Lexicon Visualization Library and JavaScript for Scientific Data Visualization

Ibrahim Tanyalcin
(IB)^2, Vrije Universiteit Brussel (VUB)

Carla Al Assaf
KU Leuven

Julien Ferte
(IB)^2, Université Libre de Bruxelles (ULB)

François Ancien
(IB)^2, ULB

Taushif Khan
(IB)^2, VUB

Guillaume Smits
(IB)^2, ULB

Marianne Rooman
(IB)^2, ULB

Wim Vranken
(IB)^2, VUB

It is becoming increasingly challenging to efficiently visualize and extract useful insight from complex and big datasets. JavaScript stands out as a suitable programming choice that offers mature libraries, easy implementation, and extensive customization, all of which stay in the shadow of new and rapid developments in the language. To illustrate the use of JavaScript in a scientific context, this article elaborates on Lexicon, a collection of JavaScript libraries for generating interactive visualizations in bioinformatics and other custom libraries.

In the last two decades, the data collected under the umbrella of data science has grown exponentially in volume, directly or indirectly affecting almost all branches of fundamental science. This is easily observable from the emergence of publicly available large datasets such as 1000 Genomes and Ensembl Genomes (www.internationalgenome.org/data, ensemblgenomes.org), American Economic Association (www.aeaweb.org/resources/data), Google Finance (www.google.com/finance), and CERN Open Data Portal (opendata.cern.ch).

Copublished by the IEEE CS and the AIP
1521-9615/18/$33 ©2018 IEEE

THE STATE OF BIG DATA

The 1000 Genomes project perfectly illustrates the vastness of the variant dataset and how important each piece of data can be. The phase 1 release of the project had 1,092 individuals, each with an average genome size of ~3.2 gigabases (Gb). The total number of variants (differences with respect to the reference genome) was ~37.9 million. Each of these variants can be very important as it is theoretically sufficient to have a single mutation in a protein coding region to disrupt normal cellular metabolism.

Part of the challenge is dealing with this vast amount of data; the other part is designing intuitive and interactive data visualization tools that help researchers leverage computation's power to efficiently use this data for the benefit of the public. Therefore, the challenge is creating not only computational systems that can handle the data but also visualizations that help scientists comprehend the data.

Available Tools

Several libraries are available for data visualization; however, one must consider the desired features before choosing a programming language/library. For instance, interactive graphics or static ones, animations or no animations, custom or out-of-the-box layouts?

MATPLOTLIB from PYTHON offers several out-of-the-box layouts to plot static graphs.[1] Likewise, you can use PERL PDL and GD libraries to plot using readily available layouts or shape primitives.

The boundaries between interactive and static data visualization are blurring as the programming languages and libraries evolve. For instance, R, one of the most popular language for performing statistics for over two decades, readily plots static graphs with numerous premade layouts. Recently, the language has stepped closer to interactive graphics by introducing the Shiny framework (shiny.rstudio.com), which offers web-based interactive graphics without requiring JavaScript (JS), CSS, or HTML knowledge. Similarly, Processing (processing.org), which was originally developed in Java, is entering into web-based interactive graphics with p5.js, a high-level JS library.

Several other libraries offer a collection of library components for a specific purpose. For instance, BioJS is a large collection that offers visualization/sequence manipulation/network components for biology. Although released independently, you can find some of the Lexicon libraries under the BioJS registry (biojs.io/d/lexicon-rainbow).

Between these technologies, JS positions itself as a central tool for deploying both static and interactive graphics because JS engines are implemented in every modern browser and do not require any other dependency. And when combined with CSS, HTML5, WebGL, backward compatibility, powerful libraries, and new EcmaScript features (ES6 & ES7), the capabilities of JS become ever-extending.

Bioinformatics

Bioinformatics is an example of a field that benefits from scientific data visualization. Bioinformatics is a rapidly evolving branch of science that deals with the curation/annotation and meta-analysis of biological data and the development of new algorithms and software to decipher high-throughput information influx from in vivo and in vitro experiments (www.britannica.com/science/bioinformatics). Since the completion of the Human Genome Project and similar initiatives, sequencing databases have grown exponentially, with featured sets ranging from structural variations to epigenetics and other biologically meaningful information.[2] For instance, sequencing protein coding regions in a single human genome (which in total is 3

Gb in length, equivalent to 12 billion bits of information written with the four letters A, T, C, and G) typically yields about 40,000 variations, where only a couple of them are the cause of the disease phenotype (a biochemically, physiologically, or behaviorally observable trait that stems from a genetic/epigenetic variation). If a trait is polygenic (meaning variations in several distinct genes contribute to a phenotype), the problem becomes even more interesting. Therefore, designing interfaces and interactive visualizations that increase data's availability and accessibility to researchers, medical doctors, and others will be key.

Mutaframe

An example project using Lexicon is Mutaframe (www.mutaframe.com), a RESTful, web-based framework written in vanilla JS. It aims to visualize protein sequences and results of mutation effect predictors aided by machine learning.

Protein sequences in the human genome are challenging because the length of each protein can range from <100 amino acids to >4,000 amino acids, and at each amino acid position, theoretically 19 other amino acids can appear instead. We currently have data from all the canonical (annotated and verified) proteins in human genome, which adds up to about 30,000 sequences. Since our app is frequently used by bioinformaticians and medical doctors, it is important that the information is represented as a gradient ranging from less detailed to more detailed with components dedicated for specialized datasets. These dedicated applets (mini libraries written in JS) are collectively named Lexicon. These mini libraries can be thought of as reusable components that can communicate with one another. Projects given as example in this article are written in ES5 JS with D3 visualization library (d3js.org; version 3.5.17) that offers a low-level API as well as many readily available layouts.[3] All of the examples in this article can be found at github.com/IbrahimTanyalcin/LEXICON. A test page for the Lexicon visualization library can be found at mutaframe.com.

API DESIGN: SELECTED JAVASCRIPT CONCEPTS

JS has the flexibility of declaring and creating object properties in several different ways. Depending on the application, these objects can be tweaked and configured (see the snippet examples in Figure 1). Lexicon follows the conventional method chaining, where the properties of a main object act as getter or setter depending on the `arguments.length` and finally returns the object itself (Snippet 1).

The constructor of `lexiconObject` is wrapped around by an anonymous function (Snippet 2). Similar expressions as in Snippet 2 are collectively named immediately invoked function expression (*iife*).

