Fixed Point Calculations and Finding Roots

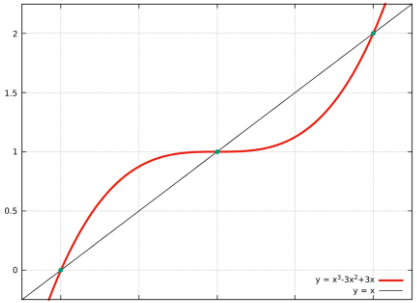
Lecture 4

Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4700/

Fixed Point of a Function

- Formally, c is a fixed point of a function f if
  - c belongs to both the domain and the codomain of f
  - $f(c) = c$
- If the fixed point iteration
  $$x_{n+1} = f(x_n), n = 0, 1, 2 \ldots$$
- Converges to a value x_{fix}, i.e. $f(x_{fix}) = x_{fix}$
- Examples:
  - $y = \cos(x)$
  - $y = x^3 - 3x^2 + 3x$
  - Many contiguous functions that intersect $y = x$



Fixed Point of a Function

- Used in
  - Economics: study the existence and stability of equilibria in game theory and economic models
  - Physics: study the behavior of dynamical systems and phase transitions
  - Computer science: design algorithms for solving equations and optimization problems
  - Engineering: design control systems and analyze the stability of structures
  - Numerical root finding
  - The vector of PageRank values of all web pages is the fixed point of a linear transformation derived from the World Wide Web's link structure
  - Etc.
- Allows us to introduce generic functions in C++

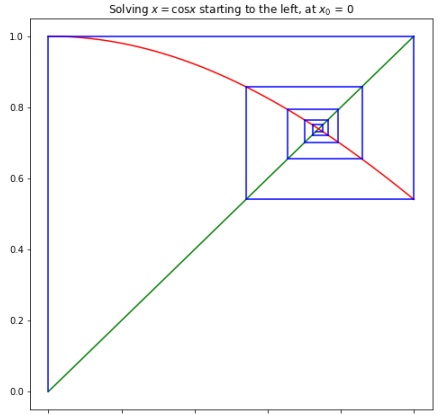
Fixed Point of a Functoin (cos(x))

- Simple fixed point iteration using double:

```
double fixed_point(int max_iterations, double init)
{
    double x = init;
    for (int i = 0; i < max_iterations; ++i)
        x = std::cos(x);
    return x;
}

int main()
{
    for (int n = 1; n != 85; ++n)
        std::println("Iterations: {}, fixed point: {}", n,
                     fixed_point(n, 0.75));
    return 0;
}
```

Fixed Point of a Function

- Geometrical interpretation



Fixed Point of a Function (cos(x))

```
Iterations: 1, fixed point: 0.75
Iterations: 2, fixed point: 0.7316888688738209
Iterations: 3, fixed point: 0.7440470847887636
Iterations: 4, fixed point: 0.7357336181872361
Iterations: 5, fixed point: 0.741338598887922
Iterations: 6, fixed point: 0.7375652963392664
Iterations: 7, fixed point: 0.7401080596152839
Iterations: 8, fixed point: 0.7383956911259041
Iterations: 9, fixed point: 0.7395493740083102
...
Iterations: 80, fixed point: 0.7390851332151603
Iterations: 81, fixed point: 0.7390851332151608
Iterations: 82, fixed point: 0.7390851332151606
Iterations: 83, fixed point: 0.7390851332151607
Iterations: 84, fixed point: 0.7390851332151607
Iterations: 85, fixed point: 0.7390851332151607
```

Fixed Point of a Function

- Simple fixed point iteration using double for an arbitrary function:

```
double fixed_point(double f(double), int max_iterations, double init)
{
    double x = init;
    for (int i = 0; i < max_iterations; ++i)
        x = f(x);
    return x;
}

int main()
{
    for (int n = 1; n != 85; ++n)
        std::println("Iterations: {}, fixed point: {}", n,
                fixed_point(/* what should go here? */, n, 0.75));
    return 0;
}
```

