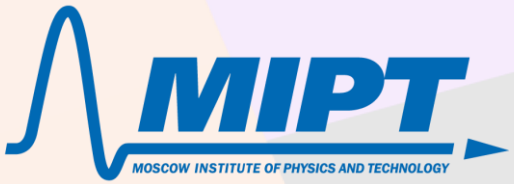
Joker<?>
2021

Math architecture in Kotlin

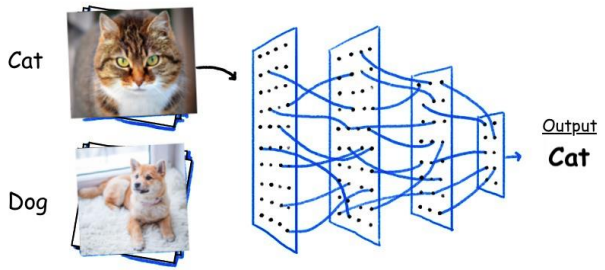
Alexander Nozik, @noraltavir

JET BRAINS Research



Why do we need Math?

- Separate cats from dogs
- Measure the mass of neutrino
- Launch a satellite
- Distribute electrical power



https://laptrinhx.com/10-papers-you-should-read-to-understand-image-classification-in-the-deep-learning-era-787400114/

$$Spectrum(U) = N \cdot \int S(E) \otimes Tr(E) \bullet R(U, E) dE + bkg$$

“Two language problem”

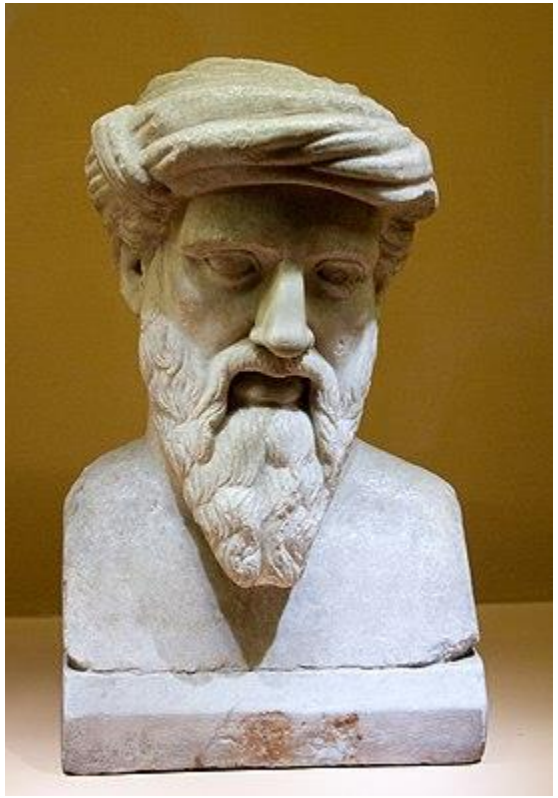
One is brilliant in math,
but lacks proper tools

https://mathstat.slu.edu/escher/index.php/History_and_Numbers

One could be super fast,
but lack proper language

https://www.eurekalert.org/multimedia/597884

What is a number?



A hard problem

$$2 + 2 = ?$$

$$2 + 2.0 = ?$$

$$2 + 1.9999999999999 = ?$$

IEEE 754

Even harder problem

$$2 + \binom{2}{2} = ?$$

$$2 + \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} = ?$$

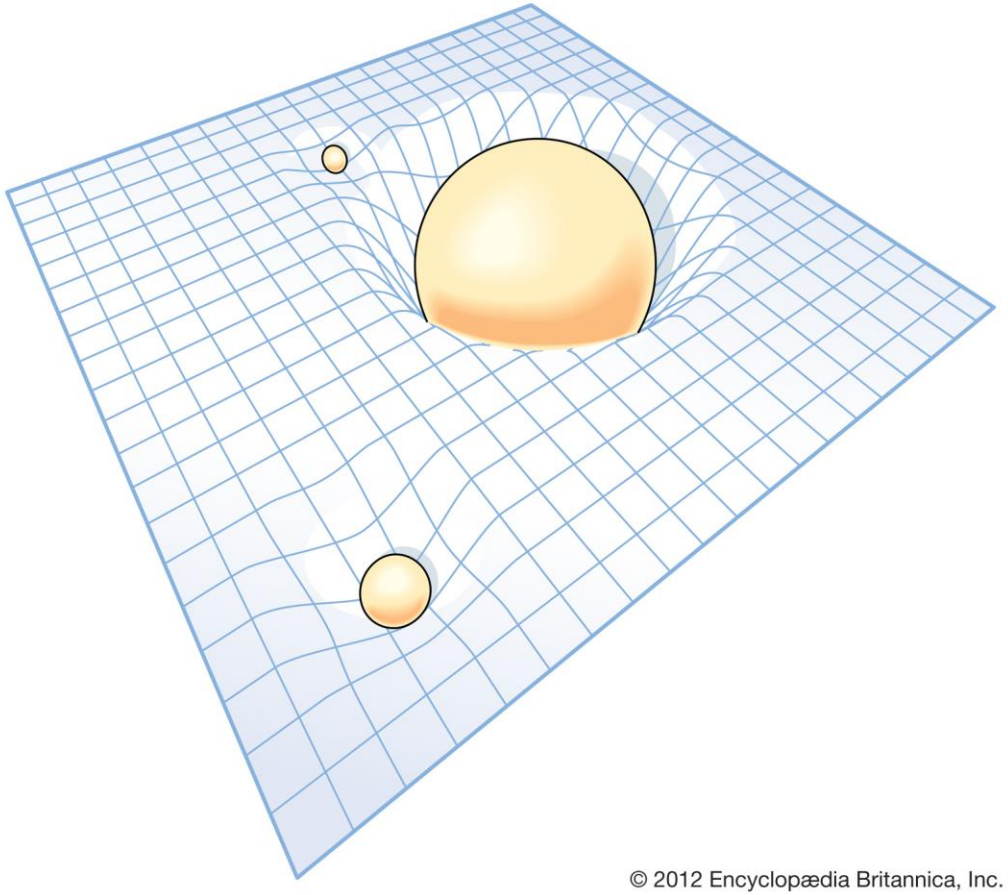
$$2 + \{x \to x^2 + 2x\} = ?$$

$$\begin{pmatrix} 0.23 & 0.5 \end{pmatrix} = ?$$

$$\sin\begin{pmatrix} 0.5 & 0 \end{pmatrix}$$

Relatively simple real-life example

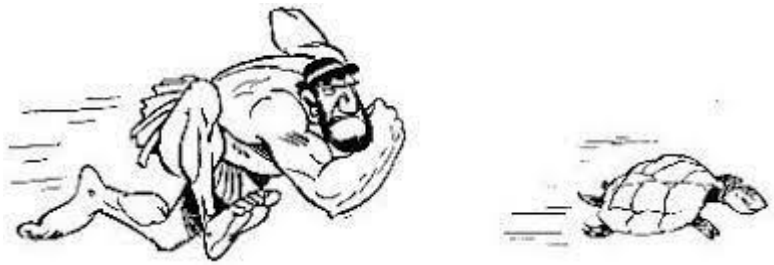
- General relativity states that the space has a curvature (at least local).
- In curved space vector $+$ works the same way as in Euclidian space, but vector $*$ (dot product) works differently.
- Sum of triangle angles is not π



https://www.britannica.com/science/relativity/Curved-space-time-and-geometric-gravitation

What we want from numbers

- Intuitive operations on numbers
- Propagation of operations to structures (vectors, matrices, functions, etc).
- Consistency (can’t perform operations on incompatible numbers).



