An implementation decoupling the storage representation from the in-memory representation

July 26, 2024, 3rd "Get Your Brain Together" Hackathon

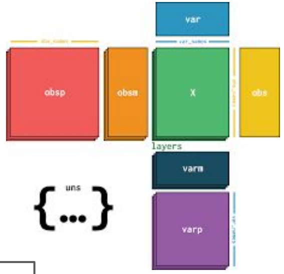
Luca Marconato on behalf of the SpatialData team: Giovanni Palla, Kevin Yamauchi, Isaac Virshup, Wouter Vierdag, Tim Treis, Josh Moore, …

EMBL

SpatialData is a solution for working with spatial multiomics datasets that bridges existing communities

scverse core

Data analysis

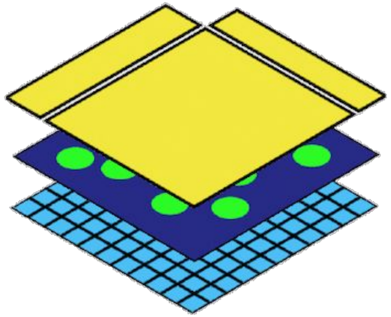


AnnData

scanpy

squidpy

SpatialData



Marconato*, Palla*, Yamauchi*, Virshup* et al. (Nature Methods, 2024)

SpatialData:
- **infrastructure for data storage, manipulation and visualization**
- **non-goal: not an analysis library**

Relies on existing Python GIS technologies

OME (Open Microscopy Environment)

