Introduction to D3.js, Part II

By Rui Li
09/07/2021

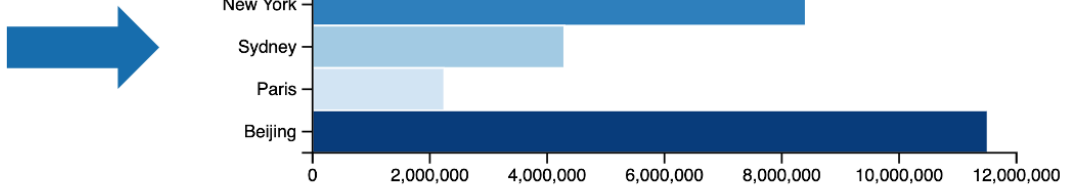
Slide Material Source Credits

- https://www.d3indepth.com/
- https://d3js.org/
- https://www.d3-graph-gallery.com/
- HTML tutorial
- Prof. Han-Wei Shen, Jiayi Xu, and Wenbin He

Recall

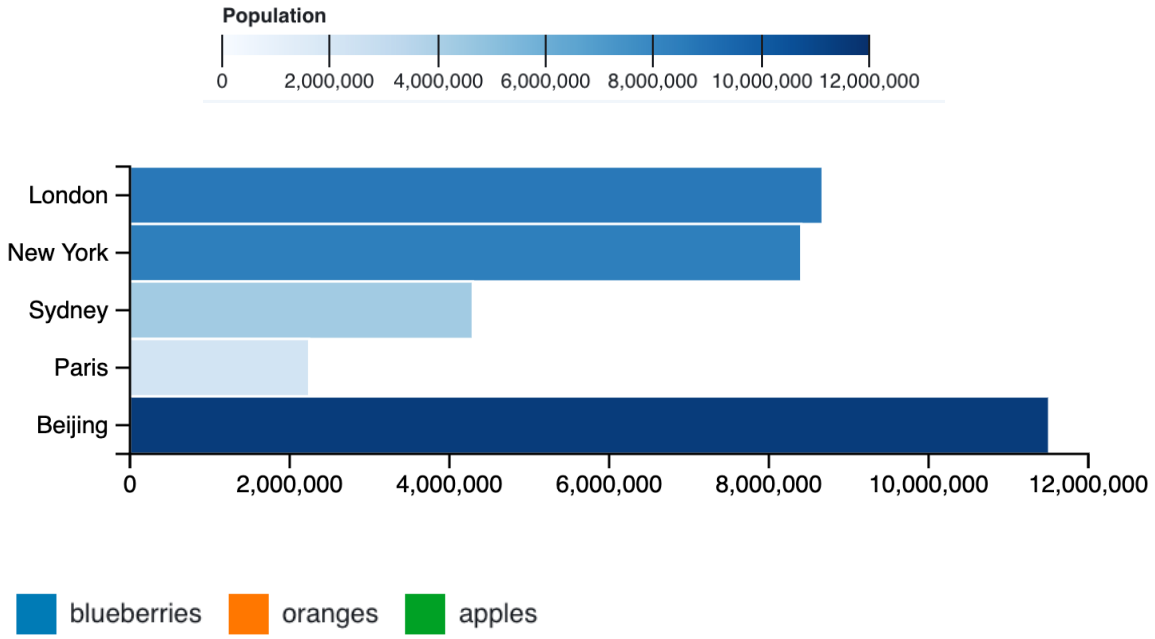
- D3 – introduction
- D3 basics
  - set up
  - data loading
  - selection
  - data binding
  - scales, color mapping
  - axis

	A	B
1	name	population
2	London	8674000
3	New York	8406000
4	Sydney	4293000
5	Paris	2244000
6	Beijing	11510000



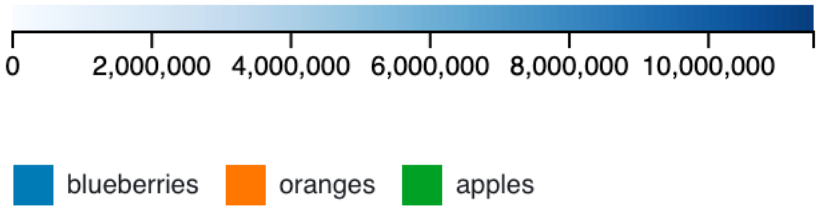
Color Legend

- From scratch
- Existing libraries
  - d3 color legend



Color Legend

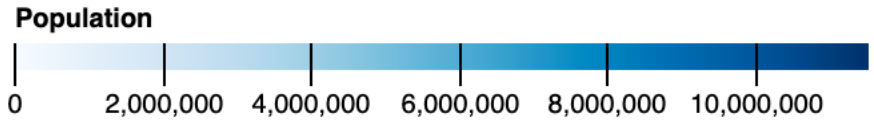
- Implement from scratch
  - render legend shapes (e.g., rectangles and circles)
  - render axis or text




```
let legendData = d3.range(0, d3.max(data.map(d => d.
population)), d3.max(data.map(d => d.population)) / 10);
let legendXScale = d3.scaleLinear()
    .domain([0, d3.max(data.map(d => d.population))])
    .range([0, 300])
d3.select('svg')
    .append("g")
    .attr("transform", "translate(50,250)")
    .selectAll('rect')
    .data(legendData)
    .join('rect')
    .attr('x', function (d, i) {
        return legendXScale(d);
    })
    .attr('width', 20)
    .attr('height', 10)
    .style('fill', function (d) {
        return colorScale(d);
    });

d3.select('svg').append("g")
    .attr("transform", "translate(50,260)")
    .call(d3.axisBottom(legendXScale).ticks(5))
```

Color Legend

- D3 – color legend - **continuous**
  - create an <g> element with an id inside your SVG
  - create the legend using the Legend function
  - append the legend to the <g> element




```
<svg>
    <g class="chart" transform="translate(50, 30)">
    </g>
    <g id="legend" transform="translate(150, 180)">

    </g>
</svg>
```

```
//continous legend
const legend = Legend(d3.scaleSequential([0, d3.max
(data.map(d => d.population))], d3.interpolateBlues), {
    title: "Population"
})

document.getElementById("legend").appendChild(legend);
```

Color Legend

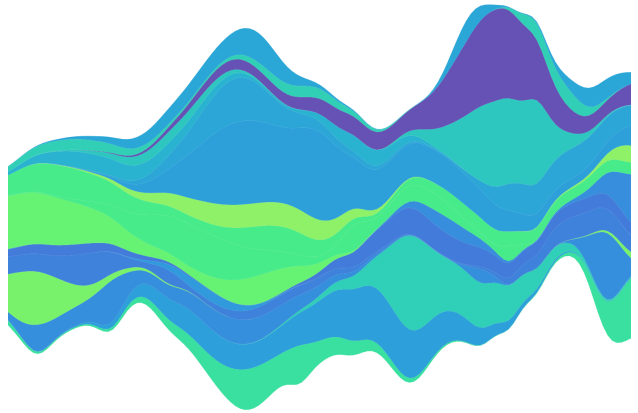
- D3 – color legend - **nominal**
  - create an element with an id
  - create the legend using the Swatches function
  - update the 's HTML content with the legend

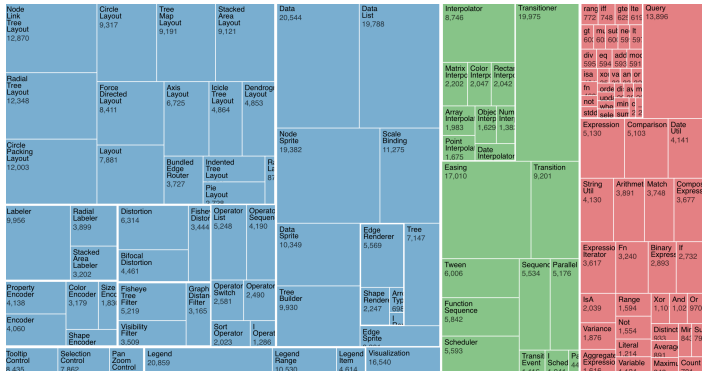
```
<div id="dis-legend">

</div>
<svg>
    <g class="chart" transform="translate(50, 30)">
```

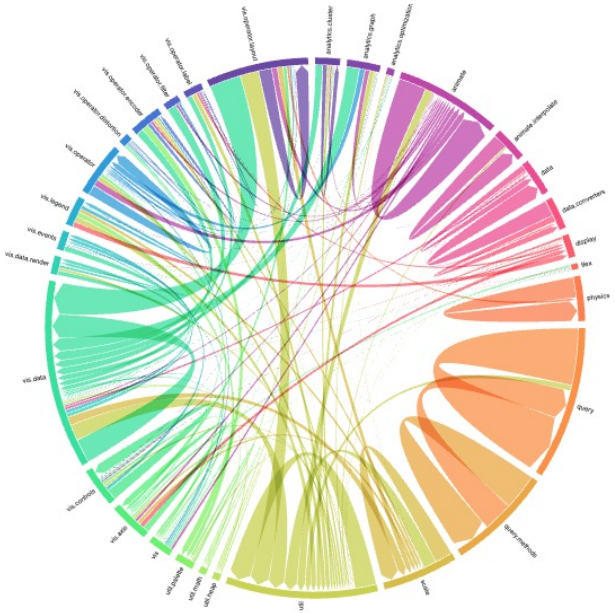

```
//discreate legend
const discreteLegend = Swatches(d3.scaleOrdinal(["blueberries",
"oranges", "apples"], d3.schemeCategory10));
d3.select("#dis-legend").html(discreteLegend);
```

Outline

- D3 shapes
  - SVG shapes
  - Line
  - Area
  - Arc
  - Symbol
- D3 layouts
  - Pie
  - Stack
  - Hierarchy
  - Chord
  - Force





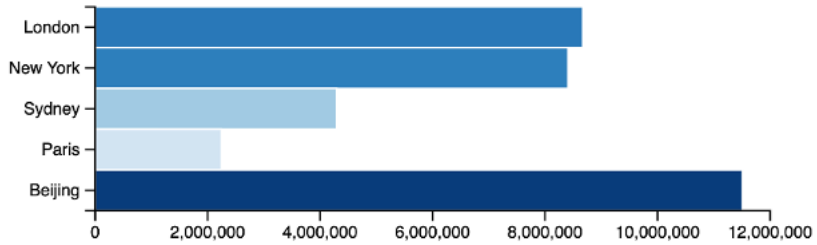


D3 SVG shapes

In D3, we could create an SVG shape directly.

- rect (rectangle)
- circle
- line

```
d3.select('.chart')
    .selectAll('rect')
    .data(data)
    .join('rect')
    .attr("x", 80)
    .attr("y", function (d, i) {
        return yScale(d.name);
    })
    .attr("width", function (d, i) {
        return xScale(d.population);
    })
    .attr("height", yScale.bandwidth())
    .style("fill", function (d, i) {
        return colorScale(d.population);
    })
    .style("stroke", "white");
```



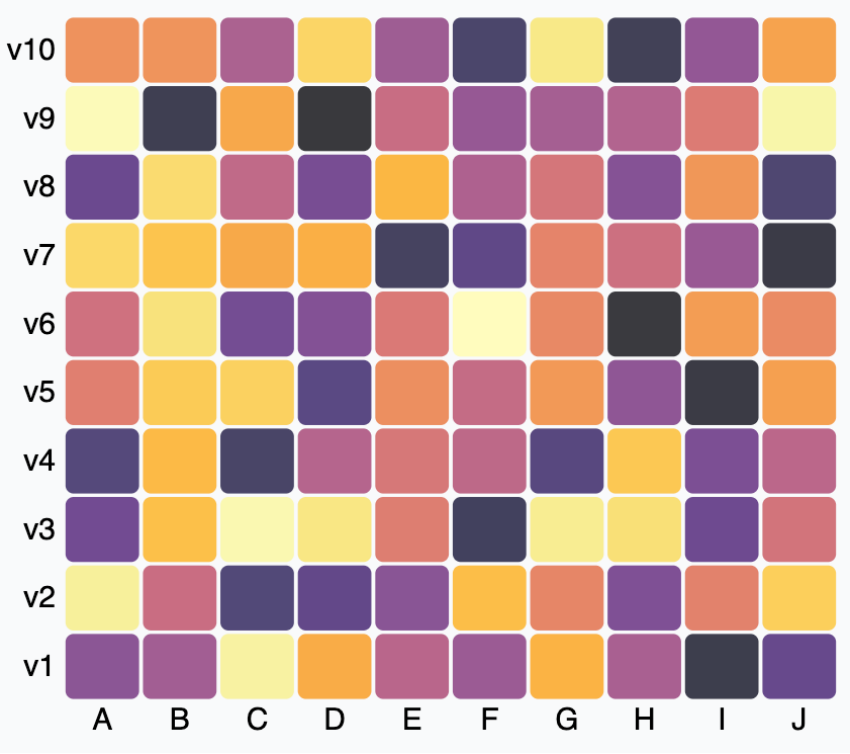
D3 SVG shapes - Rectangle

```
<rect style="fill: #69b3a2" stroke="black" x=10 y=100, width=300 height=40></rect>
```

```
// create svg element:
var svg = d3.select("#rect").append("svg").attr("width", 800).attr("height", 200)

// Add the path using this helper function
svg.append('rect')
  .attr('x', 10)
  .attr('y', 100)
  .attr('width', 300)
  .attr('height', 40)
  .attr('stroke', 'black')
  .attr('fill', '#69a3b2');
```

D3 SVG shapes - Rectangle

Example: Heatmap




```
svg.selectAll()
      .data(data, function(d) {return d.group+':'+d.variable;})
      .join("rect")
      .attr("x", function(d) { return x(d.group) })
      .attr("y", function(d) { return y(d.variable) })
      .attr("width", width )
      .attr("height", height )
      .style("fill", function(d) { return myColor(d.value)} )
```

Demo

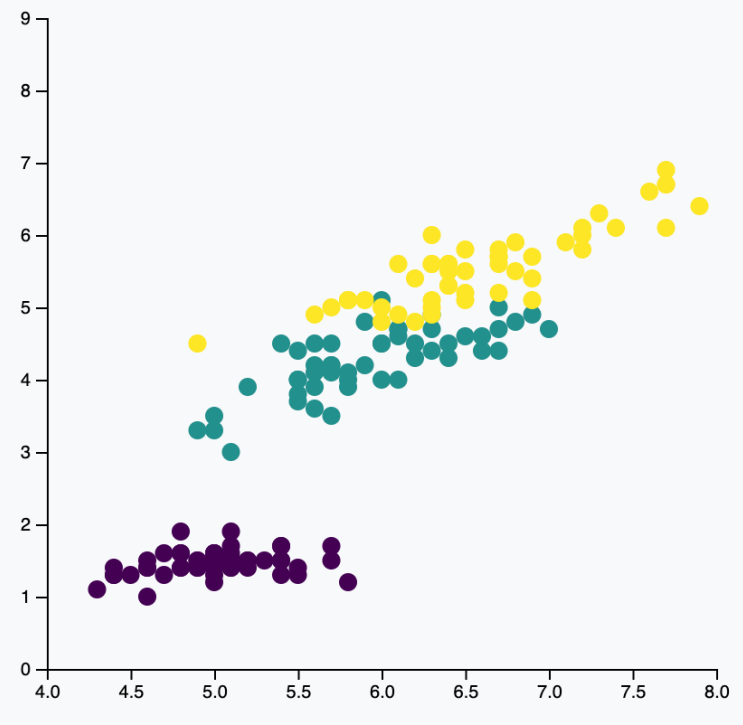
D3 SVG shapes - Circle

```
<circle style="fill: #69b3a2" stroke="black" cx=100 cy=100 r=40></circle>
```

```
svg.append('circle')
  .attr('cx', 100)
  .attr('cy', 100)
  .attr('r', 40)
  .attr('stroke', 'black')
  .attr('fill', '#69a3b2');
```

D3 SVG shapes - Circle

Example: Scatterplot

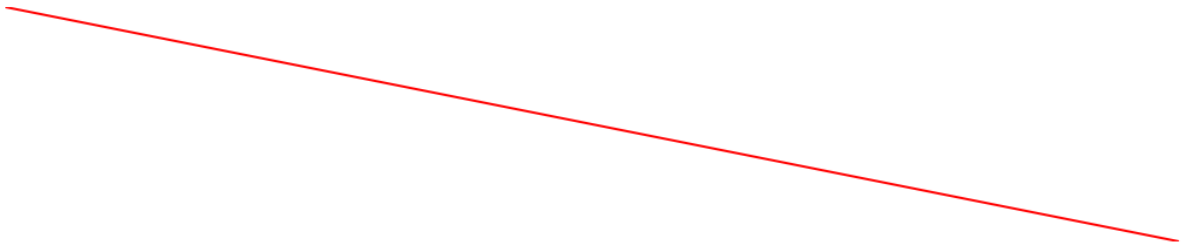



```
svg.append('g')
    .selectAll("dot")
    .data(data)
    .join("circle")
      .attr("cx", function (d) { return x(d.Sepal_Length); } )
      .attr("cy", function (d) { return y(d.Petal_Length); } )
      .attr("r", 5)
      .style("fill", function (d) { return color(d.Species) } )
```

Demo

D3 SVG shapes - Line

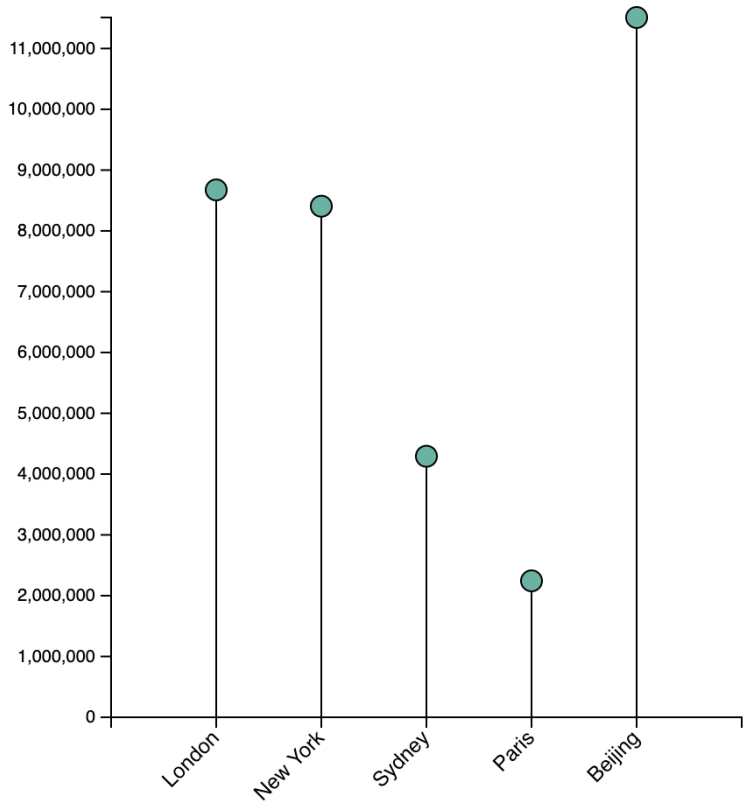
```
<line stroke="red" x0=10 y0=10, x1=500 y1=100></line>
```



```
svg.append('line')
  .attr('x1', 10)
  .attr('y1', 10)
  .attr('x2', 500)
  .attr('y2', 100)
  .attr('stroke', 'red')
```

D3 SVG shapes - Line

Example: Lollipop chart




```
// Lines in lollipop
svg
    .selectAll("myline")
    .data(data)
    .enter()
    .append("line")
    .attr("x1", function (d) {
        return x(d.name);
    })
    .attr("x2", function (d) {
        return x(d.name);
    })
    .attr("y1", function (d) {
        return y(d.population);
    })
    .attr("y2", y(0))
    .attr("stroke", "black")     // set the line colour
    .style("stroke-width", 1)    // set the stroke width
```

Demo

D3 shapes

In d3, shapes are made up of SVG path elements (d3.path)

The path element has a d attribute which defines the shape of the path.

```
moveTo(20, 20)
lineTo(120, 20)
//<path d="M 20 20 L 120 120">
```

Lines – Line generator

- lineGenerator is a function that takes **an array of coordinates** as input and outputs a **path data string**

```
var lineGenerator = d3.line();

var points = [
  [0, 80],
  [100, 100],
  [200, 30],
  [300, 50],
  [400, 40],
  [500, 80]
];

var pathData = lineGenerator(points);

d3.select('path')
  .attr('d', pathData)
  .attr('fill', 'none')
  .attr('stroke', 'black')
```

- Constructs a new line generator

Lines – Line generator

- lineGenerator is a function that takes **an array of coordinates** as input and outputs a **path data string**

```
var lineGenerator = d3.line();

var points = [
  [0, 80],
  [100, 100],
  [200, 30],
  [300, 50],
  [400, 40],
  [500, 80]
];

var pathData = lineGenerator(points);

d3.select('path')
  .attr('d', pathData)
  .attr('fill', 'none')
  .attr('stroke', 'black')
```

- Define an array of coordinates

Lines – Line generator

- lineGenerator is a function that takes **an array of coordinates** as input and outputs a **path data string**

```
var lineGenerator = d3.line();

var points = [
  [0, 80],
  [100, 100],
  [200, 30],
  [300, 50],
  [400, 40],
  [500, 80]
];

var pathData = lineGenerator(points);

d3.select('path')
  .attr('d', pathData)
  .attr('fill', 'none')
  .attr('stroke', 'black')
```

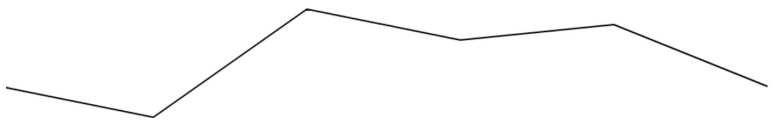
- Now call lineGenerator, passing in our data points
- *pathData is "M0,80L100,100L200,30L300,50L400,40L500,80"*
  - A path string for SVG to draw a line

Lines – Line generator

- lineGenerator is a function that takes **an array of coordinates** as input and outputs a **path data string**

```
var lineGenerator = d3.line();

var points = [
  [0, 80],
  [100, 100],
  [200, 30],
  [300, 50],
  [400, 40],
  [500, 80]
];

var pathData = lineGenerator(points);

d3.select('path')
  .attr('d', pathData)
  .attr('fill', 'none')
  .attr('stroke', 'black')
```

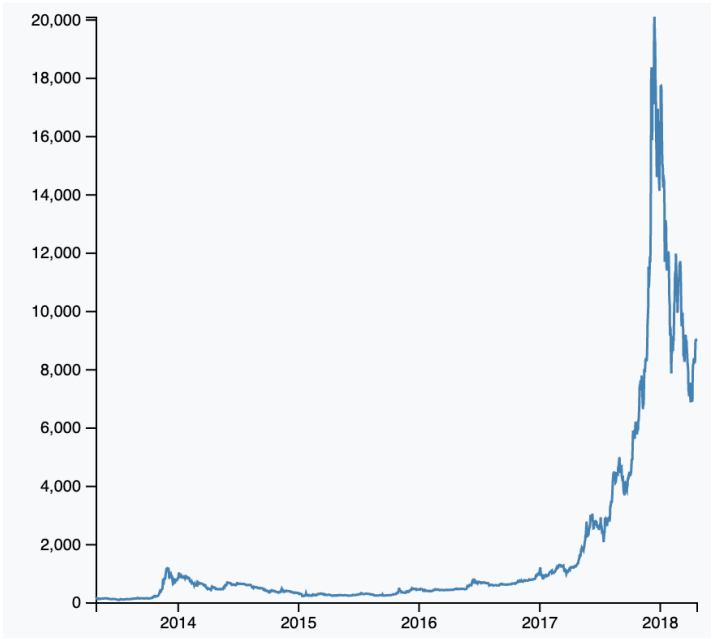
- Draw the line



Lines – Create a line chart

- Data

```
1   date,value
2   2013-04-28,135.98
3   2013-04-29,147.49
4   2013-04-30,146.93
5   2013-05-01,139.89
6   2013-05-02,125.6
7   2013-05-03,108.13
8   2013-05-04,115
9   2013-05-05,118.8
10  2013-05-06,124.66
```



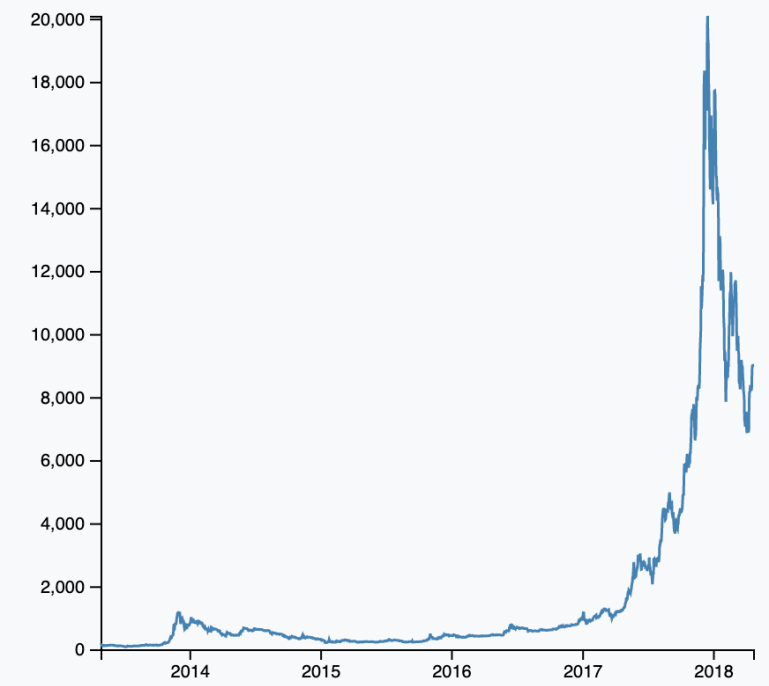
Lines – Create a line chart

Scale

- xScale: Date to width
- yScale: Price to height

Line generator

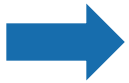
- Tell the generator how to map data [*date*, *price*] to coordinates [*x*, *y*]



Time Parser

```
//load the data
const parseTime = d3.timeParse("%Y-%m-%d");
function convertRow(d) {
    return {
        date: parseTime(d.date),
        value: +d.value
    }
}

const data = await d3.csv("datasets/linechart.csv", convertRow);
```

```
1   date,value
2   2013-04-28,135.98
3   2013-04-29,147.49
4   2013-04-30,146.93
5   2013-05-01,139.89
6   2013-05-02,125.6
7   2013-05-03,108.13
8   2013-05-04,115
```



```
▶ 0: {date: Sun Apr 28 2013 00:00:00 GMT-0400 (Eastern Daylight Time), value: 135.98}
▶ 1: {date: Mon Apr 29 2013 00:00:00 GMT-0400 (Eastern Daylight Time), value: 147.49}
▶ 2: {date: Tue Apr 30 2013 00:00:00 GMT-0400 (Eastern Daylight Time), value: 146.93}
▶ 3: {date: Wed May 01 2013 00:00:00 GMT-0400 (Eastern Daylight Time), value: 139.89}
▶ 4: {date: Thu May 02 2013 00:00:00 GMT-0400 (Eastern Daylight Time), value: 125.6}
▶ 5: {date: Fri May 03 2013 00:00:00 GMT-0400 (Eastern Daylight Time), value: 108.13}
▶ 6: {date: Sat May 04 2013 00:00:00 GMT-0400 (Eastern Daylight Time), value: 115}
▶ 7: {date: Sun May 05 2013 00:00:00 GMT-0400 (Eastern Daylight Time), value: 118.8}
▶ 8: {date: Mon May 06 2013 00:00:00 GMT-0400 (Eastern Daylight Time), value: 124.66}
```

Demo

Lines – Curve

- Draw a curve

```
line.curve(curveType)
```

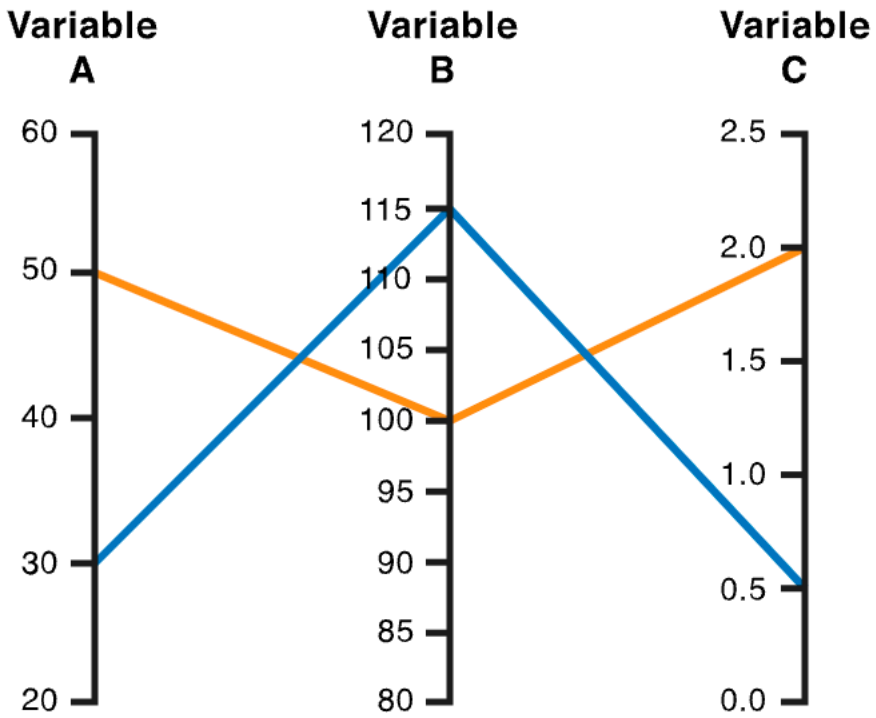
```
var lineGenerator = d3.line()
  .curve(d3.curveCardinal);
```

- Explore more curve types

Parallel coordinates plot (PCP)

- multivariate, quantitative data
- each variable is given an axis
- each axis can have a different scale
- values are plotted as a series of lines that are connected across all the axes
- how to draw PCP?

Data			
	Variable A	Variable B	Variable C
Item 1	50	100	2.0
Item 2	30	115	0.5



https://datavizcatalogue.com/methods/parallel_coordinates.html

Parallel coordinates plot (PCP)

- xScale.domain([0,1,2]).range([0,200])
- yScales
  - yScaleA
  - yScaleB
  - yScaleC
- data => [

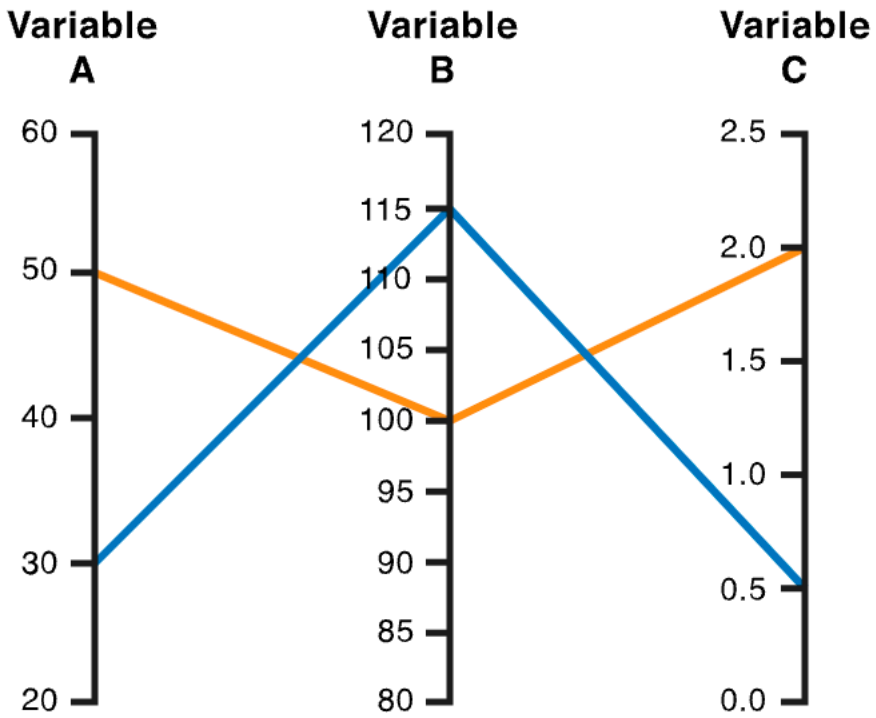
  [xScale(0), yScaleA(50)],

  [xScale(1), yScaleB(100)],

  [xScale(2), yScaleC(2.0)]

  ]

Data			
	Variable A	Variable B	Variable C
Item 1	50	100	2.0
Item 2	30	115	0.5



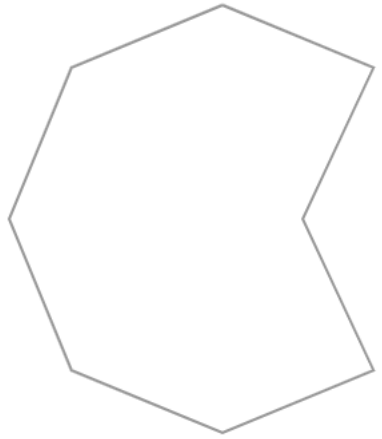
https://datavizcatalogue.com/methods/parallel_coordinates.html

Lines – Radial line

The radial line generator is similar to the line generator, but the points are formed by *angle* in radians (clockwise) and *radius*, rather than *x* and *y*

- Application: Radar graphs

```
var radialLineGenerator = d3.radialLine();

var points = [
  [0, 80],
  [Math.PI * 0.25, 80],
  [Math.PI * 0.5, 30],
  [Math.PI * 0.75, 80],
  [Math.PI, 80],
  [Math.PI * 1.25, 80],
  [Math.PI * 1.5, 80],
  [Math.PI * 1.75, 80],
  [Math.PI * 2, 80]
];

var radialLine = radialLineGenerator(points);
```



Area

The area generator outputs path that defines an area between two lines.

- Data can be encoded into *coordinates* on the two lines
- Application: Stream graphs, filled line charts

```
var areaGenerator = d3.area();

var points = [
  [0, 80],
  [100, 100],
  [200, 30],
  [300, 50],
  [400, 40],
  [500, 80]
];

var pathData = areaGenerator(points);
```

Area

The area generator outputs path that defines an area between two lines.

- .y0 and .y1 methods

```
var points = [
  {x: 0, low: 30, high: 80},
  {x: 100, low: 80, high: 100},
  {x: 200, low: 20, high: 30},
  {x: 300, low: 20, high: 50},
  {x: 400, low: 10, high: 40},
  {x: 500, low: 50, high: 80}
];

var areaGenerator = d3.area()
  .x(function(d) {
    return d.x;
  })
  .y0(function(d) {
    return d.low;
  })
  .y1(function(d) {
    return d.high;
  });

var area = areaGenerator(points);

// Create a path element and set its d attribute
d3.select('g')
  .append('path')
  .attr('d', area);
```

Area - radialArea

The radial area generator is similar to the area generator, but the points are formed by *angle* in radians (clockwise) and *radius*, rather than *x* and *y*

- Application: Filled radar graphs

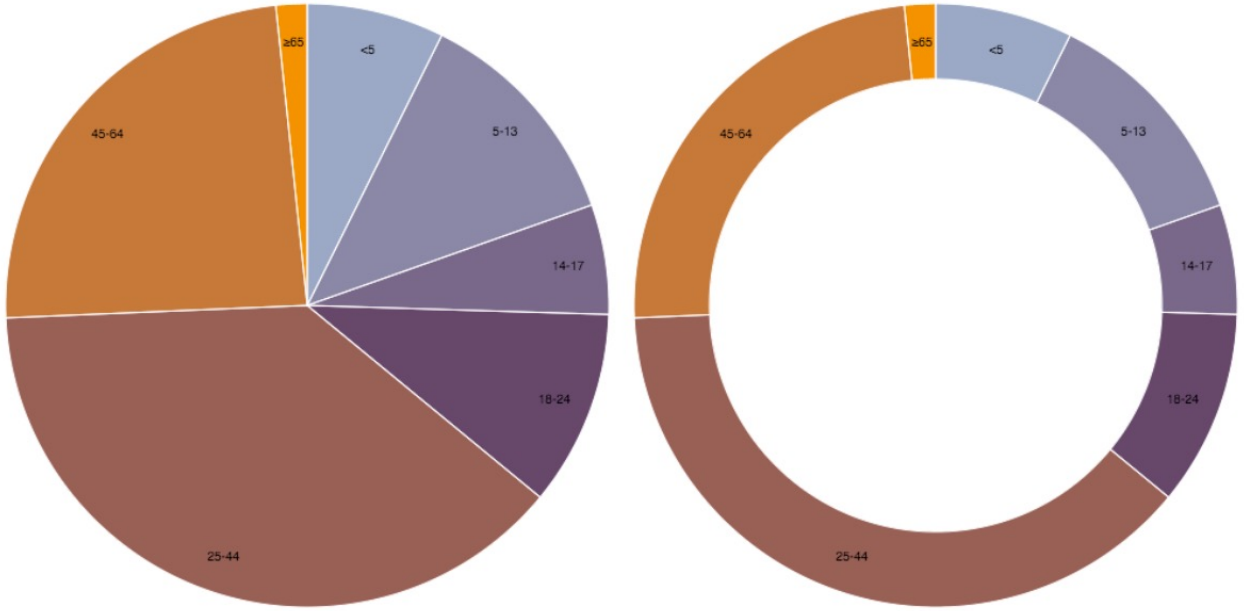
```
var points = [
  {angle: 0, r0: 30, r1: 80},
  {angle: Math.PI * 0.25, r0: 30, r1: 70},
  {angle: Math.PI * 0.5, r0: 30, r1: 80},
  {angle: Math.PI * 0.75, r0: 30, r1: 70},
  {angle: Math.PI, r0: 30, r1: 80},
  {angle: Math.PI * 1.25, r0: 30, r1: 70},
  {angle: Math.PI * 1.5, r0: 30, r1: 80},
  {angle: Math.PI * 1.75, r0: 30, r1: 70},
  {angle: Math.PI * 2, r0: 30, r1: 80}
];
```

```
var radialAreaGenerator = d3.radialArea()
  .angle(function(d) {
    return d.angle;
  })
  .innerRadius(function(d) {
    return d.r0;
  })
  .outerRadius(function(d) {
    return d.r1;
  });
```

Arc

Arc generators produce path data from *angle* and *radius* values

- Data can be encoded into *angle* and *radius*
- Application: Pie Chart, Donut Chart



Arc

- Example

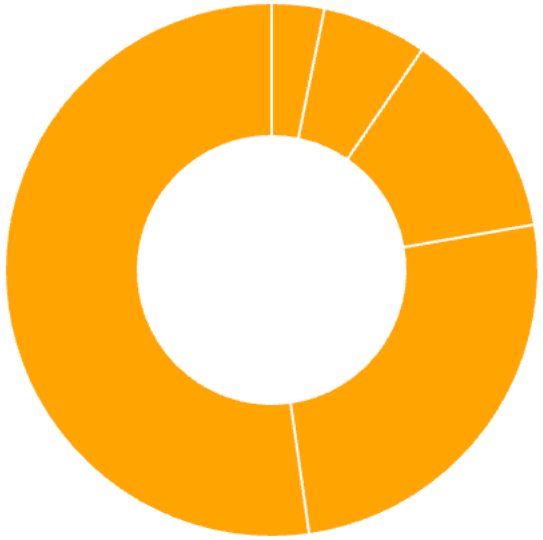
```
var arcGenerator = d3.arc();

var pathData = arcGenerator({
  startAngle: 0,
  endAngle: 0.25 * Math.PI,
  innerRadius: 50,
  outerRadius: 100
});

d3.select('g')
  .append('path')
  .attr('d', pathData)
  .attr('fill', 'orange')
```

The *angle* is specified in radians, with 0 at -*y* (12 o'clock) and positive angles proceeding clockwise.

Arc – multiple arcs

- donut chart

```
var arcGenerator = d3.arc()
  .innerRadius(20)
  .outerRadius(100)

var arcData = [
  {startAngle: 0, endAngle: 0.2},
  {startAngle: 0.2, endAngle: 0.6},
  {startAngle: 0.6, endAngle: 1.4},
  {startAngle: 1.4, endAngle: 3},
  {startAngle: 3, endAngle: 2* Math.PI}
];

d3.select('g')
  .selectAll('path')
  .data(arcData)
  .join('path')
  .attr('d', arcGenerator);
```



Symbols

The symbol generator produces path data for symbols

- example

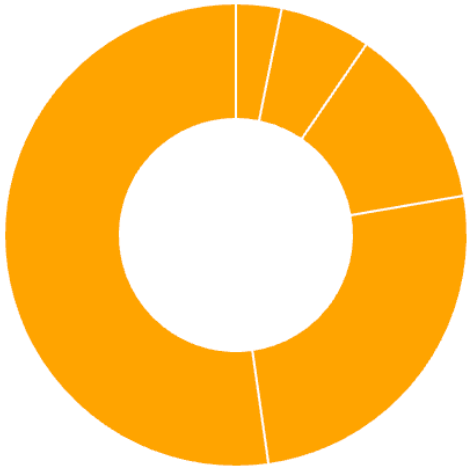
```
var symbolGenerator = d3.symbol()
  .type(d3.symbolStar)
  .size(80);

d3.select('g')
  .append('path')
  .attr('transform', 'translate(20,20)')
  .attr('d', symbolGenerator);
```

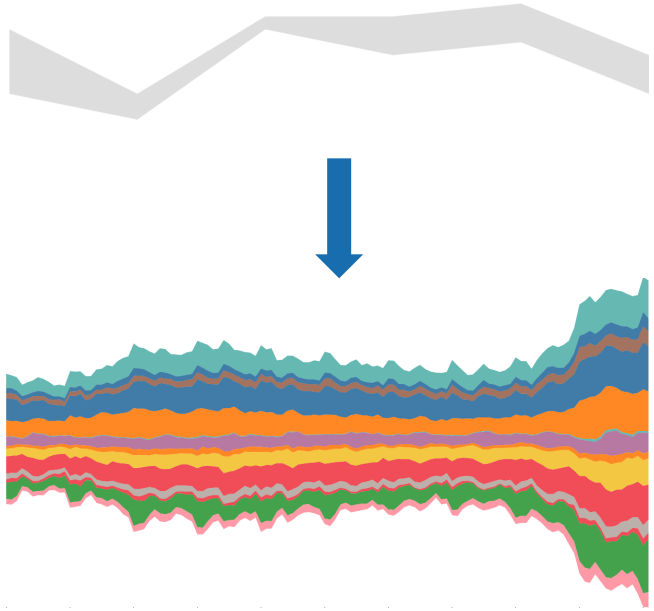

- types

What's the gap?

- We created donut charts imperatively.
- How to calculate startAngle and endAngle based on the given data automatically?

```
var arcData = [
  {startAngle: 0, endAngle: 0.2},
  {startAngle: 0.2, endAngle: 0.6},
  {startAngle: 0.6, endAngle: 1.4},
  {startAngle: 1.4, endAngle: 3},
  {startAngle: 3, endAngle: 2* Math.PI}
];
```



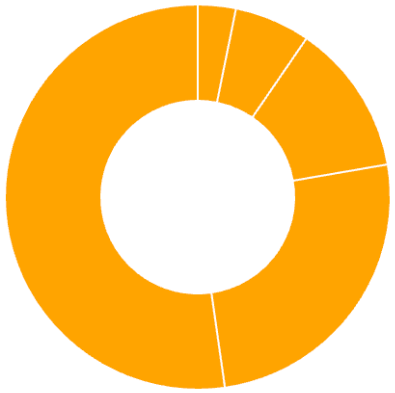


D3 Layouts

In essence, a **layout function** in D3 is just a JavaScript function that

- Takes your data as input
- Computes visual variables such as *position* and *size* to it so that we can visualize the data

```
var arcData = [
  {startAngle: 0, endAngle: 0.2},
  {startAngle: 0.2, endAngle: 0.6},
  {startAngle: 0.6, endAngle: 1.4},
  {startAngle: 1.4, endAngle: 3},
  {startAngle: 3, endAngle: 2* Math.PI}
];
```



```
var data = [10, 40, 30, 20, 60]
```

How to generate the donut chart?

Pie

Given an array of data, the pie generator computes the necessary **angles** to represent the data

- For example, we have an array of data:

  var data **=** [10, 40, 30, 20, 60, 80];

- Apply pie generator to the *data* to get *arcData*


```
var pieGenerator = d3.pie();
var arcData = pieGenerator(data);
```

```
▶ 0: {data: 10, index: 5, value: 10, startAngle: 6.021385919380437, endAngle: 6.283185307179586, …}
▶ 1: {data: 40, index: 2, value: 40, startAngle: 3.665191429188092, endAngle: 4.71238898038469, …}
▶ 2: {data: 30, index: 3, value: 30, startAngle: 4.71238898038469, endAngle: 5.497787143782138, …}
▶ 3: {data: 20, index: 4, value: 20, startAngle: 5.497787143782138, endAngle: 6.021385919380437, …}
▶ 4: {data: 60, index: 1, value: 60, startAngle: 2.0943951023931953, endAngle: 3.665191429188092, …}
▶ 5: {data: 80, index: 0, value: 80, startAngle: 0, endAngle: 2.0943951023931953, …}
```

Pie

Example: Donut Chart

- data

```
var fruits = [
  {name: 'Apples', quantity: 20},
  {name: 'Bananas', quantity: 40},
  {name: 'Cherries', quantity: 50},
  {name: 'Damsons', quantity: 10},
  {name: 'Elderberries', quantity: 30},
];
```

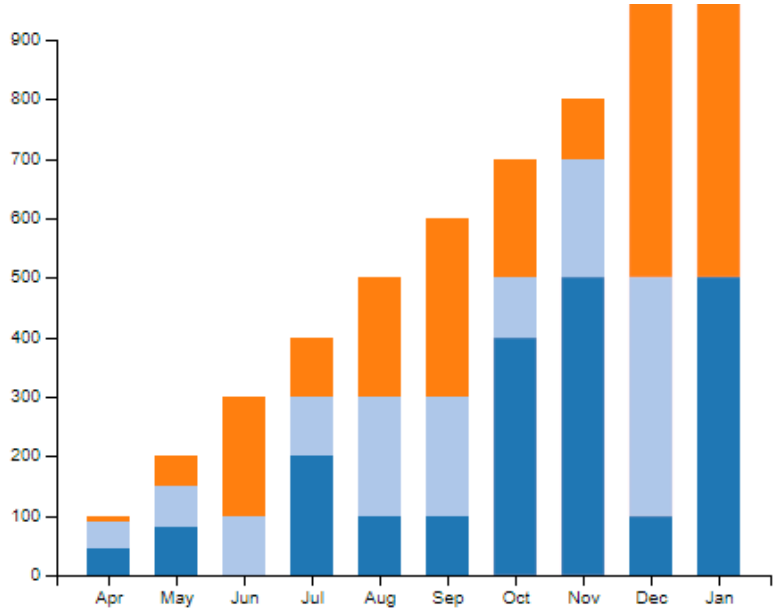
- steps
  - create the arc data (startAngles and endAngles)
  - specify the arc configurations
  - draw the pie chart

Pie

```
//create the arc data (startAngles and endAngles)
var pieGenerator = d3.pie()
  .value(function(d) {return d.quantity;})

var arcData = pieGenerator(fruits);

// specify the arc configuration
var arcGenerator = d3.arc()
  .innerRadius(60)
  .outerRadius(100);

// Create a path element and set its d attribute
d3.select('g')
  .selectAll('path')
  .data(arcData)
  .join('path')
  .attr('d', arcGenerator);
```

Stack

- Stacked graphs are used to show how a larger category is divided into smaller series/layers and what the relationship of each part has on the total amount



Stack

- d3.stack()
  - Input: an array of objects (multi-series/layer data)
  - Outputs: an array representing each series with their lower and higher values
- Example:
  - data



```
var data = [
  {day: 'Mon', apricots: 120, blueberries: 180, cherries: 100},
  {day: 'Tue', apricots: 60,  blueberries: 185, cherries: 105},
  {day: 'Wed', apricots: 100, blueberries: 215, cherries: 110},
  {day: 'Thu', apricots: 80,  blueberries: 230, cherries: 105},
  {day: 'Fri', apricots: 120, blueberries: 240, cherries: 105}
];
```

Stack

- Series
  - Three fruits
    - Series 0: Apricots
    - Series 1: Blueberries
    - Series 2: Cherries
- 1. Create a stack generator
  - Keys in generator are corresponding to keys in data

```
var stack = d3.stack()
  .keys(['apricots', 'blueberries', 'cherries']);
```

```
var data = [
  {day: 'Mon', apricots: 120, blueberries: 180, cherries: 100},
  {day: 'Tue', apricots: 60,  blueberries: 185, cherries: 105},
  {day: 'Wed', apricots: 100, blueberries: 215, cherries: 110},
  {day: 'Thu', apricots: 80,  blueberries: 230, cherries: 105},
  {day: 'Fri', apricots: 120, blueberries: 240, cherries: 105}
];
```

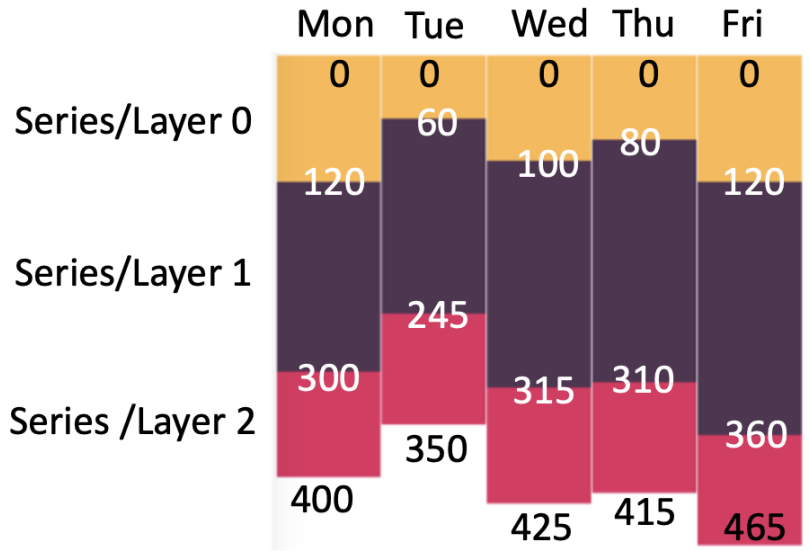
Stack

```
var data = [
  {day: 'Mon', apricots: 120, blueberries: 180, cherries: 100},
  {day: 'Tue', apricots: 60,  blueberries: 185, cherries: 105},
  {day: 'Wed', apricots: 100, blueberries: 215, cherries: 110},
  {day: 'Thu', apricots: 80,  blueberries: 230, cherries: 105},
  {day: 'Fri', apricots: 120, blueberries: 240, cherries: 105}
];
```

- Apply generator to data, we get:

[[0, 120],[0, 60],[0, 100],[0, 80],[0, 120]],// Series 0: Apricots
[[120, 300], [60, 245], [100, 315],[80, 310],[120, 360]], // Series 1: Blueberries
[[300, 400], [245, 350], [315, 425], [310, 415], [360, 465]]// Series 2: Cherries

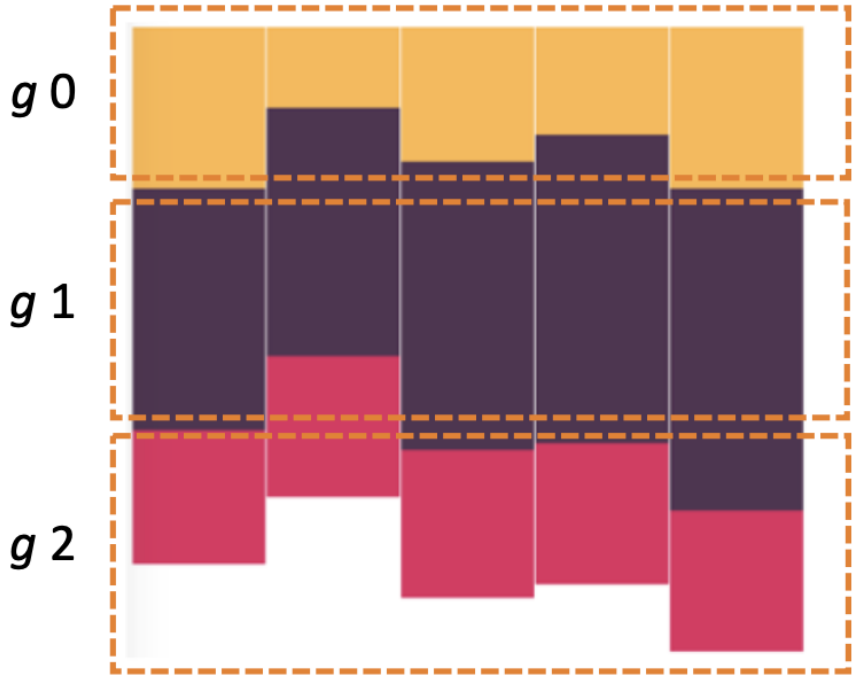
- Three arrays are the computed data for three series
  - Each array (series) has 5 tuples, which are lower and upper values for the bars of 5 days



Stack

- 2. Create a g tag for each series

```
//The colors for three fruits
var colors = ['#FBB65B', '#513551', '#de3163'];

var g = d3.select('g')
  .selectAll('g.series')
  .data(stackGenerator(data))
  .join('g')
  .style('fill', function(d, i) {
    return colors[i];
  });
```

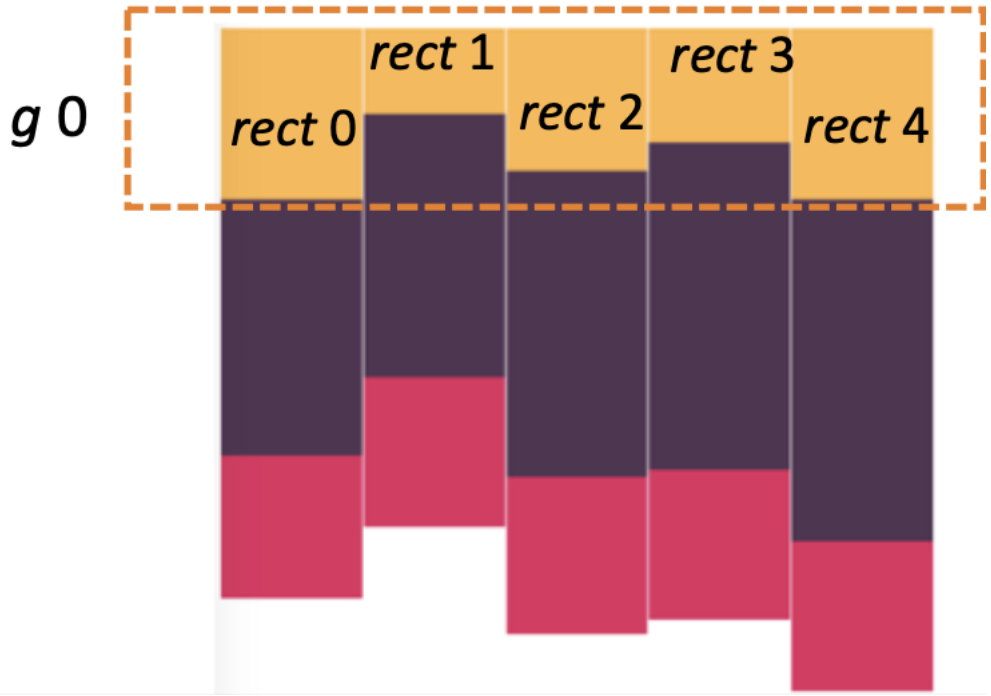


Stack

- 3. For each series (g tag), create rectangles

```
// For each series create a rect element for each day
g.selectAll('rect')
  .data(function(d) {
    return d;
  })
  .join('rect')
  .attr('width', 99)
  .attr('x', function(d) {
    return i * 100;
  })
  .attr('y', function(d, i) {
    return d[0];
  })
  .attr('height', function(d, i) {
    return d[1] - d[0];
  });
```

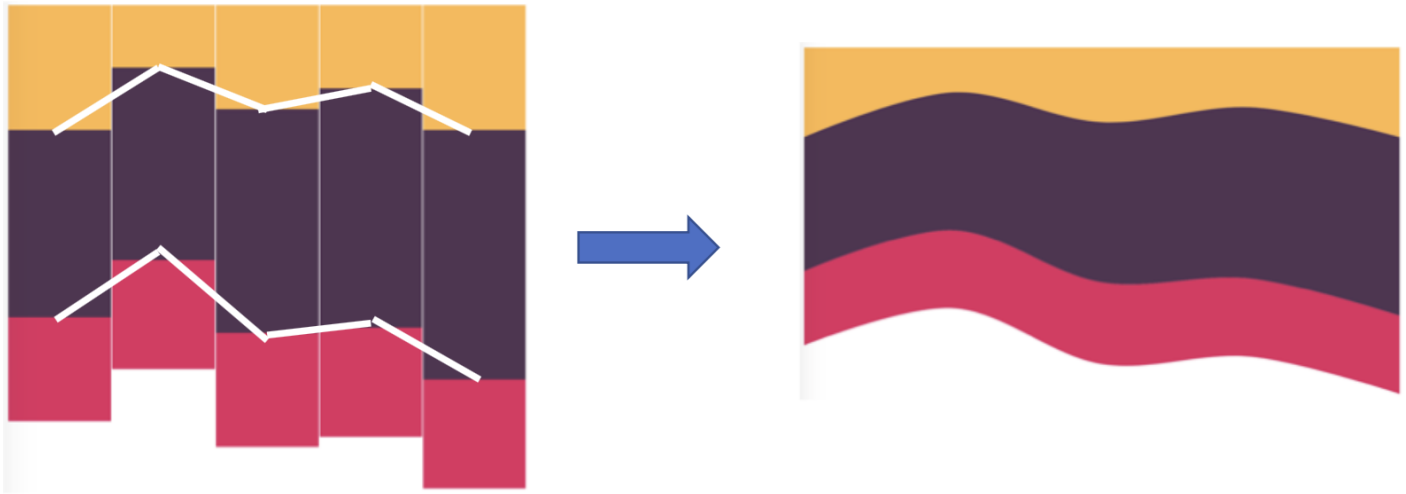
[[0, 120],[0, 60],[0, 100],[0, 80],[0, 120]]



Stack

We can generate stream graphs with the help of area generator: d3.area()

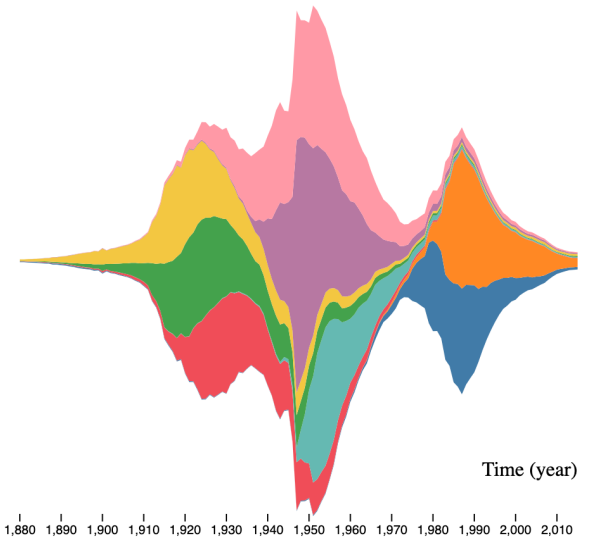
[[0, 120],[0, 60],[0, 100],[0, 80],[0, 120]]

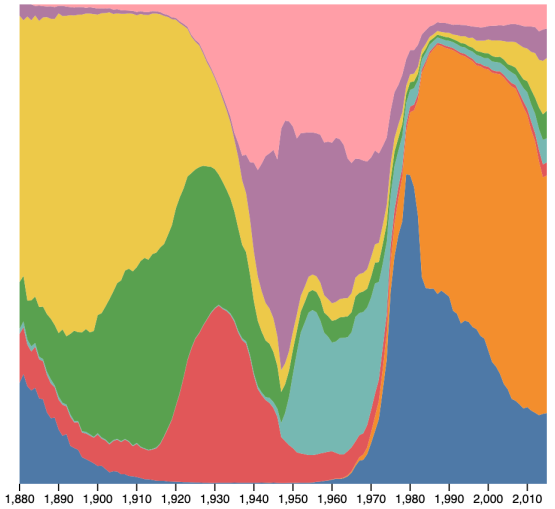


Stack Customization

- offset()

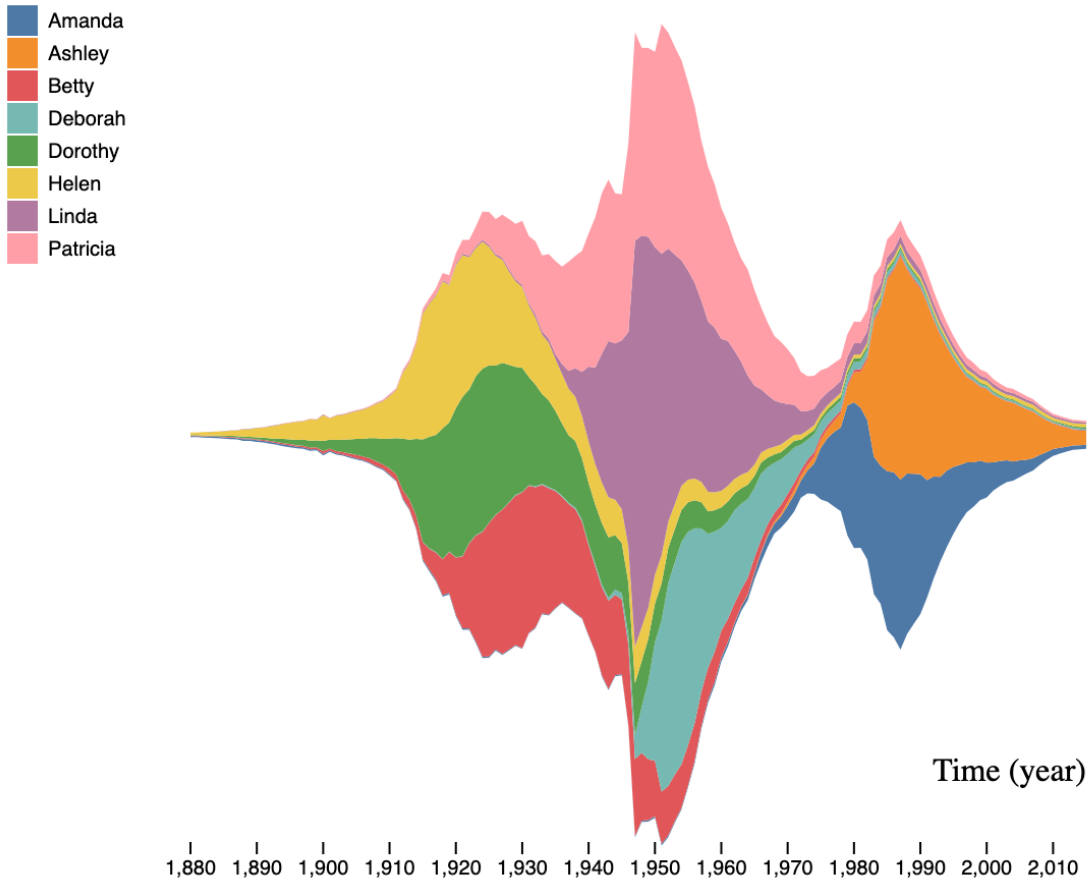
stackOffsetNone	(Default) No offset
stackOffsetExpand	Sum of series is normalised (to a value of 1)
stackOffsetSilhouette	Center of stacks is at y=0
stackOffsetWiggle	Wiggle of layers is minimised (typically used for streamgraphs)





Example - StreamGraph

- Evolution of baby names in US



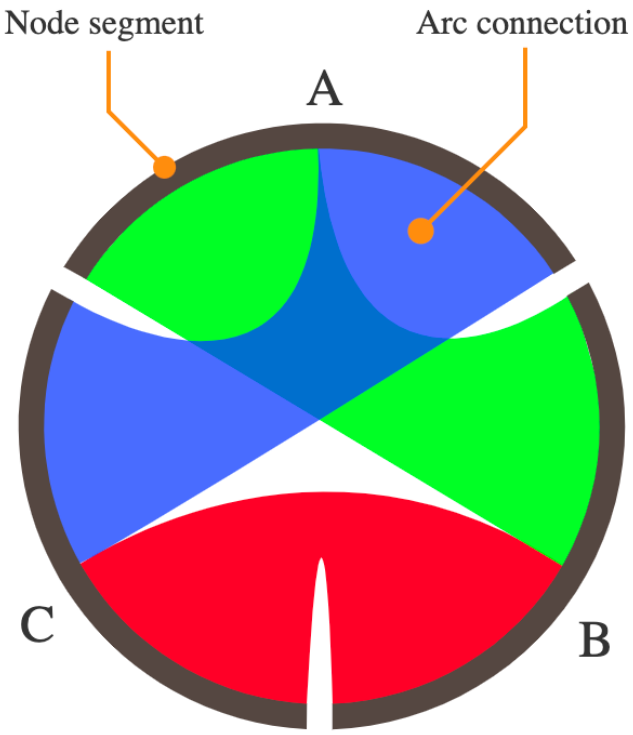
Demo

Chord

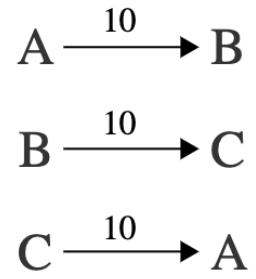
- Chord diagrams visualize links (or flows) between a group of nodes, where each flow has a numeric value.
- Example:
  - Migration flow between and within regions (2005 – 2010)

Chord - data

- Nodes are arranged along a circle
- The relationships between points are connected to each other either through the use of arcs or Bézier curves.
- Values are assigned to each connection, which is represented proportionally by the size of each arc