Any function `f` with one `double` argument returning a `double`

Using a Named Function

- Define a function of one `double` argument returning a `double`:

```
double calculate_cos(double x) {
    return std::cos(x);
}

int main()
{
    for (int n = 1; n != 85; ++n)
        std::println("Iterations: {}, fixed point: {}", n,
                     fixed_point(
                         calculate_cos,
                         n, 0.75));
    return 0;
}
```

Name this famous Person

- Alonzo Church (June 14, 1903 – August 11, 1995) was an American mathematician and logician who made major contributions to mathematical logic and the foundations of theoretical computer science.
- He is best known for the lambda calculus, Church–Turing thesis, proving the undecidability of the Entscheidungs-problem, Frege–Church ontology, and the Church–Rosser theorem.
- He developed various formalisms for computing

Using an Anonymous Function

- Simple fixed point iteration using double:

```
double fixed_point(double f(double), int max_iterations, double init) {
    double x = init;
    for (int i = 0; i < max_iterations; ++i)
        x = f(x);
    return x;
}

int main() {
    for (int n = 1; n != 85; ++n)
        std::println("Iterations: {}, fixed point: {}", n,
                     fixed_point(
                         [](double x) -> double { return std::cos(x); },
                         n, 0.75));
    return 0;
}
```

Using an Anonymous Function

- Before:

```
double calculate_cos(double x) {
    return std::cos(x);
}
```

- After:

```
auto calculate_cos(double x) -> double {
    return std::cos(x);
}
```

Using an Anonymous Function

- Before:

```
auto calculate_cos(double x) -> double {
    return std::cos(x);
}
```

- After:

```
auto calculate_cos = [](double x) -> double {
    return std::cos(x);
}
```

- A 'lambda' function is a function that has no name
  - Otherwise the same as named functions: arguments, return type, body, etc.
  - Return type is optional if it can be deduced from the code
  - Additionally capture clause: `[...]`

Using an Anonymous Function

- Simple fixed point iteration using double:

```
double fixed_point(double f(double), int max_iterations, double init) {
    double x = init;
    for (int i = 0; i < max_iterations; ++i)
        x = f(x);
    return x;
}

int main() {
    for (int n = 1; n != 85; ++n)
        std::println("Iterations: {}, fixed point: {}", n,
                     fixed_point(
                         [](double x) { return std::cos(x); },
                         n, 0.75));
    return 0;
}
```

Finding Roots with Fixed Point Iterations

- Given an equation of one variable $y = f(x)$
  - In order to find its root x ($f(x) = 0$), we use fixed point iterations as follows:
    - Convert the equation to the form $x = g(x)$
    - Start with an initial guess $x0 \approx r$, where r is the actual solution (root) of the equation
    - Iterate, using $x_{n+1} = g(x_n)$ for $n = 0, 1, 2, \ldots$
- If the sequence $(x_n)_0^\infty$ that is defined by $x_{n+1} = g(x_n)$
  - Converges to a limit r
  - And the function g is continuous at $x = r$,
  - Then the limit r is a root of $f(x)$: $f(r) = 0$.

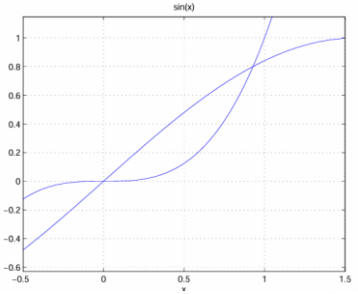
Finding Roots with Fixed Point Iterations

- The real trick for using fixed point iterations is
  - Finding a transformation of the original equation $f(x) = 0$ to $x = g(x)$ such that $(x_n)_0^\infty$ converges

- Example: $x^3 = \sin x$, some possibilities:

$$x = \frac{\sin x}{x^2}, \quad x = \sqrt[3]{\sin x}, \quad x = \sin^{-1} x^3$$

$$x = \frac{\sin x - 1}{x^2 + x + 1} + 1, \quad x = x - \frac{x^3 - \sin x}{3x^2 - \cos x}$$



