UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical &
Computer Engineering

ECE 204 *Numerical methods*

Iteration

Douglas Wilhelm Harder, LEL, M.Math.
dwharder@uwaterloo.ca
dwharder@gmail.com

Introduction

- In this topic, we will
  - Describe iteration
  - Observe how it can be used to approximate solutions to algebraic problems
  - Describe the fixed-point theorem
  - Discuss issues such as the maximum number of iterations and the tolerances we are willing to accept in our approximations

Iteration

- The next tool we will use is iteration:
  - First, we require mathematical function *f*, which may represent a numerical algorithm
  - Second, we require an initial value, or initial approximation, x_0
- An iteration occurs when start applying *f* to x_0:

$$x_1 \leftarrow f(x_0)$$
$$x_2 \leftarrow f(x_1)$$
$$x_3 \leftarrow f(x_2)$$

- We can repeat this arbitrarily often to produce a sequence of values

$$x_0, x_1, x_2, x_3, x_4, \ldots$$

  - The significance of this sequence depends on *f* and x_0

Iteration

- The function *f* may be a simple arithmetic function,
  or it may describe a complex algorithm involving a number of steps

- Here is a simple function: $f(x) \stackrel{\text{def}}{=} x^2$
  - Even here, depending on the initial value,
    the behavior will change
    - We will have initial values 0.1, 1 and 2:

    0.1, 0.01, 0.0001, 0.00000001, 0.0000000000000001, …

    1, 1, 1, 1, 1, 1, 1, 1, …

    2, 4, 16, 256, 65536, 4294967296, 18446744073709551616, …

Iteration

- We have already seen one example of iteration
  - If x approximates $\sqrt{2}$, then $\frac{2}{x}$ is on the other side of $\sqrt{2}$

$$\sqrt{2} \approx 1.4142135623730950488$$

    - Consequently, the average of these two should be a better approximation of $\sqrt{2}$, so we define

$$f(x) \stackrel{\text{def}}{=} \frac{1}{2}\left(x + \frac{2}{x}\right) = \frac{x}{2} + \frac{1}{x}$$

  - Now, $x_0 = 1.4$ has a 1% relative error if approximating $\sqrt{2}$

$x_1 = 1.414285714285714$

$x_2 = 1.414213564213564$

$x_3 = 1.414213562373095$

$x_4 = 1.414213562373095$

Iteration

- Exercise:
  - Come up with a similar formula for approximating $\sqrt{n}$ for a given positive real number n

No solution is provided, and not on the examination

Fixed-point theorem

Theorem

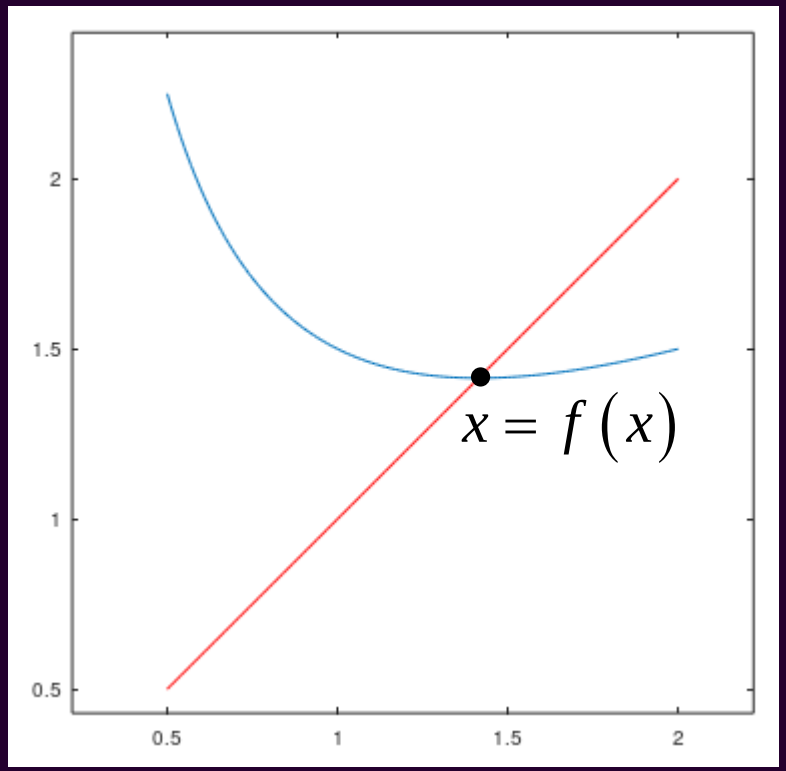
Given the equation $x = f(x)$ and given an initial value x_0,