On the other hand, you could also use JS ES6 syntax, which allows you to use `import/export` to transfer functions from other script files. Currently the browser support is not wide but there are already third-party libraries that offer this feature (Node's *require*, AMD, and so on). If you do not want script bundling prior to deploying on the client side, you might be interested in a pseudo-async module system (github.com/IbrahimTanyalcin/taskq; blog.cloudboost.io/queued-async-pseudo-modules-with-es5-812f99fed209).

One thing to note is that object properties declared as shown previously are enumerable, meaning that they will appear in a *for in* loop, for instance (including ones inherited down from the *prototype* chain, Snippet 3).

If you do not want certain properties to be enumerable or even writable, consider using `Object.defineProperty` or `Object.defineProperties` methods. These methods also allow you to fine-tune the object property or even make it both a *getter* and a *setter* (but not *writable* at the same time).

With JS, method chaining is not a must. You can use conventional *getters* and *setters* for your API. Moreover, with JS, lazy getters are not uncommon (Snippet 4). Theoretically, JS allows you create methods that implicitly set another property (Snippet 5). JS also allows extension of classes so that you can extend existing APIs within new objects (Snippet 6).

Apart from these examples, there are many more ways of creating objects in JS. With the ES6 syntax, JS allows you to use the `class` keyword for declaring classes. Although this can be viewed as syntactic sugar, it can result in a better structured code; it also extends certain features like `super` (refers to parent object, where formerly programmers encapsulated this `using var that = this` in the outer scope etc.) calls within object literals. You should consider incorporating these new features into your API design where possible.

Another possible approach is to incorporate JSON in your API design. This way you can pass simple string directives to your *constructor* function without the need for *functions*. You can even pass string directives that will later trigger other synthetic events once rendering or any other task is complete.

Throttling Function Calls

The previous section gave an example of how to initiate a lexicon applet. In most cases, the last method that is called is the *render* method, which renders the first scene. In other applications and libraries, a similar approach exists where a function is repeatedly invoked to redraw. In certain cases, calls to *render* can result in stuttering due to too many repeated calls. For any function, JS offers the `window.requestAnimationFrame (rAF)` method, which is aware of screen refresh rate and thus can throttle the amount of calls to a function. This is very useful not only in data visualization but also in browser events that fire very rapidly, such as *scroll* and *drag*. In Snippet 7, a function named *renderGlyphs* is intended to be called on *scroll* events but throttled using rAF.

In general, *requestAnimationFrame (rAF)*, *setTimeout/setInterval* and *Promises* have their own microtask queue. Therefore, the synchronous execution will execute these calls in the next event queue when the stack is empty. A *_ready_* flag is set on the called function, which is reset by an rAF (Snippet 7). If more than one call is made to the synchronous function within ~17 ms (in case of *rAFs*), it will *return* implicity.

```
/*General API pattern of
Lexicon visualization library*/
lexiconObject
.container("#someContainer")
//set width of the viewport attribute
.w(1000)
//set height of the viewport attribute
.h(200)
//set width of the style attribute
.sW("70%")
//set height of the stye attribute
.sH("200px")
/*
...Some other common methods
*/
//give an id attribute to the rendered SVG
.lexID("someID")
//internally sets _data_ variable.
.seq("ATCGATC....")
.values(
    // do something with passed _data_ and dataset
    function(_data_,dataset){
        /*
        ...
        */
    },
    dataset
    // [,additional arguments]
)
/*
Some other other common methods
*/
//synchronize this applet with other applets
.sync(someObjectReference)
//Initialize internal variables
.append()
//Render applet onto SVG
.render()
```

Snippet 1

Lexicon library API example

```
//Exporting constructed object to chosen scope
(function(){
    //define properties and do some work
    function LexiconObjectConstructor() {
        /*
        ...
        */
    }
    //Export to global or as property of some other object.
    window.lexiconObject = new  LexiconObjectConstructor;
    //or export the constructor only
    window.LexiconConstructor = LexiconObjectConstructor;
})();
```

Snippet 2

Immediately invoked function expression

```
//Enumerating object properties.
for (var i in lexiconDistribute){
    /*lexiconDistribute is the lexicon object.
    You can also test at this point whether
    a property name is inherited down from a
    parent object using Object.hasOwnProperty*/
    console.log(i)
}
/*
Will output the following key strings:
    lexID
    x
    y
    w
    h
    …
*/
```

Snippet 3

It is usually good to know which properties of a given library object are enumerable.

```
//A lazy getter
var someObjectLiteral = {
    /*remove the getter and
    replace with a DOMstring
    once called*/
    get bar(){
        delete this.bar;
        return this.bar = "foo";
    }
}
```

Snippet 4

Lazy getter.

```
//A getter that implicitly sets another property.
var someObjectLiteral = {
    bar: undefined,
    baz: 5,
    //Will implicity set bar to "foo called" while returning baz
    get foo() {
        /*You could have also referred
        to a function in outer scope,
        with this pointing to the object*/
        (function(){
            this.bar = "foo called"
        }).bind(this)();
        return this.baz;
    }
}
someObjectLiteral.bar // undefined
someObjectLiteral.foo // 5
someObjectLiteral.bar //  "foo called"
```

Snippet 5

A getter that implicitly sets another property

```
//Example of subclass creation and prototypical inheritance.
someClass () {
    /*...define some properties*/
}

someSubClass(){
    someClass.call(this)

/*...define some other properties*/
}

someSubClass.prototype = Object.create(someClass.prototype);
someSubClass.prototype.constructor = someSubClass;

//Will output false
someSubClass.prototype === someClass.prototype
//Will output true
someSubClass.prototype.__proto___ =  someClass.prototype
```

Snippet 6

Subclass creation

```
//Throttling calls to rendering function
function renderGlyphs() {
    if(!renderGlyphs._ready_){
        return;
    }
    window.requestAnimationFrame(function(){
        //perform rendering
        /*…*/
        renderGlyphs._ready_ = true
    });
    renderGlyphs._ready_ = false
}
renderGlyphs._ready_ = true;
renderGlyphs();
/* or attach it to an event that
fires rapidly */
window.addEventListener("scroll",
    renderGlyphs,
    false
);
```