Finding Roots with Fixed Point Iterations

	$g(x) = \sqrt[3]{\sin x}$	$g(x) = \frac{\sin x}{x^2}$	$g(x) = x + \sin x - x^3$	$g(x) = x - \frac{\sin x - x^3}{\cos x - 3x^2}$
x_1:	0.94408924124306	0.84147098480790	0.84147098480790	0.93554939065467
μ_1:	-0.05591075875694	-0.15852901519210	-0.15852901519210	-0.06445060934533
x_2:	0.93215560685805	1.05303224555943	0.99127188988250	0.92989141894368
μ_2:	0.21344075183985	-1.33452706115132	-0.94494313796800	0.08778771478600
x_3:	0.92944074461587	0.78361086350974	0.85395152069647	0.92886679103170
μ_3:	0.22749668328899	-1.27349109705917	-0.91668584457246	0.18109456255982
x_4:	0.92881472066057	1.14949345383611	0.98510419085185	0.92867234089417
μ_4:	0.23059142581182	-1.35803100534498	-0.95508533025939	0.18977634246913
⋮	⋮	⋮	⋮	
⋮	⋮	⋮	⋮	
x_{26}:	0.92862630873173	-0.00000000000000	0.89462921748990	0.92862630873173
μ_{26}:	0	-1.00000000000000	-0.97525571602895	NaN
x_{27}:	0.92862630873173	0	0.89614104323697	0.92862630873173
μ_{27}:	NaN	-1.00000000000000	-0.97635939022401	NaN

$$\mu_{n+1} = \frac{e_{n+1}}{e_n} \approx \frac{x_{n+1} - x_n}{x_n - x_{n-1}}$$

https://pages.cs.wisc.edu/~amos/412/lecture-note

Finding Roots with Fixed Point Iterations

```
double fixed_point(double f(double), int max_iterations, double init,
    double epsilon = 1e-10)
{
    double x = init;
    for (int i = 0; i < max_iterations; ++i) {
        double x1 = f(x);

        double error = fabs(x1 - x);
        if (error < epsilon)
            break;

        x = x1;
    }
    return x;
}
```

Default value for parameter epsilon – used when no argument supplied

Finding Roots with Fixed Point Iterations (double)

```
int main()
{
    using float_type = double;
    for (int n = 1; n != 17; ++n) {
        std::cout << n << ": fixed point: "
            << std::setprecision(std::numeric_limits<float_type>::max_digits10)
            << fixed_point([](float_type x) { return cbrt(sin(x)); },
                           n, float_type(1.0))
            << "\n";
    }
    return 0;
}
```

Finding Roots with Fixed Point Iterations (double)

```
 1: fixed point: 1
 2: fixed point: 0.94408924124306481
 3: fixed point: 0.93215560685804832
 4: fixed point: 0.92944074461587356
 5: fixed point: 0.92881472066056769
 6: fixed point: 0.92866992107978663
 7: fixed point: 0.92863640517161028
 8: fixed point: 0.92862864617144114
 9: fixed point: 0.92862684987924371
10: fixed point: 0.92862643401458511
11: fixed point: 0.92862633773639280
12: fixed point: 0.92862631544670204
13: fixed point: 0.92862631028633946
14: fixed point: 0.92862630909164634
15: fixed point: 0.92862630881505892
16: fixed point: 0.92862630875102514
17: fixed point: 0.92862630875102514
```

Finding Roots with Fixed Point Iterations (double)

```
 1: fixed point: 1
 2: fixed point: 0.94408924124306481
 3: fixed point: 0.93215560685804832
 4: fixed point: 0.92944074461587356
 5: fixed point: 0.92881472066056769
 6: fixed point: 0.928669992107978663
 7: fixed point: 0.92863640517161028
 8: fixed point: 0.92862864617144114
 9: fixed point: 0.92862684987924371
10: fixed point: 0.92862643401458511
11: fixed point: 0.928626333773639280
12: fixed point: 0.928626331544670204
13: fixed point: 0.928626331028633946
14: fixed point: 0.928626300909164634
15: fixed point: 0.928626308081505892
16: fixed point: 0.92862630875102514
17: fixed point: 0.92862630875102514
```