if the iteration of f starting with x_0 converges, then it converges to a solution of the equation $x = f(x)$

- Important:
  - The function f can be a real-valued function of a real variable
  - It can also be a vector-valued function of a vector variable
    - In this second case, the initial value must too be a vector

Fixed-point theorem

- For example, in our previous problem, $f(x) \overset{\text{def}}{=} \frac{x}{2} + \frac{1}{x}$



$$x = \frac{x}{2} + \frac{1}{x}$$

$$\frac{x}{2} = \frac{1}{x}$$

$$x^2 = 2$$

Fixed-point theorem

- We could now write a function to do this for us:

```
double fixed_point( double f( double x ), double x0 ) {
    while ( true ) {
        double previous_x0{ x0 };
        x0 = f( x0 );

        if ( x0 == previous_x0 ) {
            return x0;
        }
    }
}

double sqrt2( double x ) {
    return x/2.0 + 1.0/x;
}
```

```
int main() {
    std::cout.precision( 16 );
    std::cout
      << fixed_point( sqrt2, 1.4 )
      << std::endl;

    return 0;
}
```

Fixed-point theorem

- We could also author this in MATLAB:

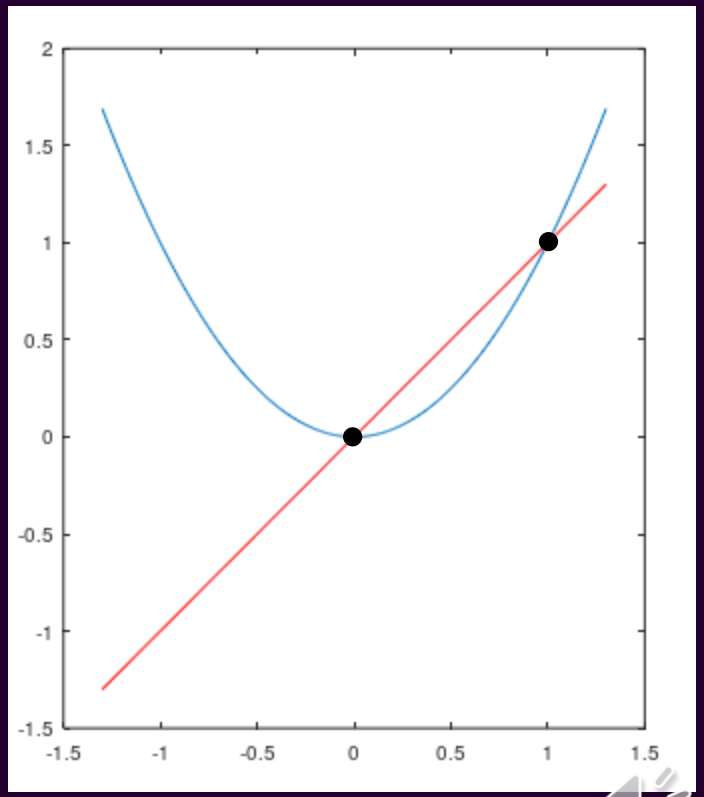
```
function [x0] = fixed_point( f, x0 )
    while true
        previous_x0 = x0;
        x0 = f( x0 );

        if x0 == previous_x0
            return;
        end
    end
end
```