NGFF

OMEZarr

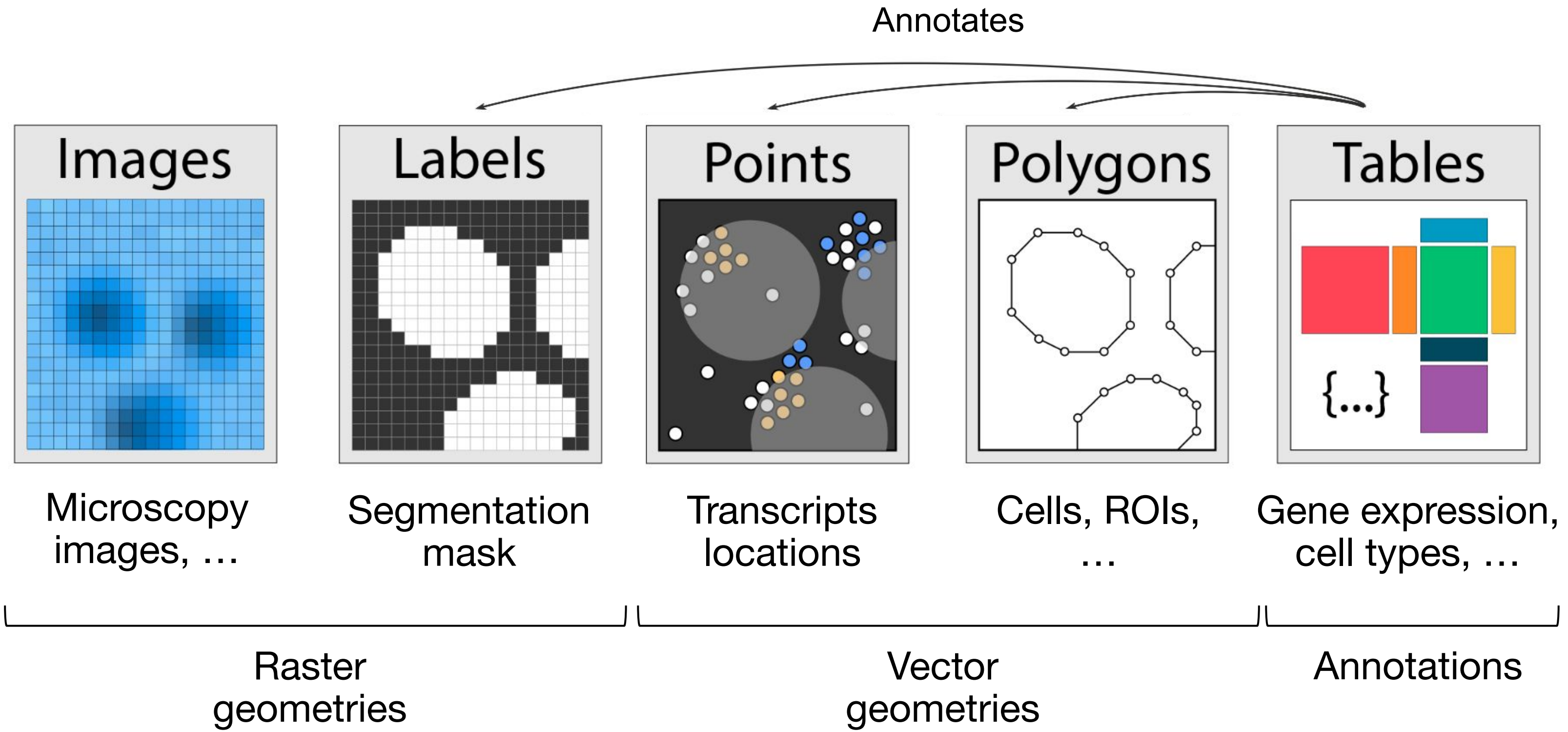
Large images, standard formats

Napari core

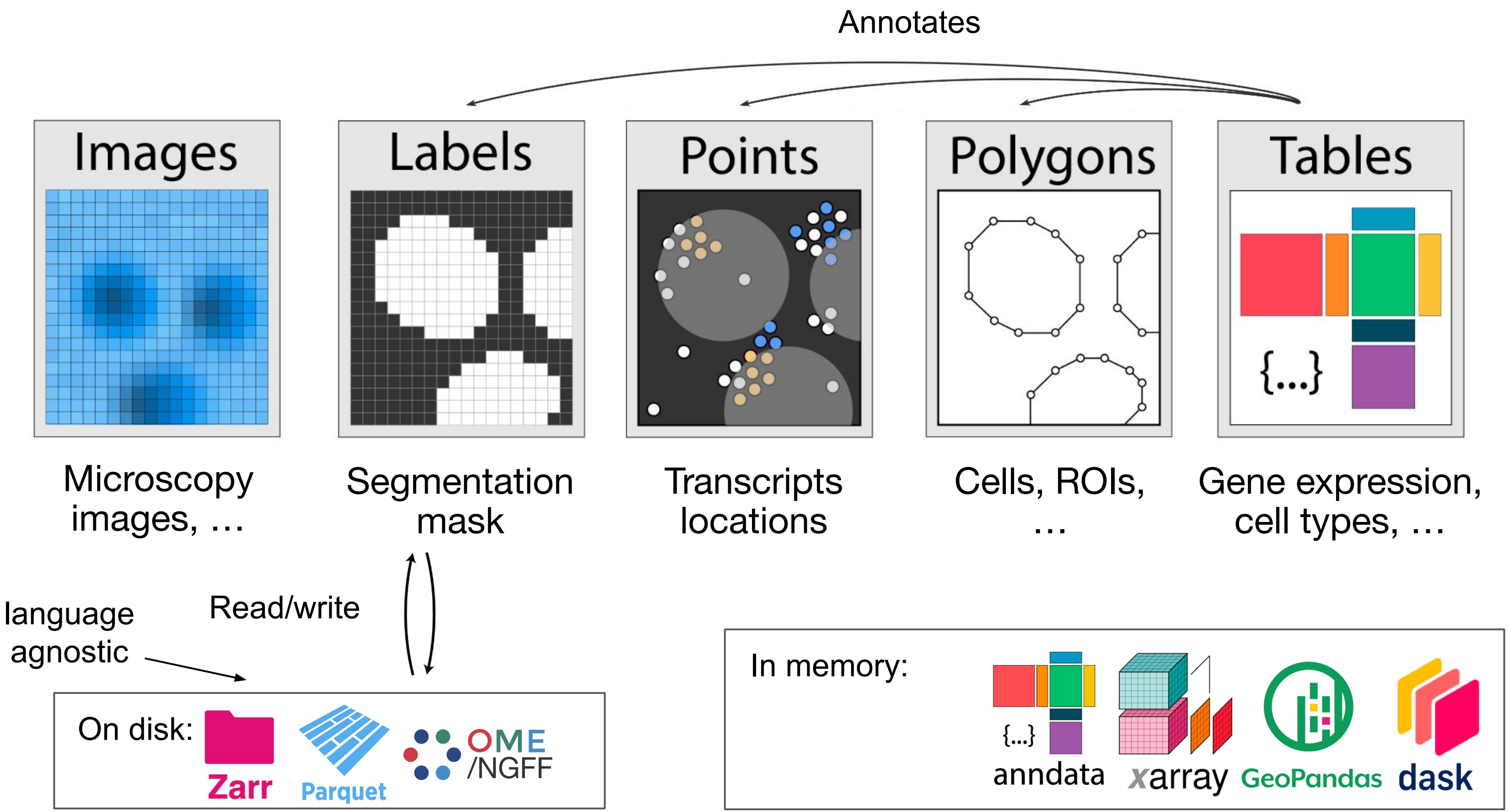
napari

Interactive visualization

Data representation is abstracted as a modular combination of reusable elements



Data representation is abstracted as a modular combination of reusable elements

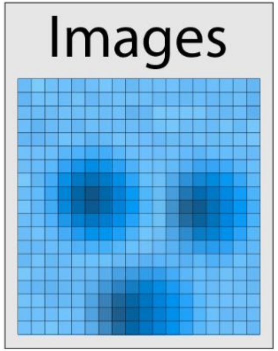


Coordinate transformations enable alignment to common coordinate systems

Transformations are defined both for raster and vector types

```
{"name" : “physical-micrometers",
        "axes" : [
            {"name": "y", "type": "space", "unit":
“micrometer"},
            {"name": "x", "type": "space", "unit":
“micrometer”}]}
```

physical space

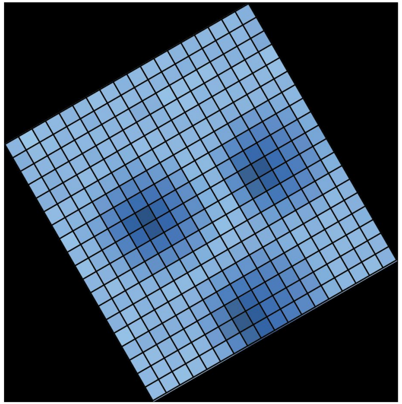


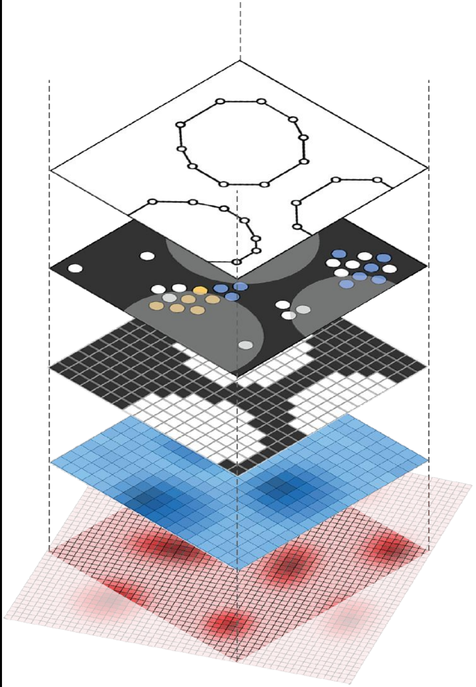

```
{“name" : “pixel-space",
        "axes" : [
            {"name": "j", "type": "space",
"discrete":      true },
            {"name": "i", "type": "space",
"discrete": true }]}
```

pixel space

```
{"name" : "pixels-to-micrometers",
  "type" : "affine",
  "values" : [
    [0.89, -0.45, 1.00],
    [0.45, 0.89, 2.00],
    [0.00, 0.00, 1.00]],
  "input_space" : "",
  "output_space":
"physical-micrometers"}
```

