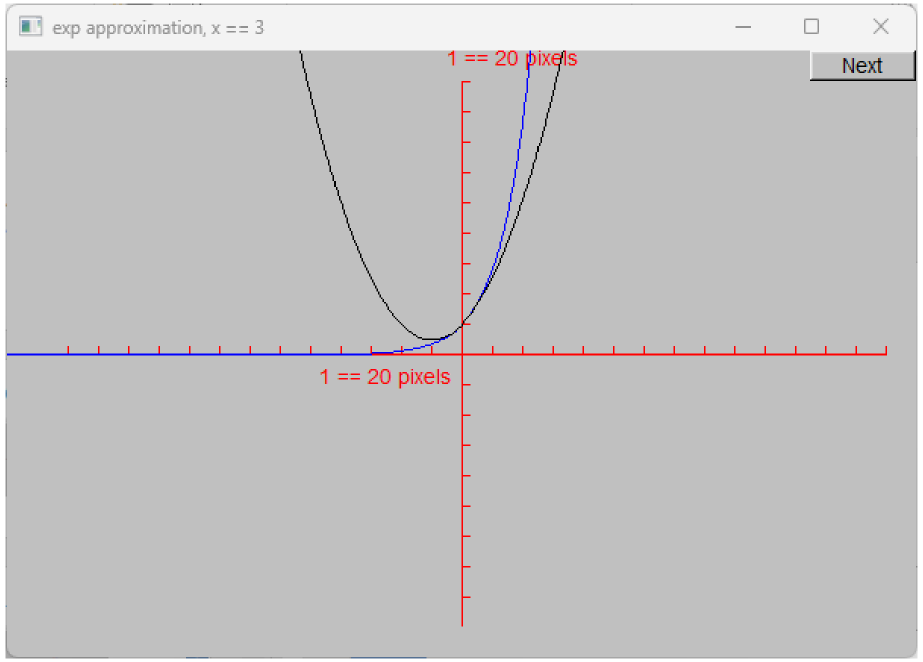
Demo n = 1



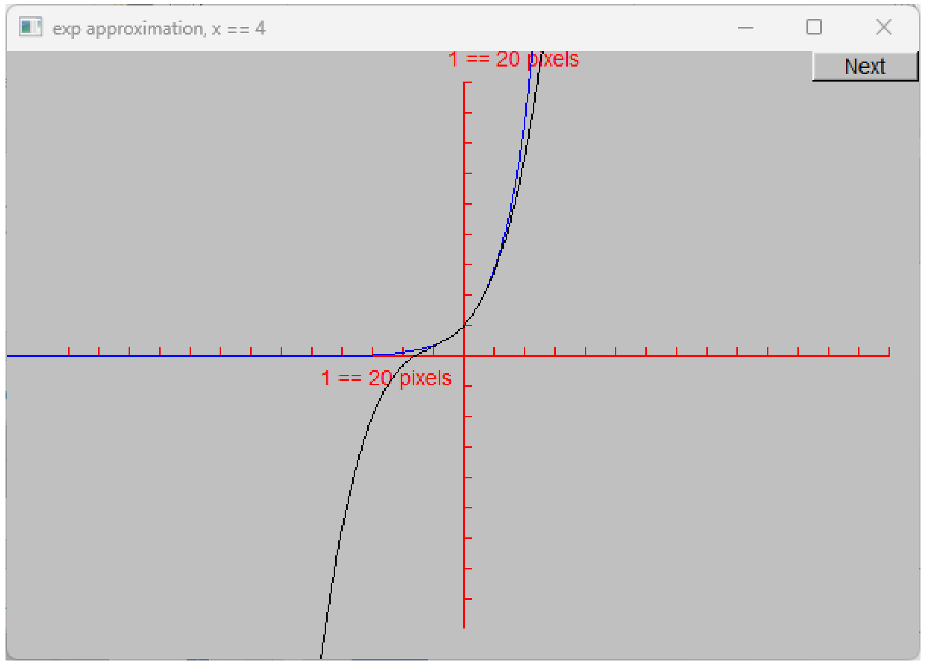
Demo n = 2



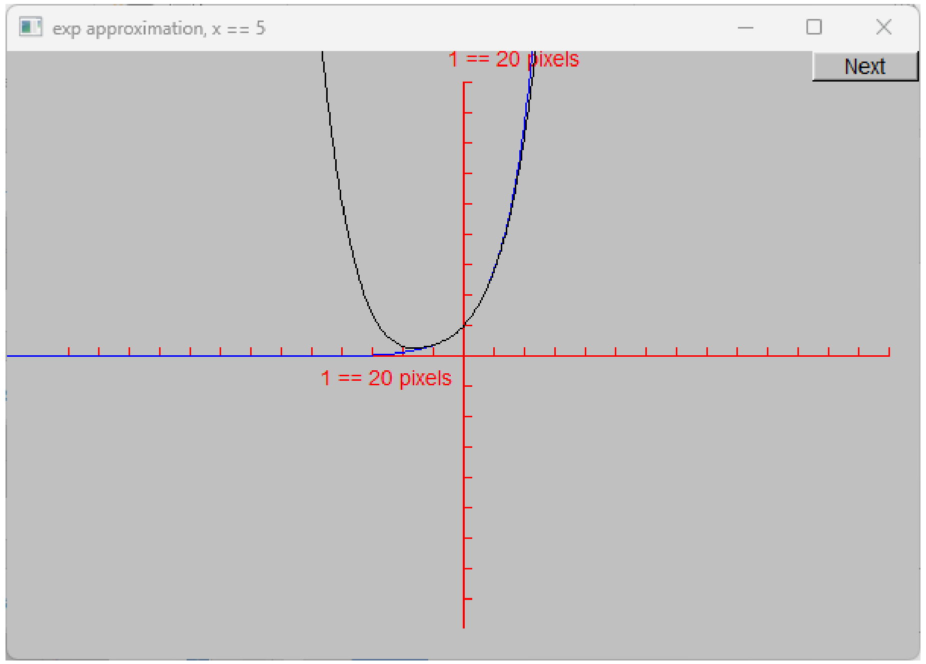