A problem of performance

Working with numeric structures

Typical newbie solution

List is not effective for primitives

```
val list = ArrayList<Int>()
for (i in 0..100) {
    list.add(i)
}
val newList = list.map { it + 1 }
```

Adding elements one by one causes multiple array copies

Unboxing and boxing of primitives

Specialized array with pre-defined size

```
val a: IntArray = IntArray(100) { it }
val b: List<Int> = a.map { it + 1 } // don't do that
```

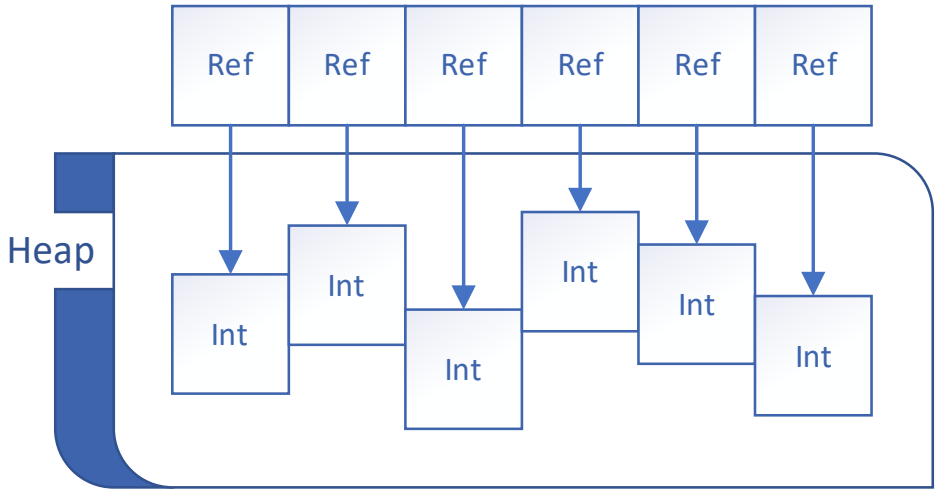
Back to list with boxing

```
val c: IntArray = IntArray(a.size) { a[it] + 1 }
```

Do that

A problem of a box

Boxed collections



Specialized arrays

- No references
- Memory must be aligned (all values specialized and of the same type)
- Processor cache likes all values in a continuous blocks.

Specialization practices

- Static polymorphic specialization. Types known in compile time. Specialized versions of containers are used
- Dynamic polymorphic specialization. Types are inferred in runtime. Specialized containers are used for known types (does not work for unknown types).

Approaches in different languages

	Static specialization	Dynamic specialization	Operations override	Broadcasting
Python	No	No	Bound to objects	No
Python + Numpy	No	Yes	Bound to objects	Automatic, flexible
C++	Yes	No (maybe with libraries)	Bound to types	No
Julia	JIT code generation		Bound to types	Semi-automatic, flexible (`.` modifier)
Java	No	Yes (with libraries)	No	No
Kotlin	Yes (inline)	Yes (with libraries)	Bound to types + scoped	No

Python style

```
import numpy

a = numpy.array([1.0, 2.0, 3.0])
b = numpy.array([1.99, 2.0, 3.0])

print(a - b)
```

```
[-0.99, 0., 0. ]
```

```
import numpy

a = numpy.array([1.0, 2.0, 3.0])
b = numpy.array([1.99, 2.0, 3.0], dtype=int)

print(a - b)
```

```
[0., 0., 0. ]
```

```
def subtract(x1, x2, *args, **kwargs): # real signature unknown; NOTE:
                                       unreliably restored from __doc__
```

- Operations are implemented in C (some algorithms in SciPy in Fortran).
- Python dynamically infers the C-type.
- And uses specialized version.

- Numpy is the core of python mathematics (all other libraries are built on top of it).

Multik

https://github.com/Kotlin/multik

Play in DataLore!

```
val b = mk.ndarray(mk[mk[1.5, 2.1, 3.0], mk[4.0, 5.0, 6.0]])
val d = mk.ndarray(doubleArrayOf(1.0, 1.3, 3.0, 4.0, 9.5, 5.0), 2, 3)
b-d
>[[0.5, 0.8, 0.0],
>[0.0, -4.5, 1.0]]

b.dtype
>DoubleDataType

b.exp()
>[[4.4816890703380645, 8.166169912567652, 20.085536923187668],
>[54.59815003314424, 148.4131591025766, 403.4287934927351]]
```

```
@JvmName("expTD")
public fun <T : Number, D : Dimension> MultiArray<T, D>.exp(): NDArray<Double, D> = mk.math.exp(this)
```

Multik inside

```
public interface MathEx {
    public fun <T : Number, D : Dimension> exp(a: MultiArray<T, D>): NDArray<Double, D>
    public fun <D : Dimension> expF(a: MultiArray<Float, D>): NDArray<Float, D>
    public fun <D : Dimension> expCF(a: MultiArray<ComplexFloat, D>): NDArray<ComplexFloat, D>
    public fun <D : Dimension> expCD(a: MultiArray<ComplexDouble, D>): NDArray<ComplexDouble, D>

    public fun <T : Number, D : Dimension> log(a: MultiArray<T, D>): NDArray<Double, D>
    public fun <D : Dimension> logF(a: MultiArray<Float, D>): NDArray<Float, D>
    public fun <D : Dimension> logCF(a: MultiArray<ComplexFloat, D>): NDArray<ComplexFloat, D>
    public fun <D : Dimension> logCD(a: MultiArray<ComplexDouble, D>): NDArray<ComplexDouble, D>

    public fun <T : Number, D : Dimension> sin(a: MultiArray<T, D>): NDArray<Double, D>
    public fun <D : Dimension> sinF(a: MultiArray<Float, D>): NDArray<Float, D>
    public fun <D : Dimension> sinCF(a: MultiArray<ComplexFloat, D>): NDArray<ComplexFloat, D>
    public fun <D : Dimension> sinCD(a: MultiArray<ComplexDouble, D>): NDArray<ComplexDouble, D>

    public fun <T : Number, D : Dimension> cos(a: MultiArray<T, D>): NDArray<Double, D>
    public fun <D : Dimension> cosF(a: MultiArray<Float, D>): NDArray<Float, D>
    public fun <D : Dimension> cosCF(a: MultiArray<ComplexFloat, D>): NDArray<ComplexFloat, D>
    public fun <D : Dimension> cosCD(a: MultiArray<ComplexDouble, D>): NDArray<ComplexDouble, D>
}
```