Snippet 7

Throttling function calls

```
/*Creating svg elements and adding
attributes without data binding.*/
 var shapeMaker = (function(){
    function maker(){
        var _this_ = this;
        this.attr = function(name,value){
            if (!name){return this;}
            this.setAttribute(name,value);
            return _this_.attr.bind(this);
        } ;
        this.make = function(el,containerNode){
            var el = document
                .createElementNS("http://www.w3.org/2000/svg",el);
            containerNode.appendChild(el);
            return this.attr.bind(el)
        };
    }
    return new maker;
})()

/*usage*/
shapeMaker
.make("text",someNode) //select container
("x",300) //set attribute x to 300
/*...*/
/*lastly call with no arguments to
get the node itself and then set the
textContent property.*/
().textContent = "textElement";
```

Snippet 8

SVG generation without data binding

```
//Example input object for lexicon-dash
var inputObj = {
    //each point has a name and value
    points:[
        {name:"charge",value:2},
        {name:"Size",value:4},
        {name:"Hydrophobicity",value:1},
        {name:"SideChain",value:3},
        {name:"Blosum62",value:0.5}
    ],
    /*specify a range for the values of the points
    this can be an array or a function,
    the values of the points are remapped
    to the specified range and allocated
    proportional height*/
    range:[0,10],
    /*specify what will be the overall fill level.
    This number is between 0-1*/
    overall:0.7,
    /*set the red threshold level, between
    0-1*/
    threshold:0.7,
    /*specify a binary category name*/
    categories:["Low","High"]
};
```

Snippet 9

Sample lexicon-dash input object

Figure 1. Lexicon API and JavaScript OOP (object oriented programming) patterns/snippets referred to throughout this article.

VISUALIZING STRINGS OF SEQUENCE INFORMATION USING LEXICON

Bioinformatics often deals with sequence information—such as DNA, RNA or protein sequences—and its annotation. The information is mostly a simple string primitive or series of strings that are aligned based on similarity. The LexiconSeq library deals with protein sequences that are mostly composed of 20 letters (one letter per amino acid) or a gap ("-") character. An example output is shown in Figure 2A.

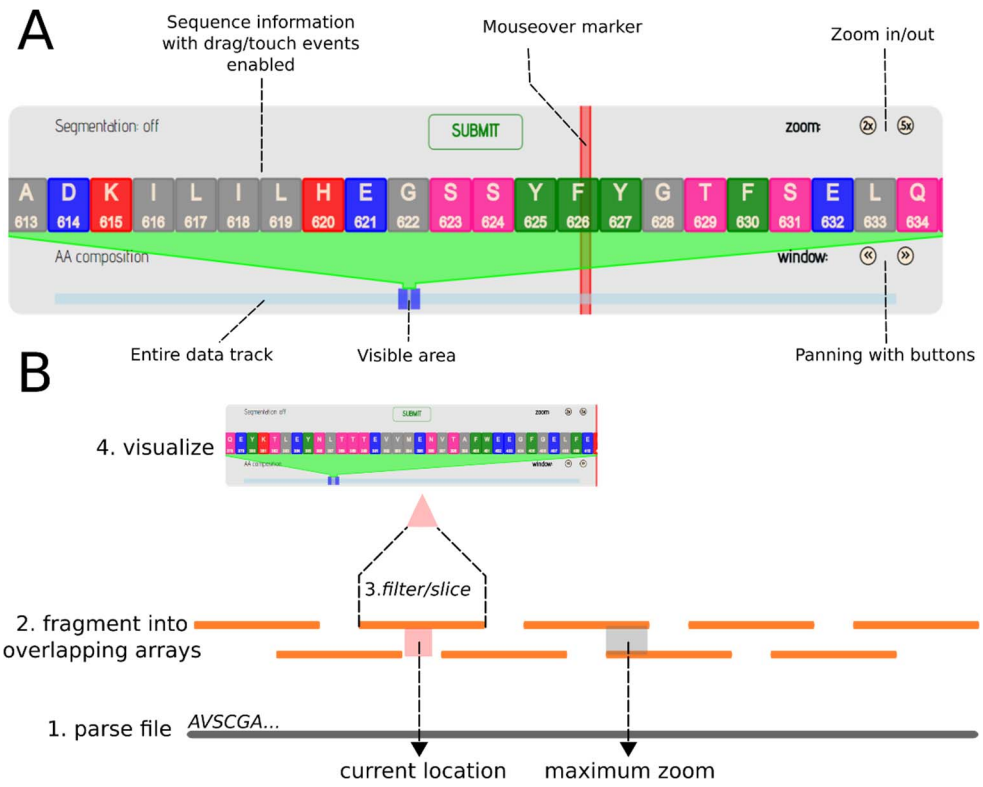

Figure 2. (A) LexiconSeq sequence visualization example. (B) LexiconSeq sequence visualization process. Parsed input is first fragmented into overlapping arrays. Subsequently selected elements within the active fragment are displayed.

Sequence information in bioinformatics is often distributed as files in FASTA format.[4] You can test several examples on mutaframe.com by typing "P36897" or "P13569" in the input box and pressing enter. Alternatively, you can drag and drop FASTA files into the input box to visualize them. In both cases, retrieved information is parsed, and *lexiconSeq...seq(..).append().render()* methods are called to render an SVG.

One important aspect in similar visualizations is whether to deploy semantic or geometric zooming. This can make a big difference in performance, especially in SVG media (as opposed to *canvas*, where the *clearRect/fillRect* method is called unless certain areas are *clipped*). The issue stems from the fact that SVG is an XML-based language and therefore each node (thus, a JS object) inherits methods/properties down the prototype chain from *Event*, *Node*, *SVGElement*, and *SVGSVGElement*, and these nodes are part of the DOM (Document Object Model). Because DOM manipulations such as adding, removing nodes, or updating attributes are expensive operations, they become bottlenecks in your application. As a rule of thumb, if you have many SVG elements, it might be better to go for geometric zoom. Otherwise, you might be able to trade-off performance for semantic zoom, where you will have more control of the individual nodes.

Semantic zooming re-renders/updates the elements on each call to *render*, whereas geometric zooming likely uses the transform attribute on the parent group for zooming and panning. Geometric zooming offers better performance, whereas semantic zooming offers layout control (you can alter the layout while zooming). The choice usually depends on the library and the application.

LexiconSeq currently uses semantic zooming. One issue with this sort of rendering is reducing the burden of DOM manipulation and other scripting processes as much as possible. One solution that this library uses is fragmenting the main data array of strings into smaller chunks of overlapping arrays where only selected elements are rendered. This effectively reduces the amount of data that a *render* function must go through before updating the DOM nodes. The process is illustrated in Figure 2B.