This must be saved to a file `fixed_point.m`

```
>> format long
>> sqrt2 = @(x)( x/2.0 + 1.0/x );
>> fixed_point( sqrt2, 1.4 )
    ans =
          1.414213562373095
```

Fixed-point theorem

- At the start of this topic, we looked at iterating $f(x) \overset{\text{def}}{=} x^2$
  - The fixed-point theorem says that if an iteration converges, then that iteration converges to a solution of $x = f(x)$
  - There are two solutions: 0 and 1
    - If $-1 < x_0 < 1$, then the iteration converges to 0
    - If $x_0 = 1$ or $x_0 = -1$, then the iteration converges to 1
    - Otherwise, the iteration diverges



Diverging sequences

- This leads to our first problem: When do we stop iterating?
  - We must parameterize our algorithms to stop iterating after a given number of iterations
  - We must also check to see if the iteration is no longer finite

Fixed-point theorem

- We could now update our C++ function:

```
double fixed_point( double f( double x ),
                    double x0,
                    unsigned int const max_iterations ) {
    for ( unsigned int iterations{0};
          iterations < max_iterations; ++iterations ) {
        double previous_x0{ x0 };
        x0 = f( x0 );

        if ( !std::isfinite( x0 ) ) {
            return NAN;
        }

        if ( x0 == previous_x0 ) {
            return x0;
        }
    }

    return NAN;
}
```


Fixed-point theorem

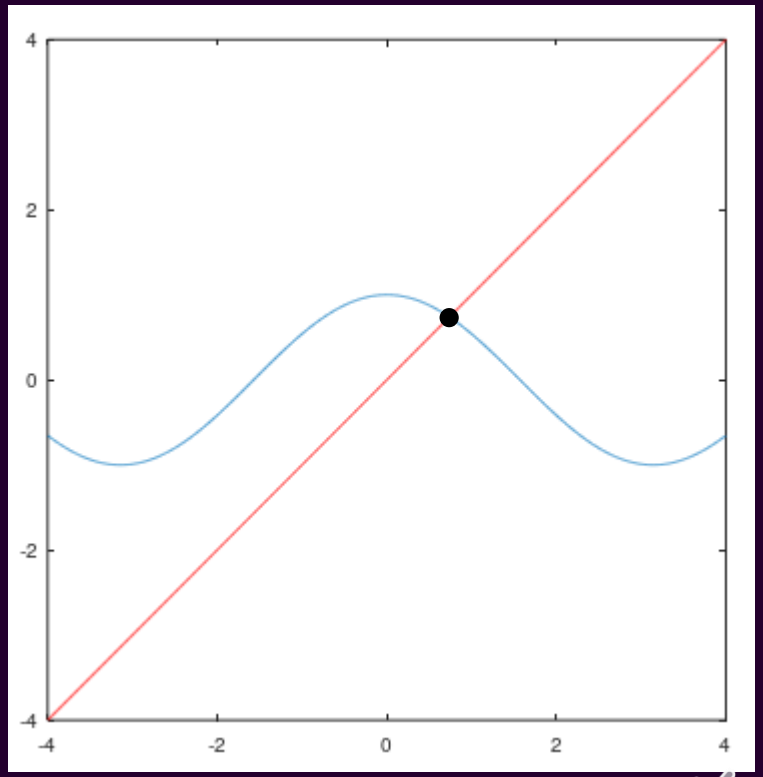
- We could also author this in MATLAB:

```
function [x0] = fixed_point( f, x0, max_iterations )
    for iterations = 1:max_iterations
        previous_x0 = x0;
        x0 = f( x0 );

        if ~isfinite( x0 )
            x0 = nan;
            return;
        end

        if x0 == previous_x0
            return;
        end
    end

    x0 = nan;
    return;
end
```

```
>> format long
>> g = @(x)( x^2 );
>> fixed_point( g, 1.4, 1000 )
    ans =
                nan
```