- Works only on Multik MultiArray.
- Has multiple backends (JVM only).
- Limited to basic types.
- Could not use existing libraries

JVM mathematics

- Commons-math
  https://commons.apache.org/proper/commons-math/
- EJML
  http://ejml.org/
- ojAlgo
  https://www.ojalgo.org/
- ND4J
  https://github.com/deeplearning4j/nd4j
- And many others

Kotlin-fist

- Koma
  https://github.com/kyonifer/koma
- KMath
- JB Multik
  https://github.com/Kotlin/multik

Into the algebra

- General mathematics branch
- A set of operations on arbitrary objects

- Algebra is an interface for the operations on data
- There could be multiple different algebras on the same data

- Algebra is an abstraction that we need

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

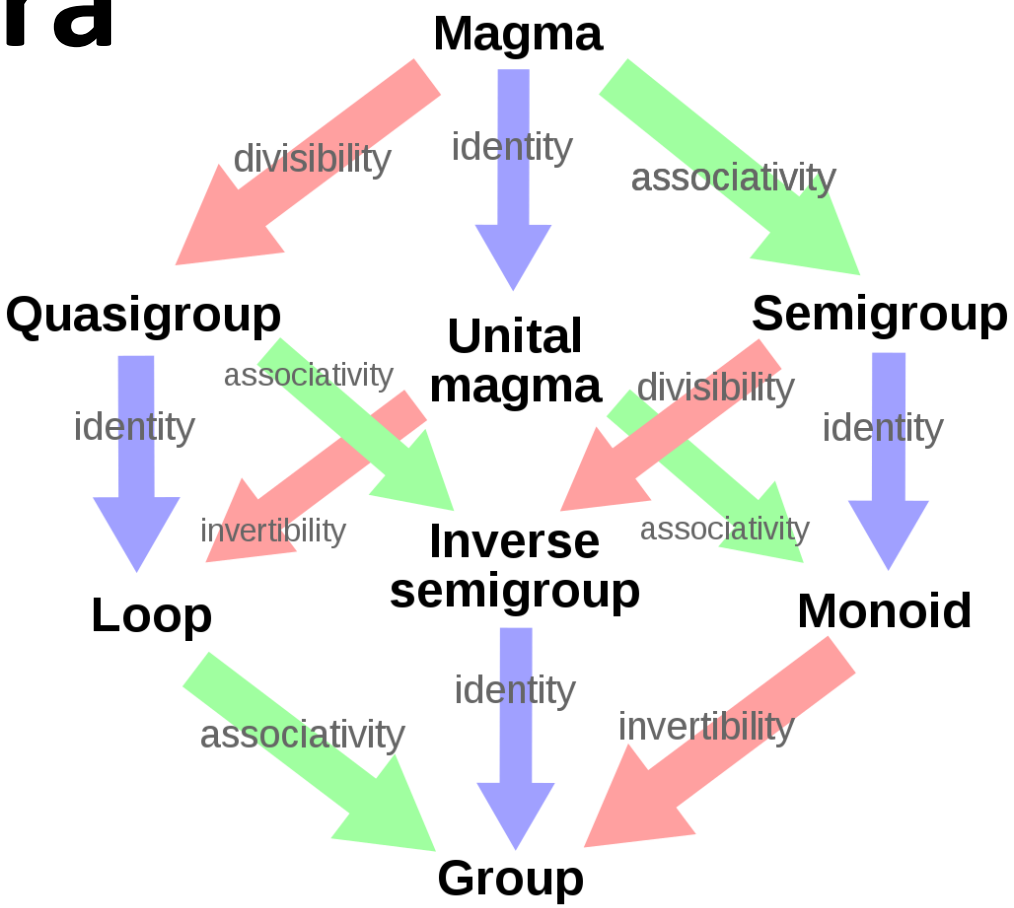
A little bit about algebra

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

Group (+, -, 0)
Ring (+, -, *, 0, 1)
Field (+, -, *, /, 0, 1)

Used in Apache Commons Math / Commons Numbers

```
interface Field // 0, 1

Interface FieldElement // field, +, -, *, /, %
```



https://en.wikipedia.org/wiki/Monoid#/media/File:Magma_to_group4.svg

Commons math /numbers generic approach

[3]
```
@file:DependsOn("org.apache.commons:commons-math3:3.6.1")
```

[4]
```
import org.apache.commons.math3.util.Decimal64
import org.apache.commons.math3.util.Decimal64Field

val field = Decimal64Field.getInstance()
```

[8]
```
val function: (Decimal64) -> Decimal64 = { x->
  x.pow(2).multiply(2.0).add(x.multiply(2.0)).add(field.one.multiply(6.0))
}
```

[9]
```
▷ 0.5s
function(Decimal64(1.0))
```

```
10.0
```

Play in DataLore!

KMath algebra

```
public interface GroupOperations<T> : Algebra<T> {

    public fun add(a: T, b: T): T

    // Operations to be performed in this context. Could be moved to extensions in case of multi-receivers.
    public operator fun T.unaryMinus(): T
    public operator fun T.unaryPlus(): T = this
    public operator fun T.plus(b: T): T = add(this, b)
    public operator fun T.minus(b: T): T = add(this, -b)
    // Dynamic dispatch of operations
    override fun unaryOperationFunction(operation: String): (arg: T) -> T
    override fun binaryOperationFunction(operation: String): (left: T, right: T) -> T
}

public interface Group<T> : GroupOperations<T> {
    public val zero: T
}

public interface RingOperations<T> : GroupOperations<T> {
    public fun multiply(a: T, b: T): T
    public operator fun T.times(b: T): T = multiply(this, b)
}

public interface Ring<T> : Group<T>, RingOperations<T> {
    public val one: T
}
```

Example: `IntRing`, `LongRing`

Operations could be used in the scope of algebra, but not outside it.

with(multi-recievers)

```
public interface GroupOperations<T> : Algebra<T> {
    public fun add(a: T, b: T): T
}

context(GroupOperations<T>)
fun <T> T.plus(b: T): T = add(this, b)

context(DoubleField)
fun Double.plus(b: Double): Double = this + b

context(ComplexField)
fun Complex.plus(b: Double): Complex = this + b.toComplex()

context(FieldND<T>)
fun <T> StructureND<T>.plus(b: T): StructureND<T> = mapInline { elementAlgebra{ it * b } }
```

Only basic operations in algebra

Operations could be statically dispatched by types

Operations could be inlined.
Inlining could solve generics problem.
Extensions are dispatched statically (in compile time)