One implication of similar techniques is that they offer minimal performance degradation with respect to input data size. For instance, Figure 3i compares the rendering performance of the different input data, one with *String* of length 500 and another with a length of 500,000.

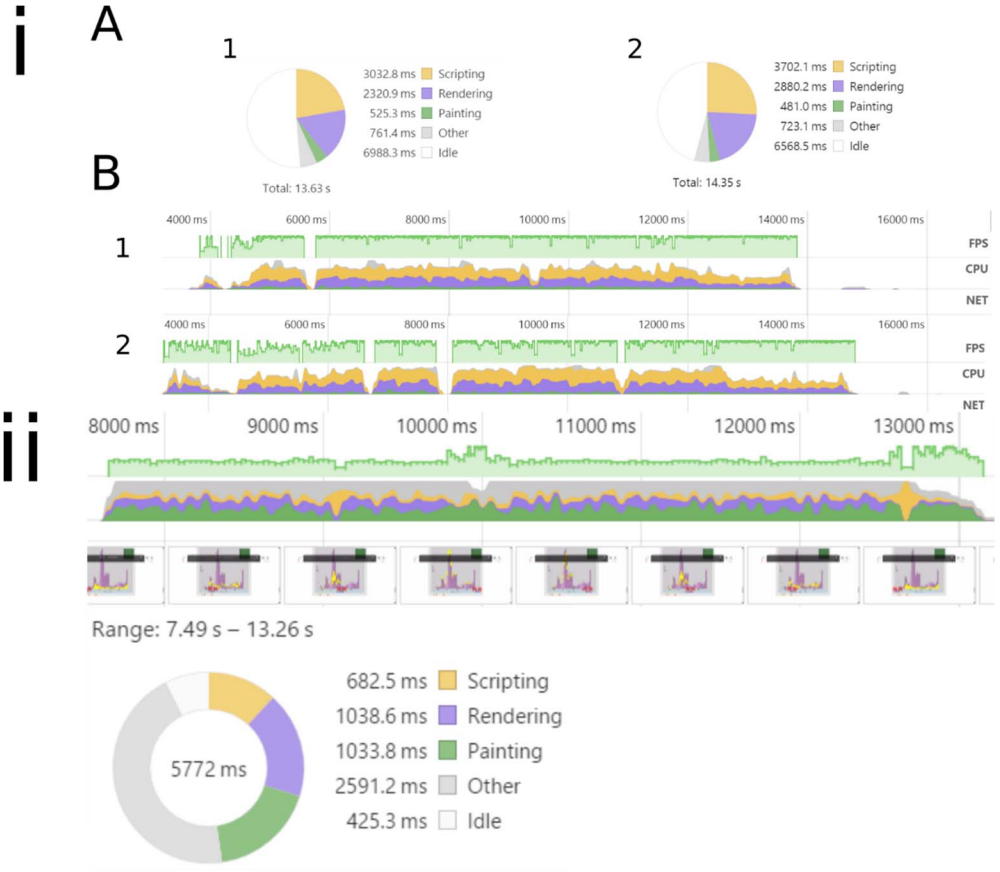

Figure 3. (i) Visualization performance comparison between sequences of 500 and 500,000 characters in length (A). Results are taken from Chrome timeline metrics (v57) while the entire length of input data is spanned within ~10 s (B). During each frame, >100 svg elements are rendered. You can download the timeline profiles from github.com/IbrahimTanyalcin. (ii) Timeline profile of a LexiconDistribute instance with approximately 10,000 svg rect elements while an active brush with 1,000 elements is dragged across the entire dataset and highlighted (yellow). You can conduct your own timeline performance measurement with Chrome using the online example at github.com/IbrahimTanyalcin/LEXICON, or see other examples at mutaframe.com.

Table 1. The metrics from Figure 3i. Regenerate these values for your own CPU profile using the live example at bl.ocks.org/ibrahimtanyalcin/24654191b656289f19c6644322658e5d.

String length	5e2	5e5
Scripting (ms)	3,032.8	3,702.1
Rendering (ms)	2,320.9	2,880.2
Painting (ms)	525.3	481.0
Idle (ms)	6,988.3	6,468.5
Other (ms)	761.4	723.1

CIRCULAR VISUALIZATION AND BEYOND FOR PROTEIN SEQUENCES

In theory, two primary conditions dictate how a protein folds (and thus gains its 3D structure to function): the solution and its amino acid sequence. With the amount of sequence data available to researchers rising, it became essential to implement tools that visualize vast amounts of sequence information. Circular visualizations have the advantage of being able to summarize the entire dataset, thus narrate the essential message to the reader. Circos was one of the first circular visualization tools developed specifically for genetics.[5] However, it was not specifically oriented for displaying protein sequences, and since it was written in perl, it was not possible to deploy interactive graphics on the web. We built on top this tool and combined it with JS, resulting in I-PV, a tool specifically for protein sequences.[6]

Researchers and medical doctors require as much data as possible regarding amino acid sequences, protein domains (regions along the protein that have functional purpose), and mutations. Therefore, we developed an add-on to I-PV (called indoril) to look at protein domains and mutations.[7] The aim was to give scores to each mutation, display these in 3D space, and link them to one another. Rather than canvas and WebGl, the add-on uses pure SVG to render shape primitives. The add-on also allows users to link certain points and change the axes simultaneously, ultimately allowing them to explore relationships between metrics and mutations. Although the end goal is to design a tool that can facilitate human-aided clustering, it remains as an experimental endeavor. A sample output can be viewed at i-pv.org/1_45/NFKB1 by clicking at the lower-left icon. The features are outlined in Figure 4 and on YouTube (www.youtube.com/watch?v=Do5MqXIi3dU)

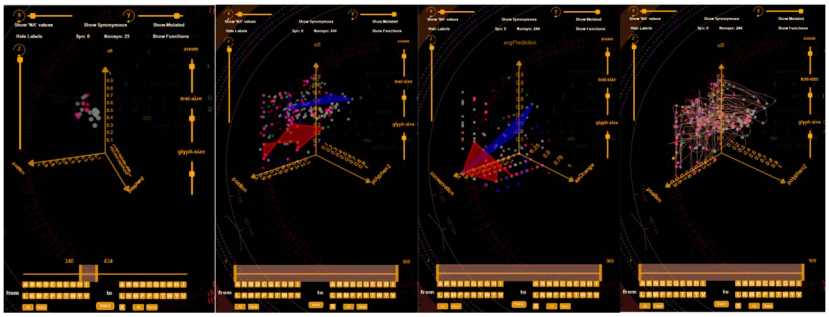

Figure 4. Indoril features. Brush through selected regions of the protein (far left). Link mutations to each other, forming simple lines or polygons (second from left) to better illustrate how selected mutations cluster. Change the axes to see how the links change with respect to different metrics (position along protein, hydrophobicity, charge, and so on). Trace points with respect to z-index (far right). You can try a sample at i-pv.org/1_45/NFKB1.