Finding Roots with Fixed Point Iterations (higher precision)

```
int main()
{
    using float_type = boost::multiprecision::cpp_bin_float_100;
    for (int n = 1; n != 17; ++n) {
        std::cout << n << ": fixed point: "
            << std::setprecision(std::numeric_limits<float_type>::max_digits10)
            << fixed_point([](float_type x) { return cbrt(sin(x)); },
                           n, float_type(1.0))
            << "\n";
    }
    return 0;
}
```

Finding Roots with Fixed Point Iterations (`boost::multiprecision::cpp_bin_float_100`)

```
 1: fixed point: 1
 2: fixed point: 0.9440892412430647727964899057914217079861772716649595536980785228715700644119345791187459362507878767 20
 3: fixed point: 0.9321556068580482494746849882634171572241699367323832152751756474431378754175361570599099896581878879 88
 4: fixed point: 0.9294407446158734846321239941392479716909962596333360720840794240121530254237239803822107788771158994 99
 5: fixed point: 0.9288147206605677817832155203052424856358124725104611199817079038256854013514075261186957546171283295 40
 6: fixed point: 0.9286699210797867293878590748854121426000610327637333612610539999971117530685777668513213034288216448 09
 7: fixed point: 0.9286364051716103722190592656384248230099660361146427240720378376748339452908522245281561140900240503 66
 8: fixed point: 0.928628646171441161484191296888214195056800070647013455516952440258806083550616805431261210169624428491
 9: fixed point: 0.9286268498792437162632585407562129258765934918293548955054266206835251635083391783729115189160786589 64
10: fixed point: 0.9286264340145850464891308329674558922213410784315368476199277825034993384605359039011135711795865026 21
11: fixed point: 0.9286263377363928252271436736736733767367597824647447567676302414734848832932411926975167102189325052 86
12: fixed point: 0.9286263154467020295601859157048312732943205632185821374292988101603539721194021417423670858844698269 87
13: fixed point: 0.9286263102863395593959911770067759934148696472066746686984379372422683278207872047557157756119931745 86
14: fixed point: 0.9286263090916463831422679732388186338023244896719776944381384351105735273506847864157049314681459743 51
15: fixed point: 0.928626308815058875585467521446095397987205995632600898761988954484349955114187085492403368952534712871
16: fixed point: 0.92862630875102515466000268936244304992809610001939213128334587882871087891532780563030761129646640302 2
17: fixed point: 0.92862630875102515466000268936244304992809610001939213128334587882871087891532780563030761129646640302 2
```

Finding Roots with Fixed Point Iterations

```
double fixed_point(double f(double), int max_iterations, double init,
    double epsilon = 1e-10)
{
    double x = init;
    for (int i = 0; i < max_iterations; ++i) {
        double x1 = f(x);

        double error = fabs(x1 - x);
        if (error < epsilon)
            break;
        x = x1;
    }
    return x;
}
```

Generic Fixed Point of a Function

```
template <typename Float>
Float fixed_point(Float f(Float), int max_iterations, Float init,
    Float epsilon = 1e-10)
{
    Float x = init;
    for (int i = 0; i < max_iterations; ++i) {
        Float x1 = f(x);              // get next iteration result

        // compare with previous iteration result
        if (fabs(x1 - x) < epsilon) break;

        x = x1;                       // update for next iteration
    }
    return x;
}
```

Generic Fixed Point of a Function

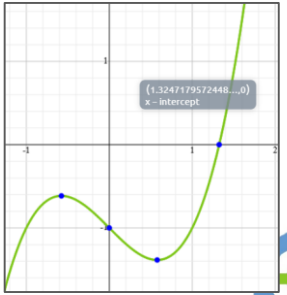
```
template <typename Data>
std::pair<Data, int> fixed_point(
    Data f(Data), bool cond(Data, Data), int n, Data init)
{
    Data x = init;
    for (int i = 0; i != n; ++i) {
        Data x1 = f(x);      // get next iteration result
        if (cond(x, x1))     // compare with previous iteration result
            return {x1, i};
        x = x1;              // update for next iteration
    }
    return {x, n};
}
```