Algebra propagation

```
public class BufferField<T, A : Field<T>>(
    override val bufferFactory: BufferFactory<T>,
    override val elementAlgebra: A,
    override val size: Int
) : BufferAlgebra<T, A>, Field<Buffer<T>> {

    override val zero: Buffer<T> = bufferFactory(size) { elementAlgebra.zero }
    override val one: Buffer<T> = bufferFactory(size) { elementAlgebra.one }

    override fun add(a: Buffer<T>, b: Buffer<T>): Buffer<T> = a.zip(b, elementAlgebra::add)
    override fun multiply(a: Buffer<T>, b: Buffer<T>): Buffer<T> = a.zip(b, elementAlgebra::multiply)
    override fun divide(a: Buffer<T>, b: Buffer<T>): Buffer<T> = a.zip(b, elementAlgebra::divide)

    override fun scale(a: Buffer<T>, value: Double): Buffer<T> = with(elementAlgebra) { a.map { scale(it, value) } }
    override fun Buffer<T>.unaryMinus(): Buffer<T> = with(elementAlgebra) { map { -it } }
}
```

In-place specialization

```
override fun StructureND<Double>.toBufferND(): DoubleBufferND = when (this) {
    is DoubleBufferND -> this
    else -> {
        val indexer = indexerBuilder(shape)
        DoubleBufferND(indexer, DoubleBuffer(indexer.linearSize) { offset -> get(indexer.index(offset)) })
    }
}
```

Check incoming type and convert it to the target algebra type if necessary.
The conversion price is paid only once per algebra scope.

```
override fun StructureND<Double>.plus(arg: Double): DoubleBufferND = mapInline(toBufferND()) { it + arg }


fun main() {
    val viktorStructure: ViktorStructureND = DoubleField.viktorAlgebra.produce(Shape(2, 2)) { (i, j) ->
        if (i == j) 2.0 else 0.0
    }
    val cmMatrix: Structure2D<Double> = CMLinearSpace.matrix(2, 2)(0.0, 1.0, 0.0, 3.0)
    val res: DoubleBufferND = DoubleField.ndAlgebra {
        exp(viktorStructure) + 2.0 * cmMatrix
    }
    println(res)
}
```

No need to watch for specific types.
Conversion is automatic.

Complex example

```
public data class Complex(val re: Double, val im: Double)


public object ComplexField : ExtendedField<Complex>, Norm<Complex, Complex>, NumbersAddOperations<Complex>, ScaleOperations<Complex>

fun main() = Complex.algebra {
    val complex = 2 + 2 * i
    println(complex * 8 - 5 * i)

    //flat buffer
    val buffer = with(bufferAlgebra){
        buffer(8) { Complex(it, -it) }.map { Complex(it.im, it.re) }
    }
    println(buffer)

    // 2d element
    val element: BufferND<Complex> = ndAlgebra.produce(2, 2) { (i, j) ->
        Complex(i - j, i + j)
    }
    println(element)

    // 1d element operation
    val result: StructureND<Complex> = ndAlgebra{
        val a = produce(8) { (it) -> i * it - it.toDouble() }
        val b = 3
        val c = Complex(1.0, 1.0)

        (a pow b) + c
    }
    println(result)
}
```

Operations are available only in this scope

Use N-dimensional field over complex field

Perform operations on 1d ND buffers

Core algebras

- BufferAlgebra. Arithmetic operations on linear structures with the fixed size.
- AlgebraND. Arithmetic operations on immutable ND structures with fixed size.
- LinearAlgebra. Linear algebra operations on matrices and vectors without size constraints.

- TensorAlgebra. For those from TensorFlow/PyTorch.
- TensorAlgebra/AlgebraND implementation for Multik.

Play in DataLore!

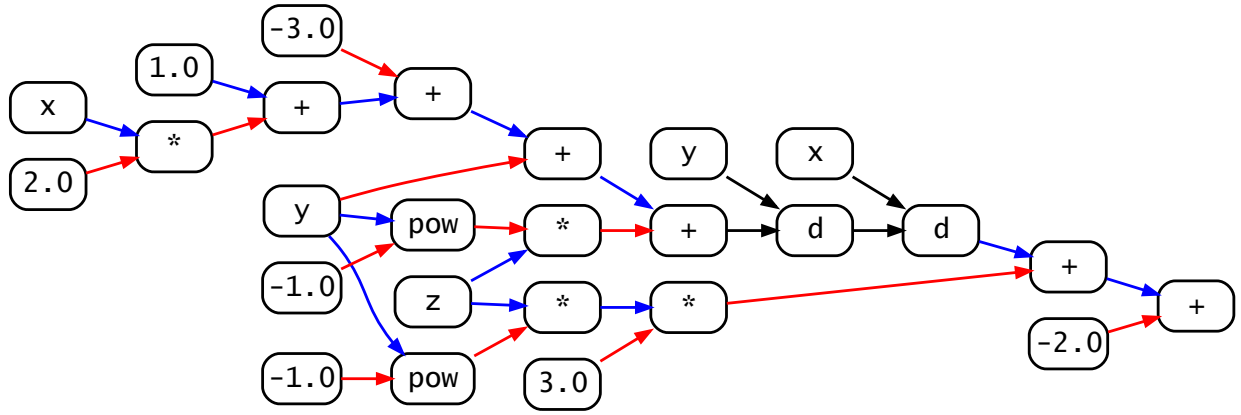
Expression algebra

```
public fun interface Expression<T> {
    public operator fun invoke(arguments: Map<Symbol, T>): T
}

private val functional = DoubleField.expression {
    val x = bindSymbol(Symobl.x)
    x * number(2.0) + 2.0 / x - 16.0 / sin(x)
}

expression(x to 1.0)
```

Need to bind symbol to this context.
Its value will be actualized when the expression is called

Symbolic expressions

MST



$$((1 + x * 2 - 3 + y + z / y).d(y).d(x) + z / y * 3 - 2)$$

https://github.com/breandan/kotlingrad

If we can operate on abstract number-like objects, why can't we operate on symbols?