Fixed-point theorem

- Suppose instead we start with $f(x) \overset{\text{def}}{=} \cos(x)$
  - This has a single solution close to $x = 0.7391$
  - Given any x_0, then the iteration of this function f will converge to that one unique solution



Iterating the cosine function

- Let us start with $x_0 = 0.7$:

0.7	
0.7648421872844885	0.7390947388395475
0.7214916395975273	0.7390786627169290
0.7508213288394496	0.7390894918051325
0.7311287725733576	0.7390821972095050
0.7444211836271648	0.7390871109407026
0.7354802004059856	0.7390838009939997
0.7415086516600415	0.7390860306147043
0.7374504531501768	0.7390845287157354
0.7401852853967579	0.7390855404131101
0.7383436103510045	0.7390848589216623
0.7395844286953486	0.7390853179825329
0.7387487096620905	0.7390850087536235
0.7393117103380089	0.7390852170539435
0.7389324891697003	0.7390850767403454
0.7391879474695492	0.7390851712572744
0.7390158723904052	0.7390851075895347
0.7391317863671112	0.7390851504768903
0.7390537062865034	0.7390851215874518
0.7391063024073616	0.7390851410477252
0.7390708732270421	0.7390851279390509

Iterating the cosine function

- If we are applying iteration,
  we don't usually need 16 significant digits
    - We seldom require more than six
  - We don't want to wait forever…
- We will make an assumption:
  - If $|x_{n+1} - x_n| < \varepsilon_{\text{step}}$, then x_{n+1} should also be close enough to whatever its converging to
    - This is not necessarily true,
      but in most cases it is sufficient

Iterating the cosine function

- Important: this is an absolute tolerance,
  and not a relative tolerance
  - If the solution is actually at $x = 0$,
    then the relative error is not even defined
  - You must be aware of what you are expecting the solution to be to use such techniques

Fixed-point theorem

- We could now update our C++ function:

```
double fixed_point( double f( double x ),
                    double x0,
                    double step_tolerance,
                    unsigned int const max_iterations ) {
    for ( unsigned int iterations{0};
          iterations < max_iterations; ++iterations ) {
        double previous_x0{ x0 };
        x0 = f( x0 );

        if ( !std::isfinite( x0 ) ) {
            return NAN;
        }

        if ( std::abs( x0 - previous_x0 ) < step_tolerance ) {
            return x0;
        }
    }

    return NAN;
}
```

Fixed-point theorem

- We could also author this in MATLAB:

```
function [x0] = fixed_point( f, x0, step_tolerance, max_iterations )
    for iterations = 1:max_iterations
        previous_x0 = x0;
        x0 = f( x0 );

        if ~isfinite( x0 )
            x0 = nan;
            return;
        end

        if abs( x0 - previous_x0 ) < step_tolerance
            return;
        end
    end

    x0 = nan;
    return;
end
```

```
>> format long
>> fixed_point( @cos, 0.7, 1e-6, 1000 )
ans =
   0.739084858921662
   0.739085133215161
```

Summary

- Following this topic, you now
  - Understand how iteration can be used to solve analytic problems
  - Have been exposed to the fixed-point theorem
  - Understand that we must limit the number of iterations
    - After all, the iteration may diverge, or take forever to converge
  - We must also set the absolute tolerance for our solution

References

[1] https://en.wikipedia.org/wiki/Iteration

[2] https://en.wikipedia.org/wiki/Fixed-point_theorem

Acknowledgments

Tazik Shahjahan for pointing out typos.

An anonymous Class of 2025 student who noticed that my iterative sequence was a geometric sequence!

Colophon

These slides were prepared using the Cambria typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`. Mathematical equations are prepared in MathType by Design Science, Inc.

Examples may be formulated and checked using Maple by Maplesoft, Inc.

The photographs of flowers and a monarch butter appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens in October of 2017 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.

Disclaimer

These slides are provided for the ECE 204 *Numerical methods* course taught at the University of Waterloo. The material in it reflects the author's best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.