transformation

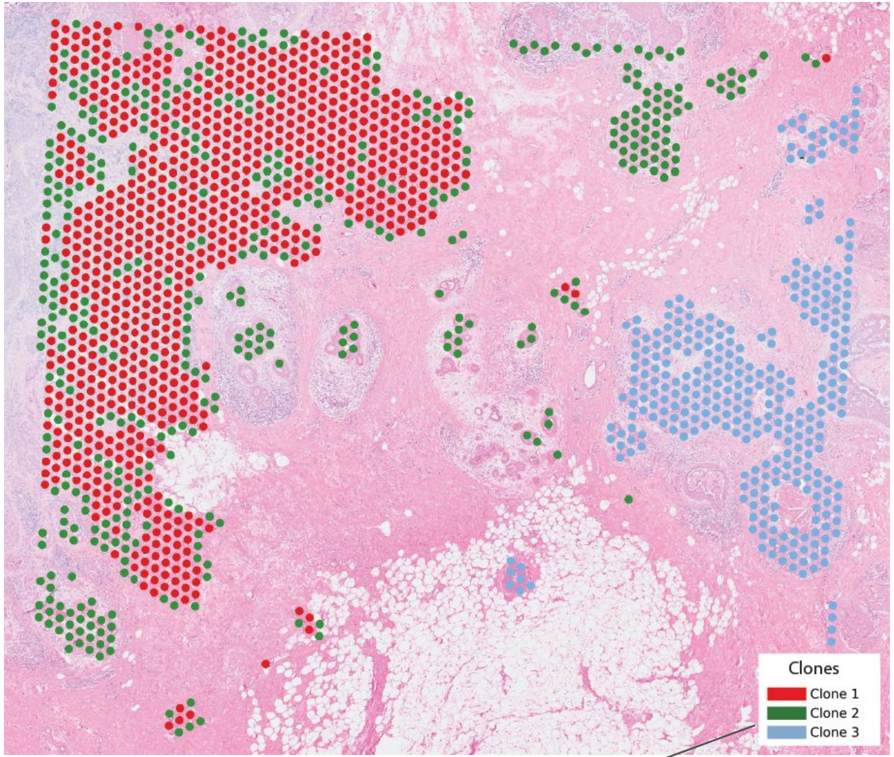


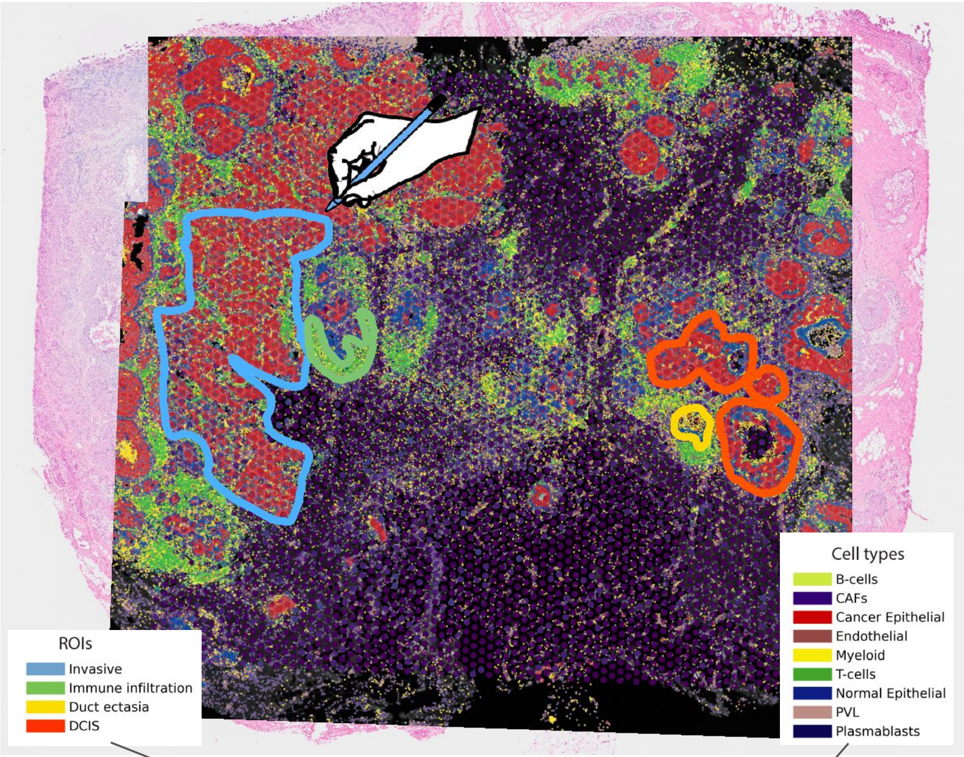


SpatialData object

Example: joint visualization of 2 Xenium + 1 Visium datasets

Transformations are defined both for raster and vector types



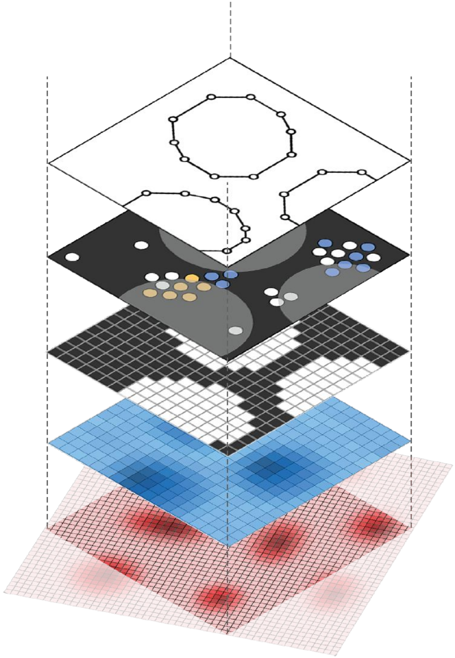


Cancer clonality (Visium)

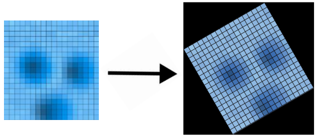
Anatomical annotations (Visium)

Cell types (Xenium)

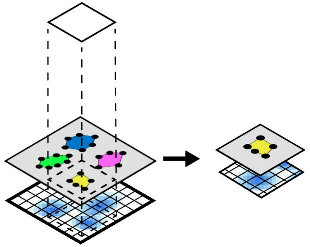
Generalized, reusable operations are defined for SpatialData objects



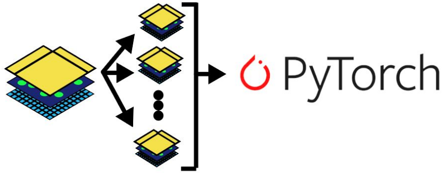
Coordinate transformations



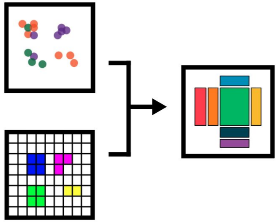
Spatial queries



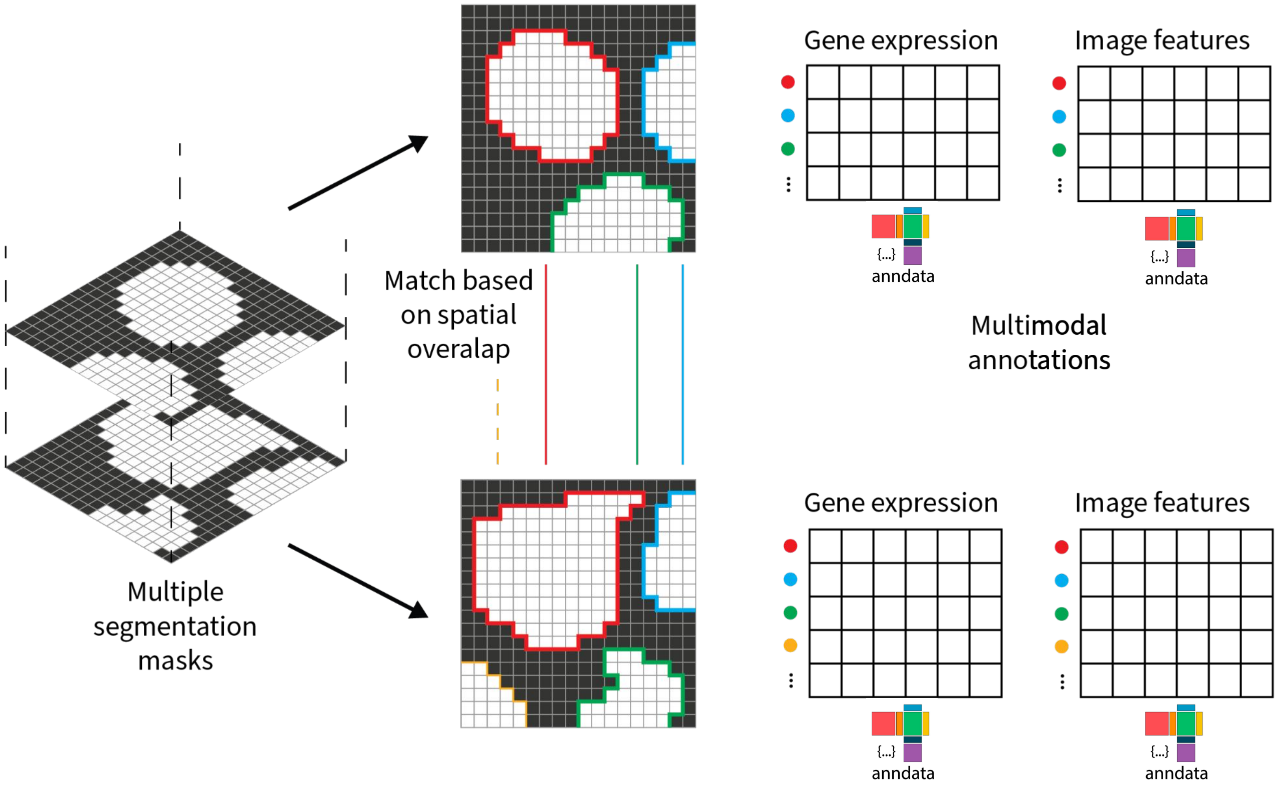
Deep learning interface



Spatial aggregations



Example use case

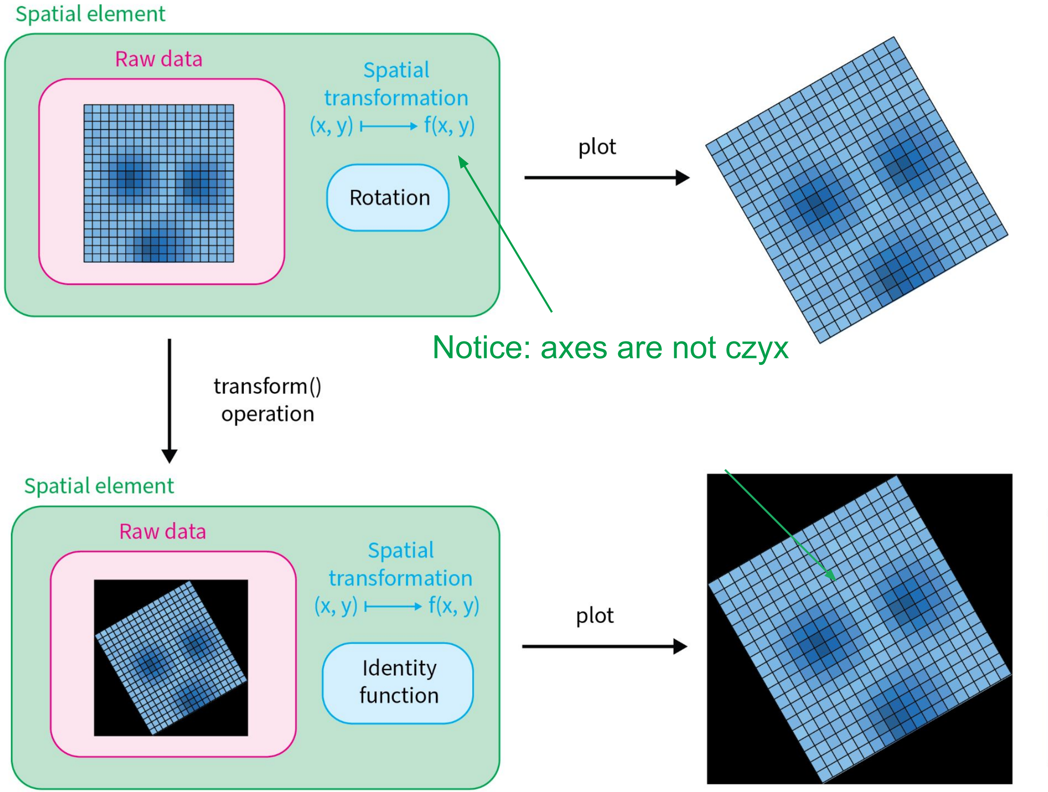


Wouter-Michiel Vierdag, Quentin Blampey

SpatialData APIs: set_transformation() vs transform()

Raw data (in-memory or on-disk)

Positioning in space (virtual)




```python
from spatialdata.transformations import set_transformation, Translation

t = Translation([10., 20.], axes=('y','z'))

set_transformation(
  element=my_image,
  transformation=t,
  to_coordinate_system="Sample 1"
)
```

Notice: axes are not czyx

```python
from spatialdata import transform

my_transformed_image = transform(my_image, to_coordinate_system='Sample 1')

# we can also manually access the transformation
from spatialdata.transformations import get_transformation

get_transformation(my_image, to_coordinate_system='Sample 1')
```

We can easily create, compose and rearrange transformations

```python
from spatialdata.transformations import Translation, Scale, Affine, Sequence

translation = Translation([1, 2], axes=("x", "y"))
scale = Scale([2, 1], axes=("y", "x"))
affine = Affine(
    [
        [4, 5, 6],
        [1, 2, 3],
        [0, 0, 1],
    ],
    input_axes=("x", "y"),
    output_axes=("y", "x"),
)
sequence = Sequence([translation, scale, affine])

print(sequence)
print(sequence.to_affine(input_axes=('x', 'y'), output_axes=('x', 'y')))
```

```
Sequence
    Translation (x, y)
        [1. 2.]
    Scale (y, x)
        [2. 1.]
    Affine (x, y → y, x)
        [4. 5. 6.]
        [1. 2. 3.]
        [0. 0. 1.]
```

```
Affine (x, y → x, y)
    [ 1.   4. 12.]
    [ 4. 10. 30.]
    [ 0.   0. 1.]
```

Transformations are defined independently of the axes of the elements they are applied to

```python
from spatialdata.transformations import Scale, MapAxis

map_axis = MapAxis({"x": "y", "y": "x"})
scale = Scale([2], axes=("z",))
sequence = Sequence([map_axis, map_axis, scale]).to_affine(
    input_axes=("x", "y", "z"), output_axes=("x", "y", "z")
)
print(sequence)
```

```
Affine (x, y, z → x, y, z)
    [1. 0. 0. 0.]
    [0. 1. 0. 0.]
    [0. 0. 2. 0.]
    [0. 0. 0. 1.]
```

```python
from spatialdata import read_zarr
from spatialdata.transformations import get_transformation,
set_transformation

sdata = read_zarr('data.zarr')

my_image = sdata['my_image']  # e.g. cyx image
my_points = sdata['my_points']  # e.g. xy points

t = get_transformation(my_image, to_coordinate_system='Sample 1')
set_transformation(my_points, transformation=t,
to_coordinate_system='Sample 1')
```

Reading/writing to disk is delegated to NGFF transformations

```python
sdata.write('data.zarr')
sdata = read_zarr('data.zarr')

# lightweight write (only transformations)
sdata.write_transformations(element_name='my_image')
```

Works also for vector data!

```python
import json

print(
    json.dumps(
        sequence.to_ngff(
            input_axes=("x", "y"),
            output_axes=("x", "y"),
        ).to_dict(),
        indent=2,
    )
)
```

NGFF requires additional informations

Assumptions to keep things less verbose for the user:

- the default coordinate system name is "global" (will change)
- the default unit is "unit"


```json
{
  "type": "affine",
  "affine": [
    [
      1.0,
      0.0,
      0.0
    ],
    [
      0.0,
      1.0,
      0.0
    ]
  ],
  "input": {
    "name": "xy",
    "axes": [
      {
        "name": "x",
        "type": "space",
        "unit": "unit"
      },
      {
        "name": "y",
        "type": "space",
        "unit": "unit"
      }
    ]
  },
  "output": {
    "name": "global",
    "axes": [
      {
        "name": "x",
        "type": "space",
        "unit": "unit"
      },
      {
        "name": "y",
        "type": "space",
        "unit": "unit"
      }
    ]
  }
}
```

First implementation, mirroring the NGFF specification: coordinate systems

https://github.com/scverse/spatialdata/blob/main/src/spatialdata/transformations/ngff

ngff_coordinate_system.py

```python
from spatialdata.transformations.ngff.ngff_coordinate_system import NgffAxis, NgffCoordinateSystem

axes = [
    NgffAxis(name="x", type="space", unit="micrometer"),
    NgffAxis(name="y", type="space", unit="micrometer"),
    NgffAxis(name="z", type="space", unit="micrometer"),
]
coordinate_system = NgffCoordinateSystem(
    name="volume_micrometers",
    axes=axes,
)
```

First implementation, mirroring the NGFF specification: coordinate transformations

https://github.com/scverse/spatialdata/blob/main/src/spatialdata/transformations/ngff

ngff_transformations.py

```python
__all__ = [
    "NgffBaseTransformation",
    "NgffIdentity",
    # "MapIndex",
    "NgffMapAxis",
    "NgffTranslation",
    "NgffScale",
    "NgffAffine",
    "NgffRotation",
    "NgffSequence",
    # "Displacements",
    # "Coordinates",
    # "InverseOf",
    # "Bijection",
    "NgffByDimension",
]
```

```python
class NgffIdentity(NgffBaseTransformation):
    """The Identity transformation from the NGFF specification."""

    def __init__(
        self,
        input_coordinate_system: Optional[NgffCoordinateSystem] = None,
        output_coordinate_system: Optional[NgffCoordinateSystem] = None,
    ) -> None:
        """
        Init the NgffIdentity object.

        Parameters
        ----------
        input_coordinate_system
            Input coordinate system of the transformation.
        output_coordinate_system
            Output coordinate system of the transformation.
        """
        super().__init__(input_coordinate_system, output_coordinate_system)

    @classmethod
    def _from_dict(cls, _: Transformation_t) -> Self:  # type: ignore[valid-type]
        return cls()

    def to_dict(self) -> Transformation_t:
        d = {
            "type": "identity",
        }
        self._update_dict_with_input_output_cs(d)
        return d
```

Possible improvement: using pydantic (see work from Davis Bennet for v0.4: JaneliaSciComp/pydantic-ome-ngff)

Adding functionalities to the NGFF transformation classes

```python
class NgffBaseTransformation(ABC):
    """Base class for all the transformations defined by the NGFF specification."""
    # ...

    @abstractmethod
    def inverse(self) → NgffBaseTransformation:
        """Return the inverse of the transformation."""

    @abstractmethod
    def _get_and_validate_axes(self) → tuple[tuple[str, ...], tuple[str, ...]]:
        """
        Get the input and output axes of the coordinate systems specified for the transformation, and check if they are
        compatible with the transformation.
        """

    @abstractmethod
    def transform_points(self, points: ArrayLike) → ArrayLike:
        """
        Transform points (coordinates).

        Notes
        -----
        This function will check if the dimensionality of the input and output coordinate systems of the
        transformation are compatible with the given points.
        """

    @abstractmethod
    def to_affine(self) → NgffAffine:
        """Convert the transformation to an affine transformation, whenever the conversion can be made."""
```

In the new implementation we have a separate transform() function

Drawbacks of staying close to the NGFF implementation

Drawbacks:

1. our APIs were too verbose:
   a. the users had to specify (or import) the axes, units, coordinate systems
   b. c vs non-c axes had to be specified
2. transformations could not be moved around: e.g. from xy points to a cyx image
3. Ambiguity around sequence transformations due to the possibility of specifying sub-transformations without an input/output coordinate system link

link to this old code

```python
from tests._core.conftest import (
    c_cs,
    cyx_cs,
    x_cs,
    xy_cs,
    yx_cs,
)

# 2d case, extending a xy→xy transformation to a cyx→cyx transformation using additional affine transformations
cyx_to_xy = Affine(
    np.array(
        [
            [0, 0, 1, 0],
            [0, 1, 0, 0],
            [0, 0, 0, 1],
        ]
    ),
    input_coordinate_system=cyx_cs,
    output_coordinate_system=xy_cs,
)
xy_to_cyx = Affine(
    np.array(
        [
            [0, 0, 0],
            [0, 1, 0],
            [1, 0, 0],
            [0, 0, 1],
        ]
    ),
    input_coordinate_system=xy_cs,
    output_coordinate_system=cyx_cs,
)

transformation = Sequence(
    [
        cyx_to_xy,
        # some alternative ways to go back and forth between xy and cyx
        # xy → cyx
        ByDimension(
            transformations=[
                MapAxis({"x": "x", "y": "y"}, input_coordinate_system=xy_cs, output_coordinate_system=yx_cs),
                Affine(
                    np.array([[0, 0], [0, 1]]),
                    input_coordinate_system=x_cs,
                    output_coordinate_system=c_cs,
                ),
            ],
            input_coordinate_system=xy_cs,
            output_coordinate_system=cyx_cs,
        ),
        # cyx → xy
        MapAxis({"x": "x", "y": "y"}, input_coordinate_system=cyx_cs, output_coordinate_system=xy_cs),
        Translation(np.array([1, 2])),
        Scale(np.array([3, 4])),
        affine,
        xy_to_cyx,
    ]
)
```

A new approach: different transformation classes in-memory

Simplification:

1. Transformations are defined independently of the input/output coordinate systems they will be eventually applied to .
   E.g. $x \mapsto x + 5$ reads as "if there is an x, translate it by 5"

Implementation:

1. We don't use coordinate systems to define transformations
2. Transformations require extra arguments. Examples:

```
Translation([5.], axes=('x',))
Affine([[1, 2]], input_axes=('i', 'j'), output_axes=('c'))
```

What didn't change:

1. Transformations are n-dimensional: any order of axes and any number
   Detail: in spatialdata we use only `'c'`, `'z'`, `'y'`, `'x'`; so we actually validate against these axes during `__init__()`
2. IO is done via NGFF transformations thanks to conversions:
   `BaseTransformation ↔ NgffBaseTransformation`

How we actually transform the elements

Simplifications:

- All transformations can be turned into Affine. This can be relaxed!

Implementation:

- We skipped byDimension and Rotation
- We always know the input axes thanks to our element schemas:
  - 2d images cyx
  - 3d images czy
  - 2d labels yx
  - 3d labels zyx
  - 2d points xy
  - 3d points xyz
  - 2d shapes xy
- We can find the output axes using [`_get_current_output_axes()`]()

```
def _get_current_output_axes(
    transformation: BaseTransformation, input_axes: tuple[str, ...]
) → tuple[str, ...]:
```

How we deal with missing/extra axes

```
translation = Translation([5, 3], axes=("x", "y"))
print(translation.to_affine(input_axes=("x", "y", "c"), output_axes=("c", "z", "y", "x")))
```

```
Affine (x, y, c → c, z, y, x)
    [0. 0. 1. 0.]
    [0. 0. 0. 0.]
    [0. 1. 0. 3.]
    [1. 0. 0. 5.]
    [0. 0. 0. 1.]
```

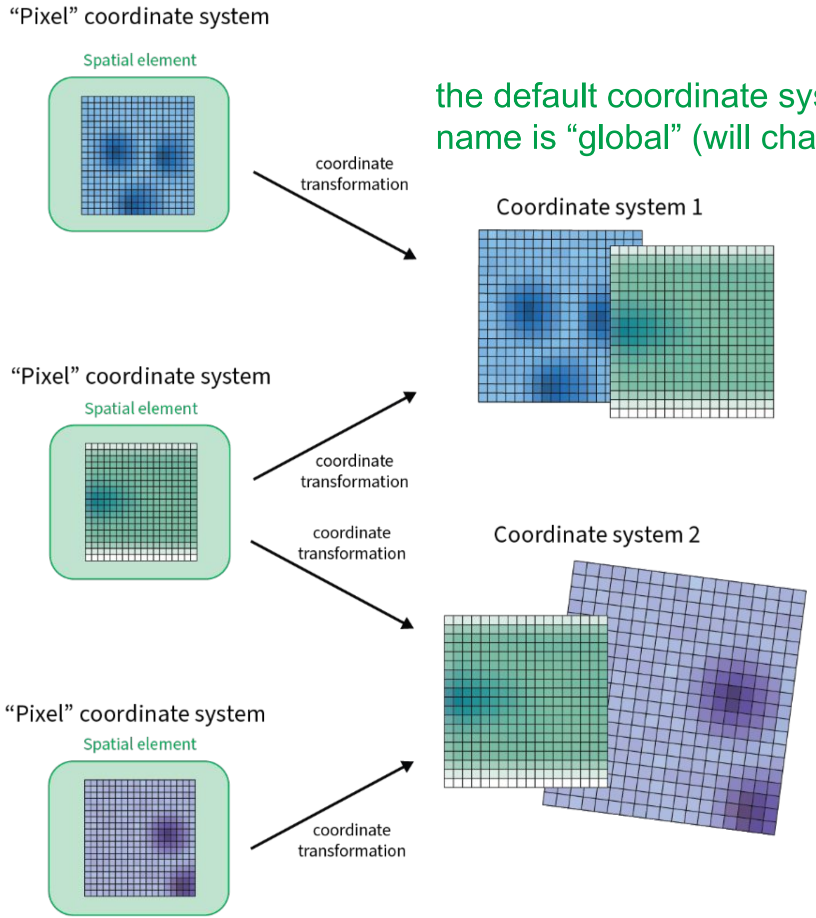
c is “passed through” (because it is present both as input and output axes but not defined in the transformation)

z is “ignored” (because it is present only in the output axes)

```
# ValueError: Input axes must be a subset of output axes.
translation.to_affine(input_axes=("x", "y", "c"), output_axes=("y", "x"))
```

```
# ValueError: The axis y is not an input of the affine transformation but it appears as output.
#   Probably you want to remove it from the input_axes of the to_affine_matrix() call.
affine = Affine(np.array([[1, 0], [2, 0], [0, 1]]), input_axes=("x",), output_axes=("x", "y"))
affine.to_affine(input_axes=("x", "y"), output_axes=("x", "y"))
```

We use transformations to map elements to coordinate systems



the default coordinate system name is "global" (will change)

- Same for vector elements
- A coordinate system is just a string
- We store transformations in the element's metadata

```
my_element.attrs['transform'] = {
    'global': Identity(),
    'Sample 1': Affine( ... )
}
```

- We plan to store (optional) coordinate system information

```
sdata.coordinate_systems = {
    'Sample 1': NgffCoordinateSystem( ... )
}
```

The transform() function

Implementation:

- Uses `dask_image.ndinterp.affine_transform` for raster data.
- Uses `geopandas.GeoSeries.affine_transform` for vector data.

Lazy and chunk-wise (but can be optimized)

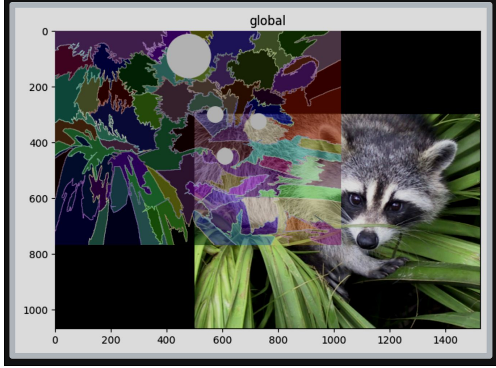
Need to use dask-geopandas at some point

Examples from the docs (note: here actually we use `matplotlib.transforms` , but the output is analogous)

Translation

The `spatialdata.transformations.Translation` transformation can be used to apply a translation to the element.

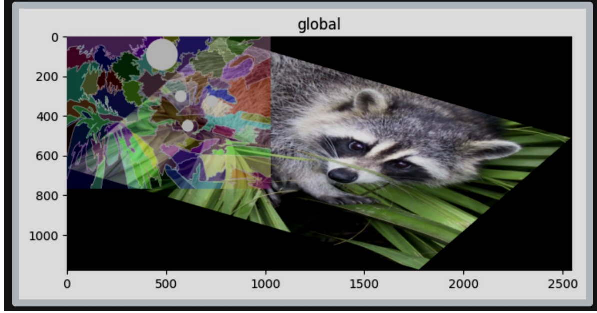
```
translation = Translation([500, 300], axes=("x", "y"))

set_transformation(sdata.images["raccoon"], translation, to_coordinate_system="global")

sdata.pl.render_images().pl.render_labels().pl.render_shapes().pl.show()
```



Affine transformation and composition

The `spatialdata.transformations.Sequence` transformation class can be used to compose transformations. This class allows to compose multiple transformations and it can be used even when the axes do not strictly match.

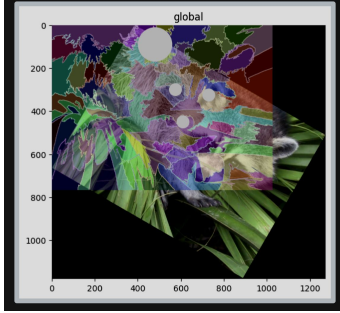
```
sequence = Sequence([rotation, scale])

set_transformation(sdata.images["raccoon"], sequence, to_coordinate_system="global")

sdata.pl.render_images().pl.render_labels().pl.render_shapes().pl.show()
```



Rotation

The `spatialdata.transformations.Affine` transformation can be used to apply an affine transformation to the elements. Let's start with a rotation.

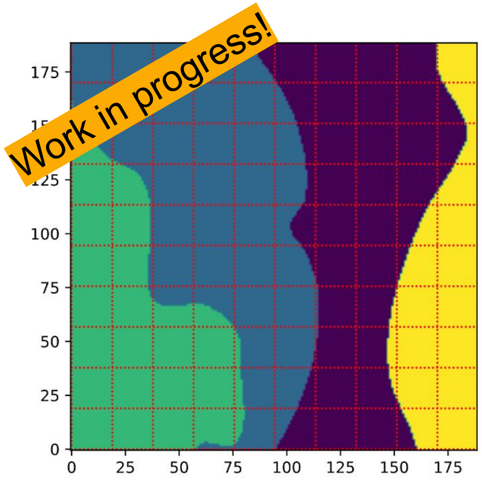
```
theta = math.pi / 6
rotation = Affine(
    [
        [math.cos(theta), -math.sin(theta), 0],
        [math.sin(theta), math.cos(theta), 0],
        [0, 0, 1],
    ],
    input_axes=("x", "y"),
    output_axes=("x", "y"),
)

set_transformation(sdata.images["raccoon"], rotation, to_coordinate_system="global")

sdata.pl.render_images().pl.render_labels().pl.render_shapes().pl.show()
```



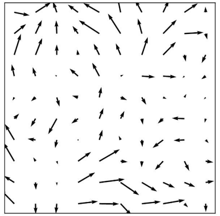
Limitations of the in-memory classes

Limitations of the in-memory transformations classes:

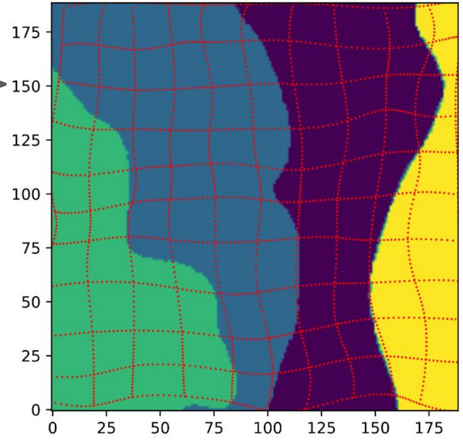
- So far, for what is implemented, none.
- Need to implement the non-linear transformations.



transform()



Displacement field
(non-linear transformation)



Read/write

OME /NGFF

Zarr

Interface to external methods:

- STalign
- EGGPLANT
- …

Tobias Graf

Limitations of the `transform()` function

Limitations of their use within `spatialdata`, and of the `transform()` function:

- `transform()` can be optimized (ad hoc algorithms, GPU)
- We allow only `'c'`, `'z'`, `'y'`, `'x'`. No `'t'` (workarounds available).
- We allow only specific orders of axes.
- We don't treat `'c'` as a spatial axis;
  - e.g. embedding a single-channel image into a multi-channel image is not allowed
  - instead, we just call `dask.array.stack()`
- No bridge: NGFF transformations ↔ xarray coordinates.

xarray coordinates

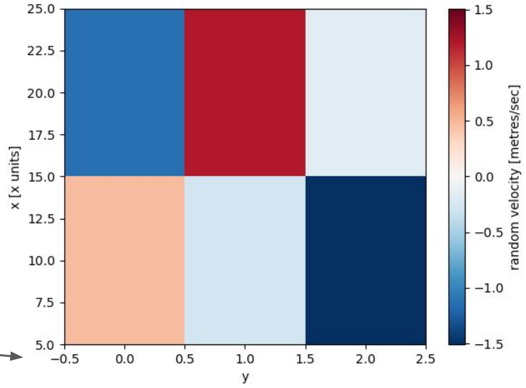
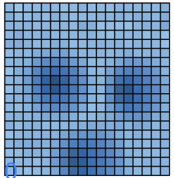


image from xarray docs

Limitations of transformations and the incoming implementation

```python
from spatialdata.transformations import Translation, Scale, Affine, Sequence

t = Affine( … )
t = Sequence([Affine( … ), Scale( … ), Translation( … )])
```



$$\begin{pmatrix} k\cos(\theta) & -k\sin(\theta) & x \\ k\sin(\theta) & k\cos(\theta) & y \\ 0 & 0 & 1 \end{pmatrix}$$

assume k = 4

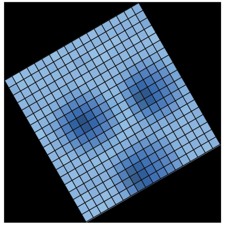
transform() transform()

Current implementation:

OME /NGFF

≈400x400
pixels

bad



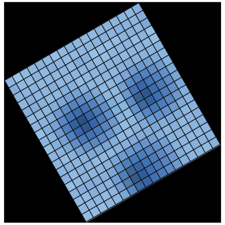
$$\begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix}$$

Better implementation:

OME /NGFF + xarray

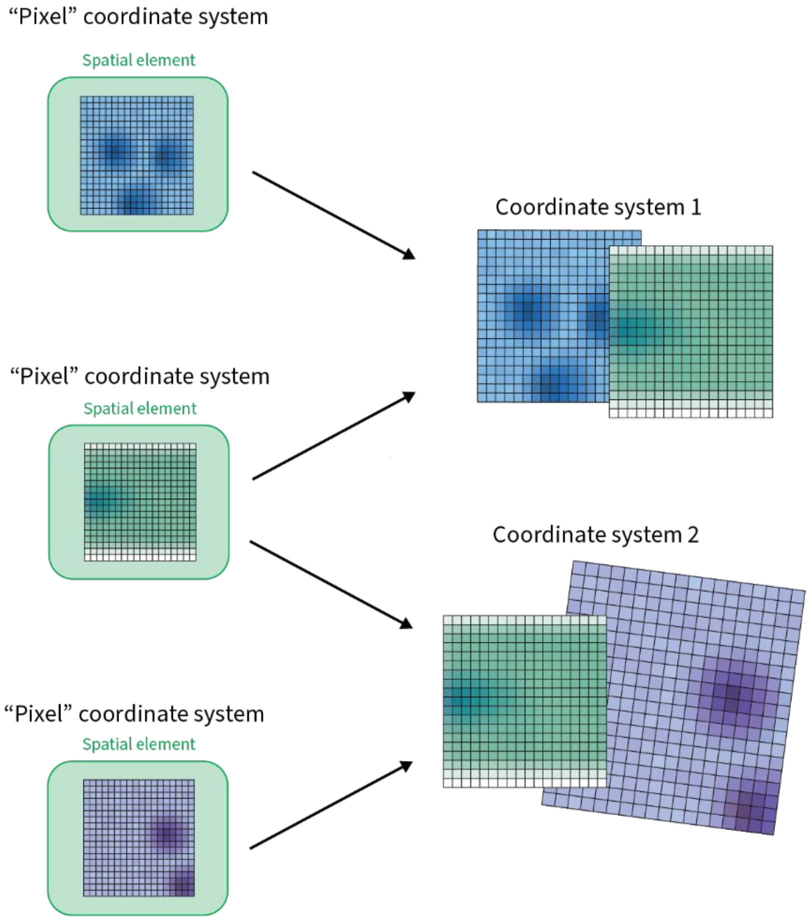
≈100x100
pixels

performant



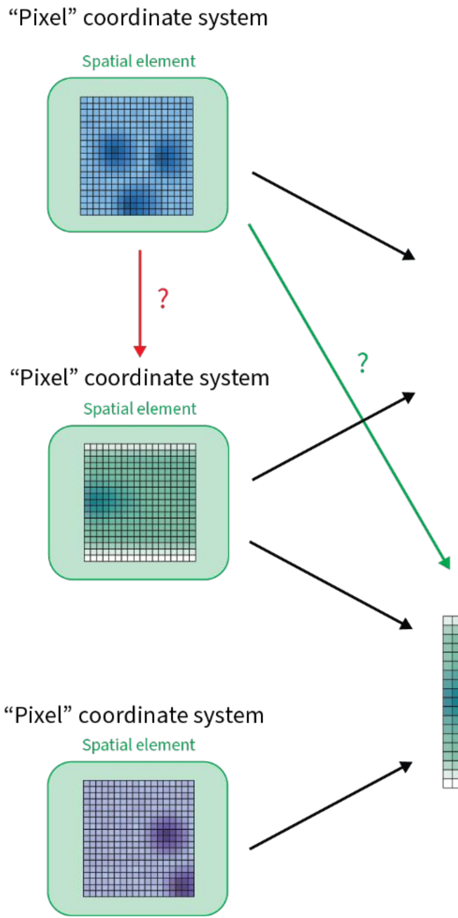
ergonomic

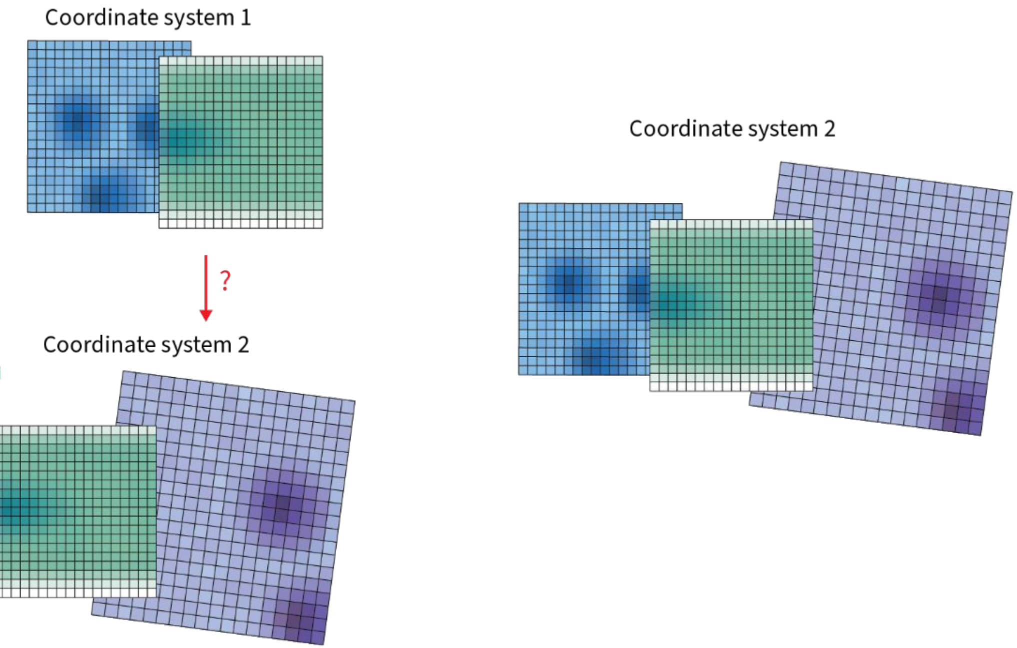
Limitations of transformations and the new implementation



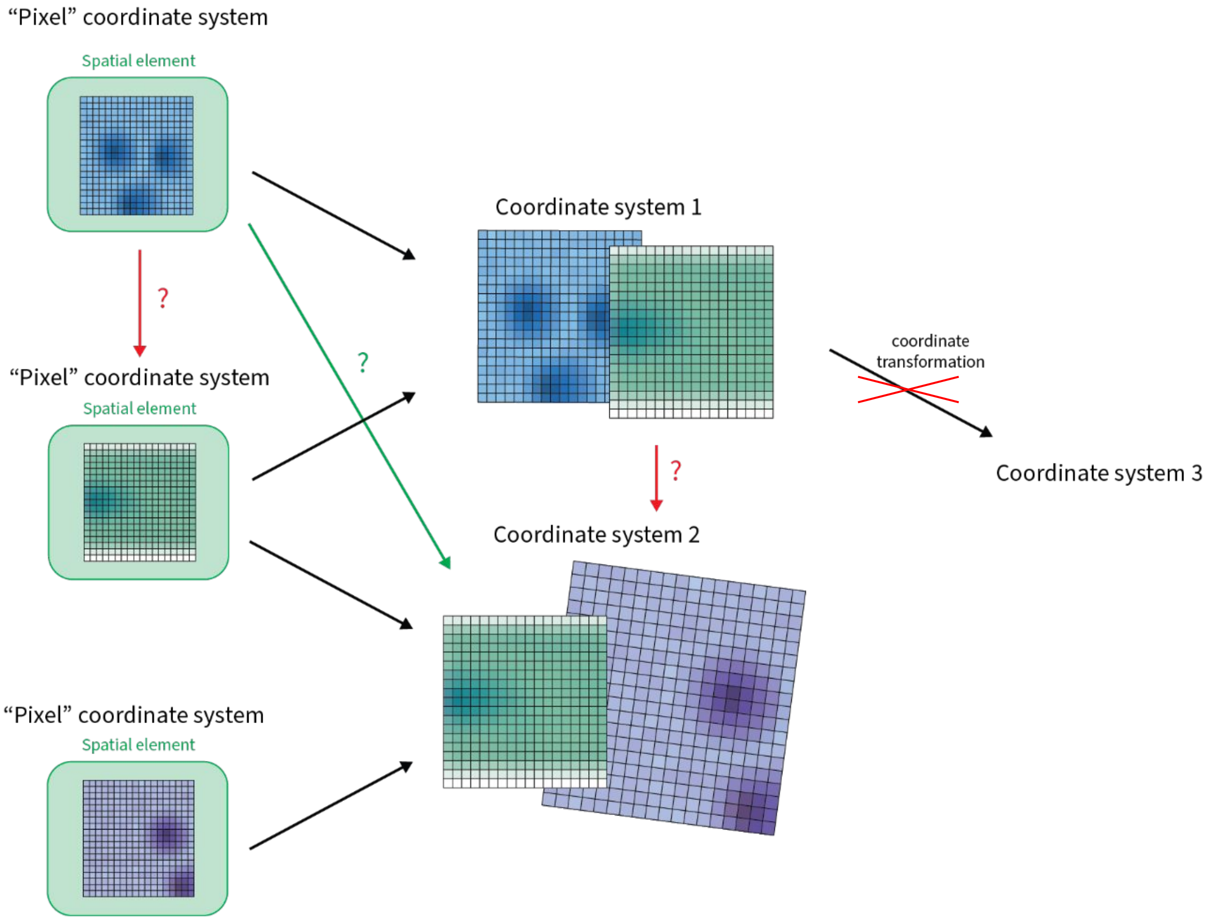
Limitations of transformations and the new implementation

- We can’t define the red transformations
- Still, we compute their values from existing transformations

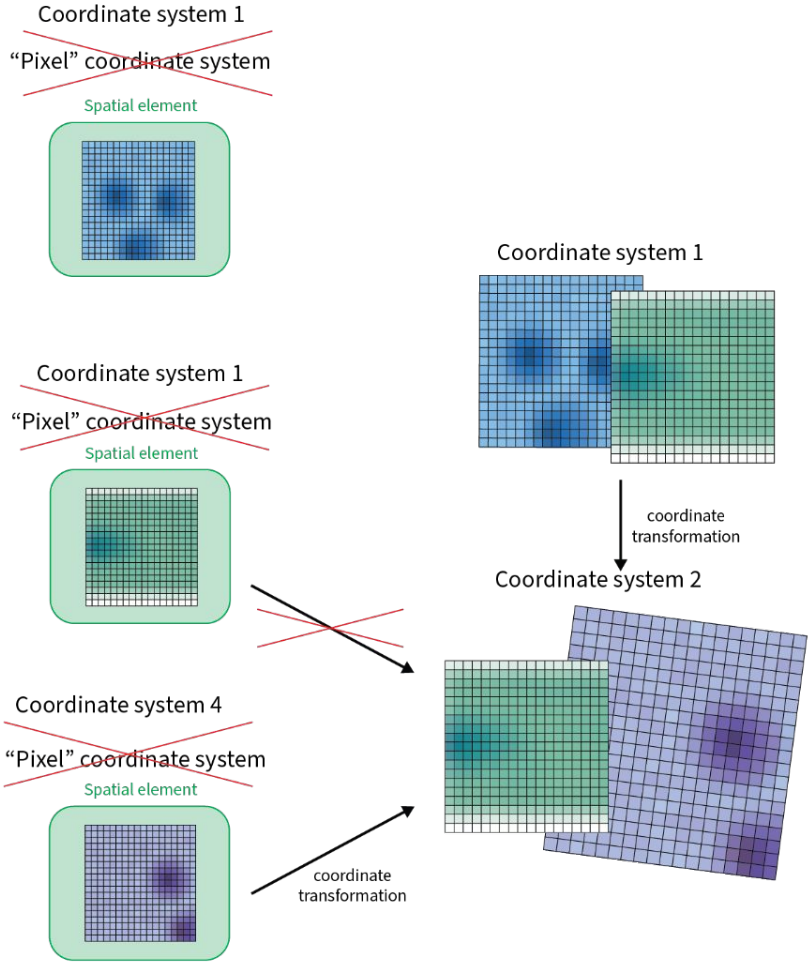




Limitations of transformations and the new implementation



Limitations of transformations and the new implementation



A bridge: NGFF transformations ↔ Xarray coordinates

The new specs maintain the old v0.4 transformations (scale+translation), used for multi-scale images.

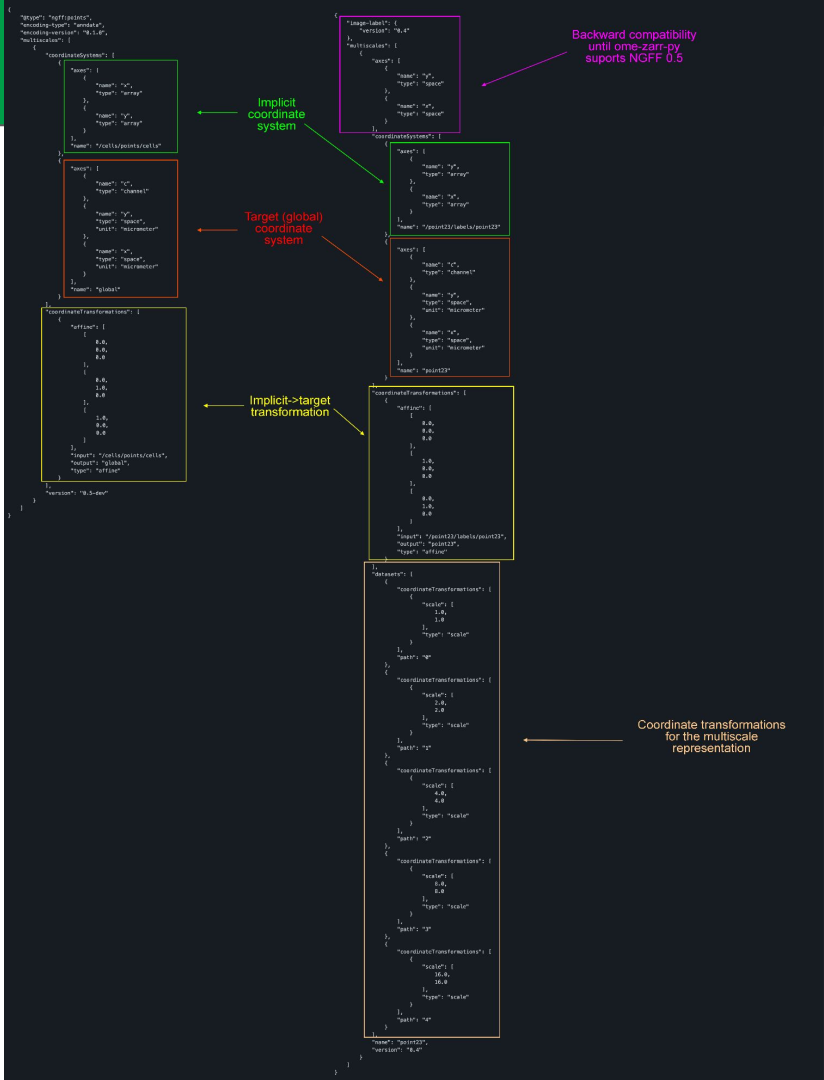
The scale+translation info represents the:

- "Canonical scale for the data"
- "Canonical orientation for the data"
- "Canonical origin for the data"

Implementation:

- After `read()` turn scale+translation into xarray coordinates.
- Before `write()` turn the xarray coordinates into scale+translation.
- `transform()` modifies the data only if necessary E.g. rotation, but not for scale or translation.

- Bonus: scale+translation can be allowed also for vector data; this can be used to define the "canonical orientation"



Conclusions and proposal on how to proceed

What we implemented:

- Ergonomic APIs that address our spatial omics use cases:
- On-disk, still NGFF
- Transformations also for points and shapes

Proposal, move code out of `spatialdata` into a more general repository:

- Tier 1: `NGFFBaseTransformation` (in particular read-write APIs)
    - could add a pydantic model
- Tier 2: `BaseTransformation` (i.e. ergonomic APIs)
- Tier 3: `transform()`
    - generalize to arbitrary axes
- Extra:
    - Converters between other formats (ITK, matplotlib, napari, …)

Announcing the first scverse conference!

Conclusions and acknowledgements

- established interoperable format for spatial omics based on OME-NGFF
- in-memory multimodal representation
- processing, visualization
- scales to large datasets


Luca Marconato


Giovanni Palla


Kevin Yamauchi

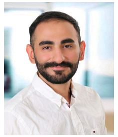

Isaac Virshup


Wouter-Michiel Vierdag


Tim Treis


Josh Moore

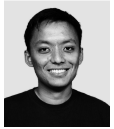

Sonja Stockhaus


Elyas Heidari


Marcela Toth


Quentin Blampey

Laurens Lehner


Rahul B. Shrestha


Benjamin Rombaut


Lotte Pollaris


Harald Vöringer


Oliver Stegle


Fabian Theis


Moritz Gerstung


Sinem Saka


Yvan Saeys


Wolfgang Huber

Nature Methods, 2024

First authors are underlined

EMBL

Oliver Stegle
Luca Marconato
Sinem Saka
Wouter-Michiel Vierdag
Wolfgang Huber
Harald Vöheringer
Constantin Ahlmann-Eltze
Mike Smith

Helmholtz Munich

Fabian Theis
Giovanni Palla
Isaac Virshup
Tim Treis
Sonja Stockhaus
Laurens Lehner
Marcella Toth
Rahul Shrestha

DKFZ

Ilia Kats
Tobias Graf
Moritz Gerstung
Elyas Heidari

Josh Moore (OME, GerBi)
Kevin Yamauchi (ETH)
Yvan Saeys (UGhent)

Lotte Pollaris (UGhent)
Benjamin Rombaut (UGhent)
Christian Tischer (EMBL)
Andreas S. Eisenbarth (EMBL)
Omer Bayraktar (Sanger)
Tong Li (Sanger)
Ilan Gold (Helmholtz)
Helena Crowell (CNAG)

Nils Eiling (UZH)
Will Moore (OME, UDundee)
Quentin Blampey (UParis Saclay)
Florian Wünnemann (UKHD)
Mark Keller (HMS)
10x Genomics team
CZI cellxgene team
…

EMBL dkfz. TUM HELMHOLTZ MUNICH UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386 ETH

spatialdata.scverse.org

Funded by
CHAN ZUCKERBERG INITIATIVE