

# Areal Data: Vector Data

HES 505 Fall 2024: Session 7

Carolyn Koehn

**Today's Plan**

The map displays the Hyde Park neighborhood in Chicago, bounded by the University of Chicago to the south and the city limits to the north. A thick black line highlights a specific area within the neighborhood, starting from the University of Chicago and extending north towards the city limits. The map includes various streets, parks, and schools. Key streets shown include E 47th St, E 48th St, E 49th St, E 50th St, E 51st St, E 52nd St, E 53rd St, E 54th St, E 55th St, E 56th St, E 57th St, E 58th St, E 59th St, E 60th St, E 61st St, E 62nd St, E 63rd St, E 64th St, E 65th St, E 66th St, E 67th St, E 68th St, E 69th St, E 70th St, E 71st St, E 72nd St, E 73rd St, E 74th St, E 75th St, E 76th St, E 77th St, E 78th St, E 79th St, E 80th St, E 81st St, E 82nd St, E 83rd St, E 84th St, E 85th St, E 86th St, E 87th St, E 88th St, E 89th St, E 90th St, E 91st St, E 92nd St, E 93rd St, E 94th St, E 95th St, E 96th St, E 97th St, E 98th St, E 99th St, E 100th St. Parks shown include Washington Park, Morgan Park, and Hyde Park. Schools shown include Harvard Elementary School, Lincoln Elementary School, and the University of Chicago. The text 'Today's Plan' is overlaid in large, dark green letters across the center of the map.

# Objectives

By the end of today, you should be able to:

- Understand **predicates** and **measures** in the context of spatial operations in **sf**
- Define valid geometries and approaches for assessing geometries in **R**
- Use **st\_\*** and **sf\_\*** to evaluate attributes of geometries and calculate measurements

# Understanding the language

# Revisiting Simple Features

- The **sf** package relies on a simple feature data model to represent geometries
  - hierarchical
  - standardized methods
  - complementary binary and human-readable encoding

type	description
<b>POINT</b>	single point geometry
<b>MULTIPOINT</b>	set of points
<b>LINestring</b>	single linestring (two or more points connected by straight lines)
<b>MULTILINestring</b>	set of linestrings
<b>POLYGON</b>	exterior ring with zero or more inner rings, denoting holes
<b>MULTIPOLYGON</b>	set of polygons
<b>GEOMETRYCOLLECTION</b>	set of the geometries above

# Revisiting Simple Features

- You already know how to access some elements of a simple feature
- `st_crs` - returns the coordinate reference system
- `st_bbox` - returns the bounding box for the simple feature

# Standardized Methods

We can categorize **sf** operations based on what they return and/or how many geometries they accept as input.

- *Output Categories*

- **Predicates:** evaluate a logical statement asserting that a property is **TRUE**
- **Measures:** return a numeric value with units based on the units of the CRS
- **Transformations:** create new geometries based on input geometries.

- *Input Geometries*

- **Unary:** operate on a single geometry at a time (meaning that if you have a **MULTI\*** object the function works on each geometry individually)
- **Binary:** operate on pairs of geometries
- **n-ary:** operate on sets of geometries

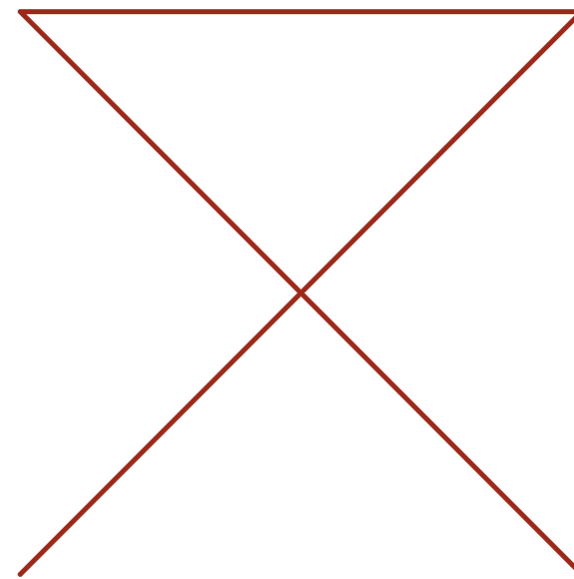
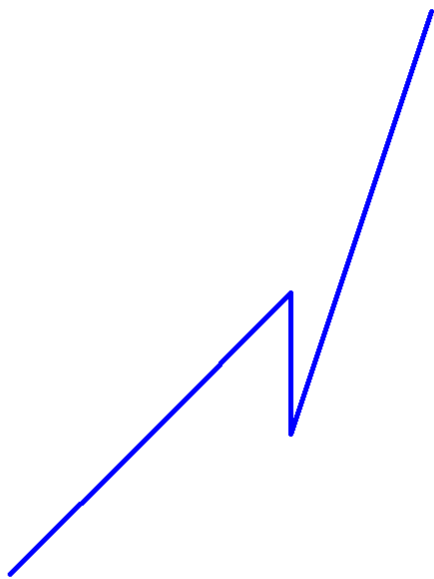
# Valid Geometries



# Remembering Valid Geometries

- A **linestring** is *simple* if it does not intersect

```
1 library(sf)
2 library(tidyverse)
3 ls = st_linestring(rbind(c(0,0), c(1,1), c(2,2), c(2,1), c(3,4)))
4
5 ls2 = st_linestring(rbind(c(0,0), c(1,1), c(2,2), c(0,2), c(1,1),
```

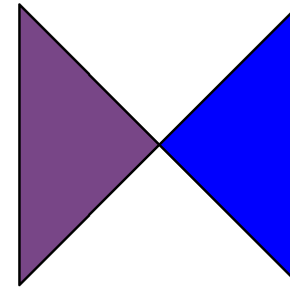
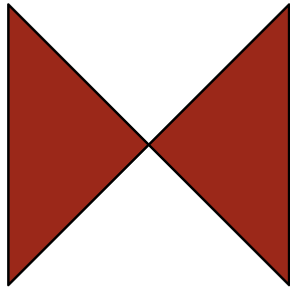


# Remembering Valid Geometries

- Valid polygons
  - Are closed (i.e., the last vertex equals the first)
  - Have holes (inner rings) that inside the the exterior boundary
  - Have holes that touch the exterior at no more than one vertex (they don't extend across a line)
    - For multipolygons, adjacent polygons touch only at points
  - Do not repeat their own path

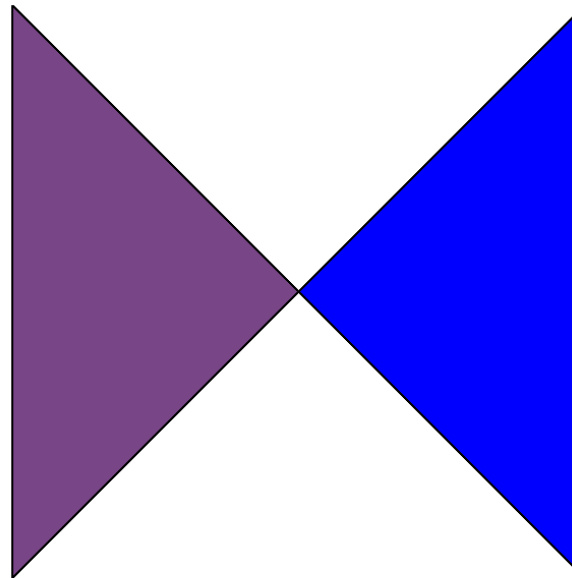
# Remembering Valid Geometries

```
1 p1 = st_as_sfc("POLYGON((0 0, 0 10, 10 0, 10 10, 0 0))")
2 p2 = st_as_sfc("POLYGON((0 0, 0 10, 5 5, 0 0))")
3 p3 = st_as_sfc("POLYGON((5 5, 10 10, 10 0, 5 5))")
```



# Remembering Valid Geometries

```
1 p4 = st_as_sfc(c("POLYGON((0 0, 0 10, 5 5, 0 0))", "POLYGON((5 5, 5 10, 0 10, 0 0))"))
2 plot(p4, col=c("#7C4A89", "blue"))
```



# Empty Geometries

- Empty geometries arise when an operation produces **NULL** outcomes (like looking for the intersection between two non-intersecting polygons)
- **sf** allows empty geometries to make sure that information about the data type is retained
- Similar to a **data.frame** with no rows or a **list** with **NULL** values
- Most vector operations require simple, valid geometries

# Predicates

# Using Unary Predicates

- Unary predicates accept single geometries (or geometry collections)
- Provide helpful ways to check whether your data is ready to analyze
- Use the **st\_** prefix and return **TRUE/FALSE**

predicate	asks...
<b>is_simple</b>	is the geometry self-intersecting (i.e., simple)?
<b>is_valid</b>	is the geometry valid?
<b>is_empty</b>	is the geometry column of an object empty?
<b>is_longlat</b>	does the object have geographic coordinates? ( <b>FALSE</b> if coords are projected, <b>NA</b> if no <b>crs</b> )
<b>is(geometry, class)</b>	is the geometry of a particular class?

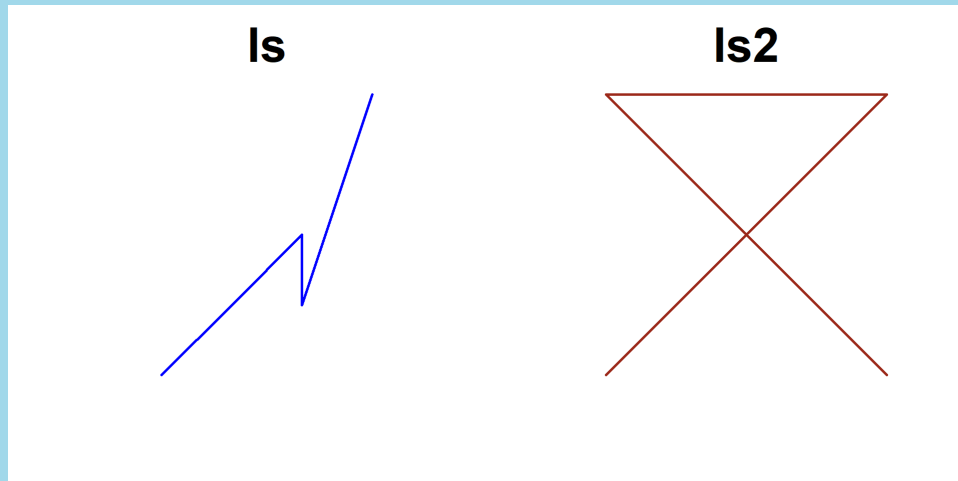


# Checking Geometries With Unary Predicates

- Before conducting costly analyses, it's worth checking for:
  1. empty geometries, using `any(st_is_empty(x))`
  2. corrupt geometries, using `any(is.na(st_is_valid(x)))`
  3. invalid geometries, using `any(na.omit(st_is_valid(x)) == FALSE)`; in case of corrupt and/or invalid geometries,
  4. in case of invalid geometries, query the reason for invalidity by `st_is_valid(x, reason = TRUE)`

Invalid geometries will require **transformation** (next week!)

# Checking Geometries With Unary Predicates

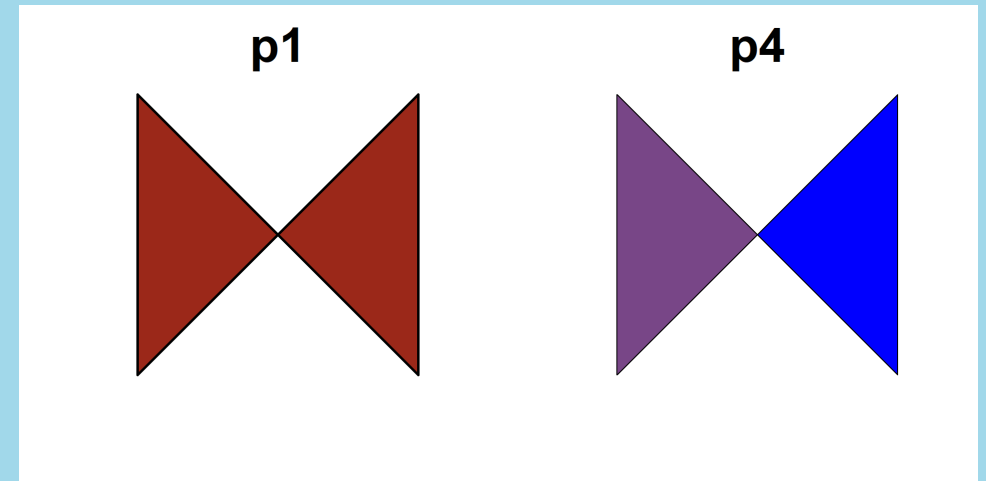


```
1 st_is_simple(ls)
```

```
[1] TRUE
```

```
1 st_is_simple(ls2)
```

```
[1] FALSE
```



```
1 st_is_valid(p1)
```

```
[1] FALSE
```

```
1 st_is_valid(p4)
```

```
[1] TRUE TRUE
```

# Unary Predicates and Real Data

```
1 library(tigris)
2 id.cty <- counties
3
4 st_crs(id.cty)$ing
```

```
[1] "NAD83"
```

```
1 st_is_longlat(id.cty)
```

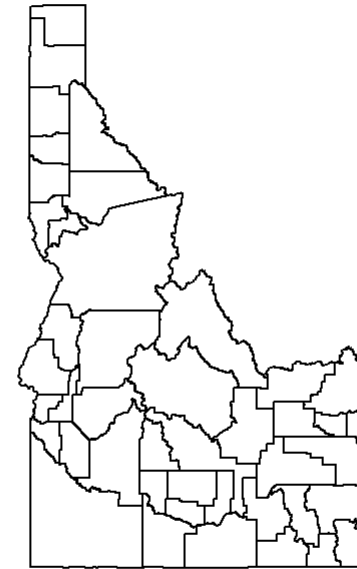
```
[1] TRUE
```

```
1 st_is_valid(id.cty)
```

```
[1] TRUE TRUE TRUE TRUE TRUE
```

```
1 all(st_is_valid(id.cty))
```

```
[1] TRUE
```



# Binary Predicates

# Binary Predicates

- Accept exactly two geometries (or collections)
- Also return **logical** outcomes
- Based on the Dimensionally Extended 9-Intersection Model (DE-9IM)

predicate	meaning	inverse of
<b>contains</b>	None of the points of A are outside B	<b>within</b>
<b>contains_properly</b>	A contains B and B has no points in common with the boundary of A	
<b>covers</b>	No points of B lie in the exterior of A	<b>covered_by</b>
<b>covered_by</b>	Inverse of <b>covers</b>	
<b>crosses</b>	A and B have some but not all interior points in common	
<b>disjoint</b>	A and B have no points in common	<b>intersects</b>
<b>equals</b>	A and B are topologically equal: node order or number of nodes may differ; identical to A contains B AND A within B	
<b>equals_exact</b>	A and B are geometrically equal, and have identical node order	
<b>intersects</b>	A and B are not disjoint	<b>disjoint</b>
<b>is_within_distance</b>	A is closer to B than a given distance	
<b>within</b>	None of the points of B are outside A	<b>contains</b>
<b>touches</b>	A and B have at least one boundary point in common, but no interior points	
<b>overlaps</b>	A and B have some points in common; the dimension of these is identical to that of A and B	
<b>relate</b>	given a mask <b>pattern</b> , return whether A and B adhere to this pattern	

# Binary Predicates

```
1 id <- states(progr  
2   filter(STUSPS ==  
3 or <- states(progr  
4   filter(STUSPS ==  
5 ada.cty <- id.cty  
6   filter(NAME == '
```

```
1 st_covers(id, ada.cty)
```

Sparse geometry binary predicate list of length 1, where the predicate was `covers`

```
1: 1
```

```
1 st_covers(id, ada.cty, sparse=FALSE)
```

```
[,1]
```

```
[1,] TRUE
```

```
1 st_within(ada.cty, or)
```

Sparse geometry binary predicate list of length 1, where the predicate was `within`

```
1: (empty)
```

```
1 st_within(ada.cty, or, sparse=FALSE)
```

```
[,1]
```

```
[1,] FALSE
```

# Measures

# Measures

## Unary Measures

- Return quantities of individual geometries

measure	returns
<code>dimension</code>	0 for points, 1 for linear, 2 for polygons, possibly <code>NA</code> for empty geometries
<code>area</code>	the area of a geometry
<code>length</code>	the length of a linear geometry

## Unary Measures

```
1 st_area(id)
```

```
2.15994e+11 [m^2]
```

```
1 st_area(id.cty[1:5,])
```

```
Units: [m^2]
```

```
[1] 2858212132 3380630278 1459359818 1726660462 1223521586
```

```
1 st_dimension(id.cty[1:5,])
```

```
[1] 2 2 2 2 2
```



# Binary Measures

- **st\_distance** returns the distance between pairs of geometries

```
1 kootenai.cty <- id.cty %>%  
2   filter(NAME == "Kootenai")  
3 st_distance(kootenai.cty, ada.cty)
```

Units: [m]

```
      [,1]  
[1,] 396433.8
```

```
1 st_distance(id.cty) [1:5, 1:5]
```

Units: [m]

```
      [,1]      [,2]      [,3]      [,4]      [,5]  
[1,]      0.0 467635.7 277227.0 132998.0      0.0  
[2,] 467635.7      0.0 319706.4 656056.0 514306.9  
[3,] 277227.0 319706.4      0.0 377105.4 336146.8  
[4,] 132998.0 656056.0 377105.4      0.0 133045.5  
[5,]      0.0 514306.9 336146.8 133045.5      0.0
```

# Practice!

1. Create a vector object for Owyhee county in Idaho (hint: use `filter`).
2. Owyhee county is in Idaho and borders Oregon. Which two predicates communicate this information? Show your code and output.
3. Print the bounding box information for Oregon and Owyhee county. Which part of this output could clue you in that they border each other?

If you finish early, try the challenge on the next slide!

# Challenge

Which two counties in Idaho are furthest from each other?  
You will need to use spatial measures from this class as well as data navigation methods that you may need to look up!