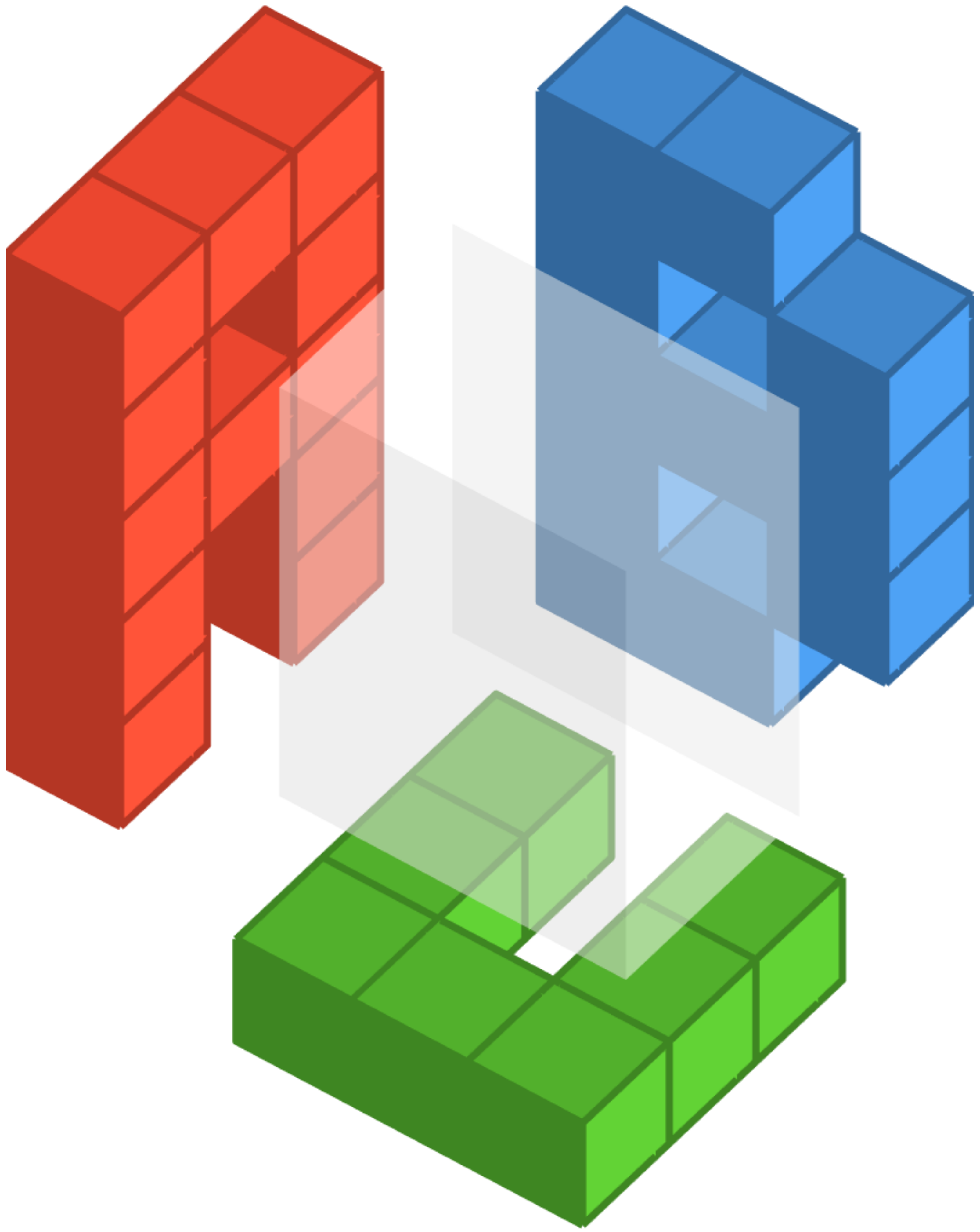
Rainbow arrays

Hypermatrix workshop



Tali Beynon, April 2023

Overview

Introduction

Basic theory of classic arrays

Array algebra

Critique of classic arrays

Rainbow arrays

Introduction

Array programming

- manipulations of higher-arity arrays ("array programming") is now an extremely common and practical concern
- used in:
  - data science, data visualization
  - data warehousing
  - machine learning, deep learning
- main players:
  - data: numpy, scipy, R, pandas
  - DL: torch, tensorflow, jax

Tensors vs arrays
In physics

- **tensors** are objects that vary over space and time
- **tensors** transform in particular ways under spacetime symmetries
- tensors are identified with **multilinear maps** between **vector spaces**
- tensors (=maps) have "input" (contravariant) and "output" (covariant) indices
  - a (p,q) tensor has p "input" and q "output" indices
  - inputs can be switched with outputs (= presence of a *metric*)
- choosing a basis for underlying vector spaces establishes unique representations as vectors, matrices, etc.
- tensors can be contracted, eliminating indices: **linear** operations

Tensors vs arrays

In computing

- **tensors** are **arrays**
- **tensors** do **not** represent maps between vector spaces
- **tensors** can be broadcasted, aggregated, and sliced
  - aggregations are typically **non-linear**
- tensors indices reflect **fundamentally different** conceptual quantities:
  - color channel vs pixel position
  - batch number vs feature dimension
  - these cannot be transformed into one another (= lack of a *metric*)

Tensors vs arrays

- we don't care about the physics meaning of tensors
- for brevity I'll use **arrays** rather than **hypermatrices**
- English definition: *collection of objects ordered in a regular way*

Basic theory

Basic theory

Scalars, vectors, matrices

- arrays have a "key space" and a "value space"
- three prototypical examples:
    - scalars (arity 0) `S = 9`
    - vectors (arity 1) `V = [1 2 3]`
    - matrices (arity 2)

```
M = [ [1 2 3]
      [4 5 6] ]
```