During the drag gesture, several calls to the function *renderGlyphs* are made, which draws the primitive SVG polygons. Several browsers can greatly benefit from throttling similar functions.

JS libraries such as D3 are in fact data-binding libraries (return objects that make it easier to append/modify nodes in batch), so they are not limited to visualizations. The enter, update, exit pattern of D3 can greatly reduce the DOM manipulation by selectively updating nodes (which got updated in D3v4 as *enter*, *update*, *update&enter*(*merge*) and *exit*; bost.ocks.org/mike/join). However, when interfaces start to contain too many control elements, you might want to resort to simple rendering of all elements on each scene. In those cases, you might write your own generator object to append nodes rather than use data-binding libraries (since you are not binding data). This will reduce the burden of scripting on the browser, resulting in higher frame rates. Snippet 8 is an example generator object that is also used internally in I-PV**.** In Snippet 8, if you are going to make too many repeated calls, you can use clojures instead of *bind* to save some computation, as *bind* will check first whether the bound function is called as a constructor or not and adjust the value of *this* accordingly.

Another example of function throttling can be seen in the LexiconRainbow module of the Lexicon library. This library generates a comparison between a set of ordinal scales and linear scales. It is sort of a parallel-coordinates diagram with variable sorting. The library generates a friction-enabled scroll bar that allows the user to navigate between different scales, which ultimately also calls the *render* method to update nodes. Similar designs can greatly benefit from throttling the drag gesture. An output is shown at Figure 5**.** You can test a sample at github.com/IbrahimTanyalcin/lexicon-rainbow.

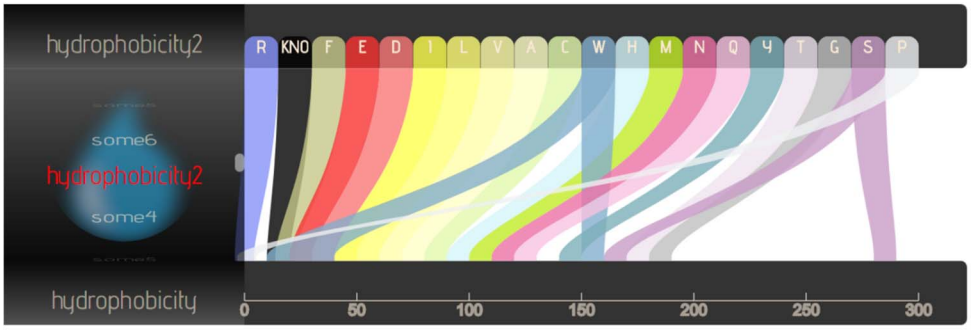

Figure 5**.** Items in ordinal scale (top) are matched to multiple regions on the linear scale (bottom). The left panel scroll bar can change both the ordinal and the linear scale on user demand, and the drag gesture is throttled using the *requestAnimationFrame* method.

Previous examples were based on SVG elements; however, the same principles can be applied to HTML elements as well.

TRANSITIONING DASHBOARDS

Dashboards can greatly simplify storytelling by narrating only part of the data. However, most of the time additional visualization is required to tell the rest of the story. In certain cases, transitioning the same dashboard's layout can be a solution. LexiconDash accepts an input object that specifies an overall score, individual data points, and a range field that can be an array of values or a function that can transform values before they are projected. A sample output and input are shown in Figure 6 and Snippet 9. Visit bl.ocks.org/ibrahimtan-yalcin/808237e5729ba4720f437fda4eab8085 for a live example.

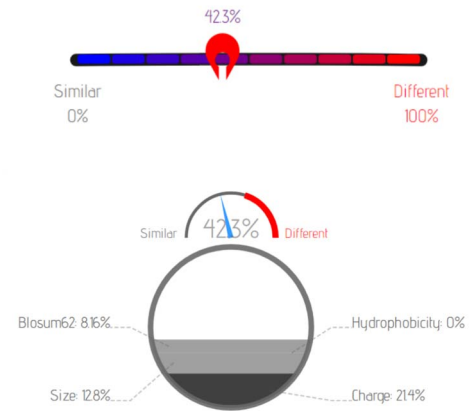

Figure 6. Transitioning dashboard between two states.

WORKING WITH DISTRIBUTIONS

Several datasets require the user to have an overview of its characteristics while still being able to have control over individual data points. While conventional distribution graphs are great at revealing trends, they tend to give little control over access to data points within individual bins. LexiconDistribute allows the user to not only generate a histogram and adjust bin size interactively but also change the layout on demand, revealing all the individual data points stacked within each bin. As opposed to LexiconSeq, LexiconDistribute uses geometric zooming, which yields better performance when more than 50,000 elements are rendered on the screen. A typical LexiconDistribute output, with a summary of its features, is shown in Figure 7.

When the individual data points are turned on, the style object of each data element (*svg rect*) is kept in memory for faster access. If the user changes the brush size or the span region, this implementation will avoid unnecessary calls to the *querySelector* or *getElementBy Id* methods. Similar applications can also benefit from *requestAnimationFrame*, where the brushing can be throtted. Figure 2ii shows the Chrome timeline profile of a LexiconDistribute instance of approximately 10,000 elements while a span region of 1,000 elements is dragged across and back through the dataset (you can try the same sample by clicking “load sample” at mutaframe.com). Although the profile is taken while several other lexicon instances are already rendered within the same page and there is no current throttling with *requestAnimationFrame*, it yields a steady 30 fps.

