Derivatives.
Important concept.
Simple to grasp in Kotlin.

Breandan Considine

@breandan

Copenhagen

Denmark

1. What are derivatives?
2. How do derivatives work?
3. What can be derived?
4. How can I derive in Kotlin?
5. What's the difference?

What are derivatives?

Leibniz: derivatives as rate of change

“It is unworthy of excellent [minds] to lose hours like slaves in the labor of calculation which could safely be relegated to anyone else if machines were used.”

-Gottfried Wilhelm Leibniz

https://en.wikipedia.org/wiki/Gottfried_Wilhelm_Leibniz

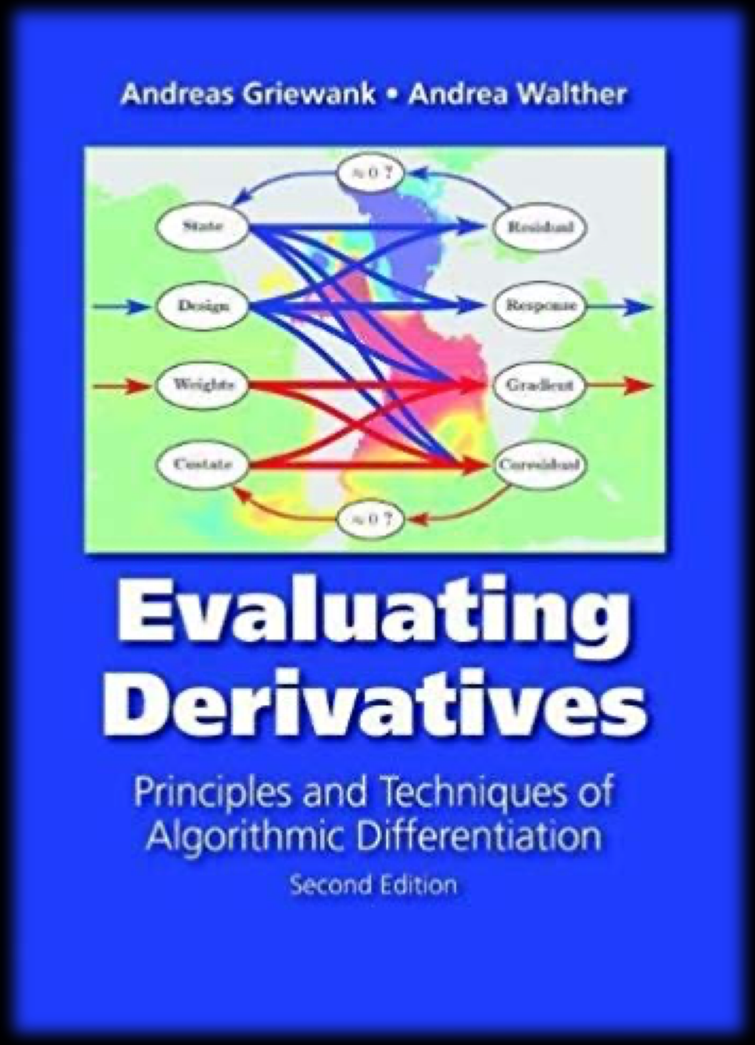
Linnainmaa: Reverse mode differentiation

Seppo Linnainmaa

http://www.helbig.dk/neucc/index.htm

Griewank: Algorithmic differentiation

Andreas Griewank



1. An algorithm for minimizing any function

1. An algorithm for minimizing any function

```
tailrec fun <I, O : Comparable<O>> minimize(
    fn: (I) -> (O), min: I, budget: Int): I =
```

1. An algorithm for minimizing any function

```
tailrec fun <I, O : Comparable<O>> minimize(
  fn: (I) -> (O), min: I, budget: Int): I =
  if (budget <= 0) min
  else minimize(fn, sample<I>().let { input ->
    if (fn(input) < fn(min)) input else min
  }, budget - 1)
```

1. An algorithm for minimizing any function

```
tailrec fun <I, O : Comparable<O>> minimize(
  fn: (I) -> (O), min: I, budget: Int): I =
  if (budget <= 0) min
  else minimize(fn, sample<I>().let { input ->
    if (fn(input) < fn(min)) input else min
  }, budget - 1)

fun <I> sample(): I = TODO()
```

2. Better algorithm (but more restrictive)

2. Better algorithm (but more restrictive)

```
interface Metric<T : Metric<T>> : Comparable<T> {
  operator fun plus(t: T): T
  operator fun minus(t: T): T
}
```

2. Better algorithm (but more restrictive)

```
interface Metric<T : Metric<T>> : Comparable<T> {
  operator fun plus(t: T): T
  operator fun minus(t: T): T
}

tailrec fun <I, O : Metric<O>> minimizeMetric(
    fn: (I) -> (O), min: I, budget: Int): I =
    if (budget <= 0) min
    else minimizeMetric(fn, wiggle(min).filter { fn(it) < fn(min) }
         .maxBy { fn(min) - fn(it) } ?: min, budget - 1)
```

2. Better algorithm (but more restrictive)

```
interface Metric<T : Metric<T>> : Comparable<T> {
  operator fun plus(t: T): T
  operator fun minus(t: T): T
}

tailrec fun <I, O : Metric<O>> minimizeMetric(
    fn: (I) -> (O), min: I, budget: Int): I =
    if (budget <= 0) min
    else minimizeMetric(fn, wiggle(min).filter { fn(it) < fn(min) }
         .maxBy { fn(min) - fn(it) } ?: min, budget - 1)

fun <I> wiggle(min: I): Sequence<I> = TODO()
```

3. Still better algorithm (even more restrictive)

3. Still better algorithm (even more restrictive)

```
interface Field<T : Field<T>> : Metric<T> {
  operator fun times(t: T): T
  operator fun div(t: T): T
}
```

3. Still better algorithm (even more restrictive)

```
interface Field<T : Field<T>> : Metric<T> {
  operator fun times(t: T): T
  operator fun div(t: T): T
}

tailrec fun <T: Field<T>> fieldMinimize(
  fn: (T) -> (T), a: T, min: T, budget: Int): T =
  if(budget <= 0) min
  else minimizeField(fn, a,
         min - (fn(min + a) - fn(min)) / a, budget - 1)
```

3. Still better algorithm (even more restrictive)

```
interface Field<T : Field<T>> : Metric<T> {
  operator fun times(t: T): T
  operator fun div(t: T): T
}

tailrec fun <T: Field<T>> fieldMinimize(
  fn: (T) -> (T), a: T, min: T, budget: Int): T =
  if(budget <= 0) min
  else minimizeField(fn, a,
         min - (fn(min + a) - fn(min)) / a, budget - 1)
```

$$\frac{d}{dx} f(x) = \frac{df}{dx} = \lim_{a \to 0} \frac{f(x+a) - f(x)}{a}$$

4. So what's the problem?

$$\frac{f(x+a)-f(x)}{a}$$

4. So what's the problem?

$$\frac{f(x+a) - f(x)}{a}$$

```
fn: (T) -> (T)
```

4. So what's the problem?

$$\frac{f(x+a) - f(x)}{a}$$

```
fn: (T) -> (T)

fn: (T, T) -> (T)
```

4. So what's the problem?

$$\frac{f(x+a) - f(x)}{a}$$

```
fn: (T) -> (T)

fn: (T, T) -> (T)

fn: (T, T, T) -> (T)
```