[1]:
```
@file:Repository("https://repo.kotlin.link")
@file:DependsOn("space.kscience:kmath-jupyter:0.3.0-dev-15")
```

[3]:
```
val expr = "exp(sqrt(x))-asin(2*x)/(2e10+x^3)/(-12)".parseMath()
expr
```

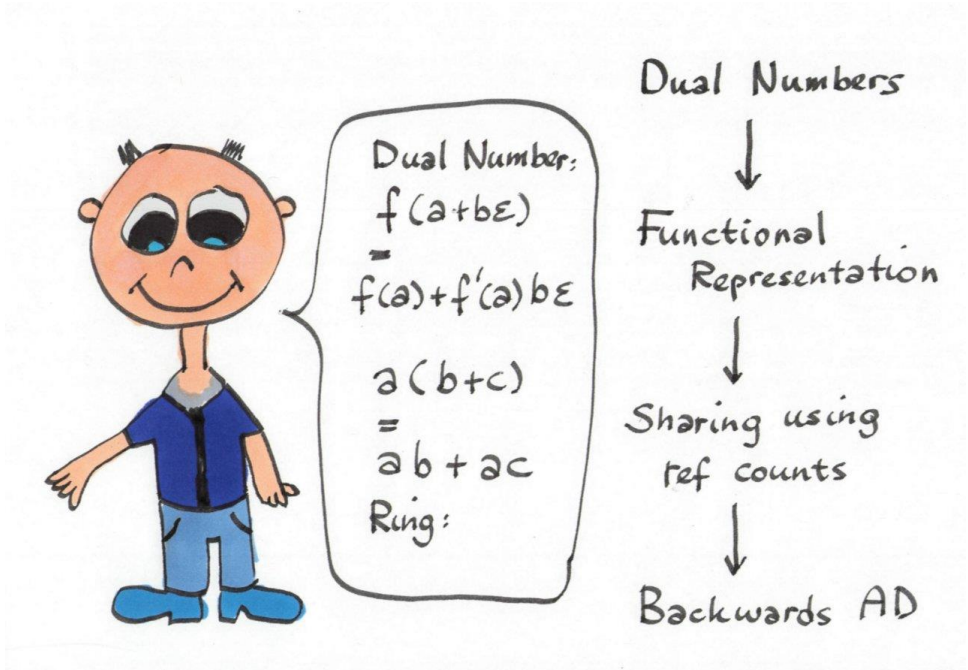
[3]:
$$\exp\left(\sqrt{x}\right)-\frac{\frac{\arcsin\left(2\ x\right)}{2\times10^{10}+x^3}}{-12}$$

[4]:
```
println(expr)
```

```
Binary(operation=-, left=Unary(operation=exp, value=Unary(operation=sqrt, value=x)), right=Binary(oper
ation=/, left=Binary(operation=/, left=Unary(operation=asin, value=Binary(operation=*, left=Numeric(va
lue=2.0), right=x)), right=Binary(operation=+, left=Numeric(value=2.0E10), right=Binary(operation=pow,
left=x, right=Numeric(value=3.0)))), right=Unary(operation=-, value=Numeric(value=12.0))))
```

Why do we need syntax tree?

- Compilation to different targets
  - LaTeX / MathML / other visualization
  - Native calls (Torch / TensorFlow)
  - Remote call / clustering
  - GPU
- Symbolic simplification
- Automatic / symbolic / hybrid differentiation



https://twitter.com/headinthebox/status/1059539103576981509?s=20

Yet another example

```
fun main(): Unit = Double.algebra {
    withNdAlgebra(2, 2) {

        //Produce a diagonal StructureND
        fun diagonal(v: Double) = produce { (i, j) ->
            if (i == j) v else 0.0
        }

        //Define a function in a nd space
        val function: (Double) -> StructureND<Double> = { x: Double -> 3 * x.pow(2) + 2 * diagonal(x) + 1 }

        //get the result of the integration
        val result = gaussIntegrator.integrate(0.0..10.0, function = function)

        //the value is nullable because in some cases the integration could not succeed
        println(result.value)
    }
}
```

Entering double algebra

Entering 2x2 ND algebra

Local function to create diagonal ND structure

A matrix-function

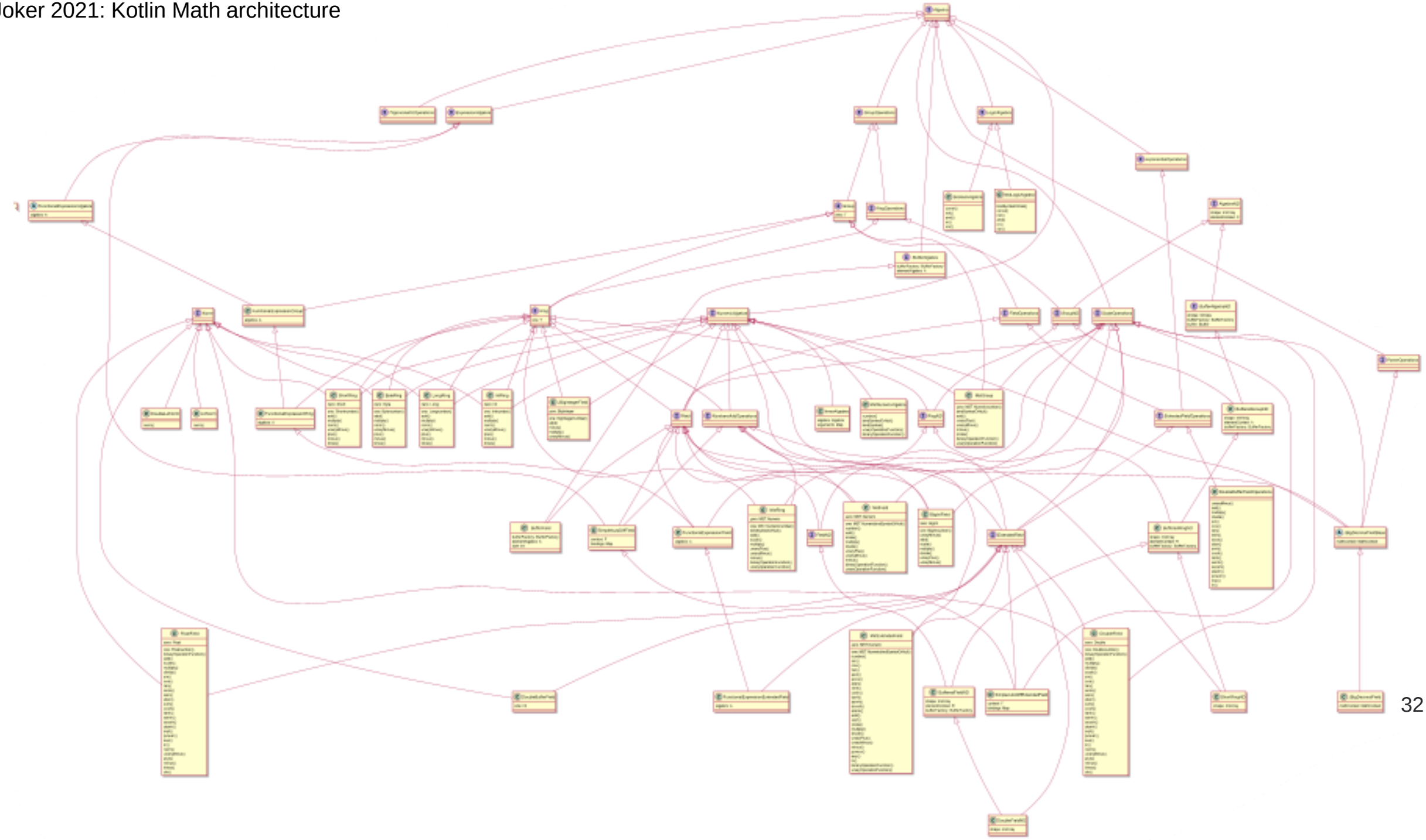
Integrating matrix function!