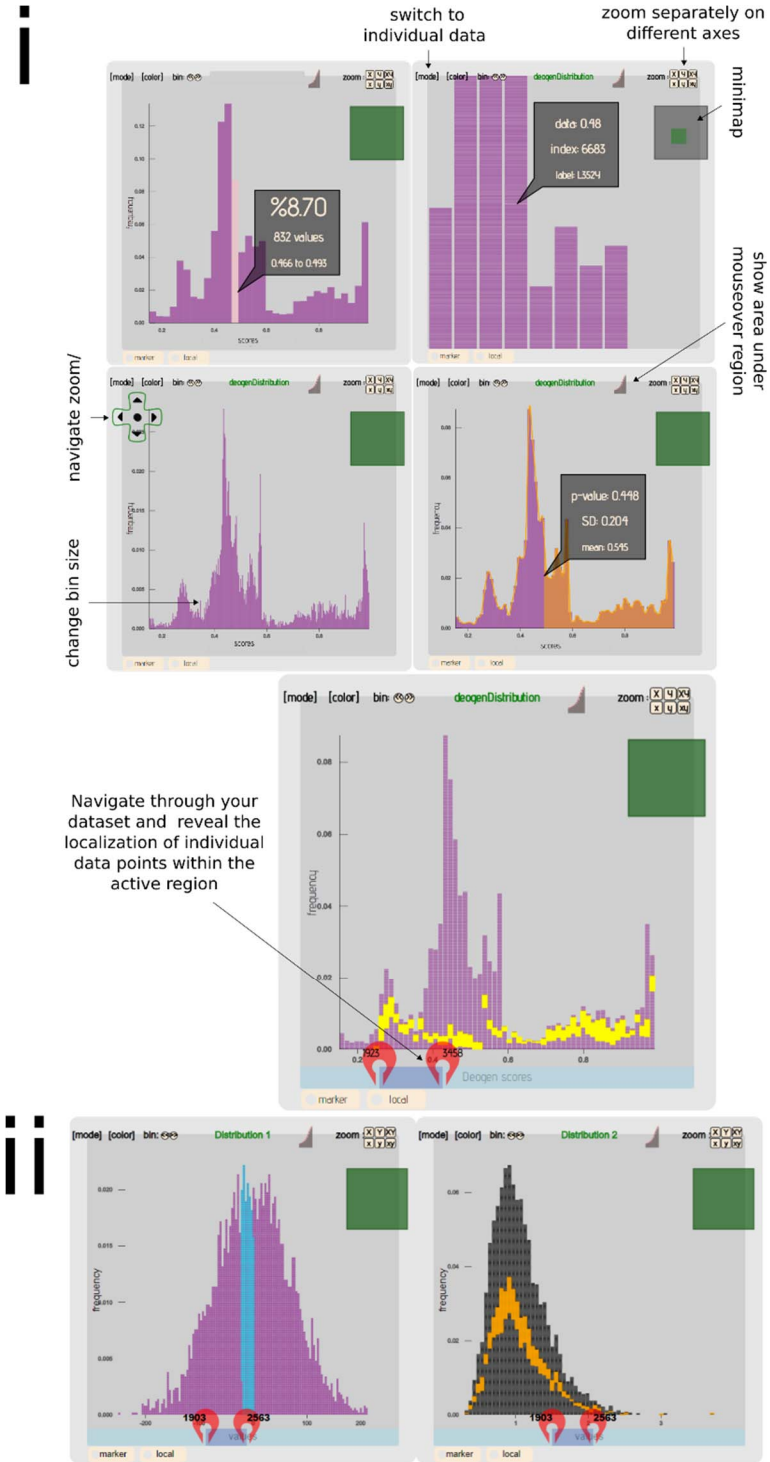

Figure 7. (i) LexiconDistribute features. (ii) Two lexicon objects are synchronized with each other. The left instance shows a normal distribution, whereas the right instance shows a lognormal distribution with half the bin size. Additionally, the values on the left are sorted in ascending order. Thus, the difference in the sorting can be revealed by brushing through the datasets. You can try a live sample at bl.ocks.org/ibrahimtanyalcin/4b99a23b09bf18123c0a1e440085af7d.

Table 2. Rendering times of 5,000, 10,000, and 20,000 elements with Intel i7-5700HQ 2.7 GHz CPU. (These measurements are a separate instance from Figure 3ii.) Regenerate metrics for your CPU by modifying the examples at github.com/IbrahimTanyalcin/LEXICON.

Data points	5e3	1e4	2e4
Range (s)	2.04–9.45	2.54–10.64	3.24–8.67
Scripting (ms)	435.7	395.2	205.7
Rendering (ms)	1261.1	1373.5	1987.9
Painting (ms)	2206.8	1387.6	1010.2
Idle (ms)	529.4	659.2	740.1
Other (ms)	2969.9	4279.2	1483.4

ONE OBJECT, MANY FACES

Aside from giving users the ability to customize the outlook, a library can also allow users to morph the layout itself. For instance, the LexiconDistribute shown in Figure 7i displays the mutation score on the *x*-axis and the frequency on the *y*-axis. The library operates on an input array where, for each data point, a data, index, and label are provided together with a transformation function (which can also be *identity function* if the user does not want to change the data). By swapping the data with the label and changing the transformation function, the user can transform the entire layout to a heat map. Combined with the *fixedBin* method of LexiconDistribute, which forces bins to contain only the specified amount (or less if there are no more points; implicitly disables bin adjustment), the user can obtain a square lattice of data points. As a result, the same dataset shown in Figure 7i can be represented as in Figure 8, but this time with the *x*-axis showing the length of protein (residue count) and *y*-axis showing the 20 possible amino acids that can replace the original residue, colored with respect to mutation score (deleterious is red, benign is blue).

