4. Functions

Last time

- Representing real numbers

- Fixed-point arithmetic
  - Integers with fixed denominator

- Floating-point numbers

$$x = \pm 2^e (1 + f); \qquad f = \sum_{n=1}^{p} 2^{-n} b_n$$

  - Special rational numbers (denominator is power of 2)

Outline for today's lecture

- What is a function?: Mathematics vs. computing
- How can we calculate elementary functions like `exp`?

Functions of real numbers

- What is a function?

Functions of real numbers

- What is a function?
- Mathematics: “Something like $f(x) = \sin(x^2) + x$”

Functions of real numbers

- What is a function?
- Mathematics: “Something like $f(x) = \sin(x^2) + x$”
- Computing: ”Something like

```
function f(x)
    return sin(x^2) + x
end
```

”

Collaborative exercise

What is a function?

1. What does $y = f(x)$ mean for a mathematician?
   - Let's think about x and y being real numbers for now
2. What is a function `y = f(x)` for a computer scientist?

Functions: Mathematically

- For a mathematician, $y = f(x)$ associates the single value y with the value x

Functions: Mathematically

- For a mathematician, $y = f(x)$ associates the single value y with the value x
- We can think of f as the set of all pairs $(x, f(x))$
  - **graph** of the function
  - a **relation** (subset of $\mathbb{R}^2$)

Functions: Mathematically

- For a mathematician, $y = f(x)$ associates the single value y with the value x
- We can think of f as the set of all pairs $(x, f(x))$
  - **graph** of the function
  - a **relation** (subset of $\mathbb{R}^2$)
- It's not clear how this association "happens"

Functions: Computationally

- Computationally a **function** is a more “active” object
- It tells us *how* to **compute**

Functions: Computationally

- Computationally a **function** is a more “active” object
- It tells us *how* to **compute**

- It has an **input** and an **output**
- It represents a **sequence of operations** that are applied, one after the other, to the input

Functions: Computationally II

- Consider the function `f(x) = sin(x^2) + x` again

Functions: Computationally II

- Consider the function `f(x) = sin(x^2) + x` again
- It is transformed into something like the following

```
function f(x)
    a = x * x
    b = sin(a)
    c = b + x
    return c
end
```

- Called a **code list** or **computational graph**

Functions: Computationally II

- Consider the function `f(x) = sin(x^2) + x` again
- It is transformed into something like the following

```
function f(x)
    a = x * x
    b = sin(a)
    c = b + x
    return c
end
```

- Called a **code list** or **computational graph**
- `f` becomes a sequence of operations – an **algorithm**
- This is what Julia actually does:
  - `@code_lowered f(3.1)` and `@code_typed f(3.1)`

What can we compute?

- If we type `exp(1.3)` in Julia, it returns a real number:

```
julia> x = 1.3;

julia> exp(x)
3.6692966676192444
```

What can we compute?

- If we type `exp(1.3)` in Julia, it returns a real number:

```
julia> x = 1.3;

julia> exp(x)
3.6692966676192444
```

- Or, rather, a **floating-point** result

What can we compute?

- If we type `exp(1.3)` in Julia, it returns a real number:

```
julia> x = 1.3;

julia> exp(x)
3.6692966676192444
```

- Or, rather, a **floating-point** result

- Here we specified an input `x = 1.3`
- I.e. `x` = $\text{fl}(1.3)$, the closest float to $13/10$
- We are asking to calculate the closest float to $\exp(x)$

Rephrase as a problem

- This is an example of a (mathematical) **problem**:

  Given an input x*, calculate* $y = f(x)$

Rephrase as a problem

- This is an example of a (mathematical) **problem**:

  Given an input x*, calculate* $y = f(x)$

- In fact, we have:
  - An approximate input $\tilde{x} = \mathrm{fl}(x)$
  - An approximate function $\widetilde{\exp}$
  - Giving an approximate output $\tilde{y}$

Rephrase as a problem

- This is an example of a (mathematical) **problem**:

  Given an input x*, calculate* $y = f(x)$

- In fact, we have:
  - An approximate input $\tilde{x} = \text{fl}(x)$
  - An approximate function $\widetilde{\text{exp}}$
  - Giving an approximate output $\tilde{y}$
- We want to solve this to some relative accuracy ϵ, i.e.

$$\frac{|\tilde{y} - y|}{|y|} < \epsilon$$

What can we compute? II

- We can increase the precision: `exp(big(1.3))`
- $\exp(x)$ is **computable**

What can we compute? II

- We can increase the precision: `exp(big(1.3))`
- $\exp(x)$ is **computable**

- We can calculate **arbitrarily good approximations**
- Given an input x, there exists an **approximation algorithm** for $\exp(x)$

What can we compute? II

- We can increase the precision: `exp(big(1.3))`
- $\exp(x)$ is **computable**

- We can calculate **arbitrarily good approximations**
- Given an input x, there exists an **approximation algorithm** for $\exp(x)$

- Which real functions can actually be computed in this sense?
- This is the subject of **computable analysis**

Collaborative exercise: Computing $\exp(x)$?