Finding Roots with Fixed Point Iterations (double)

```
int main() {
    using float_type = double;
    for (int n = 1; n != 17; ++n) {
        std::cout << n << ": fixed point: "
            << std::setprecision(std::numeric_limits<float_type>::max_digits10)
            << fixed_point(
                   [](float_type x) { return cbrt(sin(x)); },
                   [](float_type prev, float_type curr) {
                       return fabs(curr - prev) < 1e-10;
                   },
                   n, float_type(1.0))
            << "\n";
    }
    return 0;
}
```

Exercise

- Approximate a solution to $x^3 - x - 1 = 0$ on $[1, 2]$ using fixed point iteration
  - Convert the given function to the form $g(x) = x$
  - How many iterations do you need to perform to achieve an error less than 10^{-5} ?
    - Assume for the error to be: $e_{n+1} = |x_{n+1} - x_n|$ (the difference between the last to iteration results)
  - Estimate the solution
  - Demonstrate that it is indeed the solution by creating a plot that shows $y = x$, your $g(x) = x$, and f(x) = $x^3 - x - 1$ (as done on Slide 5)



Finding Square Root

Calculating Square Root

- Ancient method for calculating the square root of a:

$$x_{n+1} = average\left(x_n, \frac{a}{x_n}\right) = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

- This corresponds to calculating $\sqrt{a}$
  - Makes sense, as once $x_n = \sqrt{a}$ the equation above becomes true
- Alternatively, this can be computed using the `fixed_point` function
  - By repeated application of the averaging

Calculating Square Root

```
template <typename Float>
Float average(Float x1, Float x2) {
    return (x1 + x2) / 2;
}

template <typename Float>
Float iterate_sqrt(Float a, int max_iterations, Float init) {
    Float x = init;
    for (int i = 0; i != max_iterations; ++i)
        x = average(x, a / x);
    return x;
}
```

Calculating Square Root

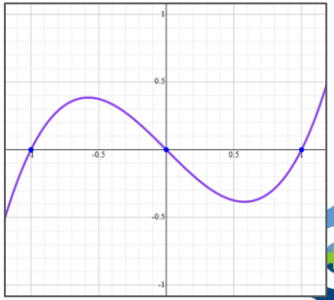
```
template <typename Float>
Float average(Float x1, Float x2) {
    return (x1 + x2) / 2;
}

template <typename Float>
Float iterate_sqrt(Float a, int n, Float init)
{
    return fixed_point([a](double x) { return average(x, a / x); }, n, init);
}
```

Capture a to make it available from inside the lambda's body

Finding Roots

Finding Roots of Functions

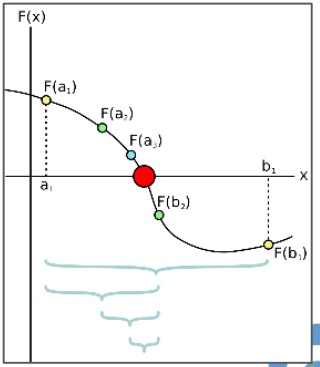
- Simplest algorithms to find roots of non-linear equations
  - Bisection method
  - Newton-Raphson method
  - Gradient Descent
- A root of a (non-linear) equation $f(x)$ is the value of x where the function intersects the x-axis ($f(x) = 0$)
- To find the root of a continuous, differentiable function $f(x)$, and assuming the root is near $x = x_0$



Bisection Method

- Based on Intermediate Value Theorem
  - If for a contiguous function $f(x)$, $\text{sign}(f(x_1)) \neq sign(f(x_2))$ for $x_1 < x_2$
  - Then there exists a x_0: $x_1 \leq x_0 \leq x_2$, such that $f(x_0) = 0$