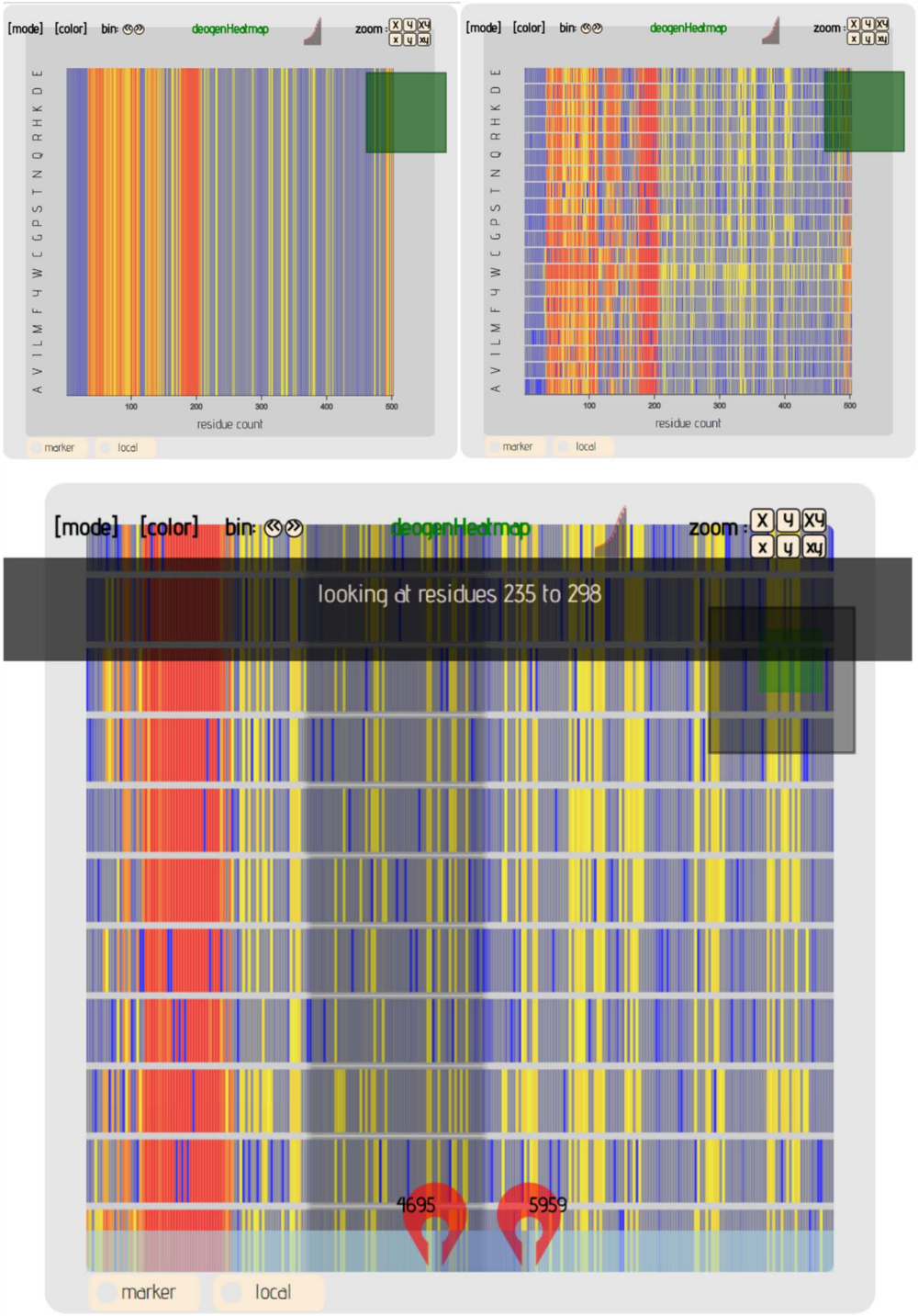

Figure 8. LexiconDistribute heat map using *fixedBin* method with 20 items per bin. Initially, the average of 20 values are shown in each bin (left). Clicking the “mode” button reveals individual data points (right). The local button turns on brushing which passes the *DOM event* (or *d3.event*), lower and upper bounds to the user-specified function to display additional information or any other operation (bottom).

FROM REUSABLE TO SYNCHRONIZABLE COMPONENTS

Being able to reuse visualizations for different datasets allows developers a certain degree of flexibility—they can mix and match based on their goals. However, it is also important that these visualization modules be able to communicate with one another. This can be achieved using vanilla JS or frameworks. However, designing the libraries in such a way that they are synchronizable with one another from the start can reduce the burden of converting the dataset on the fly. Several Lexicon libraries can already synchronize with one another, such as LexiconDistribute, LexiconSeq, LexiconPlot, and LexiconCompare. This is achieved by different visualization objects calling *render* on each other while passing the current *zoom (viewport)* and *offset*. An example is depicted in Figure 9. You can try the same example by following the instructions at github.com/IbrahimTanyalcin/LEXICON.