4. So what's the problem?

$$\frac{f(x+a) - f(x)}{a}$$

```
fn: (T) -> (T)

fn: (T, T) -> (T)

fn: (T, T, T) -> (T)
```

$\Big\}\ \mathcal{O}(n)$

4. So what's the problem?

$$\frac{f(x+a) - f(x)}{a}$$

```
fn: (T) -> (T)
fn: (T, T) -> (T)
fn: (T, T, T) -> (T)
```

$\Big\}\ \mathcal{O}(n)$

```
fn: (T) -> Pair<T, T>
fn: (T, T) -> Triple<T, T>
```

4. So what's the problem?

$$\frac{f(x+a) - f(x)}{a}$$

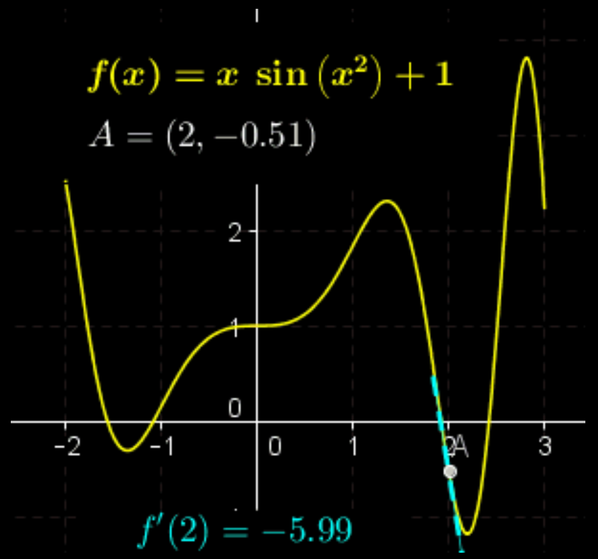
```
fn: (T) -> (T)
fn: (T, T) -> (T)
fn: (T, T, T) -> (T)
```

$\mathcal{O}(n)$

```
fn: (T) -> Pair<T, T>
fn: (T, T) -> Triple<T, T>
```

$\mathcal{O}(nm)$

How do derivatives work?



1. Sum rule

$$\frac{d}{dx}(l+r) = \frac{dl}{dx} + \frac{dr}{dx}$$

2. Product rule

$$\frac{d}{dx}(l \cdot r) = \frac{dl}{dx} \cdot r + l \cdot \frac{dr}{dx}$$

2. Chain rule

$$\frac{d}{dx}(l \circ r) = \frac{dl}{dr} \cdot \frac{dr}{dx}$$

2. Chain rule, revisited

$$f = l \circ r = l(r(x)) \qquad \frac{df}{dx} = \frac{dl}{dr} \cdot \frac{dr}{dx}$$

$$P_k(x) = \begin{cases} p_1 \circ x = x & \text{if } k = 1 \\ p_k \circ P_{k-1} \circ x & \text{if } k > 1 \end{cases} \qquad \frac{dP}{dp_1} = \frac{dp_k}{dp_{k-1}} \frac{dp_{k-1}}{dp_{k-2}} \frac{dp_{k-2}}{dp_{k-3}} \cdots \frac{dp_2}{dp_1}$$

"Forward accumulation": $$\frac{dP}{dp_1} = \frac{dp_k}{dp_{k-1}} \left(\frac{dp_{k-1}}{dp_{k-2}} \left(\frac{dp_{k-2}}{dp_{k-3}} \cdots \frac{dp_2}{dp_1} \right) \right)$$

"Reverse accumulation": $$\frac{dP}{dp_1} = \left(\left(\frac{dp_k}{dp_{k-1}} \frac{dp_{k-1}}{dp_{k-2}} \right) \frac{dp_{k-2}}{dp_{k-3}} \right) \cdots \frac{dp_2}{dp_1}$$

Forward vs. Reverse mode accumulation

Forward

$$\underbrace{\mathbf{J_h}(\mathbf{y_2})}_{n\times q}\Big(\underbrace{\mathbf{J_g}(\mathbf{y_1})}_{q\times p}\ \underbrace{\mathbf{J_f}(\mathbf{x})}_{p\times m}\Big)$$

$$q\times m$$

Cost

$$qpm + nqm$$
$$= m(qp + nq)$$

Reverse

$$\Big(\underbrace{\mathbf{J_h}(\mathbf{y_2})}_{n\times q}\ \underbrace{\mathbf{J_g}(\mathbf{y_1})}_{q\times p}\Big)\underbrace{\mathbf{J_f}(\mathbf{x})}_{p\times m}$$

$$n\times p$$

Cost

$$nqp + npm$$
$$= n(qp + pm)$$

cf. Breleux et al., https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9244-deep-learning-with-myia.pdf

Forward vs. Reverse mode accumulation

Forward mode is good when there are **few inputs**.

- **Easy to implement: dual numbers.**

$$x \to \left(y_1, \frac{dy_1}{dx}\right) \to \left(y_2, \frac{dy_2}{dx}\right) \to \left(y_3, \frac{dy_3}{dx}\right)$$

Reverse mode is good when there are **few outputs**.

- **Hard to implement: execution is reversed.**

$$x \to y_1 \to y_2 \to y_3 \to \frac{dy_3}{dy_2} \to \frac{dy_3}{dy_1} \to \frac{dy_3}{dx}$$

cf. Breleux et al., https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9244-deep-learning-with-myia.pdf

Differentiation rules in Kotlin

```
class D<X: Fun<X>>(val f: Fun<X>): Fun<X>(f) {
  fun Fun<X>.df(): Fun<X> = when (this) {


    is Sum   -> left.df() + right.df()
    is Prod  -> left.df() * right + left * right.df()
    is Comp  -> left.df()(right) * right.df()

  }
}
```

Differentiation rules in Kotlin

```
class D<X: Fun<X>>(val f: Fun<X>): Fun<X>(f) {
  fun Fun<X>.df(): Fun<X> = when (this) {
    is Const -> Zero()
    is Var   -> One()
    is Sum   -> left.df() + right.df()
    is Prod  -> left.df() * right + left * right.df()
    is Comp  -> left.df()(right) * right.df()
    is D     -> f.df()
  }
}
```

Functions in Kotlin

```
sealed class Fun<X : Fun<X>>(open val sVars: Set<Var<X>> = emptySet())
                            : Field<Fun<X>>, (Bindings<X>) -> Fun<X> {
    constructor(fn: Fun<X>) : this(fn.sVars)
    constructor(vararg fns: Fun<X>) : this(fns.flatMap { it.sVars }.toSet())

    override operator fun plus(addend: Fun<X>): Fun<X> = Sum(this, addend)
    override operator fun times(multiplicand: Fun<X>): Fun<X> = Prod(this, multiplicand)
    override operator fun div(divisor: Fun<X>): Fun<X> = this * divisor.pow(-One<X>())

    override operator fun invoke(bnds: Bindings<X>): Fun<X> =
         Composition(this, bnds).run { if (bnds.isReassignmentFree) evaluate else this }

    open operator fun invoke(): Fun<X> = invoke(Bindings())

    open fun d(v1: Var<X>): Fun<X> = Derivative(this, v1)
    open fun d(v1: Var<X>, v2: Var<X>): Vec<X, D2> =
         Vec(Derivative(this, v1), Derivative(this, v2))
    open fun d(vararg vars: Var<X>): Map<Var<X>, Fun<X>> =
         vars.map { it to Derivative(this, it) }.toMap()
}
```