- To compute $\exp(x)$ to accuracy ϵ, we need an **algorithm**

Computing `exp`

1. What is the mathematical definition of $\exp(x)$? (The version of the definition that might be useful for computing.)
2. Can we implement that on the computer? Why (not)?
3. How could we write a function `my_exp` to compute exp to a given precision?
4. Which type of arithmetic operations do we need?

The exp function

- The function $\exp(x)$ is defined as a **power series**:

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

where $n! := 1 \times 2 \times \cdots \times n$ is the **factorial** function

The exp function

- The function $\exp(x)$ is defined as a **power series**:

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

  where $n! := 1 \times 2 \times \cdots \times n$ is the **factorial** function

- Alternative (equivalent) definition: $\exp' = \exp$

  where $f' :=$ the derivative of f

The exp function

- The function exp(x) is defined as a **power series**:

$$\exp(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

where $n! := 1 \times 2 \times \cdots \times n$ is the **factorial** function

- Alternative (equivalent) definition: $\exp' = \exp$

where $f' :=$ the derivative of f
- The power series converges for *all* real x
- In fact, for all *complex* x!

Computing the exp function

- Clearly we *cannot* calculate an infinite sum
- We must *approximate* the result

Computing the exp function

- Clearly we *cannot* calculate an infinite sum
- We must *approximate* the result

- **Truncate** to calculate the *finite* sum

$$f_N(x) = \sum_{n=0}^{N} \frac{x^n}{n!} = 1 + x + \tfrac{1}{2}x^2 + \tfrac{1}{6}x^3 + \cdots + \tfrac{1}{N!}x^N$$

Computing the exp function

- Clearly we *cannot* calculate an infinite sum
- We must *approximate* the result

- **Truncate** to calculate the *finite* sum

$$f_N(x) = \sum_{n=0}^{N} \frac{x^n}{n!} = 1 + x + \tfrac{1}{2}x^2 + \tfrac{1}{6}x^3 + \cdots + \tfrac{1}{N!}x^N$$

- This is now a **polynomial**

Polynomials

- **Polynomials** are the *simplest* functions
- Foundation of most of numerical computation (and mathematics)

Polynomials

- **Polynomials** are the *simplest* functions
- Foundation of most of numerical computation (and mathematics)

- $f(x) = a_0 + a_1\, x + \cdots + a_n x^n$

  is a (univariate) **polynomial of degree** n

Polynomials

- **Polynomials** are the *simplest* functions
- Foundation of most of numerical computation (and mathematics)

- $f(x) = a_0 + a_1\,x + \cdots + a_n x^n$

  is a (univariate) **polynomial of degree** n
- There are a *finite* number $n + 1$ of terms

Polynomials II

- To *calculate* $f(x)$, need only $+$ and $\star$ defined on x
  - i.e. x must belong to a **ring**

Polynomials II

- To *calculate* $f(x)$, need only $+$ and $\star$ defined on x
  - i.e. x must belong to a **ring**
- Polynomials are the *only* functions we can calculate!

Polynomials II

- To *calculate* $f(x)$, need only $+$ and $\star$ defined on x
  - i.e. x must belong to a **ring**
- Polynomials are the *only* functions we can calculate!
- We will need to use polynomials to approximate all other functions!
- Does this make sense?

Approximating functions with polynomials

- We will (implicitly) apply the following amazing theorem:

- Weierstrass approximation theorem:

  any continuous *function on a finite interval*
  can be approximated ***uniformly*** *by a polynomial*
  to within any given distance ϵ

Approximating functions with polynomials

- We will (implicitly) apply the following amazing theorem:

- Weierstrass approximation theorem:

  any continuous *function on a finite interval*
  can be approximated ***uniformly*** *by a polynomial*
  to within any given distance ϵ

- **Uniformly** means that the distance $\|f - p\|_\infty < \epsilon$:

$$\|f - p\|_\infty := \max_{x \in [a,b]} |f(x) - p(x)| < \epsilon$$

Approximating functions with polynomials

- We will (implicitly) apply the following amazing theorem:

- Weierstrass approximation theorem:

  any continuous *function on a finite interval*
  can be approximated ***uniformly*** *by a polynomial*
  to within any given distance ϵ

- **Uniformly** means that the distance $\|f - p\|_\infty < \epsilon$:

$$\|f - p\|_\infty := \max_{x \in [a,b]} |f(x) - p(x)| < \epsilon$$

- NB: This “uniform” condition fails for *infinite* intervals

How can we *find* polynomial approximations?

- The beautiful Weierstrass approximation theorem is…

How can we *find* polynomial approximations?

- The beautiful Weierstrass approximation theorem is…
  …completely useless!

How can we *find* polynomial approximations?

- The beautiful Weierstrass approximation theorem is…
  …completely useless!
- It does not tell us how to *find* the polynomial p!

How can we *find* polynomial approximations?

- The beautiful Weierstrass approximation theorem is…
  …completely useless!
- It does not tell us how to *find* the polynomial p!
- A major theme of numerical computation:
  - **find methods** (algorithms) to **compute** polynomial approximations of a function f

Summary

- We want to evaluate real functions
- **Approximate** them using functions that we understand

Summary

- We want to evaluate real functions
- **Approximate** them using functions that we understand

- Polynomials are those functions

Summary

- We want to evaluate real functions
- **Approximate** them using functions that we understand

- Polynomials are those functions

- We need to find approximation algorithms for **functions**