Everyone has their own algebra

- Universal lightweight math (including bigint and complex/quaternions): KMath-core
- Performant matrix operations on JVM: KMath-EJML
- Effective ND operations: KMath-ND4J
- Natively optimized SIMD operations: KMath-Viktor
- Bridge to optimized native libraries: KMath-GSL (Iaroslav Postovalov): https://github.com/mipt-npm/kmath-gsl

- One can use multiple connectors
- The same architectural patter could be used anywhere, where multiple implementations are involved.

Joker 2021: Kotlin Math architecture



Performance requirements for maths

- No boxing!
- No fragmentation in memory layout!
- SIMD!

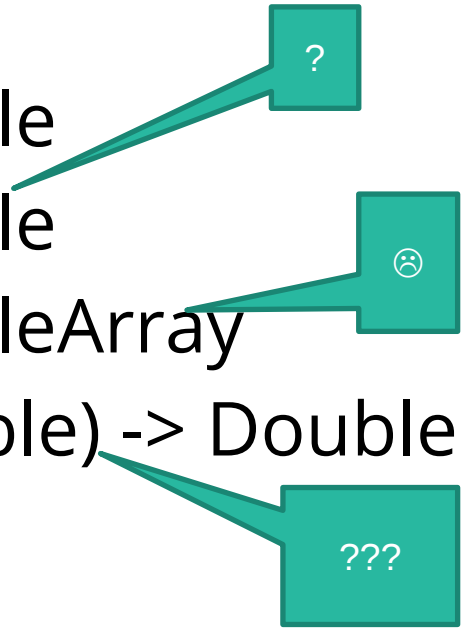
JVM to Kotlin translation

Java

- Double
- double
- double[]
- interface UnivariateFunction{
  double evaluate(double x)
  }

Kotlin

- Double
- Double
- DoubleArray
- (Double) -> Double



When you concerned about performance – Benchmark!

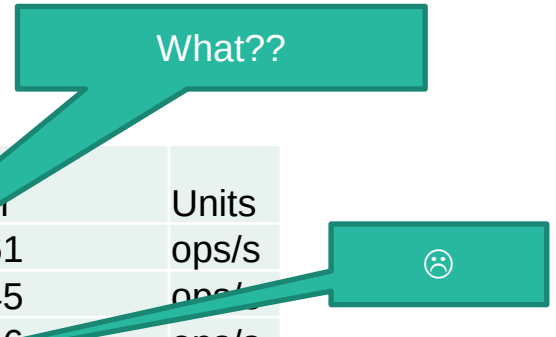
https://github.com/Kotlin/kotlinx-benchmark

```
@State(Scope.Benchmark)
internal class DotBenchmark {
    companion object {
        val random = Random(12224)
        const val dim = 1000
        //creating invertible matrix
        val matrix1 = Double.algebra.linearSpace.buildMatrix(dim, dim) { i, j ->  if (i <= j) random.nextDouble() else 0.0 }
        val matrix2 = Double.algebra.linearSpace.buildMatrix(dim, dim) { i, j ->  if (i <= j) random.nextDouble() else 0.0 }
        val cmMatrix1 = CMLinearSpace { matrix1.toCM() }
        val cmMatrix2 = CMLinearSpace { matrix2.toCM() }
        val ejmlMatrix1 = EjmlLinearSpaceDDRM { matrix1.toEjml() }
        val ejmlMatrix2 = EjmlLinearSpaceDDRM { matrix2.toEjml() }
    }

    @Benchmark fun cmDotWithConversion(blackhole: Blackhole) = CMLinearSpace { blackhole.consume(matrix1 dot matrix2) }

    @Benchmark fun cmDot(blackhole: Blackhole) = CMLinearSpace { blackhole.consume(cmMatrix1 dot cmMatrix2) }

    @Benchmark fun ejmlDotWithConversion(blackhole: Blackhole) = EjmlLinearSpaceDDRM { blackhole.consume(matrix1 dot matrix2) }

    @Benchmark fun ejmlDot(blackhole: Blackhole) = EjmlLinearSpaceDDRM { blackhole.consume(ejmlMatrix1 dot ejmlMatrix2) }

    @Benchmark fun bufferedDot(blackhole: Blackhole) = with(DoubleField.linearSpace(Buffer.Companion::auto)) { blackhole.consume(matrix1 dot matrix2) }

    @Benchmark fun doubleDot(blackhole: Blackhole) = with(Double.algebra.linearSpace) { blackhole.consume(matrix1 dot matrix2) }
}
```

Automatic conversion to CM format

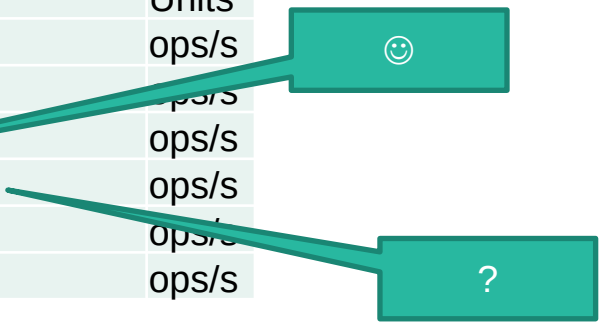
Liberica-16

Benchmark	Mode	Score		Error	Units
DotBenchmark.bufferedDot	thrpt	0.293	±	0.061	ops/s
DotBenchmark.cmDot	thrpt	0.260	±	0.045	ops/s
DotBenchmark.cmDotWithConversion	thrpt	0.706	±	0.616	ops/s
DotBenchmark.doubleDot	thrpt	0.285	±	0.013	ops/s
DotBenchmark.ejmlDot	thrpt	2.889	±	0.089	ops/s
DotBenchmark.ejmlDotWithConversion	thrpt	2.470	±	0.109	ops/s



GraalVM-11

Benchmark	Mode	Score		Error	Units
DotBenchmark.bufferedDot	thrpt	1.187	±	0.033	ops/s
DotBenchmark.cmDot	thrpt	0.499	±	0.114	ops/s
DotBenchmark.cmDotWithConversion	thrpt	0.747	±	0.593	ops/s
DotBenchmark.doubleDot	thrpt	0.948	±	0.712	ops/s
DotBenchmark.ejmlDot	thrpt	2.737	±	0.127	ops/s
DotBenchmark.ejmlDotWithConversion	thrpt	2.436	±	0.134	ops/s

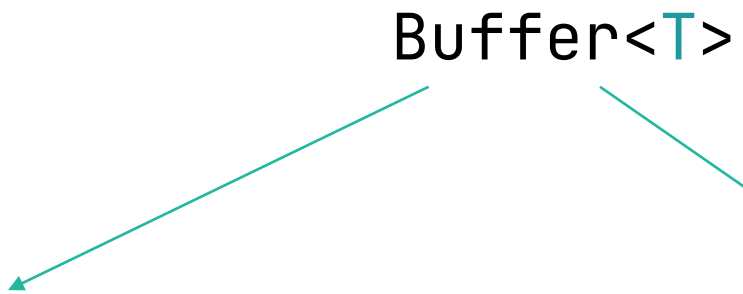


Why is the difference?