Demo n = 3



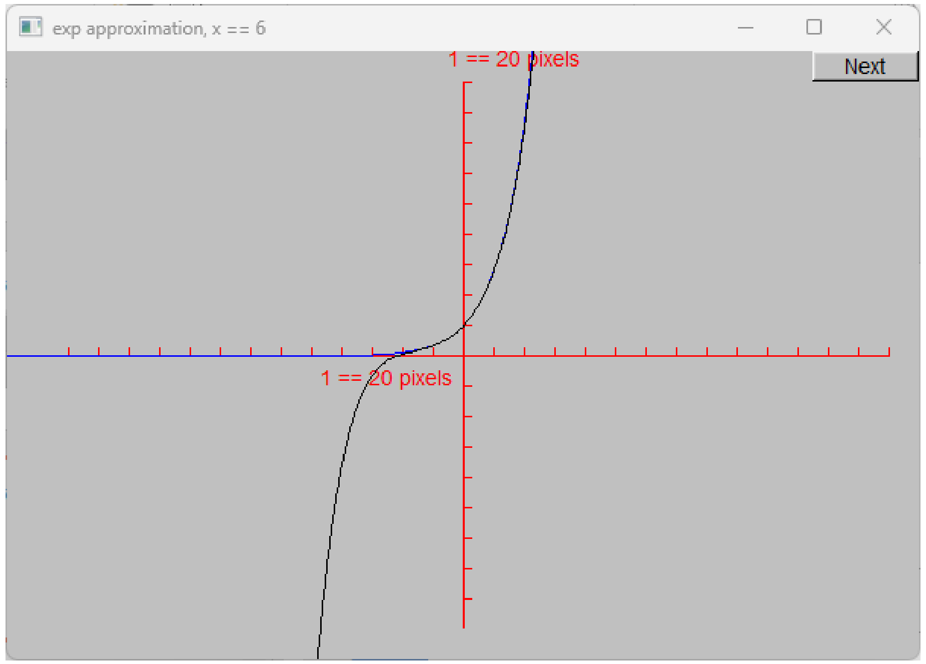
Demo n = 4



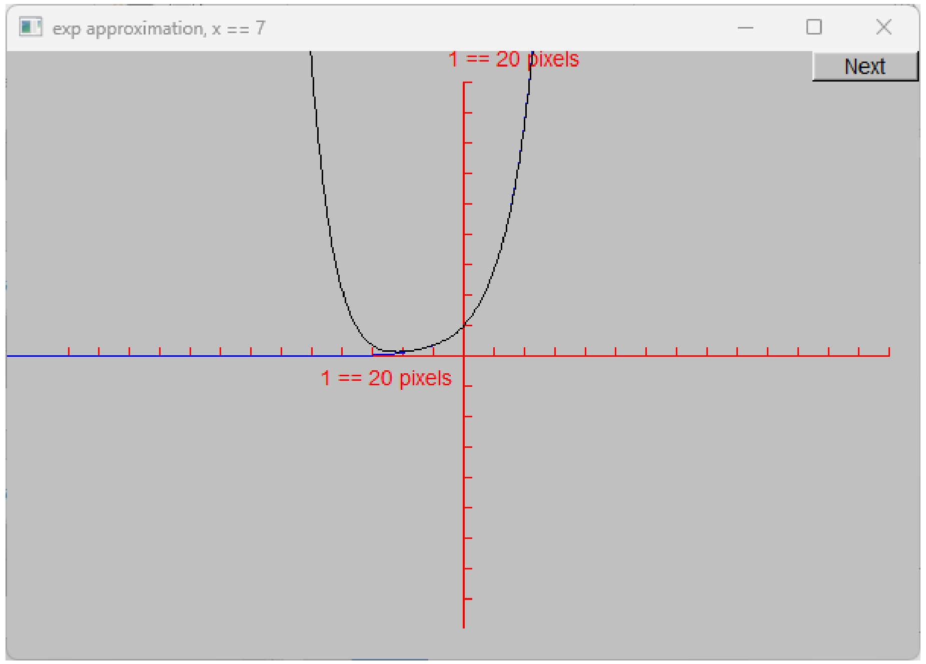
Demo n = 5



