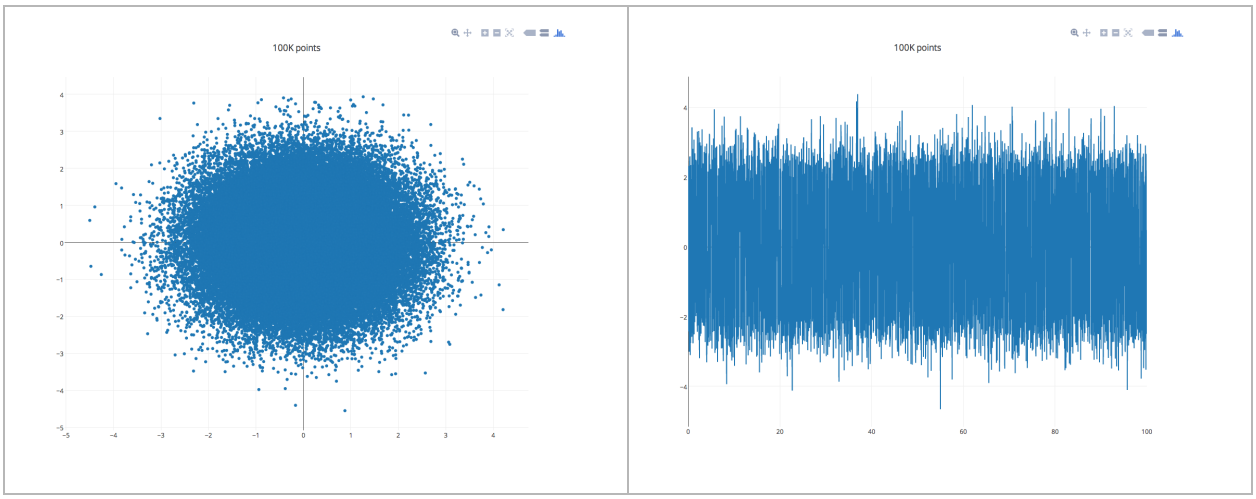
plotly

Visualizing Data: Scalable Interactivity

The best data visualizations illustrate hidden information and structure contained in a data set. As access to large data sets has grown, so has the need for interactive and scalable solutions which connect computer science, mathematics, and industry.

Plotly is an online data visualization tool designed to quickly render graphs in web browsers. Our user base comprises of a wide variety of clients from industry and scientific labs. As our clients have access to larger data sets, demand is growing to enable the rendering of huge two-dimensional plots.

Our primary goal for this workshop is to find an executable solution that will allow us to improve our rendering to allow data sets of up to 1 million elements, while preserving interactive and exportable features that our clients expect.



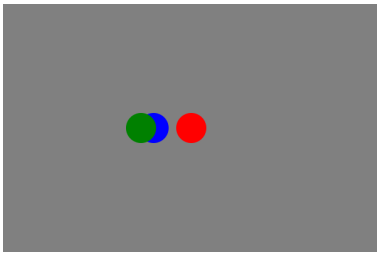
Current technology utilized by Plotly's graphing tools:

Plotly currently utilizes two methods for rendering graphics. For two-dimensional plots, we use D3.js, a JavaScript graphing library that utilizes SVG (scalable vector graphics). Our three-dimensional plots are rendered using WebGL, a low-level graphics API that uses GPUs directly for extremely fast rendering.

Benchmarks: Our tool renders approximately 100,000 points in the browser using SVG, however, interactivity begins to slow around 30,000 points. A more comprehensive list of current benchmarks can be found at: https://plot.ly/benchmarks/

SVG is a vector markup language for describing two-dimensional graphics. Graphics are drawn based upon a list of SVG elements, which are essentially polygons/spline curves with associated drawing commands. For example the element `<circle cx=”150” cy=”100” r=”80” fill=”red” />` is an SVG element that describes a circle with red fill, with given x and y coordinates, and given radius, r. SVG draws the elements in the order that they appear in the list.