Method	
95.4%	space.kscience.kmath.linear.**DotPerformanceKt.main**(String[])
95.4%	space.kscience.kmath.linear.**DotPerformanceKt.main**()
95.0%	space.kscience.kmath.linear.**BufferedLinearSpace.dot**(Structure2D, Structure2D)
92.2%	space.kscience.kmath.linear.**BufferedLinearSpace.buildMatrix**(int, int, Function3)
92.2%	space.kscience.kmath.nd.**BufferAlgebraND.produce**(Function2)
92.2%	space.kscience.kmath.nd.**BufferAlgebraND.produce**(Function2)
92.2%	space.kscience.kmath.linear.**BufferedLinearSpaceKt$linearSpace$1.invoke**(Object, Object)
92.2%	space.kscience.kmath.linear.**BufferedLinearSpaceKt$linearSpace$1.invoke-CZ9oacQ**(int, Function1)
91.0%	space.kscience.kmath.nd.**BufferAlgebraND$produce$1.invoke**(Object)
91.0%	space.kscience.kmath.nd.**BufferAlgebraND$produce$1.invoke**(int)
90.6%	space.kscience.kmath.linear.**BufferedLinearSpace$buildMatrix$1.invoke**(Object, Object)
90.6%	space.kscience.kmath.linear.**BufferedLinearSpace$buildMatrix$1.invoke**(Ring, int[])
90.2%	space.kscience.kmath.linear.**BufferedLinearSpacedot2$1.invoke**(Object, Object, Object)
90.2%	space.kscience.kmath.linear.**BufferedLinearSpacedot2$1.invoke**(Ring, int, int)
26.2%	space.kscience.kmath.operations.**DoubleField.plus**(Object, Object)
26.2%	java.lang.**Double.valueOf**(double)

https://pikabu.ru/story/houston_we_have_a_problem_2666170

Memory layout



Specialized buffer

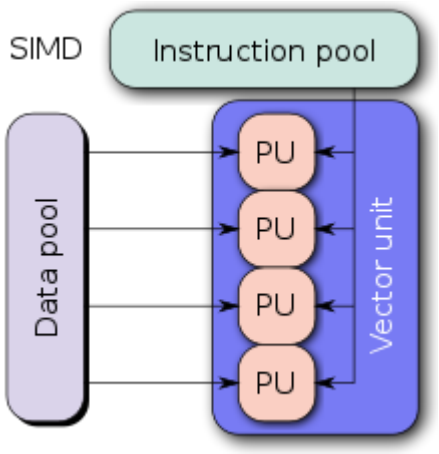
```
@JvmInline
public value class DoubleBuffer(
    public val array: DoubleArray
) : MutableBuffer<Double> {
    override val size: Int get() = array.size

    override operator fun get(index: Int): Double = array[index]

    override operator fun set(index: Int, value: Double) {
        array[index] = value
    }

    override operator fun iterator(): DoubleIterator = array.iterator()

    override fun copy(): DoubleBuffer = DoubleBuffer(array.copyOf())
}
```

Generic buffer

```
public class ListBuffer<T>(public val list: List<T>) : Buffer<T> {

    public constructor(size: Int, initializer: (Int) -> T)
        : this(List(size, initializer))

    override val size: Int get() = list.size

    override operator fun get(index: Int): T = list[index]
    override operator fun iterator(): Iterator<T> = list.iterator()
}
```

Does not work that well if generic buffer is used.
Need to explicitly convert to specialized buffer.

SIMD?



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

```
@Benchmark
fun realFieldAddition(blackhole: Blackhole) = with(realField) {
    var res: StructureND<Double> = one
    repeat(n) { res += 1.0 }
    blackhole.consume(res)
}
```

//github: JetBrains-Research/**viktor**

```
@Benchmark
fun viktorFieldAddition(blackhole: Blackhole) = with(viktorField) {
    var res: StructureND<Double> = one
    repeat(n) { res += 1.0 }
    blackhole.consume(res)
}
```

Benchmark	Mode	Cnt	Score	Error	Units
ViktorBenchmark.automaticFieldAddition	thrpt	5	5.104	± 0.112	ops/s
ViktorBenchmark.rawViktor	thrpt	5	4.797	± 0.410	ops/s
ViktorBenchmark.realFieldAddition	thrpt	5	4.918	± 0.634	ops/s
ViktorBenchmark.viktorFieldAddition	thrpt	5	5.088	± 0.552	ops/s

Modern JVM mostly solves the SIMD problem.

Looking forward to try https://openjdk.java.net/jeps/414 (Vector API)

Solutions to performance problem

Static specialization

- Use inline functions
- Use extensions because they do static type dispatch
- Use wrappers for specialized libraries (EJML, Commons-math, Torch, etc).
- Convert structures on use.

Dynamic specialization

```
@Suppress("UNCHECKED_CAST")
public inline fun <T : Any> auto(
    type: KClass<T>, size: Int, initializer: (Int) -> T
): Buffer<T> = when (type) {
    Double::class -> MutableBuffer.double(size) {
        initializer(it) as Double
    } as Buffer<T>

    Short::class -> MutableBuffer.short(size) {
        initializer(it) as Short
    } as Buffer<T>

    Int::class -> MutableBuffer.int(size) {
        initializer(it) as Int
    } as Buffer<T>

    Long::class -> MutableBuffer.long(size) {
        initializer(it) as Long
    } as Buffer<T>

    Float::class -> MutableBuffer.float(size) {
        initializer(it) as Float
    } as Buffer<T>
    else -> boxing(size, initializer)
}
```

Conclusions

- Kotlin mathematics ecosystem is young but healthy
- It could rival Python for data science
- Julia for computational finance
- C++ for automation

Make new VS use existing not clear

- Make new core mathematics library

OR

- Use existing platform libraries

Context oriented design solves the problem of operations binding

And partially solves performance/specialization problems

- Performance could be made pretty good on JVM.
- But one needs to control and check it constantly.
- Not sure about Android.

val q: Questions?