Basic theory
Notation

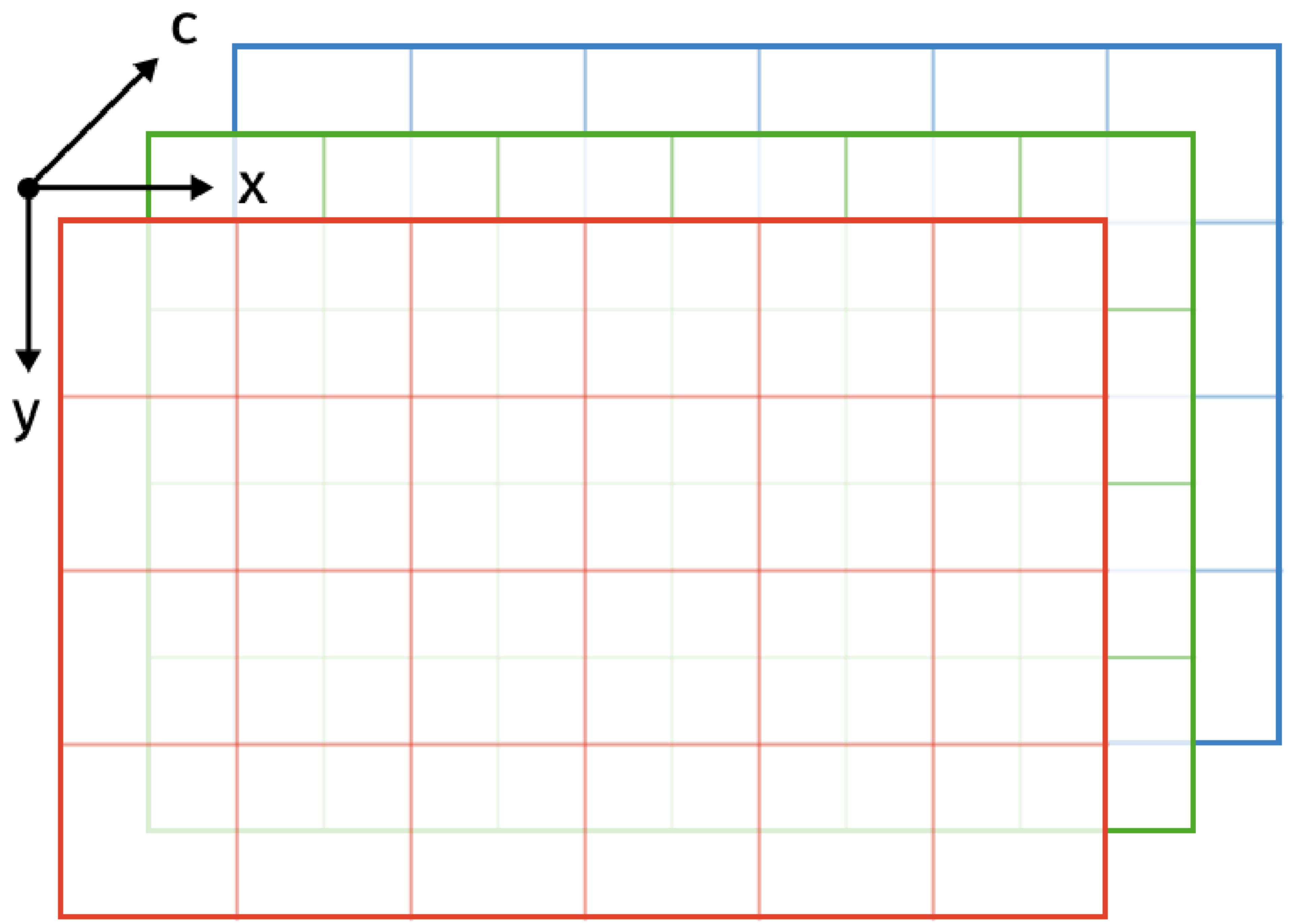
- we use the convention of **nested lists** to write arrays with multiple axes
  - no axes
    ```
    S =       9
    ```
  - 1 axis
    ```
    V =    [1 2 3]
    ```
  - 2 axes
    ```
    M = [ [1 2 3]
          [4 5 6] ]
    ```
- more deeply nested lists correspond to higher-numbered axes
- we can use colors to make this correspondence clearer:

```
M = [ [1 2 3]        axis 2
      [4 5 6] ]      axis 1
```

Basic theory

Example: color images

- images are very common data structures
- these have 3 axes: `x,y,c`
  - `x` is the x-position of a pixel ∈ `1..W`
  - `y` is the y-position of a pixel ∈ `1..H`
  - `c` is the color channel ∈ `{red, green, blue}` or conventionally `{1,2,3}`
  - value is the intensity ∈ `[0, 1]` ⊂ $\mathbb{R}$
- but in ordinary arrays, axes are ordered, not named...