- Let's assume root x of contiguous function $f(x)$ is inside the interval $[x_1, x_2)$
- Divide interval in two sections: $[x_1, x_0)$ and $[x_0, x_2)$
- If $\text{sign}(f(x_1)) \neq sign(f(x_0))$
  - Repeat dividing for interval $[x_1, x_0)$
  - Otherwise repeat dividing for interval $[x_0, x_2)$
- Continue until $f(x_0) < eps$



Bisection Method

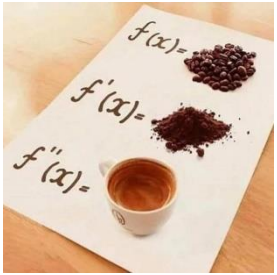
```
std::pair<double, size_t> bisection (double f(double), size_t max_iterations,
                                     double xn, double xm, double epsilon)
{
    for (size_t step = 0; step != max_iterations; ++step) {
        double fn = f(xn), fm = f(xm);
        if (std::fabs(fn) <= epsilon) return {xn, step};
        if (std::fabs(fm) <= epsilon) return {xm, step};

        double next_x = (xm + xn) / 2;
        if (std::signbit(fn) == std::signbit(f(next_x)))
            xn = next_x;
        else
            xm = next_x;
    }
    return {0.0, max_iterations};
}
```

Bisection Method

```
auto bisection(double f(double), size_t max_iterations, double xn_init,
    double xm_init, double epsilon)
{
    using value_type = std::pair<double, double>;
    return fixed_point(
        [f](value_type bounds) {                        // get next iteration result
            double fn = f(bounds.first);
            double next_x = (bounds.first + bounds.second) / 2;
            if (std::signbit(fn) == std::signbit(f(next_x)))
                bounds.first = next_x;
            else
                bounds.second = next_x;
            return bounds;
        },
        [f, epsilon](value_type, value_type curr) {     // check for termination condition
            return std::fabs(f(curr.first)) <= epsilon ||
                   std::fabs(f(curr.second)) <= epsilon;
        },
        max_iterations, value_type(xn_init, xm_init));
}
```

Newton's Method

Newton-Raphson Method

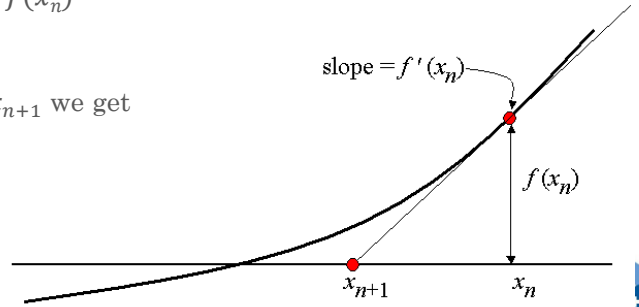


- Tangent line to the graph of $f(x)$ at the point $x = x_n$
  - has slope $f'(x_n)$ and goes through the point $(x_n, f(x_n))$
- This line has equation:

$$y = f'(x_n) \cdot (x - x_n) + f(x_n)$$

- Now, setting $y = 0$ and $x = x_{n+1}$ we get

$$x_{n+1} = x_n - \frac{f'(x_n)}{f(x_n)}$$



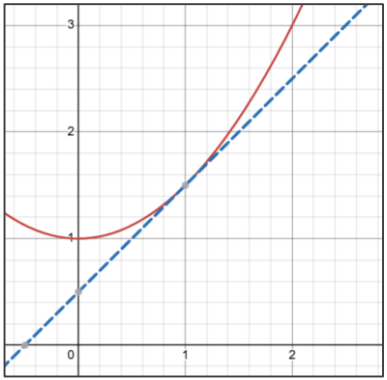
Gradient

- For univariate function (one independent variable)
  - The gradient at x is simply the value of the first derivative of f at x:

$$f'(x) = \frac{df(x)}{dx}$$

Gradient: Interpretation

- The gradient of a function at a given point is the slope of the tangent at that point:



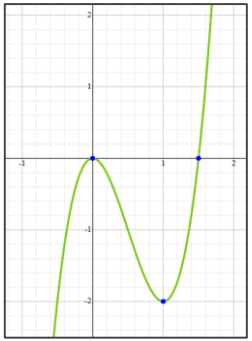
Gradient Examples, Finding Extrema

- For univariate function (one independent variable)

$$f(x) = x^4 - x^3 + 4, \qquad \frac{df(x)}{dx} = 4x^3 - 6x^2 = x^2(4x - 6)$$

The minima and maxima of $f(x)$ can be found by finding the solution of:

$$f'(x) = \frac{df(x)}{dx} = 0 \rightarrow x = 0, x = 1.5$$



Newton-Raphson Method

```
std::pair<double, size_t> newton(double f(double), double f_prime(double),
                                 size_t max_iterations, double xn, double epsilon)
{
    for (size_t step = 0; step != max_iterations; ++step) {
        double fn = f(xn);
        if (fabs(fn) <= epsilon)
            return {fn, step};

        double gn = f_prime(xn);

        xn = xn - gn / fn;
    }
    return {0.0, max_iterations};
}
```

Newton-Raphson Method

```
// Calculate slope (gradient) of f(x) at x.
double f_prime(double f(double), double x, double epsilon)
{
    double delta = epsilon / 50;
    return (f(x + delta) - f(x - delta)) / (2 * delta);
}
```

Newton-Raphson Method

```
std::pair<double, size_t> newton(double f(double),
    double f_prime(double(double), double, double),
    size_t max_iterations, double xn, double epsilon)
{
    for (size_t step = 1; step != max_iterations; ++step) {
        double fn = f(xn);
        if (fabs(fn) <= epsilon)
            return {fn, step};

        double gn = f_prime(f, xn, epsilon);
        xn = xn - gn / fn;
    }
    return {0.0, max_iterations};
}
```

Newton-Raphson Method

```
std::pair<double, size_t> newton(double f(double),
    double f_prime(double(double), double, double),
    double init, unsigned int max_iterations = 100, double epsilon = 0.0001)
{
    return fixed_point(
        [f, f_prime, epsilon](double xn) {
            double gn = f_prime(f, xn, epsilon);
            return xn - gn / f(xn);
        },
        [f, epsilon](double, double curr) {
            return fabs(f(curr)) <= epsilon;
        },
        max_iterations, init);
}
```

Halley's Method

- Very similar, just includes second derivative

$$x_{n+1} = x_n - \frac{2 \cdot f(x_n) \cdot f'(x_n)}{2 \cdot f'(x_n)^2 - f(x)f''(x)}$$

- Requires contiguous second derivative
- This method guarantees
  - at least cubic convergence rate

Schröder's Method

- Very similar, just includes second derivative

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{f''(x_n) \cdot f(x_n)^2}{2 \cdot f'(x_n)^3}$$

- This method guarantees
  - at least quadratic convergence rate
  - is known to work well in the presence of multiple roots

On Infinitely Many Algorithms for Solving Equations

Exercises

- Implement Halley's method
- Implement Schröder's method

Gradient Descent

Gradient

- For multi-variate function (more than one independent variable)
  - The vector of partial derivatives along each of the variable's axes is defined as:

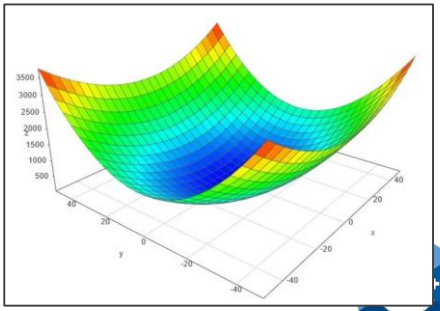
$$\nabla f(p) = \begin{bmatrix} \frac{\delta f(p)}{\delta x_1} \\ \vdots \\ \frac{\delta f(p)}{\delta x_n} \end{bmatrix}, p = (x_1, x_2, \ldots x_n)$$