Compile-time shape safety

Math	Infix	Prefix	Postfix	Operator Type Signature
$\mathbf{A}(\mathbf{B})$ $\mathbf{A} \circ \mathbf{B}$	`a(b)`			$(\texttt{a} : \mathbb{R}^\tau \to \mathbb{R}^\pi,\ \texttt{b} : \mathbb{R}^\lambda \to \mathbb{R}^\tau) \to (\mathbb{R}^\lambda \to \mathbb{R}^\pi)$
$\mathbf{A} \pm \mathbf{B}$	`a + b` `a - b`	`plus(a, b)`		$(\texttt{a} : \mathbb{R}^\tau \to \mathbb{R}^\pi,\ \texttt{b} : \mathbb{R}^\lambda \to \mathbb{R}^\pi) \to (\mathbb{R}^? \to \mathbb{R}^\pi)$
$\mathbf{AB}$	`a * b` `a.times(b)`	`times(a, b)`		$(\texttt{a} : \mathbb{R}^\tau \to \mathbb{R}^{m \times n},\ \texttt{b} : \mathbb{R}^\lambda \to \mathbb{R}^{n \times p}) \to (\mathbb{R}^? \to \mathbb{R}^{m \times p})$

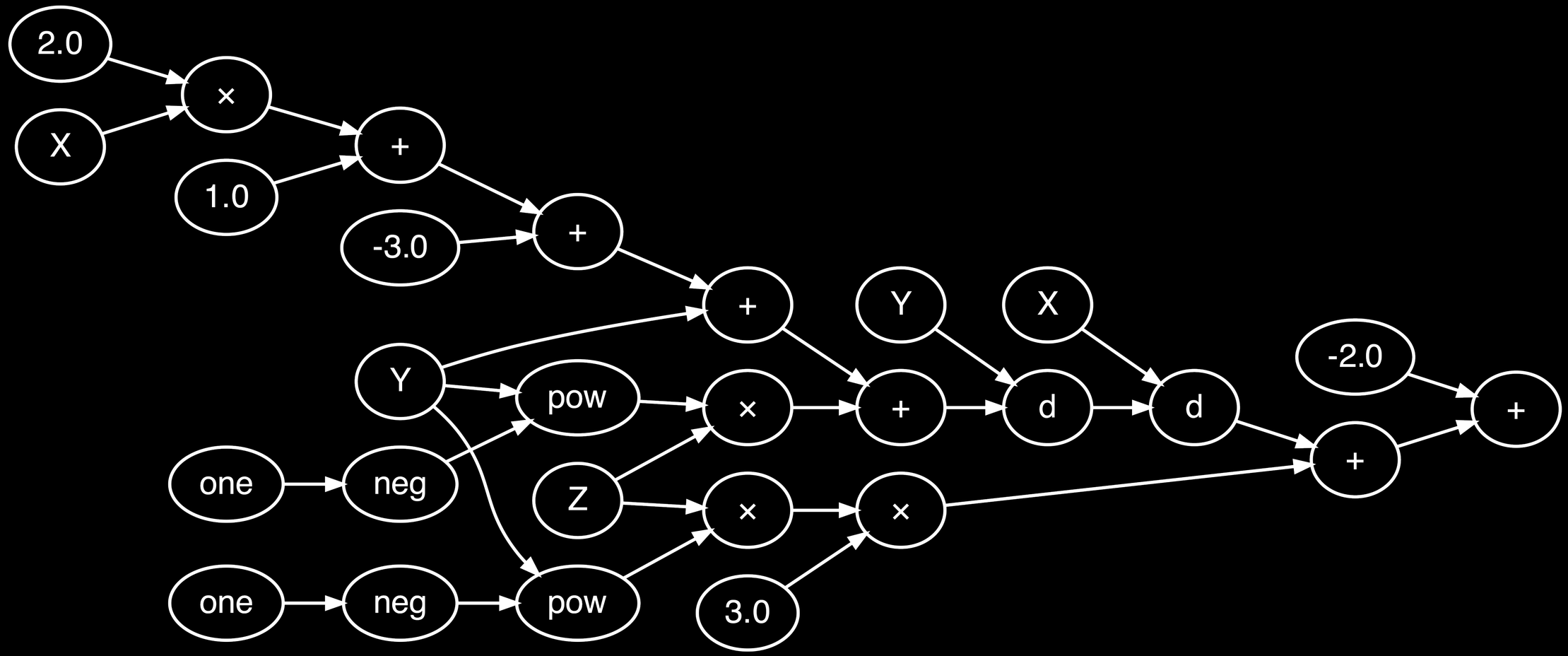
$\sin(\mathrm{a})$
$\cos(\mathrm{a})$
$\tan(\mathrm{a})$
		`sin(a)`
		`cos(a)`
		`tan(a)`
			`a.sin()`
			`a.cos()`
			`a.tan()`
				$(\texttt{a} : \mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$

$\log_b \mathbf{A}$	`a.log(b)`	`log(a, b)`		$(\texttt{a} : \mathbb{R}^\tau \to \mathbb{R}^{m \times m},\ \texttt{b} : \mathbb{R}^\lambda \to \mathbb{R}^{m \times m}) \to (\mathbb{R}^? \to \mathbb{R})$
$\mathbf{A}^b$	`a.pow(b)`	`pow(a, b)`		$(\texttt{a} : \mathbb{R}^\tau \to \mathbb{R}^{m \times m},\ \texttt{b} : \mathbb{R}^\lambda \to \mathbb{R}) \to (\mathbb{R}^? \to \mathbb{R}^{m \times m})$
$\frac{da}{db}, \frac{\partial a}{\partial b}$ $D_b a$	`a.d(b)` `d(a)/d(b)`	`grad(a)[b]`		$\big(\texttt{a} : C(\mathbb{R}^\tau \to \mathbb{R}),\ \texttt{b} : C(\mathbb{R}^\lambda \to \mathbb{R})\big) \to (\mathbb{R}^? \to \mathbb{R})$
∇a		`grad(a)`	`a.grad()`	$\big(\texttt{a} : C(\mathbb{R}^\tau \to \mathbb{R})\big) \to (\mathbb{R}^\tau \to \mathbb{R}^\tau)$
$\nabla_{\mathbf{B}} a$	`a.d(b)` `a.grad(b)`	`grad(a, b)`		$\big(\texttt{a} : C(\mathbb{R}^\tau \to \mathbb{R}),\ \texttt{b} : C(\mathbb{R}^\lambda \to \mathbb{R}^n)\big) \to (\mathbb{R}^? \to \mathbb{R}^n)$

Lazy Composition

```
class Composition<X : Fun<X>>(val fn: Fun<X>, val bindings: Bindings<X>) : Fun<X>() {
    val evaluate by lazy { apply() }
    override val sVars: Set<Var<X>> by lazy { evaluate.sVars }

    fun Fun<X>.apply(): Fun<X> =
        bindings.sMap.getOrElse(this) {
            when (this) {
                is Var -> this
                is Const -> this
                is Prod -> left.apply() * right.apply()
                is Sum -> left.apply() + right.apply()
                is Power -> base.apply() pow exponent.apply()
                is Derivative -> df().apply()
                is Composition -> fn.apply().apply()
            }
        }
}
```