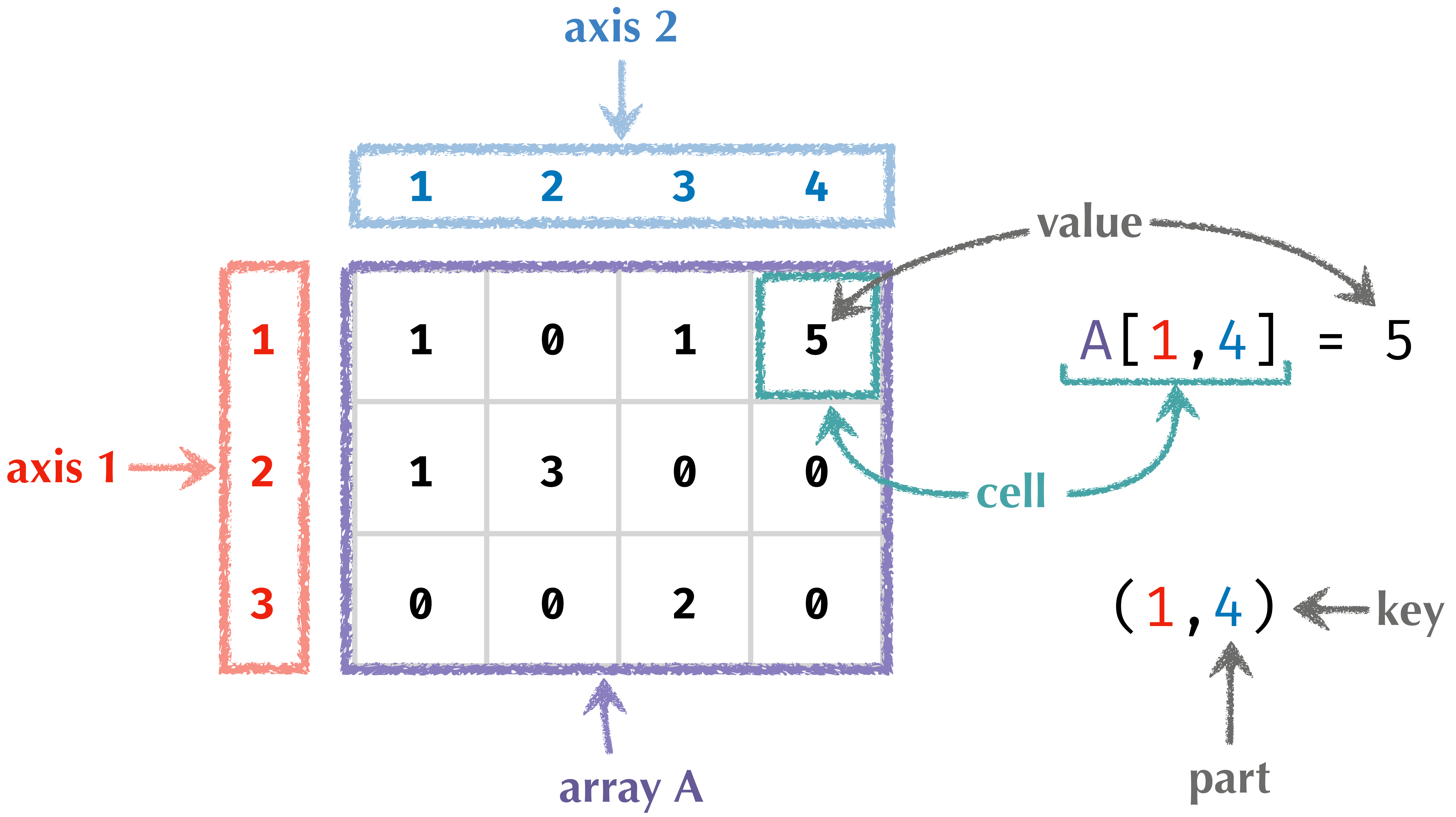
Basic theory

Example: color images

- but with classical arrays, axes are ordered, not named...
- these axes are ordered in two common ways:
  - `y,x,c`
  - `c,y,x`
- semantically, a distinction without a difference, but required to know for (classical) array programming

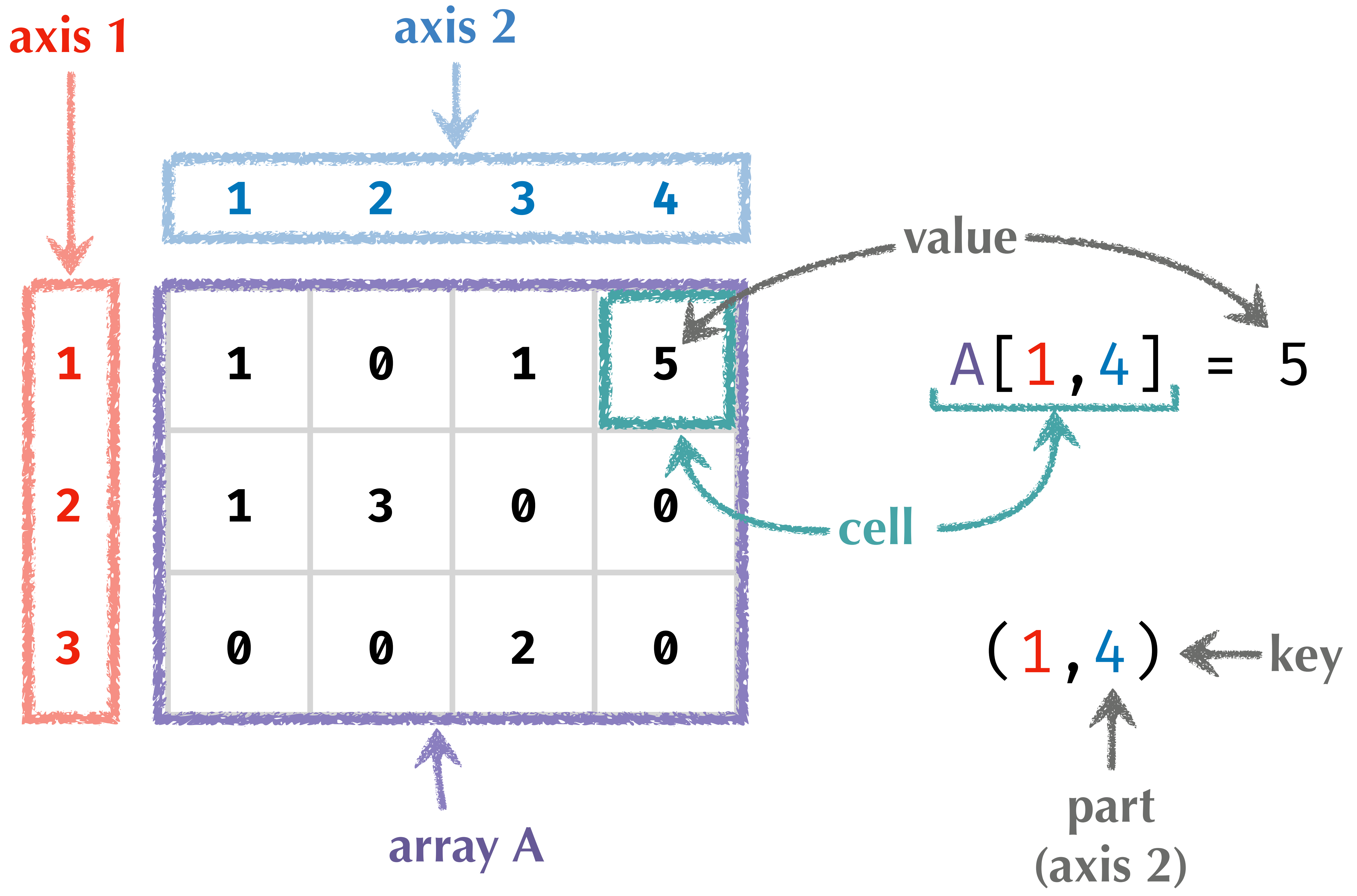
Basic theory

Array terminology



Basic theory

Array terminology



array	collection of **cells**
cell	slot in an **array**; labeled by **key**; filled with a **value**
key	position of a cell in an array; a tuple of **parts**
axis	key tuple position