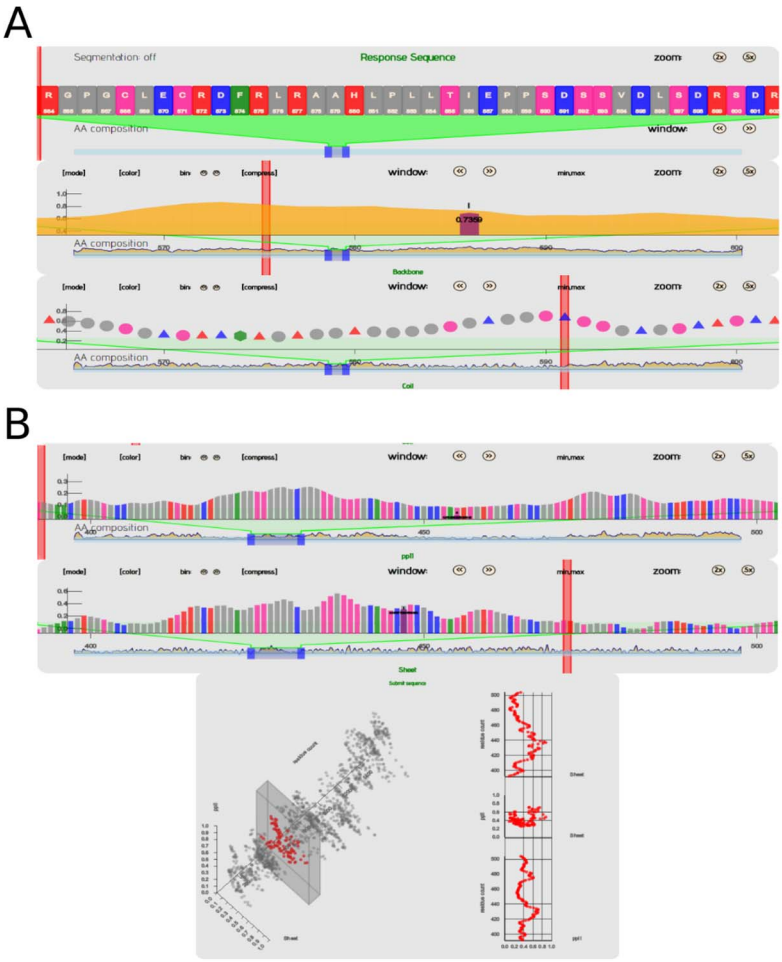

Figure 9. Synchronization of Lexicon modules. (A) LexiconSeq (top) and LexiconPlot (bottom) are synchronized with each other; so when a mouse or touch gesture occurs, they respond together. (B) Some lexicon modules, like LexiconCompare (bottom) are initialized upon calling *sync.* The top and bottom Lexicon modules are compared to each other by calling sync on LexiconCompare (bottom), which automatically gets input from the two other modules and renders the svg. If *sync* is called from any of modules from Panel B, passing in one of the modules from A, then the whole set of six modules (panels A and B together) are all synchronized (for example, when you drag the box to the bottom of panel B, all visualizations will update). Watch the tutorial video at www.youtube.com/watch?v=8leI7J4Y-zI.

One important aspect of synchronization is that you should avoid circular references because this can bloat the call stack. You can write your custom scripts to take care of similar problems. A sample solution can be found at github and at deogen2.mutaframe.com/synctor.js .

SUMMARY

There are multiple ways to perform the same tasks in JS. Although this can sometimes be detrimental, it nevertheless allows the language to expand while still preserving backward compatibility. In this article, we illustrated a few examples of how JS can be used in a scientific context. Media such as canvas or WebGl and other influential libraries or technologies can also be coupled with JS to extend its usage in science.

We also described here our Lexicon visualization library, which is a collection of reusable graph components that can be coupled with one another. Lexicon can be used in bioinformatics a well as extended to other scientific branches.

One of the biggest advantages of JS is that it provides access to a tremendous number of libraries and costs nothing to set up. Incorporating this language into scientific routines can help researchers broaden the range of their services as well as increase their efficiency when developing new technologies.

ACKNOWLEDGMENTS

This work was supported by the European Regional Development Fund (ERDF) and the Brussels-Capital Region-Innoviris within the framework of the Operational Programme 2014-2020 through the ERDF- 2020 project ICITY-RDI.BRU

REFERENCES

1. J.D. Hunter, “Matplotlib: A 2D Graphics Environment,” *Computing in Science & Engineering*, vol. 9, no. 3, 2007, pp. 90–95.
2. “Identification and Analysis of Functional Elements in 1% of the Human Genome by the ENCODE Pilot Project,” *Nature*, vol. 447, no. 7146, 2007, pp. 799–816.
3. M. Bostock, V. Ogievetsky,, and J. Heer, “D3: Data-Driven Documents,” *IEEE Trans. on Visualization & Computer Graphics*, vol. 17, no. 12, 2011, pp. 2301–2309.
4. D.J. Lipman and W.R. Pearson, “Rapid and Sensitive Protein Similarity Searches,” *Science*, vol. 227, no. 4693, 1985, pp. 1435–14341.
5. M.I. Krzywinski et al., “Circos: An Information Aesthetic for Comparative Genomics,” *Genome Res.*, vol. 19, no. 9, 2009, pp. 1639–1645.
6. I. Tanyalcin et al., “I-PV: a CIRCOS Module for Interactive Protein Sequence Visualization,” *Bioinformatics*, vol. 32, no. 3, 2016, pp. 447–449.
7. I. Tanyalcin et al., “Indoril: An I-PV Add-On for Visualization of Point Mutations on 3D Cartesian Coordinates,” *bioRxiv*, vol. 148122, 2017; doi.org/doi.org/10.1101/148122.

ABOUT THE AUTHORS

Ibrahim Tanyalcin is a postdoctoral researcher at the $(IB)^2$ Interuniversity Institute of Bioinformatics in Brussels. Tanyalcin received a PhD in malformations of cortical development from Vrije Universiteit Brussel. Contact him at itanyalc@vub.ac.be.

Carla Al Assaf is a PhD student working on essential thrombocythemia at KU Leuven. Contact her at dr.carla.assaf@gmail.com.

Julien Ferte was a postdoctoral researcher at the (IB)2 Interuniversity Institute of Bioinformatics in Brussels. He received a PhD in computer science from the University of Aix-Marseille I. Contact him at iqc.ferte@gmail.com.

François Ancien is a PhD student at the (IB)2 Interuniversity Institute of Bioinformatics in Brussels, studying methods of predicting the deleteriousness of mutations in the human exome from structural information. He received a master's degree in bioinformatics from the Université de Liège. Contact him at f.ancien6@gmail.com.

Taushif Khan is a postdoctoral researcher at the (IB)2 Interuniversity Institute of Bioinformatics in Brussels, working on the identification of protein topology from sequence. He received a PhD in computational biology and bioinformatics from Jawaharlal University. Contact him at taushifkhan@gmail.com.

Guillaume Smits is a clinical director assistant at Queen Fabiola Children University Hospital and is affiliated with the Université Libre de Bruxelles and the (IB)2 Interuniversity Institute of Bioinformatics in Brussels. His research interests include machine learning to decipher the genetic architecture of oligo- to polygenic disorders, the BRiDGEIris: BRussels big data platform for sharing and discovery in clinical genomics, deciphering oligo- to polygenic genetic architecture in brain developmental disorders, and a prediction of coding variant effects using protein dynamics and stability information. Smits received a PhD in the medical sciences from the Université Libre de Bruxelles. Contact him at guillaume.smits@huderf.be.

Marianne Rooman is the research director of the Belgian Scientific Research Fund and a professor in the Faculty of Sciences and the School of Engineering at Université Libre de Bruxelles (ULB). Her current research involves the development and application of tools for rational protein design, and for the identification and rationalization of disease-causing mutations in human proteins. She is also active in the mathematical modeling of biological networks using deterministic and stochastic differential equations. Rooman received PhDs in theoretical high-energy physics and structural bioinformatics from Université Libre de Bruxelles. Contact her at mrooman@ulb.ac.be.

Wim Vranken is a research professor at Vrije Universiteit Brussel (VUB) and VUB director of the (IB)2 VUB/ULB Interuniversity Institute of Bioinformatics. His research centers on the prediction of biophysical and other characteristics of proteins from their sequence. Vranken received a PhD in Chemistry from the Universiteit Gent. Contact him at wim.vranken@vub.be.