Dataflow graphs



Kraphviz by Stefan Niederhauser: https://github.com/nidi3/graphviz-java#kotlin-dsl

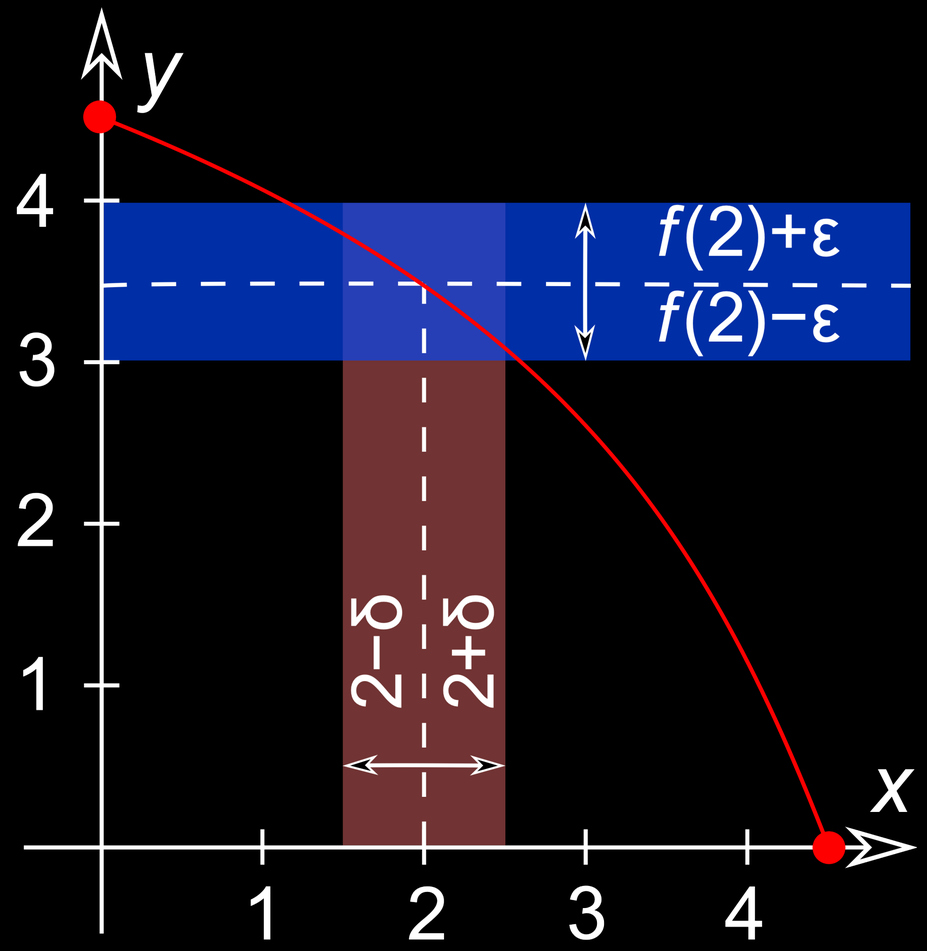
What can be derived?

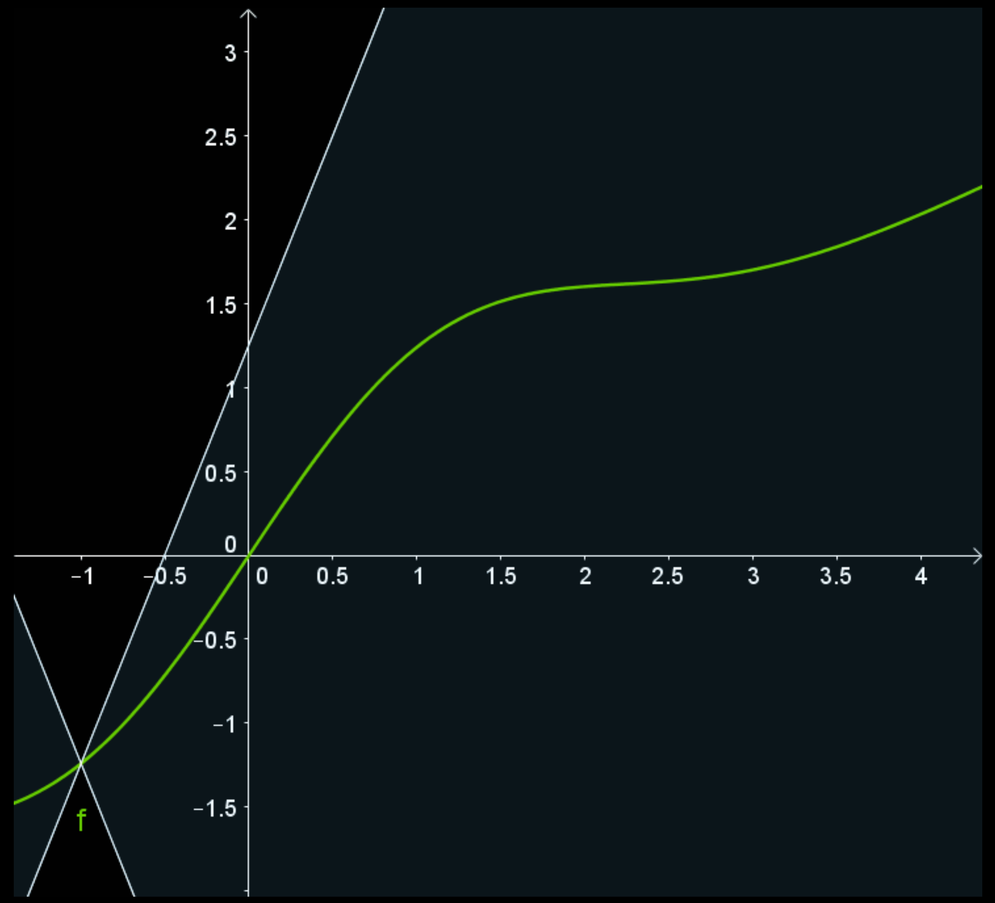
Optimization is just one perspective

“With four parameters I can fit an elephant, and with five I can make him wiggle his trunk.”

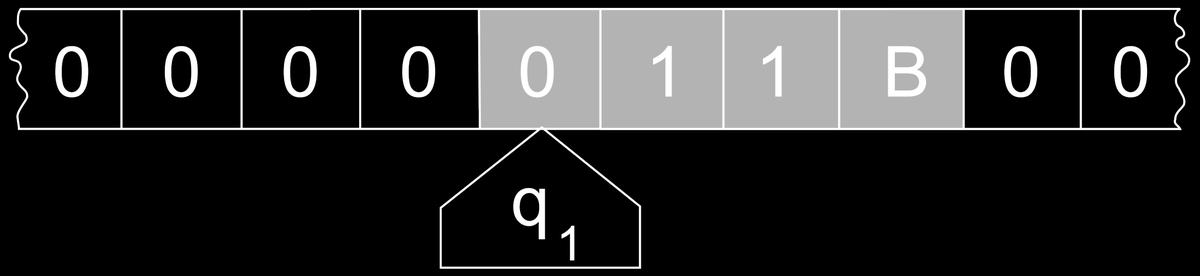
–John Von Neumann

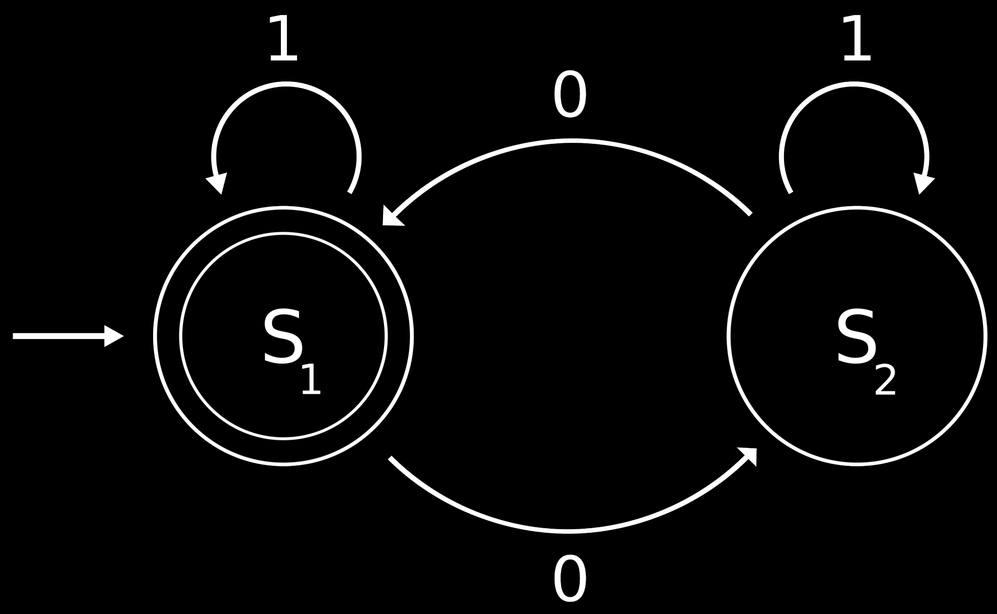
Continuity: bounded differences





Computer science





The world is discrete

Point Cloud Voxelization
(Tommy Hinks)

Physicists

Hamiltonian mechanics

$$\frac{d\mathbf{p}}{dt} = -\frac{\partial \mathcal{H}}{\partial \mathbf{q}}, \frac{d\mathbf{q}}{dt} = -\frac{\partial \mathcal{H}}{\partial \mathbf{p}}$$

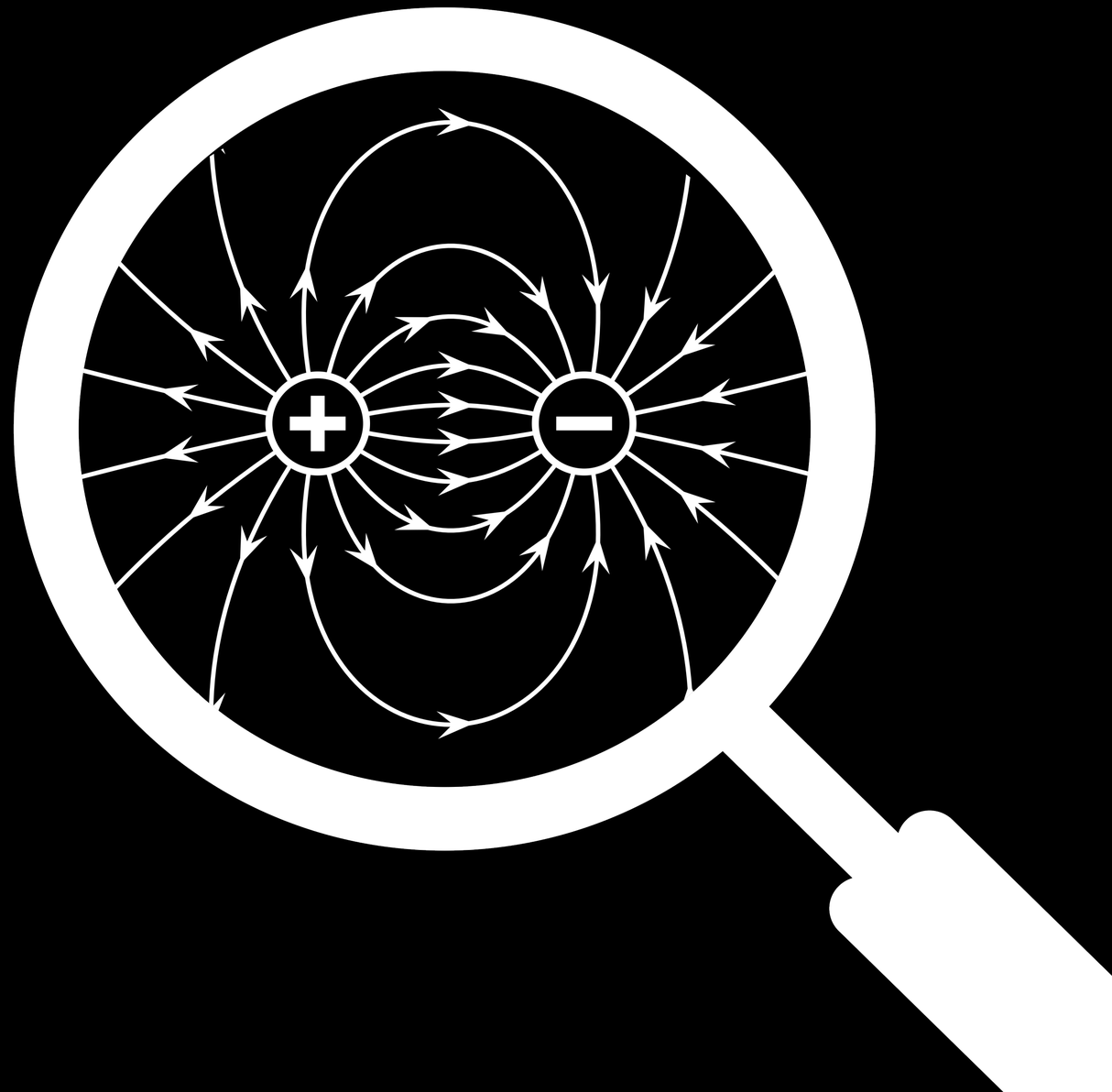
Wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right)$$

Pendulum with friction

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} + 2r\omega \frac{\mathrm{d}x}{\mathrm{d}t} + \omega^2 x = 0$$

World is differentiable

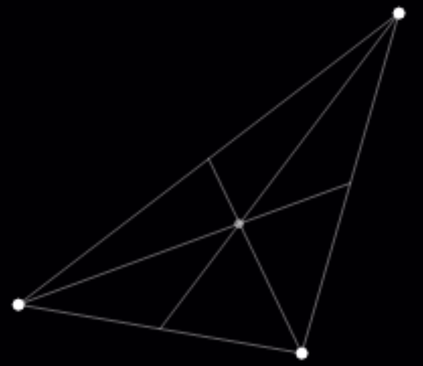


Nonlinear dynamics: no closed form solution

Double Pendulum

https://en.wikipedia.org/wiki/Double_pendulum

Three Body Orbit

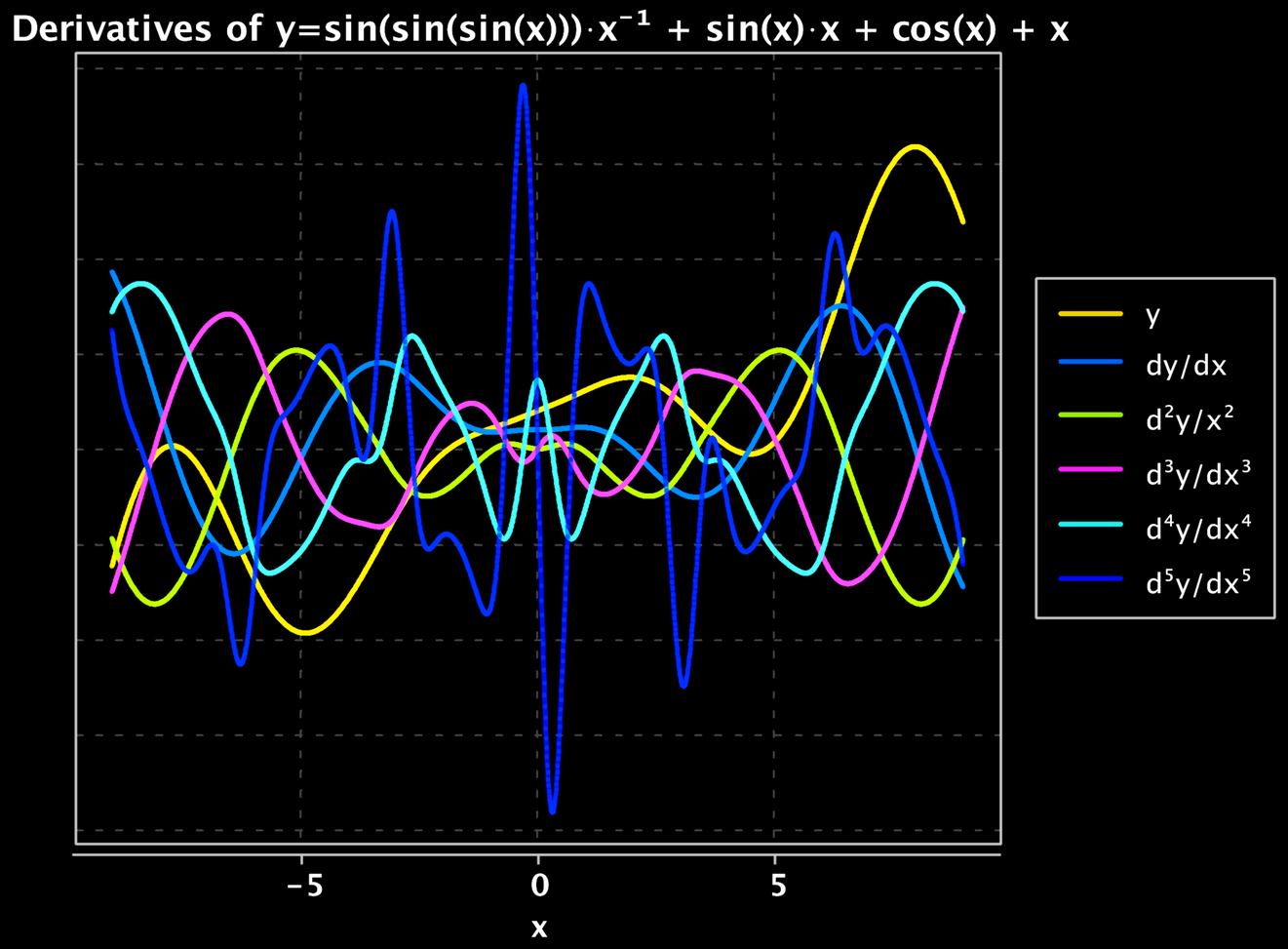


https://en.wikipedia.org/wiki/Three-body_problem

How do I derive in Kotlin?

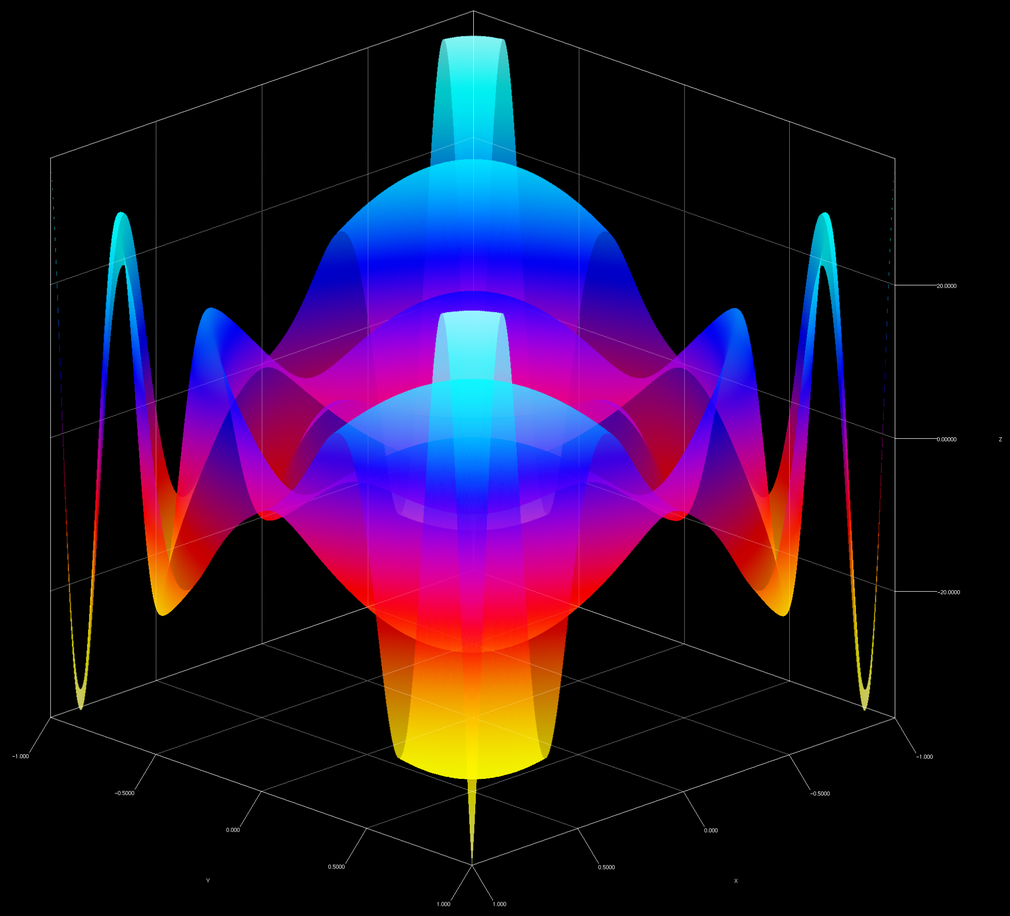
Arbitrarily high order differentiation

```
with(DoublePrecision) {
  val y = sin(sin(sin(x))) / x + sin(x) * x + cos(x) + x
  val `dy/dx` = d(y) / d(x)
  val `d²y/dx²` = d(`dy/dx`) / d(x)
  val `d³y/dx³` = d(`d²y/dx²`) / d(x)
  val `d⁴y/dx⁴` = d(`d³y/dx³`) / d(x)
  val `d⁵y/dx⁵` = d(`d⁴y/dx⁴`) / d(x)

  val xs = (-9.0..9.0 step 0.01).toList()
  val ys = xs.run {
    arrayOf(
      map { y(it) },
      map { `dy/dx`(it) },
      map { `d²y/dx²`(it) },
      map { `d³y/dx³`(it) },
      map { `d⁴y/dx⁴`(it) },
      map { `d⁵y/dx⁵`(it) }
    )
  }.map { it.toDoubleArray() }
```