Basic theory
Simplifying assumption

- part spaces are usually sets of the form `1 .. n = {1,2, … ,n}`
- key space is the space of tuples formed from these part spaces
  - for n-array, key space can be denoted by a **shape** written $\langle s_1, s_2, \ldots, s_n \rangle$
  - a matrix with 3 rows and 2 columns has shape `⟨3,2⟩`

    `⟨3,2⟩ = { (1,1), (1,2), (2,1), (2,2), (3,1), (3,2) }`

  - this is the Cartesian product of part spaces `1 .. 3` and `1 .. 2`
  - a scalar has a shape `⟨⟩` ; key space is singleton set `⟨⟩ = {()}`
- abstracting, we can consider general part spaces that aren't consecutive natural numbers -- the total order on ℕ is usually not used or needed

Basic theory

Arrays as functions

- we can see an array A as a **function** from its key space to its value space
- arrays are just **lookup tables**, to use computer science terminology
- array algebra is the algebra that manipulates lookup tables
- an n-array is just a function of n variables
- or equivalently: a unary function with a single argument that is an n-tuple
- cell value `A[1,2,3]` is shorthand for function application `A( (1,2,3) )`
- later: rainbow arrays replace this **tuple** with a **record**

Array examples

Examples

name	arity	example	shape	entries
scalar	0	9	()	() → 9
vector	1	[9 1 5]	(3)	(3) → 5
matrix	2	[[9 1 5] [3 2 6]]	(2,3)	(1,1) → 9 (2,3) → 6
3-array	3	[[[0 1] [0 10]] [[2 3] [20 30]] [[4 5] [40 50]]]	(3,2,2)	(1,1,1) → 0 (2,1,1) → 2 (3,2,2) → 50
4-array	4	[[[[1 2]]] [[[3 4]]] [[[5 6]]]]	(3,1,1,2)	(1,1,1,1) → 1 (2,1,1,1) → 3 (3,1,1,2) → 6

Array algebra

Cellwise definitions

- define some array ("output array") that depends on other arrays ("input arrays")
- to do this, **work backwards**: derive output cell from input cells
- examples:
  - `M[i,j] ≡ B[i] + C[j]`
  - `V[i] ≡ V[i] * M[i]`
  - `M[i,j] ≡ V[i] - V[j]`
  - `M[i,j] ≡ S[]`
- cellwise definitions are the most flexible kind of definition
  - but can be decomposed into other, primitive definitions

Colorful notation

- cellwise definitions: instead of using symbols to indicate parts:

```
P[i,j] ≡ A[i] + B[j]
```

- often we will use colors to indicate parts:

```
P[●,●] ≡ A[●] + B[●]
```

- at a glance, we can easily see matching parts...
- note: this is just a visual aid, it's not semantically meaningful
- this also gestures to the rainbow at the end of the talk

Operations

- unary operations: have a single input array
- three common unary operations, corresponding to how they modify # of axes
  - **transposition**: output has = # *of* axes as input
  - **broadcasting**: output has > # axes than input
  - **aggregation**: output has < # axes than input
  - **folding**: output has < # axes than input
- n-ary operations: have multiple input arrays
  - **elementwise**: output has = # axes as every input
  - **picking**: output has = # axes as first input

Unary operations

Broadcasting

- "smear" lower-arity arrays across additional novel axes
- "repeats" cells across the novel axes
- example:
  - broadcast a scalar (0-array) into a vector (1-array):
  - broadcast a vector into a matrix: XXX missing image
- cellwise:
  - scalar → vector: `V[i] ≡ S[]`
  - vector → matrix: `M[r,c] ≡ V[r]`   `M[r,c] ≡ V[c]`
  - scalar → matrix: `M[r,c] ≡ S[]`

Broadcasting

- broadcasting notated by $A^{a \to s}$
  - `a` is the position in axis list where an axis of size `s` will be inserted
  - $A^{1 \to s}$ indicates adding axis at beginning
  - $A^{n \to s}$ indicates adding axis just before position *n*
- example, vector `U = [1 2 3]`

```
U²→³ = [[1 1 1]     U¹→³ =    [[1 2 3]
        [2 2 2]                [1 2 3]
        [3 3 3]]               [1 2 3]
```

- example: scalar S = 9

$$S^{1\to2,2\to3} = 9^{1\to2,2\to3} = [9\ 9]^{2\to3}$$
$$= [[9\ 9\ 9]\ [9\ 9\ 9]]$$

Broadcasting

- from the function perspective, $A^{n\to}$ takes one more argument than `A`, but drops it and calls `A`

$$A^{n\to}[\ \dots,\ \bullet_{n-1},\ \bullet_{n},\ \bullet_{n+1},\ \dots\ ]$$

$$\equiv\quad A[\ \dots,\ \bullet_{n-1},\ \bullet_{n+1},\ \dots\ ]$$

Transposition

"Axis yoga"

- transposition re-arranges the order of axes:
  - transpose a matrix: $M^T[i,j] \equiv M[j,i]$
  - change image convention: $I'[x,y,c] \equiv I[c,x,y]$
- notate this as a superscript describing the axis permutation A^σ:
  - $A^\sigma[\bullet_1, \bullet_2, \ldots, \bullet_n] = A[\bullet_{\sigma(1)}, \bullet_{\sigma(2)}, \ldots, \bullet_{\sigma(n)}]$
  - $M^T = M^{(1,2)}$
  - $I' = I^{(1,2,3)}$

Aggregation

- aggregation w.r.t. any commutative monoid
  - for fields: sum, mean
  - for semirings:
    - $\mathbb{R}$, $\mathbb{Z}$, $\mathbb{N}$: plus, times, min, max
    - $\mathbb{B}$: and $\wedge$, or $\vee$, xor $\veebar$
- subscript picks the axis to aggregate (= remove):
  - $\mathtt{sum}_2(\mathtt{F})[{\color{red}\bullet},{\color{blue}\bullet}] \equiv \sum_{\color{green}\bullet} \mathtt{F}[{\color{red}\bullet},{\color{green}\bullet},{\color{blue}\bullet}] = \mathtt{F}[{\color{red}\bullet},1,{\color{blue}\bullet}] + \mathtt{F}[{\color{red}\bullet},2,{\color{blue}\bullet}] + \ldots$
  - $\mathtt{min}_1(\mathtt{G})[{\color{green}\bullet}] \equiv \mathtt{min}_{\color{red}\bullet}\ \mathtt{G}[{\color{red}\bullet},{\color{green}\bullet}] = \mathtt{min}(\mathtt{G}[1,{\color{green}\bullet}],\ \mathtt{G}[2,{\color{green}\bullet}],\ \ldots)$
  - $(\wedge_1 \mathtt{F})[] \equiv \mathtt{and}_{\color{red}\bullet}\ \mathtt{F}[{\color{red}\bullet}] = \mathtt{F}[1] \wedge \mathtt{F}[2] \wedge \ldots$