Gradient Examples

- For multi-variate function (more than one independent variable)
  - The vector of partial derivatives along each of the variable's axes

$$f(x,y) = 0.5x^2 + y^2, \qquad \frac{\delta f(x,y)}{\delta x} = x, \qquad \frac{\delta f(x,y)}{\delta y} = 2y$$

$$\nabla f(x,y) = \begin{bmatrix} x \\ 2y \end{bmatrix}$$



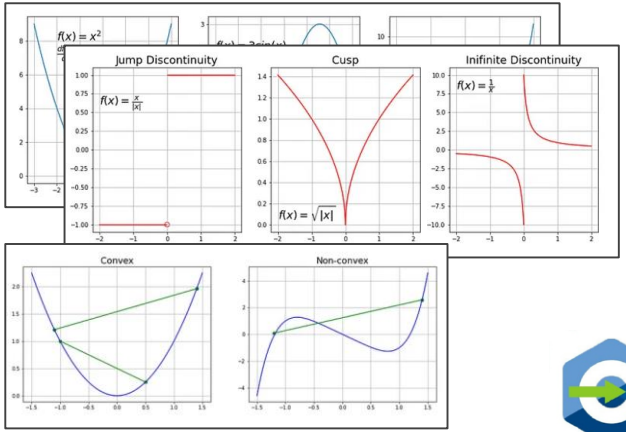
Gradient Descent

- Gradient Descent is an algorithm for finding minima/maxima for convex and differentiable functions
- Differentiable:

$$\frac{df(x)}{dx} \; is \; defined$$

- Convex:

$$\frac{d^2 f(x)}{dx} > 0$$



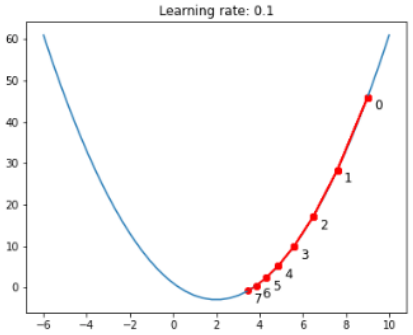
Gradient Descent

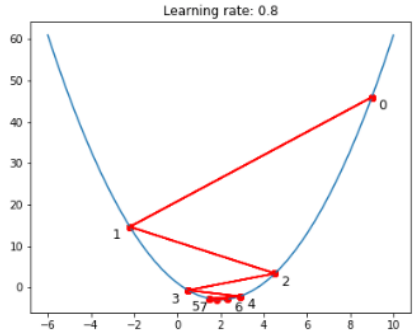
- Iteratively
  - Calculate the next point using the gradient $\nabla f(p_n)$ at the current position p_n
  - Scale the gradient (by a learning rate η)
  - Subtract obtained value from the current position (make a step)

$$p_{n+1} = p_n - \eta \nabla f(p_n)$$

- Where η: learning rate, controls step size
  - The smaller learning rate the longer GD converges
  - If learning rate is too big the algorithm may not converge to the optimal point (jump around) or even diverge completely

Gradient Descent





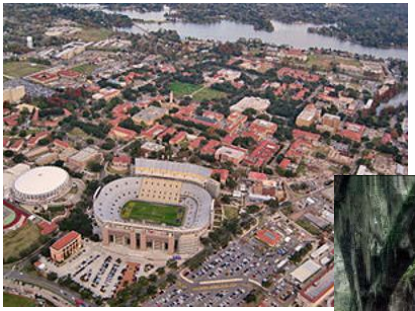
Gradient Descent

```
std::pair<double, unsigned int> gradient_descent(
    double f(double), double xn, unsigned int max_iterations = 100,
    double epsilon = 0.0001, double learning_rate = 0.01)
{
    return fixed_point(
        [&](double xn) {
            double gn = f_prime(f, xn, epsilon);
            return xn - learning_rate * gn;
        },
        [&](double prev, double curr) {
            return fabs(curr - prev) <= epsilon;
        },
        max_iterations, xn);
}
```

Capture all required variables by reference: [&]

Exercises

- Implement gradient descent for functions of two (or more) variables
  - Create a plot visualizing your results for varying learning rates
- Implement a second order algorithm that finds extrema for a given function
  - I.e. find the root of the derivative



LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP