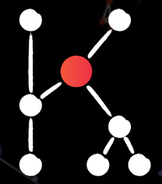


DATA-DRIVEN DOCUMENTS

KIRELL BENZI, PH.D

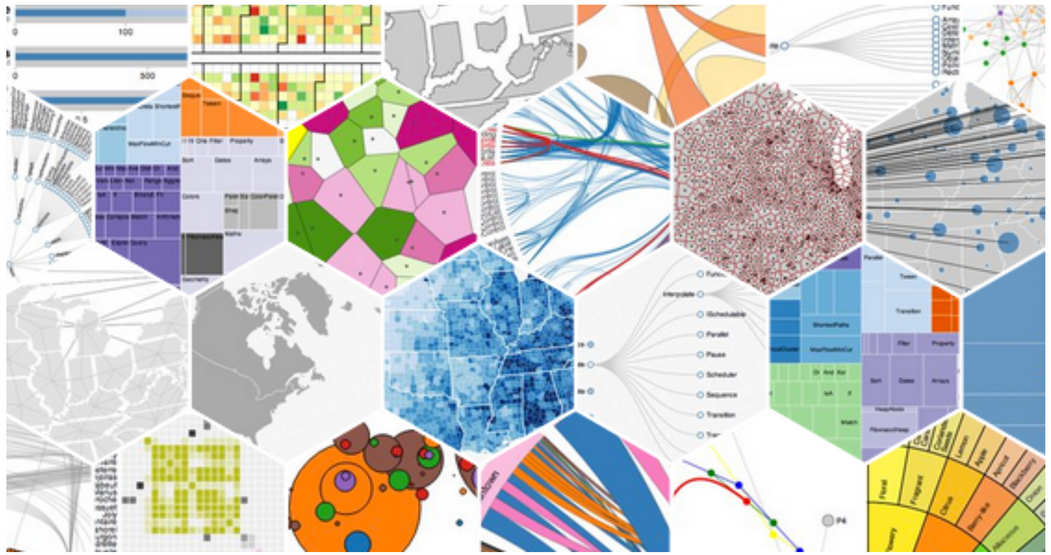
@KirellBenzi

www.kirellbenzi.com

D3.js is a Javascript library for creating data visualizations

Offers a data-driven approach to manipulate the DOM

Modular design with a lot of different modules



Why learn D3?

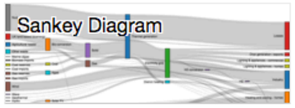
Manipulating an HTML document is tedious

SVG, Canvas are error-prone and difficult on your own

D3 is the standard for data-viz => hirable skills

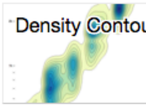
Vibrant open-source community

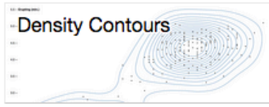
Municipalities of Mexico

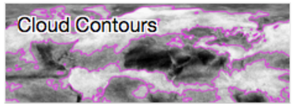


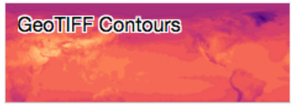
Adjustable Link Strength

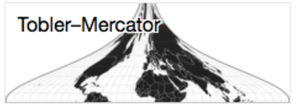
Density Contours II

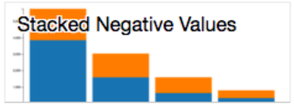


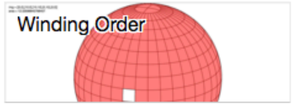


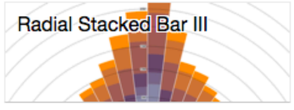


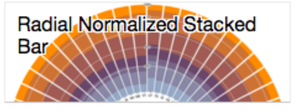


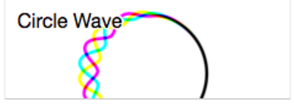


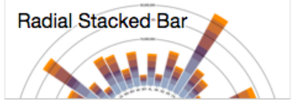


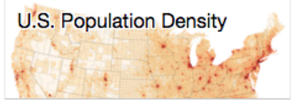


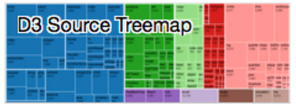










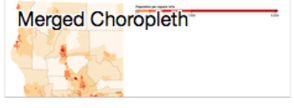


Localized Time Axis II

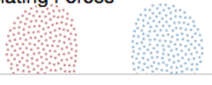
Localized Time Axis

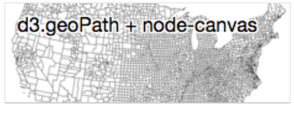
Modifying a Force Layout II

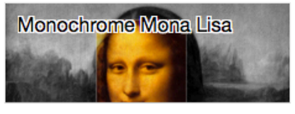
Clipped Conic Conformal



Isolating Forces



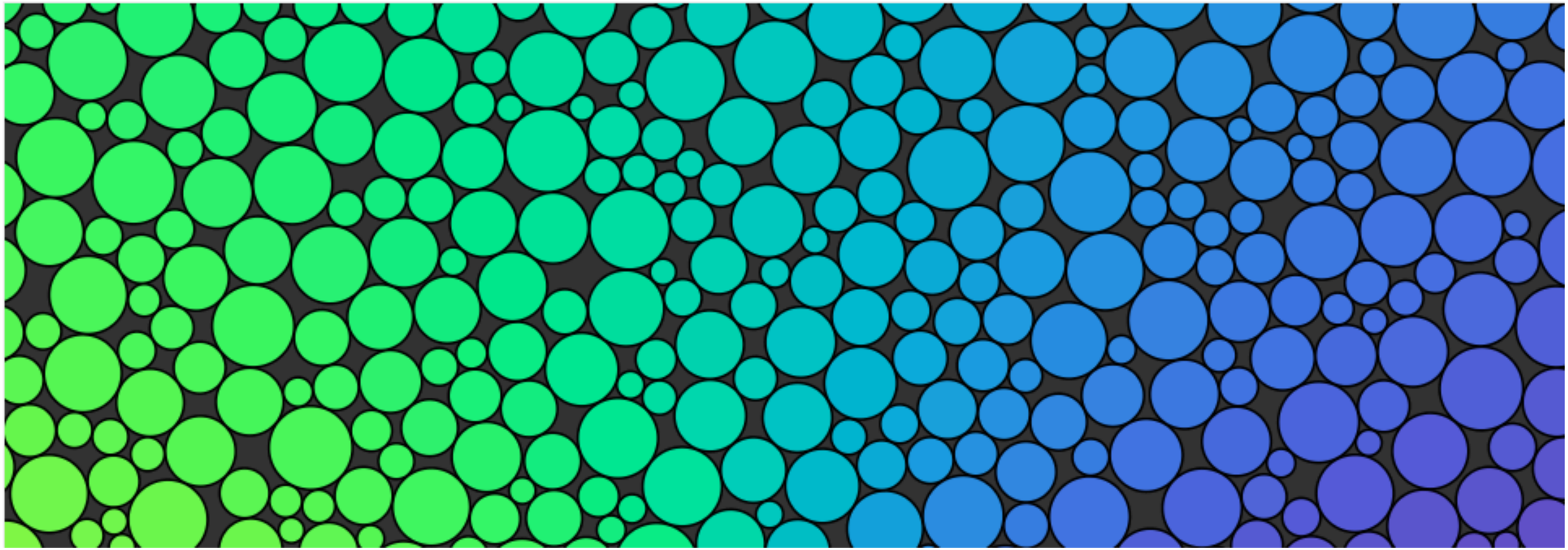




Reporting an Issue

Example

Live example



- Arrays (Statistics, Search, Transformations, Histograms)
- Axes
- Brushes
- Chords
- Collections (Objects, Maps, Sets, Nests)
- Colors
- Dispatches
- Dragging
- Delimiter-Separated Values
- Easings
- Forces
- Number Formats
- Geographies (Paths, Projections, Spherical Math, Spherical Shapes, Streams, Transforms)
- Hierarchies
- Interpolators
- Paths
- Polygons
- Quadtrees
- Queues
- Random Numbers
- Requests
- Scales (Continuous, Sequential, Quantize, Ordinal)
- Selections (Selecting, Modifying, Data, Events, Control, Local Variables, Namespaces)
- Shapes (Arcs, Pies, Lines, Areas, Curves, Links, Symbols, Stacks)
- Time Formats
- Time Intervals
- Timers
- Transitions
- Voronoi Diagrams
- Zooming

Module overview

D3 selection

```
<p>Look at me</p>
<p id="par2">Yeah me!</p>
<p class="yo">Yo</p>
<p class="yo">Hello</p>
<script>
    // Select by tag
    let p = d3.select("p");
    p.style("color", "blue");

    // Select by ID
    p = d3.select("#par2");
    p.style("color", "green");

    // Select by class
    p = d3.selectAll(“.yo");
    p.style("color", "red");
</script>
```

D3 offers a unified **declarative interface** that allows to describe the what and not the how.

D3 selectors works like document.querySelector

D3 selection

Selection methods d3.select, d3.selectAll return the current selection or a new selection

D3 allows to bulk-modify the content of a selection for arbitrary properties like ***style***, ***attr***, etc. or the textual content of the elements

```
selection.property(name[, value])
selection.text([value])
```

```
<p>Look at me</p>
<p id="par2">Yeah me!</p>
<p class="yo">Yo</p>
<p class="yo">Hello</p>
<script>
  // we don't have to iterate over each element
  let ps = d3.selectAll(“p")
              .style("color", "red");
</script>
```

Append and modify elements

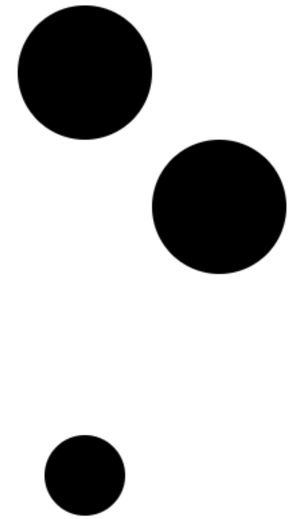
We can modify the DOM and append elements and modify their properties altogether.

```
d3.select("body")
  .append("svg")
    .attr("width", 960)
    .attr("height", 500)
  .append("g")
    .attr("transform", "translate(20,20)")
  .append("rect")
    .attr("width", 920)
    .attr("height", 460);
```

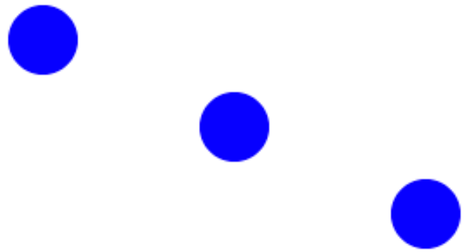
Dynamical properties

Properties can also be modfied dynamically with functions

```
<svg height="500" width="500">
    <circle cx="100" cy="100" r="50"></circle>
    <circle cx="200" cy="200" r="50"></circle>
    <circle cx="100" cy="400" r="30"></circle>
</svg>
```



```
let circles = d3.selectAll("circle")
                .style("fill", "blue");

circles.attr('cx', (d, i) => 110 * (i+1))
       .attr('cy', (d, i) => 50 * (i+1))
       .attr('r', 20);
```



Binding data (data-join)

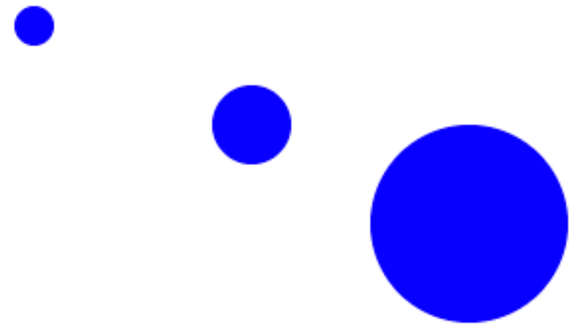
The strength of D3 allows us to link or ***bind*** data to DOM elements

The most important concepts after a data-join are ***update***, ***enter***, ***exit***

Here we ***update*** existing elements according to the data

```
<svg height="500" width="500">
    <circle cx="100" cy="100" r="50"></circle>
    <circle cx="200" cy="200" r="50"></circle>
    <circle cx="100" cy="400" r="30"></circle>
</svg>
```

```
const radii = [10, 20, 50]; // radius plurar form
let circles = d3.selectAll("circle")
                .data(radii)
                .attr('cx', (d, i) => 110 * (i+1))
                .attr('cy', (d, i) => 50 * (i+1))
                .attr('r', (d, i) => d) // 10, 20, 50
                .style("fill", "blue");
```

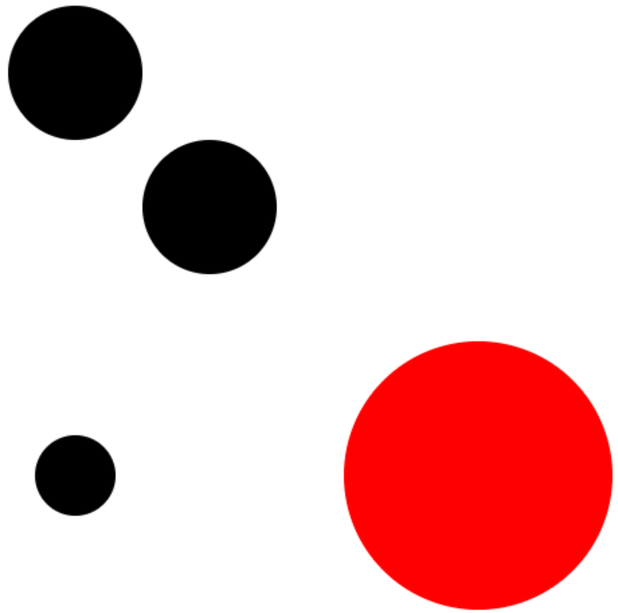


Enter enter()

enter() describes what to do when new data arrives.

New data means what is not currently bound to DOM elements

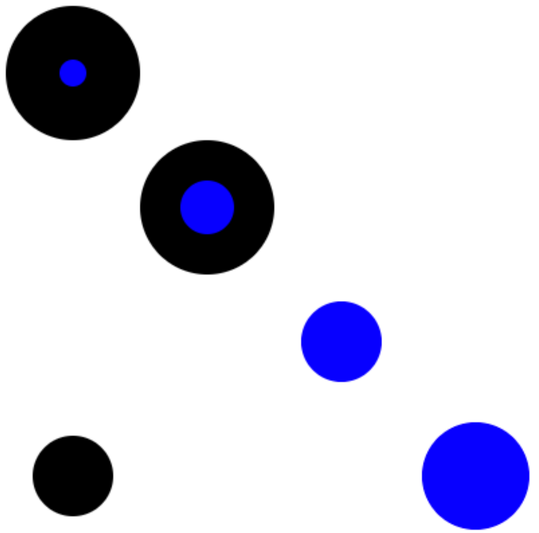
```
let svg = d3.select("svg");
let circles2 = svg.selectAll("circle")
                  // only 3 circles in our SVG
                  .data([10, 20, 50, 100])
                  .enter()
                  .append('circle')
                    .attr('cx', (d, i) => 100 * (i+1))
                    .attr('cy', (d, i) => 100 * (i+1))
                    .attr('r', d => d) // 100
                    .style("fill", "red");
```



Selecting non existing elements

If we select elements that are not yet defined, all the dataset can be found after the enter() section

```
let whatever = svg.selectAll('I_DONT_EXIST_YET')
                  .data([10, 20, 30, 40])
                  .enter()
                  .append('circle')
                    .attr('cx', (d, i) => 100 * (i+1))
                    .attr('cy', (d, i) => 100 * (i+1))
                    .attr('r', d => d)
                    .style("fill", "blue");
```



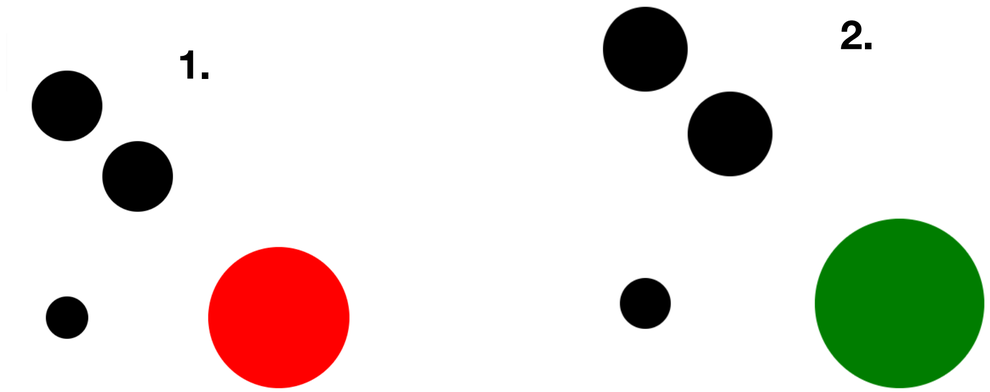
Exit

exit() is the opposite of enter()

We enter the exit() section if previously bound data is now removed from the dataset

```
let svg = d3.select("svg");
let data = [10, 20, 50, 100];
let circles2 = svg.selectAll("circle")
                  .data(data)
                  .enter()
                  .append('circle')
                    .attr('cx', (d, i) => 100 * (i+1))
                    .attr('cy', (d, i) => 100 * (i+1))
                    .attr('r', d => d) // 100
                    .style("fill", "red");
```

```
data = [10, 20, 50];
circles2.data(data)
        .exit()
        .style("fill", "green")
        // .remove() // if you want to remove the elem
        ;
```



General Update Pattern

Live example

More d3

Formatting numbers

```
for (const i = 0; i < 10; i++) {
  console.log(0.1 * i);
}
```

```
0
0.1
0.2
0.30000000000000004
0.4
0.5
0.6000000000000001
0.7000000000000001
0.8
0.9
```

```
const f = d3.format(".1f");
for (const i = 0; i < 10; i++) {
  console.log(f(0.1 * i));
}
```

```
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
```

More formatting

`[[fill]align][sign][symbol][0][width][,][.precision][type]`

- `>` - Forces the field to be right-aligned within the available space. (Default behavior).
- `<` - Forces the field to be left-aligned within the available space.
- `^` - Forces the field to be centered within the available space.
- `=` - like `>`, but with any sign and symbol to the left of any padding.

The *sign* can be:

- `-` - nothing for zero or positive and a minus sign for negative. (Default behavior.)
- `+` - a plus sign for zero or positive and a minus sign for negative.
- `(` - nothing for zero or positive and parentheses for negative.
- ` ` (space) - a space for zero or positive and a minus sign for negative.

The *symbol* can be:

- `$` - apply currency symbols per the locale definition.
- `#` - for binary, octal, or hexadecimal notation, prefix by `0b`, `0o`, or `0x`, respectively.

- `e` - exponent notation.
- `f` - fixed point notation.
- `g` - either decimal or exponent notation, rounded to significant digits.
- `r` - decimal notation, rounded to significant digits.
- `s` - decimal notation with an SI prefix, rounded to significant digits.
- `%` - multiply by 100, and then decimal notation with a percent sign.
- `p` - multiply by 100, round to significant digits, and then decimal notation with a per
- `b` - binary notation, rounded to integer.
- `o` - octal notation, rounded to integer.
- `d` - decimal notation, rounded to integer.
- `x` - hexadecimal notation, using lower-case letters, rounded to integer.
- `X` - hexadecimal notation, using upper-case letters, rounded to integer.
- `c` - converts the integer to the corresponding unicode character before printing.
- ` ` (none) - like `g`, but trim insignificant trailing zeros.

https://github.com/d3/d3-format#format

Examples

```
d3.format(".2")(42);  // "42"
d3.format(".2")(4.2); // "4.2"
d3.format(".1")(42);  // "4e+1"
d3.format(".1")(4.2); // "4"
```

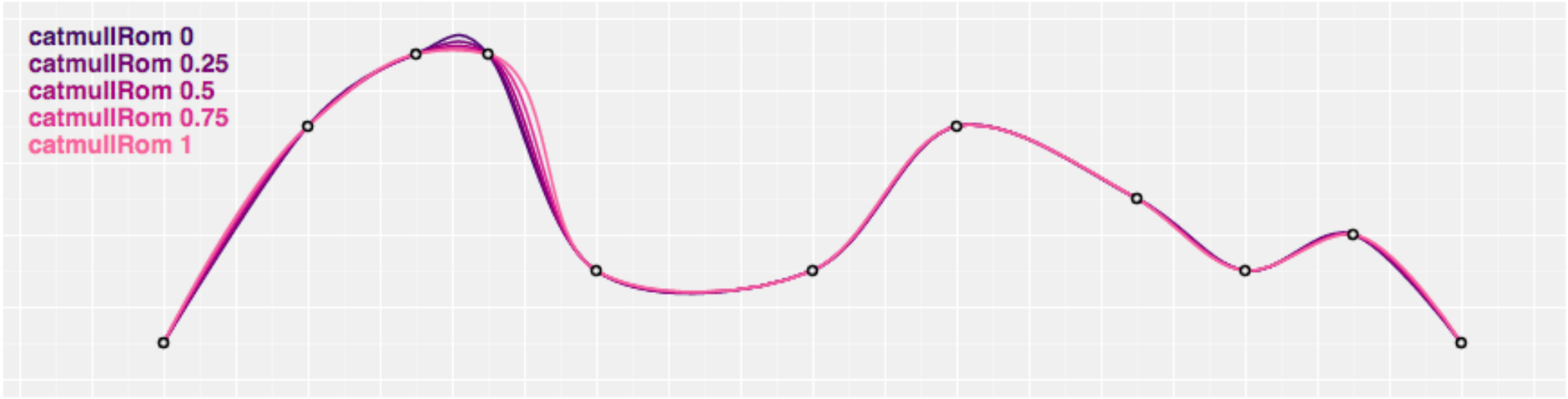
```
d3.format(".0%")(0.123);  // rounded percentage, "12%"
d3.format("($.2f")(-3.5); // localized fixed-point currency, "(£3.50)"
d3.format("+20")(42);     // space-filled and signed, "                 +42"
d3.format(".^20")(42);    // dot-filled and centered, ".........42........."
d3.format(".2s")(42e6);   // SI-prefix with two significant digits, "42M"
d3.format("#x")(48879);   // prefixed lowercase hexadecimal, "0xbeef"
d3.format(",.2r")(4223);  // grouped thousands with two significant digits, "4,200"
```

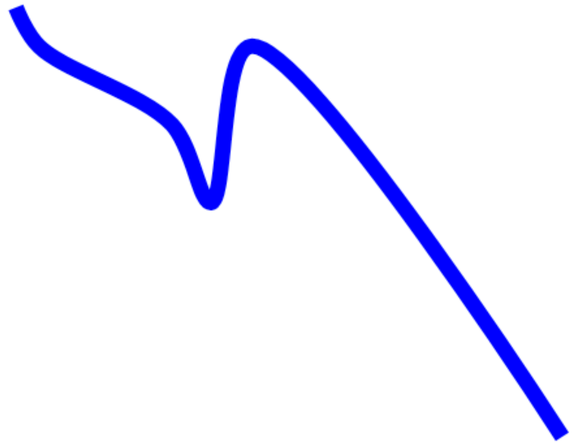
Paths

```
const data = [ { "x": 10,   "y": 15}, { "x": 13,  "y": 20},
               { "x": 30,  "y": 30}, { "x": 35,  "y": 40},
               { "x": 40,  "y": 20}, { "x": 80, "y": 70}];

const lineGenerator = d3.line()
                        .x(d => d.x)
                        .y(d => d.y);

const svgContainer = d3.select("body")
                       .append("svg")
                         .attr("width", 200)
                         .attr("height", 200);

const lineChart = svgContainer.append("path")
                              .attr("d", lineGenerator(data))
                              .attr("stroke", "blue")
                              .attr("stroke-width", 2)
                              .attr("fill", "none");
```

Curves

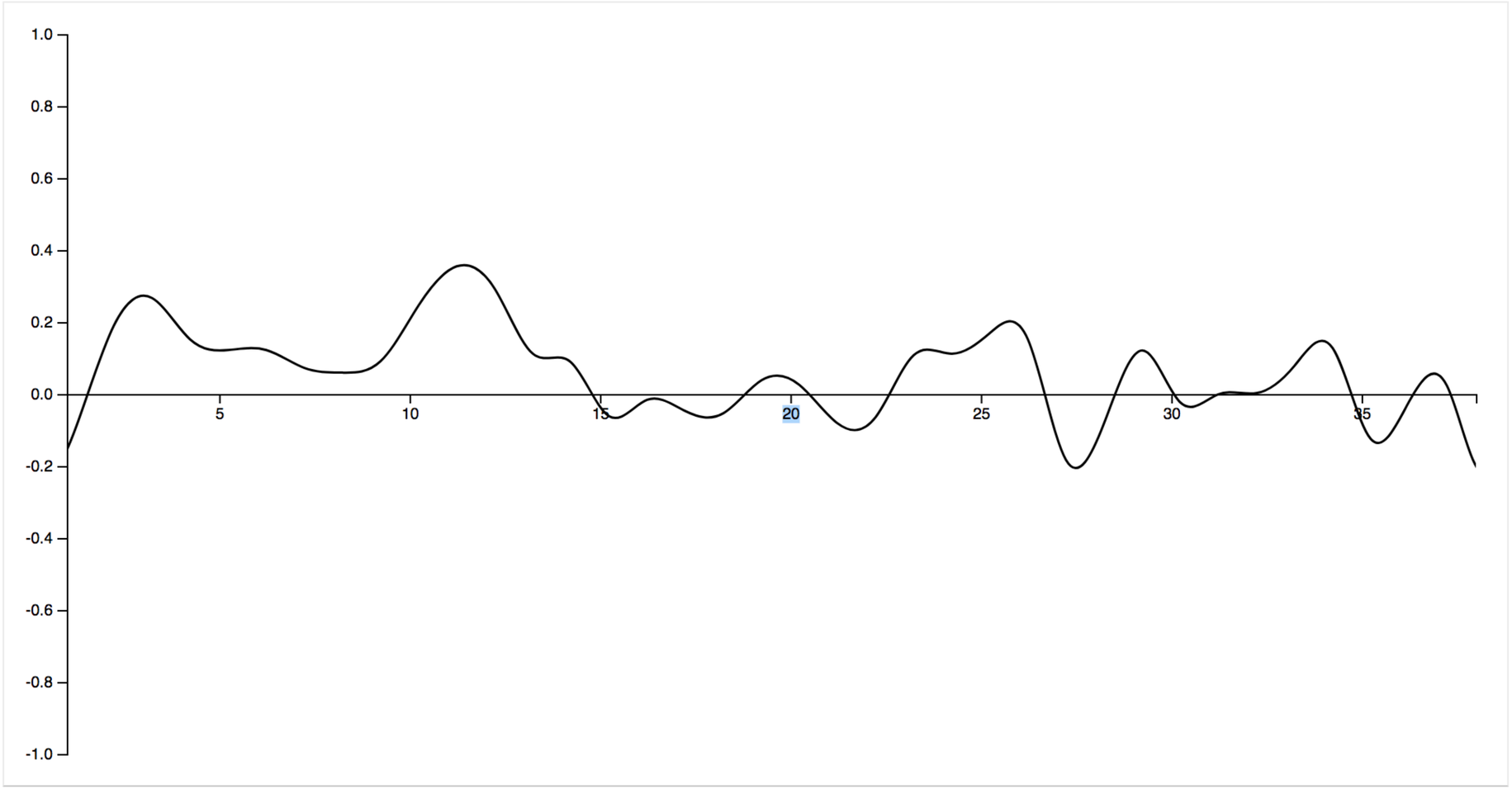
```
const lineGenerator = d3.line()
                        .x(d => d.x)
                        .y(d => d.y)
                        .curve(d3.curveCatmullRom.alpha(0.5));
```





https://github.com/d3/d3-shape/blob/master/README.md#curves

Spline transition



Reading data

Reading data in d3

d3 offers a nice API for HTTP requests

Main usage is to parse a dataset from a server

Reading JSON files

```
d3.json("data.json", function(error, data) {
  // ...
}
```

```
d3.json("data.json", function(data) {
  // ...
}
```

Reading CSV files

```
Year,Make,Model,Length
1997,Ford,E350,2.34
2000,Mercury,Cougar,2.38
```

```
d3.csv(url, row, callback);

function row(d) {
  return {
    year: new Date(+d.Year, 0, 1), // convert "Year" column to Date
    make: d.Make,
    model: d.Model,
    length: +d.Length // convert "Length" column to number
  };
}
```

```
[
  {"Year": "1997", "Make": "Ford", "Model": "E350", "Length": "2.34"},
  {"Year": "2000", "Make": "Mercury", "Model": "Cougar", "Length": "2.38"}
]
```

Load multiple datasets

If you wish to consolidate your dataset, you may want to fetch data from several sources.

d3 offers a clean way to do wait for asynchronous operation (similar to promises) using **d3.queue**

“A queue evaluates zero or more **deferred** (delayed) asynchronous tasks with configurable concurrency: you control how many tasks run at the same time.” [d3 doc]

Load multiple datasets

```
d3.queue()
  .defer(d3.csv, "/data/cities.csv")
  .defer(d3.tsv, "/data/animals.tsv")
  .await(analyze);

function analyze(error, cities, animals) {
  if(error) { console.log(error); }

  console.log(cities[0]);
  console.log(animals[0]);
}
```

http://learnjsdata.com/read_data.html

Lo

The most essential utility library: **Lodash**

Well-written list of utility functions to make JS easier and more batteries-included.

Most functions are faster than the standard library!

Emphasis on the functional side of Javascript, promotes **immutability** and **composition**

Some nice examples

```
_.flattenDeep([1, [2, [3, [4]], 5]]);
// => [1, 2, 3, 4, 5]
```

```
_.zipObject(['a', 'b'], [1, 2]);
// => { 'a': 1, 'b': 2 }
```

```
const users = [
  { 'user': 'barney',  'age': 36, 'active': true },
  { 'user': 'fred',    'age': 40, 'active': false },
  { 'user': 'pebbles', 'age': 1,  'active': true }
];
```

```
_.forEach
_.map
_.filter
_.reduce
```

```
_.find(users, function(o) { return o.age < 40; });
// => object for 'barney'

// The `_.matches` iteratee shorthand.
_.find(users, { 'age': 1, 'active': true });
// => object for 'pebbles'
```

```
_.chunk(['a', 'b', 'c', 'd'], 2);
// => [['a', 'b'], ['c', 'd']]
```

Some nice examples

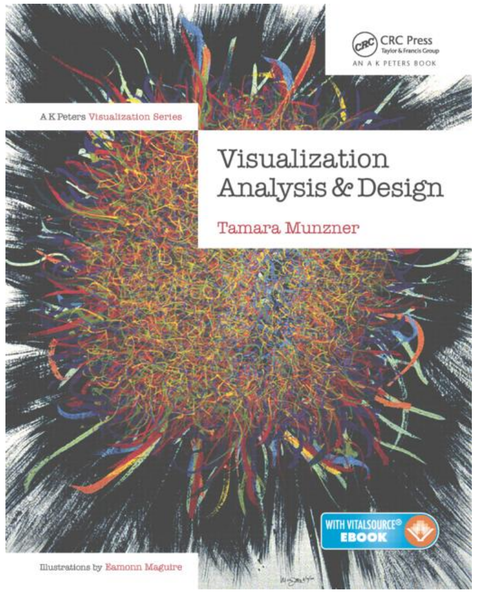
```
// Fetch the name of the first pet from each owner
const ownerArr = [{
    "owner": "Colin",
    "pets": [{"name":"dog1"}, {"name": "dog2"}]
}, {
    "owner": "John",
    "pets": [{"name":"dog3"}, {"name": "dog4"}]
}];

// Array's map method.
ownerArr.map(function(owner){
   return owner.pets[0].name;
});

// Lodash, improved map
_.map(ownerArr, 'pets[0].name');
```

```
_.times(5, () => {
    // do something 5 times
});
```

https://colintoh.com/blog/lodash-10-javascript-utility-functions-stop-rewriting

Homework

Read Visualization Analysis and Design chapter 2



Read Interactive Data Visualization for the Web chapter 4 to 7 included