	A	B	C
A		10	10
B	10		10
C	10	10	

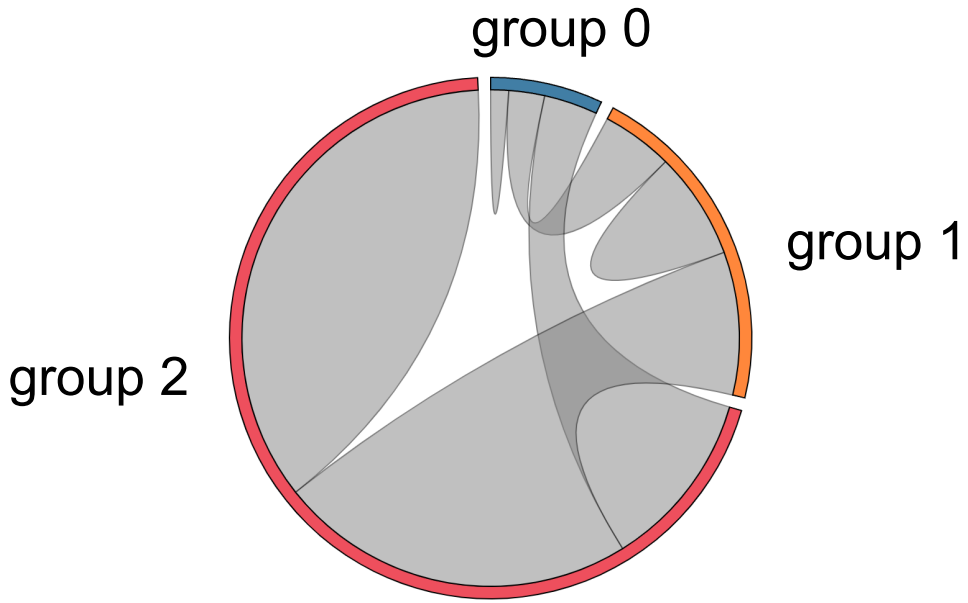


https://datavizcatalogue.com/methods/chord_diagram.html

Chord

- The data needs to be in the form of an n x n matrix (where n is the number of items)
  - First row represents flows from the 1st item to the 1st, 2nd and 3rd items etc.

```
var data = [
  [10, 20, 30],
  [40, 60, 80],
  [100, 200, 300]
];
```



Chord

Draw a chord layout

- d3.chord()
  - Compute *startAngle* and *endAngle* for each chord
  - padAngle(): set padding angle (gaps) between adjacent groups

```
var chordGenerator = d3.chord();
var chords = chordGenerator(data);
```

```
// [object Array] (6)
[// [object Object]
{
  "source": {
    "index": 0,
    "startAngle": 0,
    "endAngle": 0.07337125365689984,
    "value": 10
  },
  "target": {
    "index": 0,
    "startAngle": 0,
    "endAngle": 0.07337125365689984,
    "value": 10
  }
},// [object Object]
```

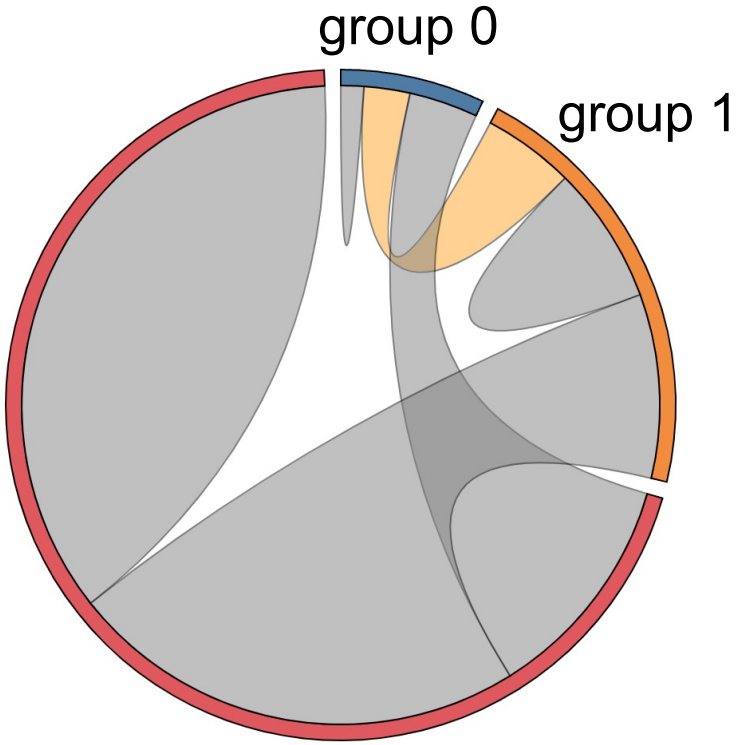
Chord

Draw a chord layout

```
var data = [
  [10, 20, 30],
  [40, 60, 80],
  [100, 200, 300]
];
```

group 0 -> group 1: 20

group 1 -> group 0: 40

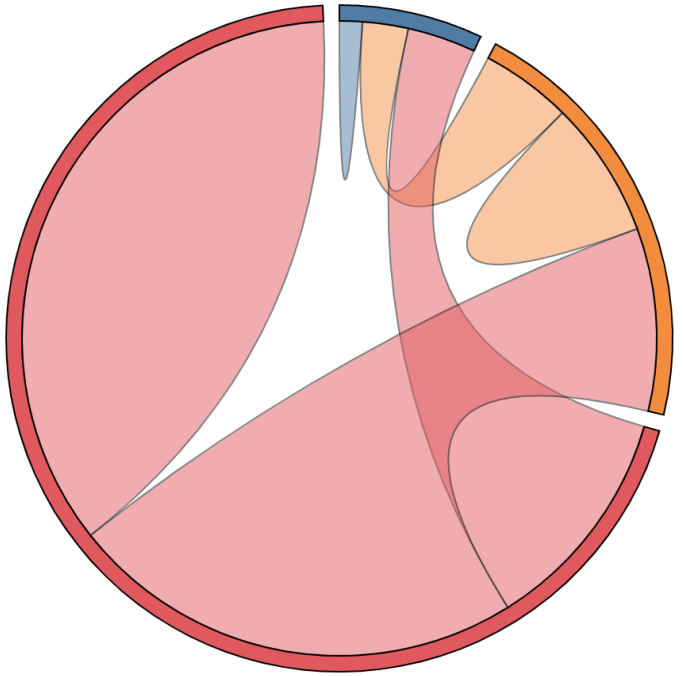



```
{
  "source": {
    "index": 1,
    "startAngle": 0.480227521941399,
    "endAngle": 0.7737125365689983,
    "value": 40
  },
  "target": {
    "index": 0,
    "startAngle": 0.073371253656899 84,
    "endAngle": 0.2201137609706995,
    "value": 20
  }
```

Chord

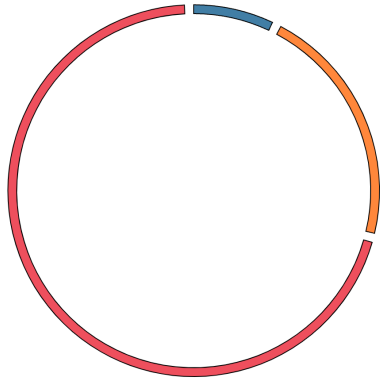
- d3.ribbon
  - Converts the chord properties (*startAngle* and *endAngle*) into *path* data so that we can draw chords by *SVG*
  - radius(): controls the radius of the final layout

```
var ribbonGenerator = d3.ribbon()
    .radius(200);

d3.select('g')
  .selectAll('path')
  .data(chords)
  .join('path')
  .attr('d', ribbonGenerator)
```



Chord

- Group arcs

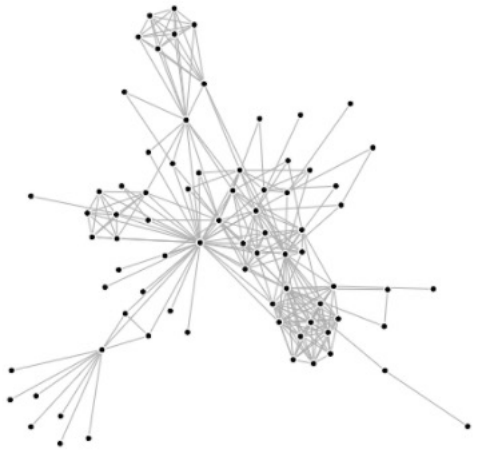


```
svg.selectAll('group')
    .data(chords.groups)
    .join('path')
    .style("fill", function(d,i){
        return groupColors[i];
    })
    .style("stroke", "black")
    .attr("d", d3.arc()
        .innerRadius(200)
        .outerRadius(210)
    )
```


```
▼groups: Array(3)
  ▼0:
     endAngle: 0.43808466479854186
     index: 0
     startAngle: 0
     value: 60
    ▶[[Prototype]]: Object
  ▼1:
     endAngle: 1.8023386591941675
     index: 1
     startAngle: 0.48808466479854185
     value: 180
    ▶[[Prototype]]: Object
  ▼2:
     endAngle: 6.233185307179586
     index: 2
     startAngle: 1.8523386591941675
     value: 600
    ▶[[Prototype]]: Object
   length: 3
```

D3 Force Layout

- D3's force layout uses a physics-based simulator for positioning visual elements.
  - all elements can be configured to repel one another
  - elements can be attracted to center(s) of gravity
  - linked elements can be set a fixed distance apart (e.g., network visualization)
  - elements can be configured to avoid intersecting one another (collision detection)
  - example 1, example 2



Voronoi

- In mathematics, a Voronoi diagram is a partitioning of a plane into regions based on the distance to points in a specific subset of the plane
  - Application: Partition a plane based on points
  - Implementation