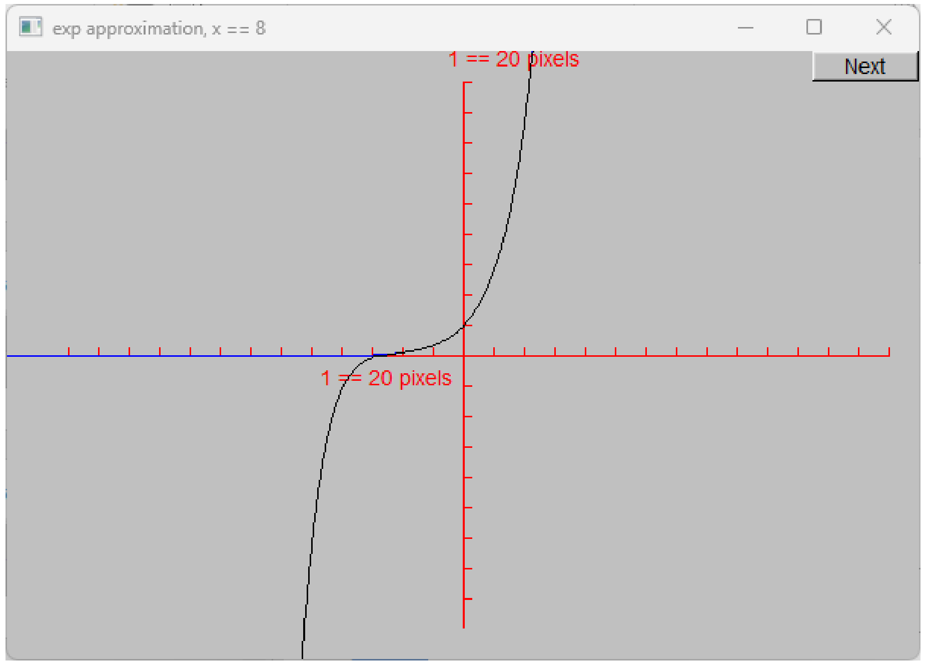
Demo n = 6



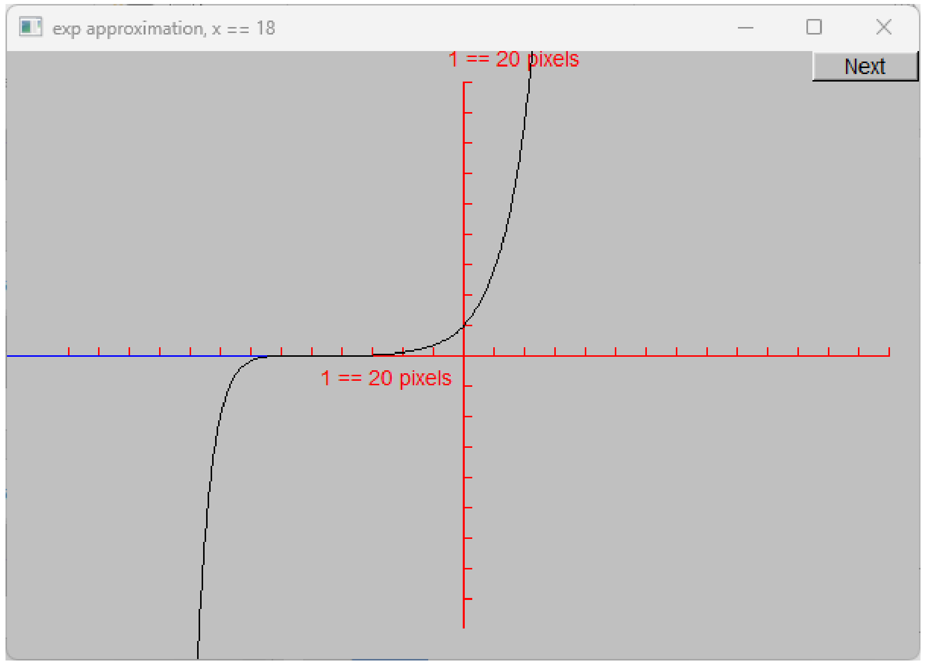
Demo n = 7



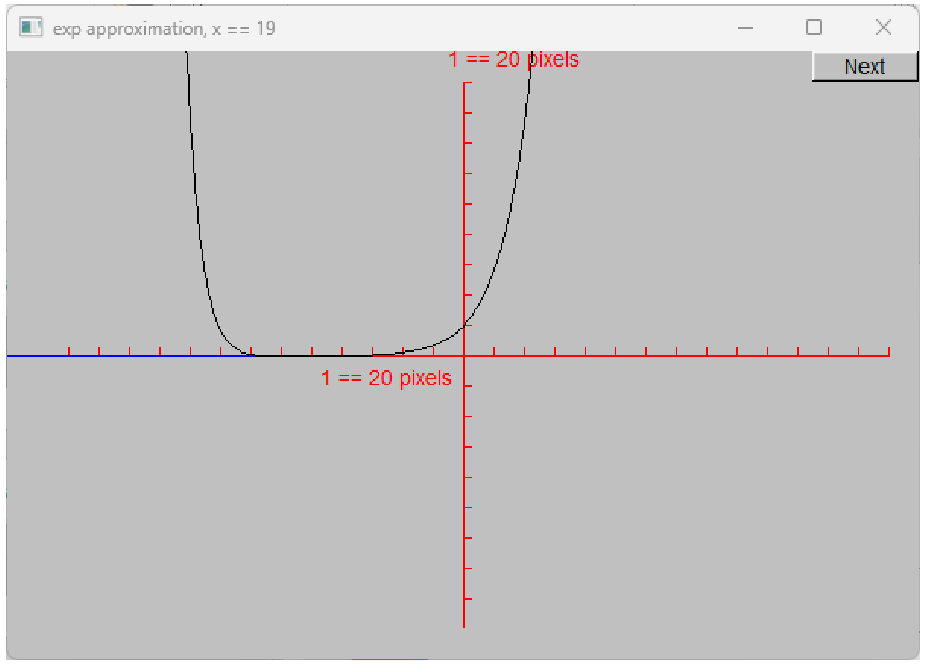
Demo n = 8



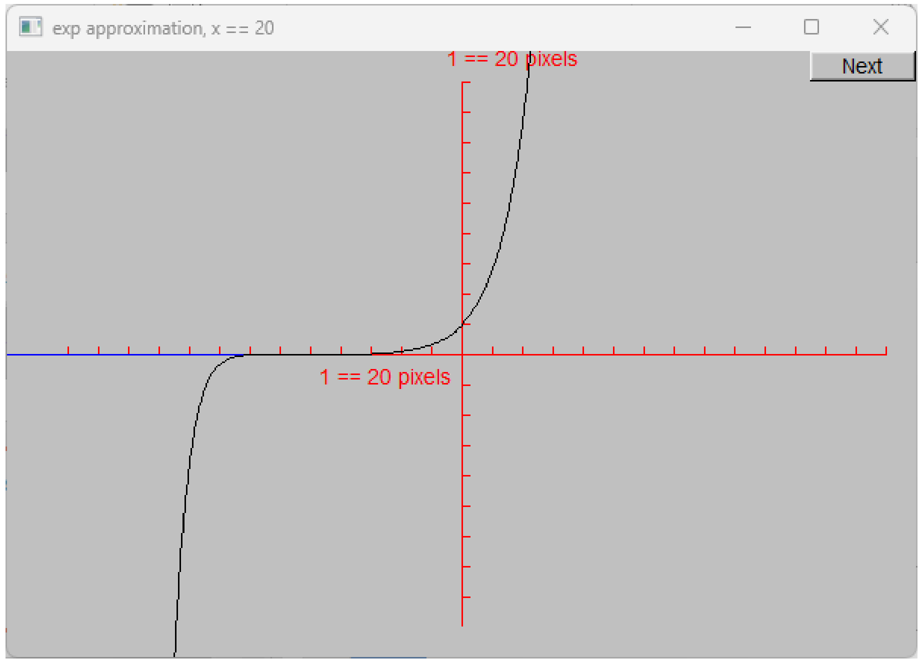
Demo n = 18



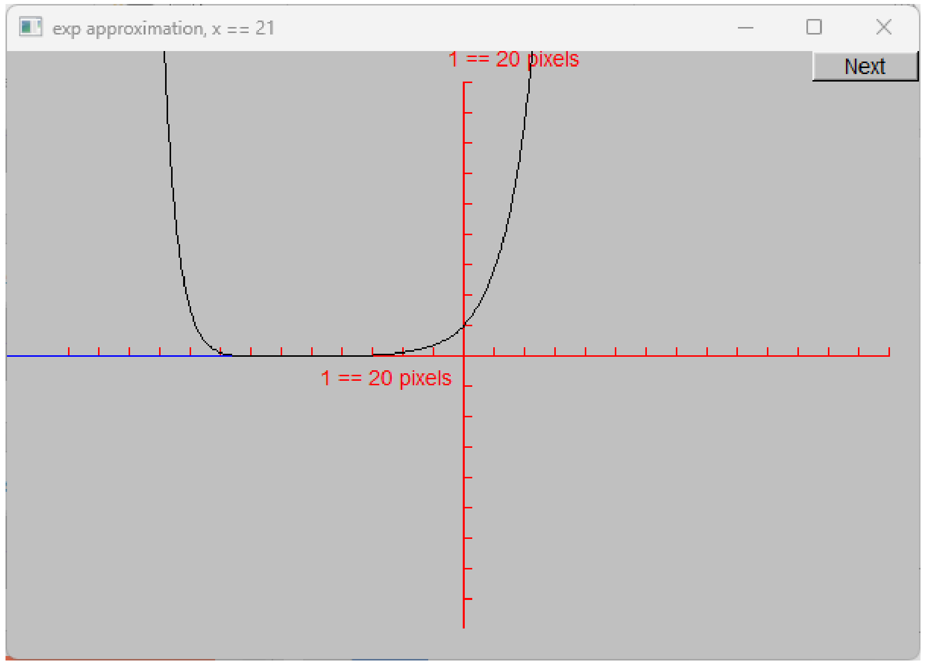
Demo n = 19



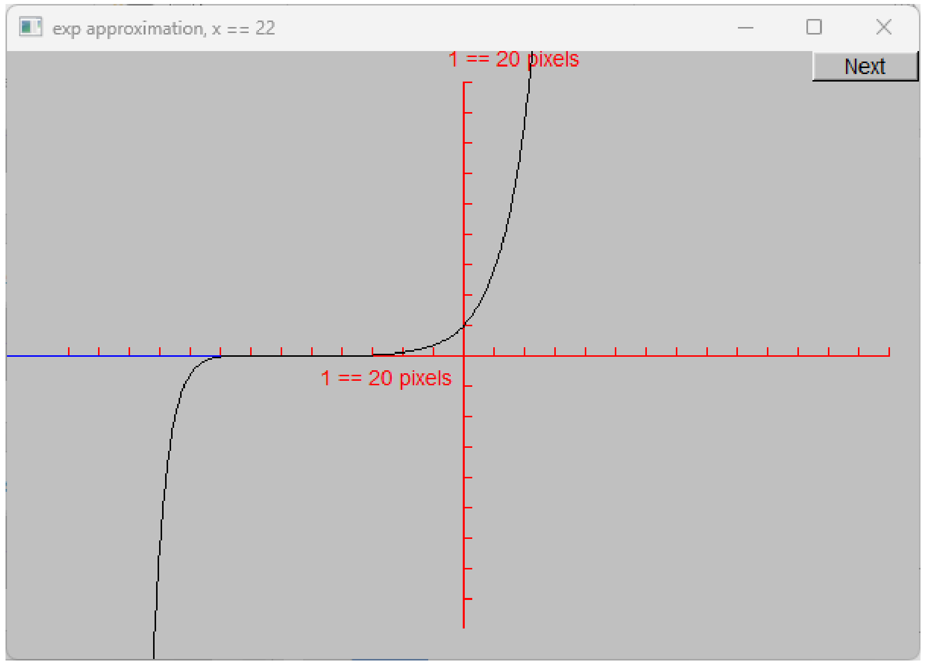
Demo n = 20



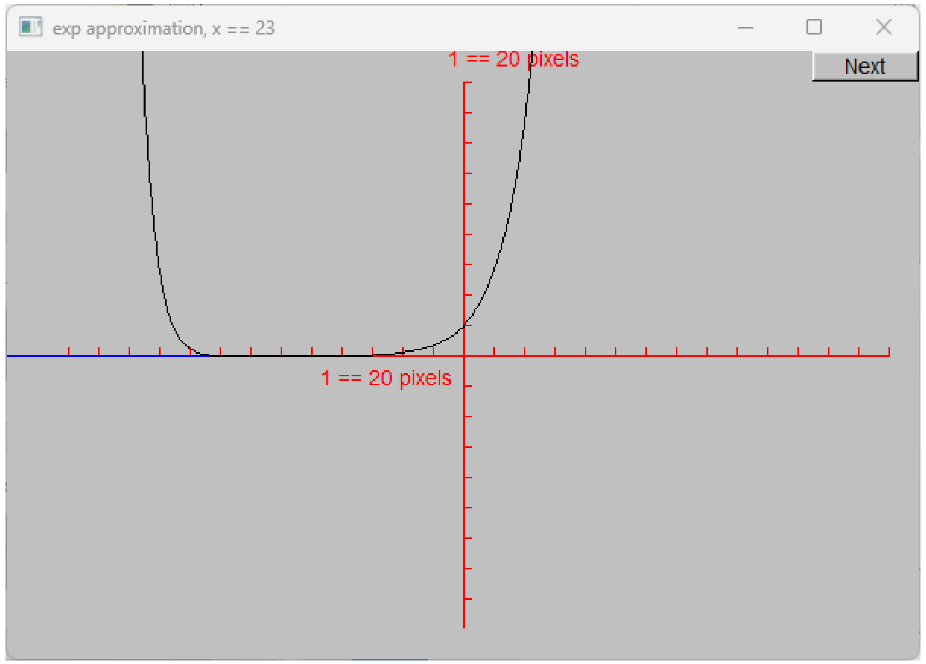
Demo n = 21



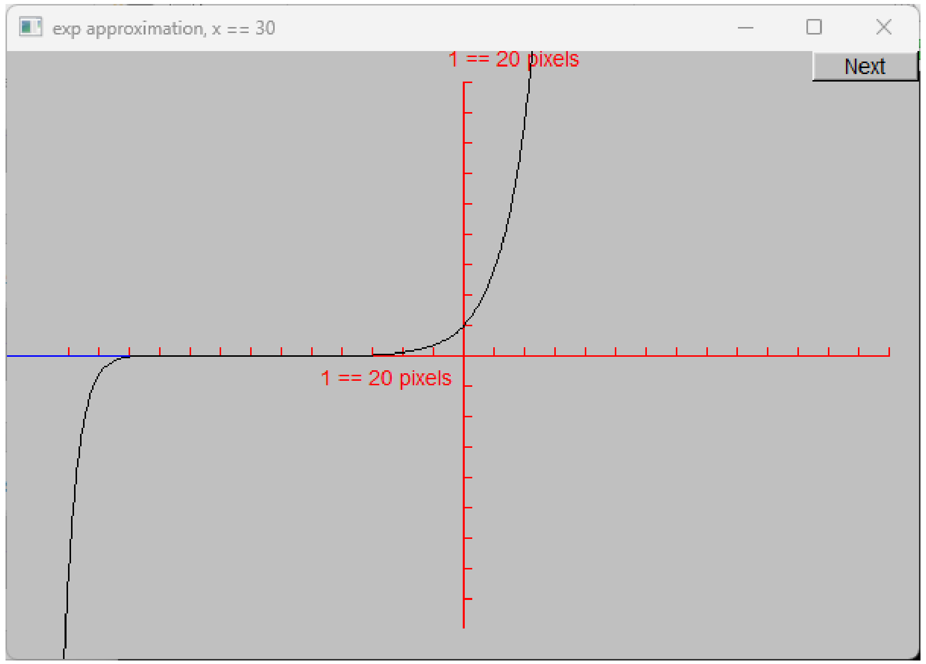