Example: To produce the image



We use the code:

```
<svg version="1.1"
     baseProfile="full"
     width="300" height="200"

xmlns="http://www.w3.org/2000/svg">

  <rect width="100%" height="100%" fill="grey" />

  <circle cx="150" cy="100" r="12" fill="red" />

  <circle cx="120" cy="100" r="12" fill="blue" />

  <circle cx="110" cy="100" r="12" fill="green" />

</svg>
```

Because every SVG element is drawn, complicated SVG files take longer to render. For Plotly, this means that to draw a scatter plot with 300,000 points, each point of the scatter plot is an SVG element.

There are several important benefits to SVG, from which Plotly benefits: First Plotly graphs take advantage of the graphing library D3.js, which is highly developed and offers many interactive options. Second, SVG renders on any device, is very stable, is resolution independent and supports fast development.

There are some disadvantages to SVG, most notably the drawing complicated images is slow for plots with many points. Similarly updating these figures during user interactions is time consuming.

WebGL is a low level graphics API that uses GPUs directly for extremely fast rendering. It is based on OpenGL, but is designed for the web. To render an image in WebGL, functions send data to your GPU for processing, and images are drawn on top of a canvas element. Shapes are created by scripts that contain vertex and *fragment shaders*, that assign colors to each pixel contained in the shape.

WebGL is fast, offers high performance rendering, good interactivity and excellent control. On the other hand, development is expensive (it takes more time to develop features than in SVG), and support is limited. Also it is not as well suited as SVG for generating archival figures, as WebGL renderings are resolution dependent.

Server side: In addition to drawing considerations (whether using SVG or WebGL), our solution will likely require thought about how Plotly stores plots, and how stored data is downloaded to the browser. Specifically, we require an efficient way to store and retrieve data server-side based on a plot viewport.

Goals:

Initial goals for this project involve finding graphing solutions for two-dimensional scatter plots and line graphs for data sets of around 1 million points.

In the short term, we would like this solution to be implemented using SVG. The outline of a solution may look something like the following:

1. An efficient way to store and retrieve data server-side based on a plot viewport.
2. A way to decide on the server which points should be sent to the client, so that the plot looks effectively just as it would if you plotted everything.
3. The client will then query the server on zoom/pan events to update the data.

In the longer term, we expect that WebGL will be an important component of our 2D graphing library.

Important considerations:

- It is essential that we maintain existing features. In particular, users must be able to highlight individual points (this is done by hovering your mouse over a point in your plot). We also need to preserve advances styling features like custom glyphs, colors, sizes, borders, etc.
- Performance constraints:
    - Allow at most O(n polylog(n)) steps during preprocessing, where n is the number of data points.
    - Storage of at most O(n) points.
    - Algorithms must be simple enough to allow implementation in a reasonable amount of time in order to be cost effective.

Problems:

Scatter plots:

Definition: Given a set of n vectors in 2D, and a shape S (which we call the *marker)*, a scatter plot is the image formed by the union of n copies of S translated by each vector.

1. SVG: Given a scatter plot with circular markers, find the curve consisting of the minimal number of circular arc-segments representing the boundary of the scatter plot.

   Motivation: How can we reduce the number of curves that need to be rendered, while preserving the exact image of the plot?

   Variants:
   a. Warm up: Suppose that the marker shape is a circle with constant radius.
   b. Allow the radii of the markers to vary.
   c. Allow the marker shape to vary (crosses, stars, etc.).
   d. Allow the marker colors to vary.
   e. Allow the markers to have borders of different colors
   f. Suppose the markers are partially transparent. It is possible to compute a minimal set of arcs which will render correct opacity?
2. WebGL: We have the following general questions to investigate:
   a. How should we support prioritized/progressive loading?
   b. What is the best way to implement user interactivity and selections?
   c. How should transparent shapes be handled?

Line plots:

Definition: A line plot is a piecewise linear curve.

Note: We currently use a min/max decimation algorithm that gives us better performance than scatter plots.

1. SVG

   Motivation: How can we reduce the number of elements that need to be rendered, while preserving the exact image of the plot?
   a. Warm up: Filter a curve by removing nearby points.
   b. Suppose a line plot has dense regions. How can lines be replaced by polygons?

    c. In using decimation algorithms, how can support of hovering, line colors, widths, etc. be preserved?
2. WebGL:
    a. How can streaming graphs be implemented?
    b. How should we support prioritized/progressive loading?