Arbitrarily high order differentiation

```
with(DoublePrecision) {
  val f = sin(10 * (x * x + pow(y, 2))) / 10
  val z = d(f) / d(x)
  val n = d(d(z) / d(y)) /d(x)

  n(xc, yc) ^with
}
```



Type safe vector manipulation

```
val vf1 = Vec(y + x, y * 2)
val vf3 = Vec(1.0, 2.0, 3.0)
vf1 ⁎ vf3
val bh = x * vf1 + Vec(1.0, 3.0)
bh(y to 2.0, x to 4.0)
val vf2 = Vec(x, y)
val q = vf1 + vf2 + Vec(0.0, 0.0)
val z = q(x to 1.0).magnitude()(y to 2.0)

val vfd = vf2 ⊙ Vec(x, x)
val mf1 = vf3.d(x, y)
```

Type safe matrix manipulation

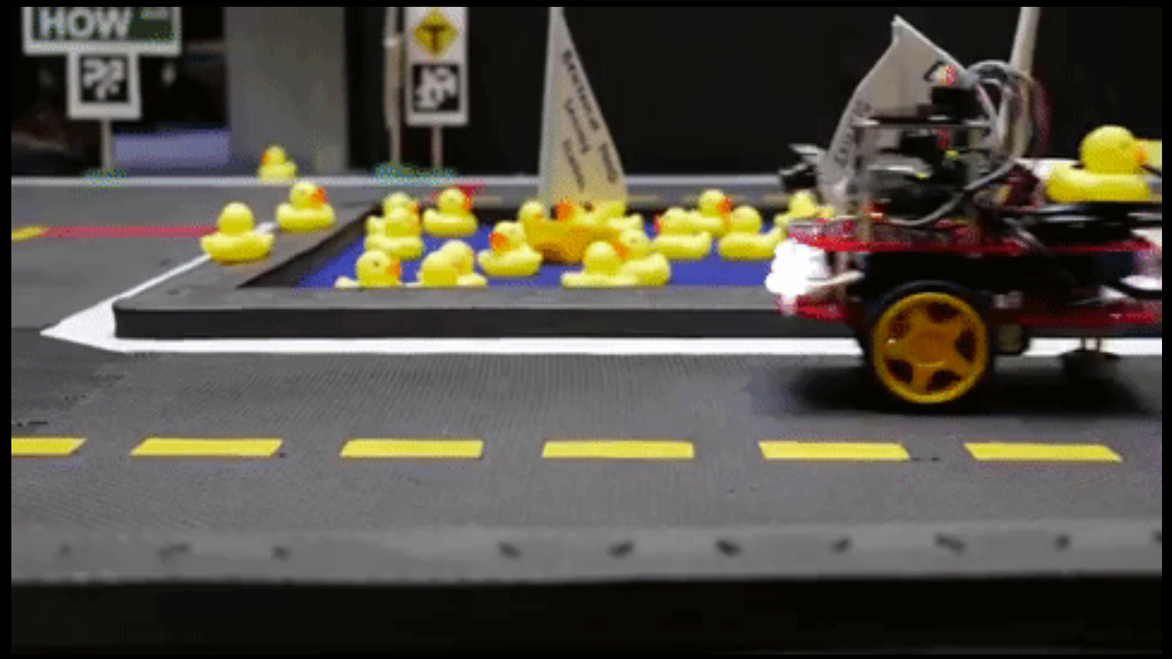
```
val mf1 = Mat2x1(y * y, x * y)
val mf2 = Mat1x2(vf2)
val qr = mf2 * Vec(x, y)
val mf3 = Mat3x2(x, x, y, x, x, x)
val mf4 = Mat2x2(vf2, vf2)
val mf5 = Mat2x2(y * y, x * x, x * y, y * y)
val mf6 = mf4 * mf5 * mf1
println(mf1 * mf2) // 2*1 x 1*2
println(mf1 ≈* vf1) // 2*1 x 2
println(mf2 * vf1) // 1*2 x 2
println(mf3 * vf1) // 3*2 x 2
println(mf3 ≈* mf3) // 3*2 x 3*2
```

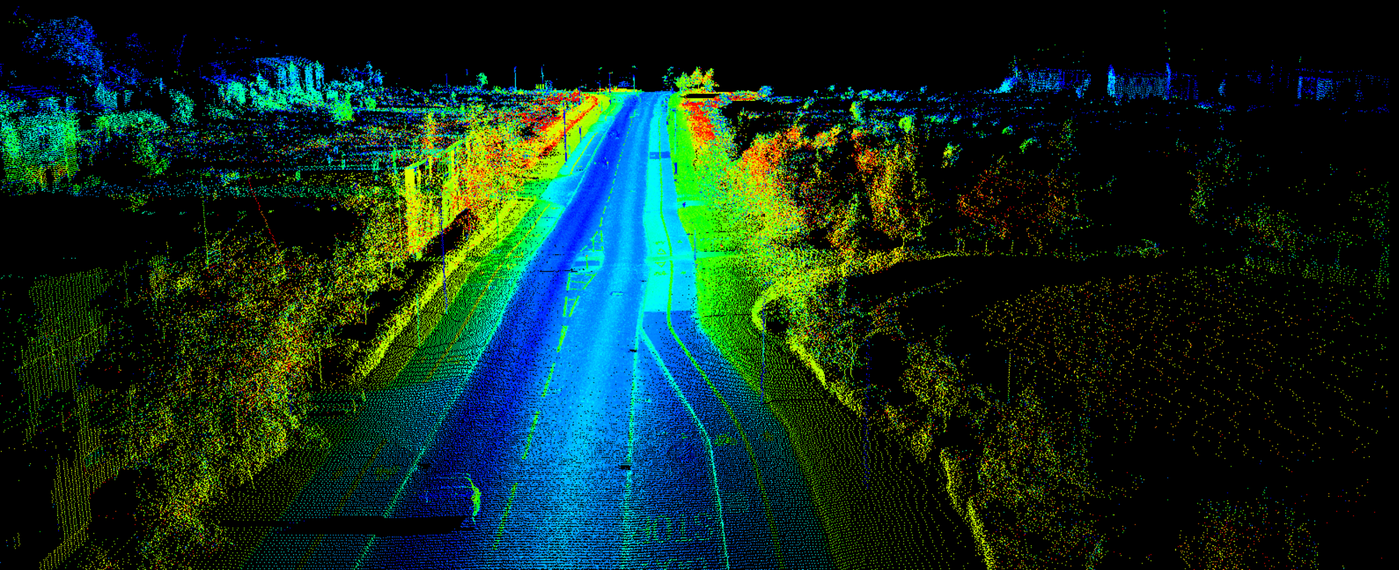
Type safe currying (experimental)