Folding

- see an array `A` as a map $A : \mathbb{K} \to \mathbb{V}$ from key space $\mathbb{K}$ to value space $\mathbb{V}$
- e.g. a real `M` of shape $\langle 3,2 \rangle$ is a map $A : \langle 3,2 \rangle \to \mathbb{R}$
  - $\langle 3,2 \rangle$ stands for the set of tuples $\{ (i,j) \mid i \in 1..3,\ j \in 1..2 \}$
  - `M = [ [1 2] [3 4] [5 6] ]` is a vector-of-vectors in two ways:
    - 3-vector of row vectors with cells `[1 2]`, `[3 4]`, `[5 6]`
    - 2-vector of column vectors with cells: `[1 3 5]`, `[2 4 6]`
- this corresponds to **folding** the map $A : \langle 3,2 \rangle \to \mathbb{R}$ into:
  - a 3-vector whose cells are 2-vectors $A : \langle 3 \rangle \to \langle 2 \rangle \to \mathbb{R}$
  - a 2-vector whose cells are 3-vectors $A : \langle 2 \rangle \to \langle 3 \rangle \to \mathbb{R}$

Folding

- the isomorphism between `A:X×Y→Z` and `A:X→Y→Z` is called currying; (generalized) currying of lookup tables is **array folding**
- we denote folding the n'th axis of A with $A^{n>}$

$$
\begin{aligned}
& A^{n>}[\ \ldots\ , {\color{gray}\bullet}_{n-1},\quad {\color{gray}\bullet}_{n+1},\ \ldots\ ][{\color{red}\bullet}] \\
\equiv\ & A[\ \ldots\ , {\color{gray}\bullet}_{n-1}, {\color{red}\bullet}_{n}, {\color{gray}\bullet}_{n+1},\ \ldots\ ]
\end{aligned}
$$

- folding the n'th axis moves that axis into the value space; cells become vectors
- folding multiple axes simultaneously make cells into arbitrary arrays:
  - example: fold the 1st and 3rd axes of a 3-array A, giving a vector of matrices:

$$A^{1,3>}[{\color{green}\bullet}][{\color{red}\bullet},{\color{blue}\bullet}] \equiv A[{\color{red}\bullet},{\color{green}\bullet},{\color{blue}\bullet}]$$

Folding

row vectors of a matrix

```
M = [ [1 2]
      [3 4]
      [5 6] ]
```

- cellwise:

`M2>[●][●] ≡ M[●,●]`

- evaluate `M2>[3]`

```
M2>[3][1] = M[3,1] = 5
M2>[3][2] = M[3,2] = 6

M2>[3] = [5 6]
```

- `M2>[3]` is the *third* row vector of `M`
- `M2>` is a vector of row vectors of `M`

Folding

column vectors of a matrix

```
M = [ [1 2]
      [3 4]
      [5 6] ]
```

- cellwise:

$$M^{1>}[\bullet][\bullet] \equiv M[\bullet,\bullet]$$

- evaluate $M^{1>}[1]$

$$M^{1>}[1][1] = M[1,1] = 1$$
$$M^{1>}[1][2] = M[2,1] = 3$$
$$M^{1>}[1][3] = M[3,1] = 5$$

$$M^{1>}[1] = [1\ 3\ 5]$$

- $M^{1>}[1]$ is the *first* column vector of M
- $M^{1>}$ is a vector of column vectors of M

N-ary operations

Elementwise

- combine arrays w.r.t. any n-ary operation
  - for fields: unary operations `-□`, `1/□`
  - for semirings:
    - ℝ, ℤ, ℕ: n-ary plus, times, min, max
    - 𝔹: n-ary and, or, xor, unary not
- e.g. cellwise definitions:
  - `(A + B)[●] ≡ A[●] * B[●]`
  - `(A ∧ B)[●,●] ≡ A[●,●] ∧ B[●,●]`
  - `(¬A)[●,●,●] ≡ ¬A[●,●,●]`

"Tensor product"

- tensor product of two vectors via broadcasting + elementwise

$$
\begin{aligned}
(U \otimes V)[{\color{red}\bullet},{\color{green}\bullet}] &\equiv U[{\color{red}\bullet}\quad] \ast V[\quad {\color{green}\bullet}] \\
&= U^{2\to}[{\color{red}\bullet},{\color{green}\bullet}] \ast V^{1\to}[{\color{red}\bullet},{\color{green}\bullet}] \\
&= (U^{2\to} \ast V^{1\to})[{\color{red}\bullet},{\color{green}\bullet}] \\
&\therefore \\
U \otimes V &= U^{2\to} \ast V^{1\to}
\end{aligned}
$$

Matrix multiplication

- matrix multiplication `M·N` via broadcasting + elementwise + aggregation
    - $$\begin{aligned}(M \cdot N)[{\color{red}\bullet},{\color{blue}\bullet}] &= \textstyle\sum_{\color{green}\bullet}\ M[{\color{red}\bullet},{\color{green}\bullet}\quad] * N[\quad{\color{green}\bullet},{\color{blue}\bullet}] \\ &= \textstyle\sum_{\color{green}\bullet}\ M^{3\to}[{\color{red}\bullet},{\color{green}\bullet},{\color{blue}\bullet}] * N^{1\to}[{\color{red}\bullet},{\color{green}\bullet},{\color{blue}\bullet}] \\ &= \textstyle\sum_{\color{green}\bullet}\ (M^{3\to} * N^{1\to})[{\color{red}\bullet},{\color{green}\bullet},{\color{blue}\bullet}] \\ &= \mathrm{sum}_2(M^{3\to} * N^{1\to})[{\color{red}\bullet},{\color{blue}\bullet}] \\ &\therefore \\ M \cdot N &= \mathrm{sum}_2(M^{3\to} * N^{1\to})\end{aligned}$$

Picking

- using an array of positions P to pick cells in another array A
- written A[P]
- value space of picking array P must be key space of target array A

$$\begin{aligned} P &: \mathbb{K} \to \langle s1, s2, \ldots \rangle \\ A &: \phantom{\mathbb{K} \to{}} \langle s1, s2, \ldots \rangle \to \mathbb{V} \\ A[P] &: \mathbb{K} \to \mathbb{V} \end{aligned}$$

- cellwise definition: (A[P])[●,●,●, …] = A[P[●,●,●, …]]
- this is just ordinary function composition of lookup tables!

Picking

- examples: picking from a vector `A = [10 20 30]`

```
P = 2                   A[P] = 20
P = [3 1 2]             A[P] = [30 10 20]
P = [[1] [2]]           A[P] = [[10] [20]]
```

- examples: picking from a matrix `A = [[10 20] [30 40]]`

```
P = (1,1)                   A[P] = 10
P = [(1,1) (2,2) (2,1)]     A[P] = [10 40 30]
P = [[ (2,1) ]]             A[P] = [[ 30 ]]
```

Critique

Key point: keys are tuples

- Classic arrays = cells identified by tuples of parts
- Tuples are ordered lists
- Is this a good choice?

Why tuples?

Why are tuples a good choice?

- They are simple, familiar data structures
- Positionally-ordered arguments are the norm in programming
- Make machine implementation easy:
  - Arrays must be laid out in consecutive positions in linear memory (RAM)
  - This requires an ordering of axes to decide how to compile an abstract key like `(3,1,2)` from shape `(3,3,3)` into an offset into memory:

```
offset = (3-1) * 9 + (1-1) * 3 + (2-1) = 19
```

Why not tuples?

Why are tuples not a good choice?

- compositions of arrays require **matching** corresponding axes from the arrays
  - getting this matching right (e.g. color channel of images with color channel of a tinting operation) may require fiddly transposition + broadcasting
- throws away semantic information (e.g. axis 3 = color channel), yielding endless bugs and tedious documentation to keep track of axes
- similar situation to early days of programming:
  - registers in a CPU are **numbered**, but humans like to use **named variables**
  - the allocation of variables to registers constantly changes
  - this is why we moved from **assembly code** to **high level programming langauges**

Rainbow arrays

Records

- solution: replace key **tuples** with key **records**
  - tuple: `(5, 3, 2)`
  - record: `(a=5 b=3 c=2)`
- the tuple has **components** labeled by **1**, **2**, and **3**
- the record has **fields** labeled by **`a`**, **`b`**, and **`c`**

Records

- relationship to axes:
  - tuples: axis **1** associated with the **1st** slot of every key tuple
  - records: axis **a** associated with the "**a**" field of every key record
- relationship to shapes:
  - `⟨3,2,4⟩ ≡ { (i,j,k) | 1≤i≤3, 1≤j≤2, 1≤k≤4 }`
  - `⟨a=3 b=2 c=4⟩ ≡ { (a=i b=j c=k) | 1≤i≤3, 1≤j≤2, 1≤k≤4 }`

Records

Rainbow notation

- instead of writing `(a=5 b=3 c=2)` we *color code* the fields:

  `a b c`

- and then use these colors to distinguish fields:

  `(5 3 2)`

- note there are no commas, as this is not a tuple (where order of components matters), but a rainbow notation for a record (which has no order of fields)
- similarly, the shape of a matrix with 3 rows and 4 columns is

  `⟨3 4⟩ ≡ ⟨4 3⟩ ≡ ⟨row=3 column=4⟩`

Records

Rainbow notation

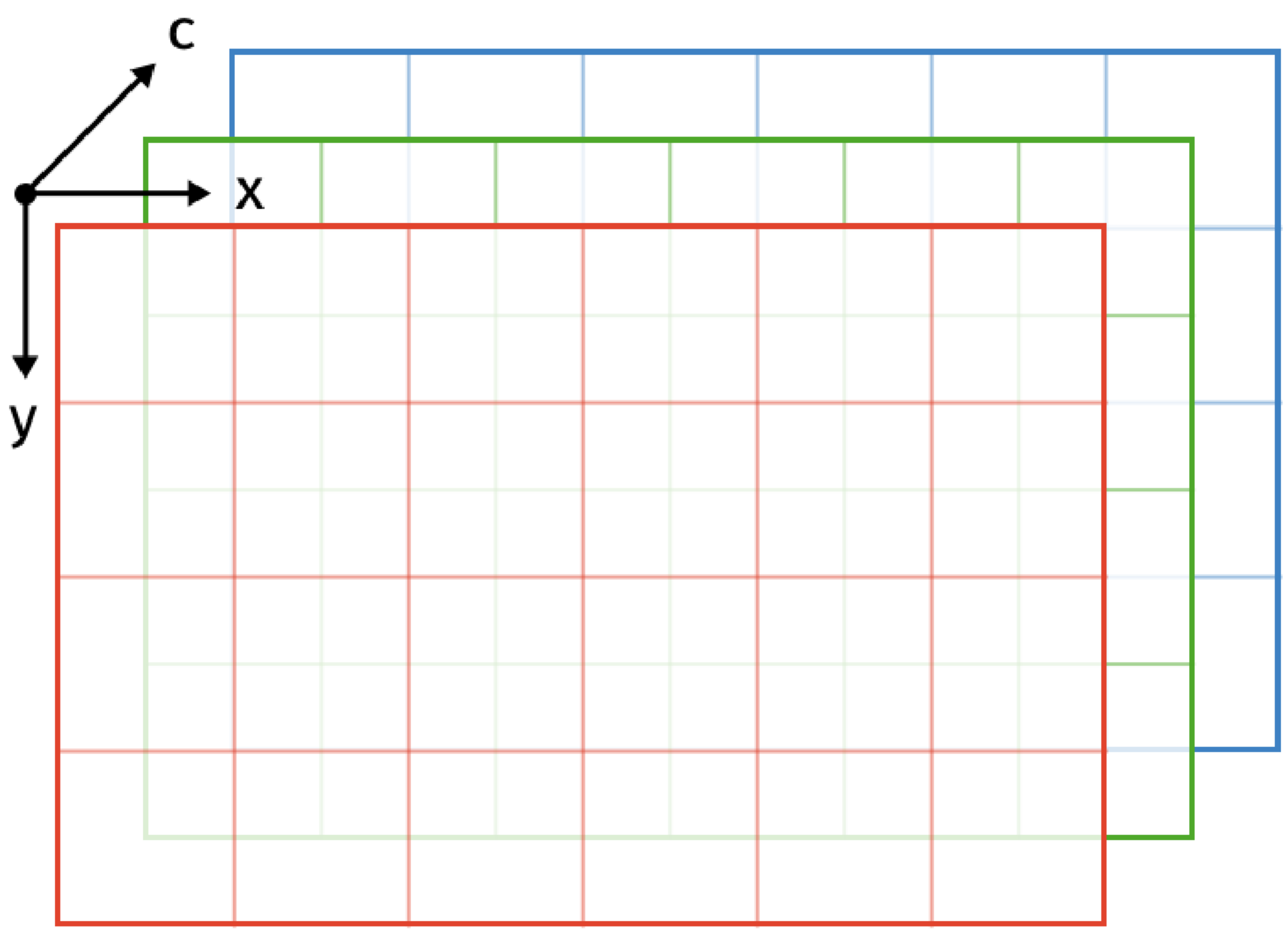
- for array lookup, we replace `A[i,j,k]` with `A[🔴🟢🔵]`
- notice again the lack of commas, since `A[🔴🟢🔵] ≡ A[🟢🔴🔵] ≡ A[🔵🔴🟢] ≡ …`
- cell value `A[🔴🟢🔵]` is shorthand for function application `A( (🔴🟢🔵) )`

```
A[🔴🟢🔵] ≡ A( (🔴🟢🔵) ) ≡ A( (r=🔴 g=🟢 b=🔵 ) )
```

- in colorful notation `A[🔴,🟢,🔵]` color is a visual aid; only order is meaningful
- in rainbow notation `A[🔴🟢🔵]` order is meaningless; only color is meaningful
- to denote the colors of an array (spectrum?), write `A : ⟨🔴🟢🔵⟩`

Records

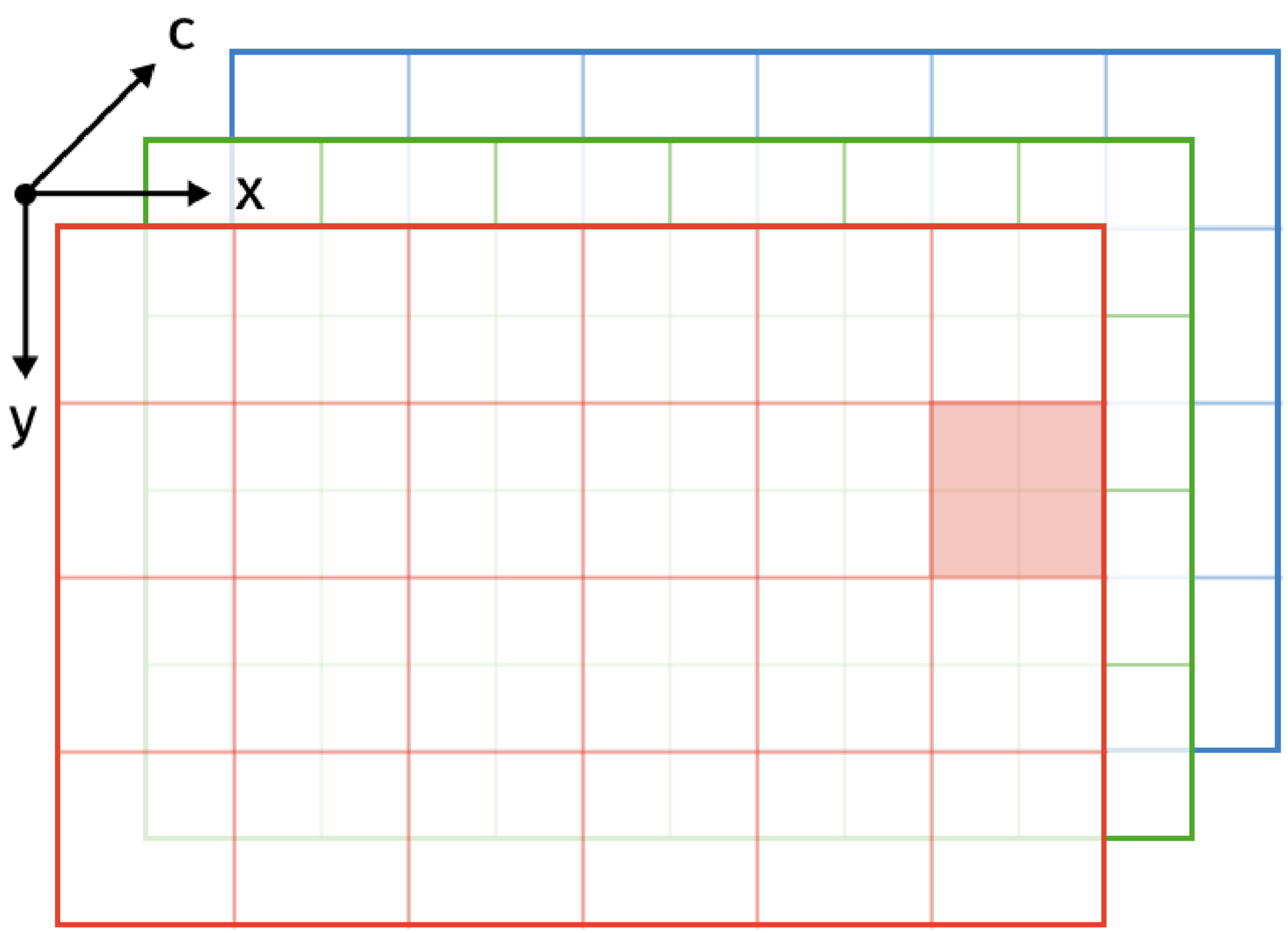
- example: a color image of 4 pixels high by 6 pixels wide



- in tuple formalism: image has shape ⟨`4,6,3`⟩ under y, x, c convention
- in record formalism: image has shape ⟨`y=4 x=6 c=3`⟩

Records

- example: a color image of 4 pixels high by 6 pixels wide



- in tuple formalism: highlighted sub-pixel has key `(2,6,1)`
- in record formalism: highlighted sub-pixel has shape ⟨`y=2 x=6 c=1`⟩

Reorganizing the API

- how do rainbow arrays reformulate our algebra?
- transposition is meaningless, since axes do not have order
- axes can however be **recolored**, a new operation
- aggregation is unchanged
- folding is unchanged
- elementwise operation automatically broadcasts over missing colors
- broadcasting is hence unnecessary
- we eliminate one operation from our API
- we also gain semantic clarity, since the axes preserve their meaning across compositions

Recoloring

- transposition *reordered* axes but preserved arity; recoloring is similar
- we have a matrix `M` : ⟨●●⟩ but we want a matrix `M̂` : ⟨●●⟩
- apply a map σ=`{`●↦●`}` to "translate" keys of `M̂` to keys of `M`
- cellwise: $\texttt{M}^{\sigma}$`[i j] ≡ M[i j]`
- conceptually, σ : ⟨●●⟩→⟨●●⟩ renames a field of a key record:
  - if underlying field names are `r`, `g`, `b`

    σ`((r=i b=j)) = (r=i g=j)`
  - in rainbow notation:

    σ`((i j)) = (i j)`

Recoloring as picking

- this is a special case of **picking**
- e.g., if we want to recolor `M`$:\langle 2\ 3\rangle$ to `M̂`$:\langle 2\ 3\rangle$, we can using picking matrix:

```
P = [ [ (1 1) (1 2) (1 3) ]
      [ (2 2) (2 3) (2 3) ] ]
```

- this has the property that `P[i j] = (i j)` as needed, so `M̂ = M[P]`
- this is also true of transposition: a transposition is a particular kind of picking in which we look up the transposed keys in the original key
- we can **also** express broadcasting (and diagonal-taking) as a special case of recoloring, if we allow the map σ to be a more general relation than a function (specifically, it must be the pre-image of a total function)

Elementwise

- rainbows: elementwise and broadcasting are *combined*
- rule: broadcast all arrays to have common set of colors, then apply operation cellwise
- result has *union* of colors of inputs
- yields unique array op for each value op (by "lifting")

Elementwise

vector times vector

- shared color:

```
    U : ⟨●⟩
    V : ⟨●⟩
U * V : ⟨●⟩

(U * V)[●] ≡ U[●] * V[●]

[1 2 3] * [0 1 2] = [0 2 6]
```

Elementwise
vector times scalar

- scalar has no colors, so no sharing!

```
    S : ⟨⟩
    V : ⟨●⟩
S * V : ⟨●⟩

(S * V)[●] ≡ S[] * V[●]

5 * [1 2 3] = [5 10 15]
```

Elementwise

vector times vector

- no shared color:

```
    U : ⟨●⟩
    V : ⟨●⟩
U * V : ⟨●●⟩

(U * V)[●●] ≡ U[●] * V[●]

[1 2 3] * [0 1 2] = [[0 0 0] [1 2 3] [2 4 6]]
```

Elementwise

matrix times matrix

- 2 shared colors:

```
    M : ⟨●●⟩
    N : ⟨●●⟩
M ∗ N : ⟨●●⟩

(M ∗ N)[●●] ≡ M[●●] ∗ N[●●]

[[1 2]   * [[0 1]   =  [[0 2]
 [3 4]]     [1 0]]      [3 0]]
```

(In the ⟨●●⟩ and [●●] forms, the first dot is red and the second green.)

Elementwise

matrix times matrix

- 1 shared colors:

```
    M : ⟨●●⟩
    N : ⟨●●⟩
M ∗ N : ⟨●●●⟩

(M ∗ N)[●●●] ≡ M[●●] ∗ N[●●]

[[1 2]  ∗ [[0 1]  =  [[1*[0 1] 2*[1 0]]   =  [[[0 1] [2 0]]
 [3 4]]    [1 0]]     [3*[0 1] 4*[1 0]]]      [[0 3] [4 0]]]
```

Elementwise

matrix times matrix

- 0 shared colors:

```
    M : ⟨🔴🟢⟩
    N : ⟨🔵🟣⟩
M * N : ⟨🔴🟢🔵🟣⟩

(M * N)[🔴🟢🔵🟣] ≡ M[🔴🟢] * N[🔵🟣]

[[1 2]   * [[0 1]   = [ [ [[0 1] [1 0]] [[0 2] [2 0]] ]
 [3 4]]     [1 0]]      [ [[0 3] [3 0]] [[0 4] [4 0]] ] ]
```

Elementwise

Example: matrix multiplication

- if M and N share one color, we can obtain matrix multiplication via:

```
  M : ⟨●●⟩
  N : ⟨●●⟩
M·N : ⟨●●⟩

M·N ≡ sum(M * N)
```

- if M and N share no colors, we obtain Kronecker product of matrices
- if they share both colors, we obtain Hadamard product

Elementwise

Example: tinting an image

- for color coding x, y, c image array `I :⟨●●●⟩`
- for a tinting factor `T :⟨●⟩` such `T = [1.0, 1.0, 0.5]` as which halves blue channel, we can apply the tint simply as:

```
I * T
```

- this is simpler and more straightforward than the classic picture, which requires broadcasting to account for x and y axes

Other operations

- aggregation, folding, picking remain as before
- however, we color these operations rather than subscript them
- e.g. for `F :⟨●●●⟩` we can "sum over green":

```
sum(F)[●●] ≡ ∑● F[●●●] = F[●1●] + F[●2●] + …
```

Takeaways

Advantages

- rainbow array algebra keeps semantic meaning (e.g. color channel, batch number, time) attached to array axes, and abandons axis order
- this leads to fewer fundamental operations, greater clarity
- compositional properties of this alternative formulation are underexplored (e.g. categorical foundation)
- the future of array programming: various deep learning practictioners (e.g. one of the inventors of Torch) are pushing for labeled axes to become the standard

Future directions

- alternative diagrammatic formulation in terms of part / key dataflow
  - e.g. taking the diagonal is copying of flow, broadcasting is deleting a flow
  - flows compose
  - categorical foundations, and connections to profunctors
- software library for Mathematica
- explain connections to hypergraph rewriting
  - e.g. matrix multiplication measures combinatorics of graph composition
  - adjacency arrays of hypergraphs are... higher-arity arrays, obviously

References

- Rush: "Tensors considered harmful"
- Maclaurin, Paszke et al: Dex project
- Chiang, Rush, Barak: "Named Tensor Notation"
- Hoyer et al: XArray project
- Zapata-Carratala, Arsiwalla, Beynon: "Heaps of Fish"