Demo n = 22



Demo n = 23

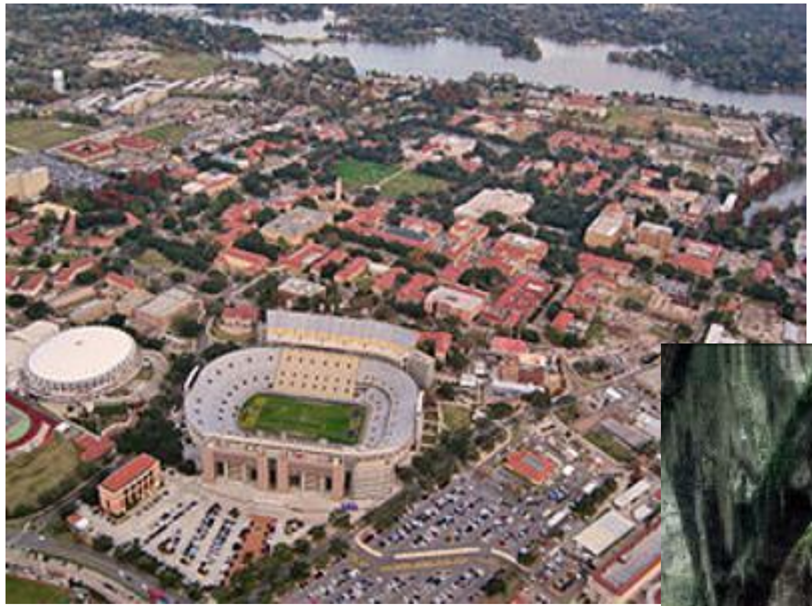


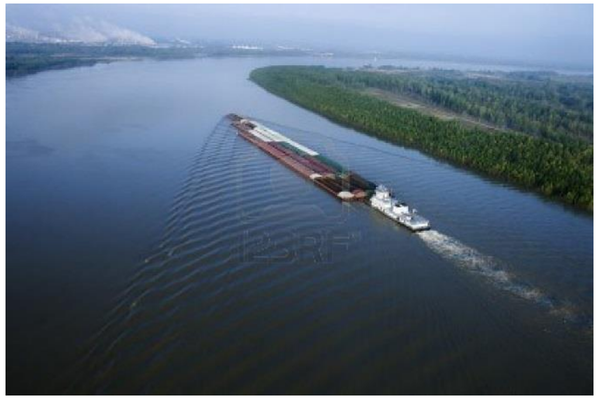
Demo n = 30



Next Lecture

- Graphical user interfaces
- Windows and Widgets
- Buttons and dialog boxes


LSU CENTER FOR COMPUTATION & TECHNOLOGY



STE||AR GROUP