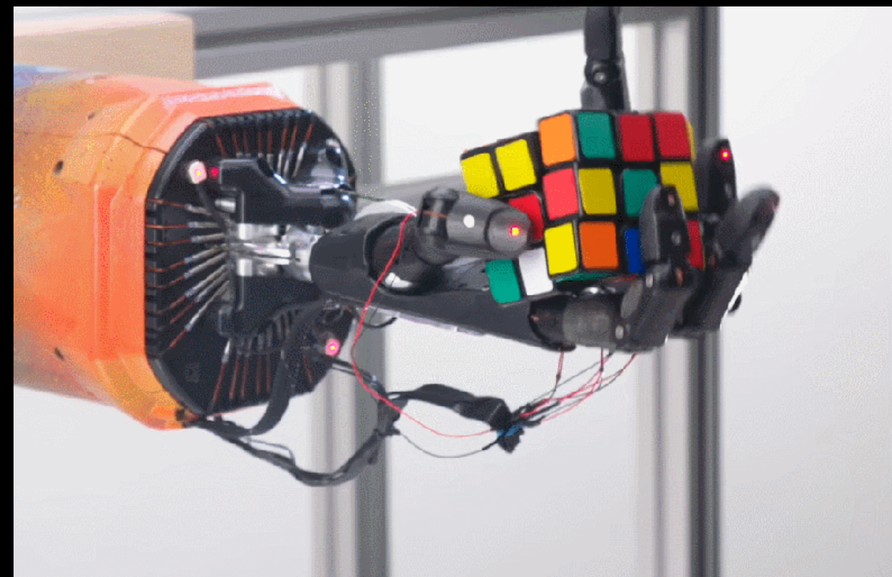
```
val q = X + Y + Z + Y + 0.0
val totalApp = q(X to 1.0, Y to 2.0, Z to 3.0)
val partialApp = q(X to 1.0, Y to 1.0)(Z to 1.0)
val partialApp2 = q(X to 1.0)(Y to 1.0, Z to 1.0)
val partialApp3 = q(Z to 1.0)(X to 1.0, Y to 1.0)

val t = X + Z / Z + Y + 0.0
val v = t(Y to 4.0)
val l = t(X to 1.0)(Z to 2.0)
val r = t(X to 1.0)(Z to 2.0)(Y to 3.0)

val o = X + Z + 0.0
val k = o(Y to 4.0) // Does not compile
val s = (o(X to 1.0) + Y)(Z to 4.0)(Y to 3.0)
```

What's the difference?

Code is the interface





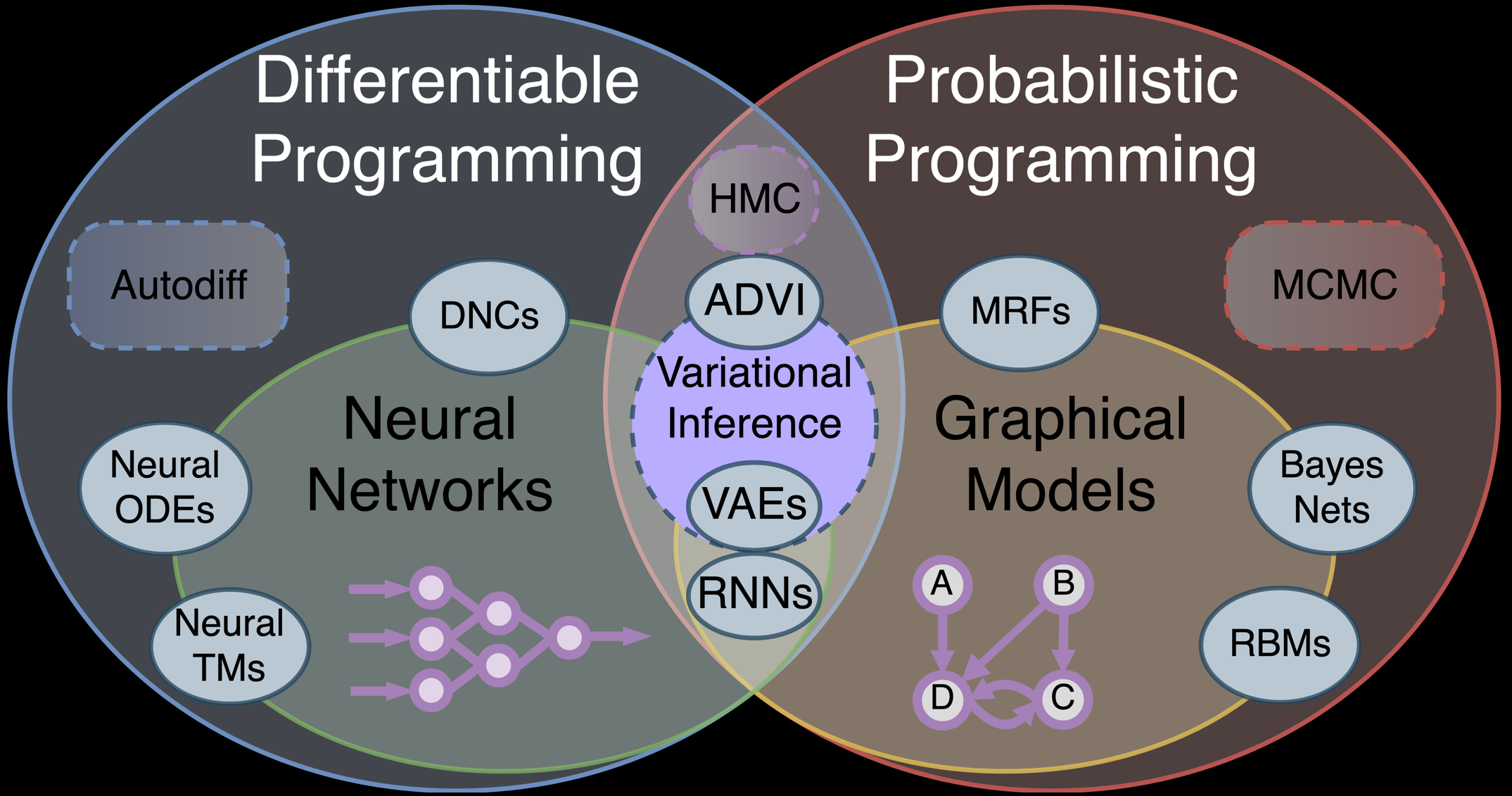


Differentiable Programming

“Neural networks are not just another classifier, they represent the beginning of a fundamental shift in how we write software.”

-Andrej Karpathy

Why do derivatives matter?



Learn more at:

Kotlin∇

kg.ndan.co

Interested in mathematical abstractions?

KMath

github.com/mipt-npm/kmath

Special Thanks

Liam Paull
Michalis Famelis
Jin Guo

THANK YOU
AND REMEMBER TO VOTE

@breandan

#KotlinConf