Post scriptum

Tensor algebra

(by Rolan Grinis and his team)

```
fun main(): Unit = Double.tensorAlgebra.withBroadcast {  // work in context with broadcast methods
    val seed = 100500L
    // assume x is range from 0 until 10
    val x = fromArray( intArrayOf(10), DoubleArray(10) { it.toDouble() }  )
    // take y dependent on x with noise
    val y = 2.0 * x + (3.0 + x.randomNormalLike(seed) * 1.5)
    // stack them into single dataset
    val dataset = stack(listOf(x, y)).transpose()
    // normalize both x and y
    val xMean = x.mean()
    val yMean = y.mean()
    val xStd = x.std()
    val yStd = y.std()
    val xScaled = (x - xMean) / xStd
    val yScaled = (y - yMean) / yStd
    // save means ans standard deviations for further recovery
    val mean = fromArray(
        intArrayOf(2),
        doubleArrayOf(xMean, yMean)
    )
    val std = fromArray(
        intArrayOf(2),
        doubleArrayOf(xStd, yStd)
    )
    // calculate the covariance matrix of scaled x and y
    val covMatrix = cov(listOf(xScaled, yScaled))
    // and find out eigenvector of it
    val (_, evecs) = covMatrix.symEig()
    val v = evecs[0]
    // reduce dimension of dataset
    val datasetReduced = v dot stack(listOf(xScaled, yScaled))
    // we can restore original data from reduced data;
    // for example, find 7th element of dataset.
    val n = 7
    val restored = (datasetReduced[n] dot v.view(intArrayOf(1, 2))) * std + mean
}
```

Differentiation

```
//Create a uniformly distributed x values like numpy.arrange
val x = 1.0..100.0 step 1.0
//Perform an operation on each x value (much more effective, than numpy)
val y = x.map { it ->
    val value = it.pow(2) + it + 1
    value + chain.next() * sqrt(value)
}

// create same errors for all xs
val yErr = y.map { sqrt(it) }

// compute differentiable chi^2 sum for given model ax^2 + bx + c
val chi2 = DSProcessor.chiSquaredExpression(x, y, yErr) { arg ->
    //bind variables to autodiff context
    val a = bindSymbol(a)
    val b = bindSymbol(b)
    //Include default value for c if it is not provided as a parameter
    val c = bindSymbolOrNull(c) ?: one
    a * arg.pow(2) + b * arg + c
}

//minimize the chi^2 in given starting point. Derivatives are not required; they are already included.
val result = chi2.optimizeWith(
    CMOptimizer,
    mapOf(a to 1.5, b to 0.9, c to 1.0),
    FunctionOptimizationTarget.MINIMIZE
)
```

Differentiation is required to do things like optimization. Several ways to differentiate:

- **Analytic**
  Derivatives are computed manually
- **Symbolic**
  Differentiate symbolic expression
- **Automatic**
  Erik Meijer - Inside Every Calculus Is A Little Algebra Waiting To Get Out

Decompilation - special

```
private fun mapSpecial(): Double
{
    val array = DoubleArray(3) {
it.toDouble() }
    return array[2] + 1.0
}
```

```
private final static mapSpecial()D
  L0
   LINENUMBER 12 L0
   ICONST_3
   ISTORE 1
   ILOAD 1
   NEWARRAY T_DOUBLE
   ASTORE 2
   ICONST_0
   ISTORE 3
  L1
   ILOAD 3
   ILOAD 1
   IF_ICMPGE L2
   ALOAD 2
   ILOAD 3
   ILOAD 3
   ISTORE 4
   ISTORE 7
   ASTORE 6
  L3
   ICONST_0
   ISTORE 5
  L4
   LINENUMBER 12 L4
   ILOAD 4
   I2D
  L5
  L6
   DSTORE 8
   ALOAD 6
   ILOAD 7
   DLOAD 8
   DASTORE
   IINC 3 1
   GOTO L1
  L2
   ALOAD 2
  L7
   LINENUMBER 12 L7
   ASTORE 0
  L8
   LINENUMBER 13 L8
   ALOAD 0
   ICONST_2
   DALOAD
   DCONST_1
   DADD
   DRETURN
  L9
   LOCALVARIABLE it I L3 L6 4
   LOCALVARIABLE $i$a$-<init>-
BufferDemoKt$mapSpecial$array$1 I L4 L6 5
   LOCALVARIABLE array [D L8 L9 0
   MAXSTACK = 4
   MAXLOCALS = 10
```

```
private static final double mapSpecial() {
   byte var1 = 3;
   double[] var2 = new double[var1];

   for(int var3 = 0; var3 < var1; ++var3) {
      int var5 = false;
      double var8 = (double)var3;
      var2[var3] = var8;
   }

   return var2[2] + 1.0D;
}
```

Decompilation - generic

```
private fun mapGeneric(): Double {
    val list = List(3) {
        it.toDouble()
    }
    return list[2] + 1.0
}
```

```
private final static mapGeneric()D
  L0
   LINENUMBER 17 L0
   ICONST_3
   ISTORE 1
  L1
   ICONST_0
   ISTORE 2
  L2
   ICONST_0
   ISTORE 3
   NEW java/util/ArrayList
   DUP
   ILOAD 1
   INVOKESPECIAL java/util/ArrayList.<init> (I)V
  …
  L5
   LINENUMBER 30 L5
   ASTORE 12
  L6
   ICONST_0
   ISTORE 11
  L7
   LINENUMBER 17 L7
   ILOAD 10
   I2D
  L8
  L9
   INVOKESTATIC java/lang/Double.valueOf (D)Ljava/lang/Double;
   ASTORE 13
   ALOAD 12
   ALOAD 13
  …
  L13
   LINENUMBER 18 L13
   ALOAD 0
   ICONST_2
   INVOKEINTERFACE java/util/List.get (I)Ljava/lang/Object; (itf)
   CHECKCAST java/lang/Number
   INVOKEVIRTUAL java/lang/Number.doubleValue ()D
   DCONST_1
   DADD
   DRETURN
  L14
   LOCALVARIABLE it I L6 L9 10
   LOCALVARIABLE $i$a$-List-BufferDemoKt$mapGeneric$list$1 I L7 L9 11
   LOCALVARIABLE list Ljava/util/List; L13 L14 0
   MAXSTACK = 4
   MAXLOCALS = 14
```

INVOKESTATIC java/lang/Double.valueOf (D)Ljava/lang/Double;
INVOKEVIRTUAL java/lang/Number.doubleValue ()D

```
private static final double mapGeneric() {
   byte var1 = 3;
   boolean var2 = false;
   boolean var3 = false;
   ArrayList var4 = new ArrayList(var1);
   boolean var5 = false;
   boolean var6 = false;
   int var14 = 0;

   for(byte var7 = var1; var14 < var7;
++var14) {
      boolean var9 = false;
      int var11 = false;
      Double var13 = (double)var14;
      var4.add(var13);
   }

   List list = (List)var4;
   return
((Number)list.get(2)).doubleValue() + 1.0D;
}
```

double -> Double

Double -> double