

From interactive visualization web-application to JavaScript library: a roadmap for parasol.js

Produced by:

Kasprzyk Research Group

Civil, Environmental and Architectural Engineering Dept.

University of Colorado Boulder

josephkasprzyk.wordpress.com

Last updated: May 8, 2018

Table of contents

- [Introduction](#)
- [Features](#)
 - [Import data](#)
 - [K-means clustering](#)
 - [Linked brushing](#)
 - [Reset brushes](#)
 - [Marking](#)
 - [Clear markings](#)
 - [Reorderable axes](#)
 - [Hide and show axes](#)
 - [Set axes limits](#)
 - [Keep and Remove selection](#)
 - [Export selection as csv](#)
 - [Explore selection](#)
- [Discussion](#)

Introduction

parasol is a new tool providing interactive visualization and statistical analysis methods for multiobjective optimization problems. *parasol* is built on D3.js, a JavaScript library for data-driven documents, and the associated parallel coordinates library, d3.parcoords.js. This document describes the current state of *parasol* as a web-application and provides a

comprehensive outline of the mechanics of each feature, including an overview of all associated code and intended functionality.

As a web-application, *parasol* serves as an approachable and streamlined tool for decision makers to explore many-objective optimizations. However, as a Javascript library, *parasol* could provide the same functionality in a more versatile manner. Therefore, in addition to describing the functionality of *parasol*, the relevance of each feature to a library implementation and potential course for transition are additionally addressed throughout.

Features

Import data

Currently, only .csv file uploads are supported. In the future, this may extend to other file formats and database connections. Once the data has been uploaded, the uploader must be removed due to an issue with re-uploading data without reloading the webpage.

Note: If the user is building a personalized tool through the library, they will likely have a specific dataset to automatically import. In this case, functionality is already handled by `d3.csv`.

Current implementation details

The current csv uploader functions as a button and is the first thing the user sees when opening the app. They must click the button and select a csv file from their directory before the plots and grid can be produced. Once a file is selected, the app enters the `visualize` function which drives the interactive capabilities, and the uploader is removed.

```
// create button
<input type="file" id="uploader">
```

```
var uploader = document.getElementById("uploader");
var reader = new FileReader();

reader.onload = function(e) {
  var contents = e.target.result;
  var data = d3.csv.parse(contents);

  // visualize data with default to 3 clusters
  visualize(data, n_objs = 3, k = 4);

  // remove uploader button
```

```
uploader.parentNode.removeChild(uploader);  
};  
  
uploader.addEventListener("change", handleFiles, false);  
  
function handleFiles() {  
  var file = this.files[0];  
  reader.readAsText(file);  
};
```

Proposed library API

This feature is relevant for a two main purposes. It is convenient for those who would like to test their code on multiple files without having to edit the file path, and to those using *parasol* to develop a web application similar to what we have done here.

In the library, this should be a function that takes no argument, and simply appends the uploader button to the webpage.

```
parasol.import()
```

K-means clustering

K-means clustering partitions the imported data into into k clusters in which each datapoint belongs to the cluster with the nearest mean. In this way, clustering groups the solutions by a measure of statistical similarity. These clusters are denoted by corresponding categorical colors depending on the number of clusters specified. The default value is set to 3 clusters, with a maximum value of 6 clusters. This cap is due in part to the loss of relevance with too many clusters, and to the limited number of catagorical colors in the "Dark2" color pallete. Of course, the latter can be easily remedied should we choose to increase the maximum number of clusters.

Current implementation details

Clustering is implemented using the *ML* library.

```
var kmeans = ML.Clust.kmeans;
```

In a library-like functionality, the web-app developer currently specifies the number of clusters in the call to the `visualize` function.

```
visualize(data, n_objs = 3, k = 4);
```

The first task of the `visualize` function is to setup the clusters. We use the *Underscore* library to obtain data in the necessary array format for the *ML* library, perform clustering on the array version of the data, and append clusters id's to the original data.

```
// max clusters = 6
k = (k <= 6) ? k : 3;

// coerce data to array of arrays for clustering
var clust_form = [];
data.forEach(function(d,i) { clust_form[i] = _.values(d) });

// perform default clustering
var km = kmeans(clust_form, k);
data.forEach(function(d,i) { d.cluster = km.clusters[i]; });
```

We then define a color palette based on the number of specified clusters.

```
// choose default categorical color scheme
var color_scheme = d3.scale.ordinal()
  .range(colorbrewer.Dark2[k])

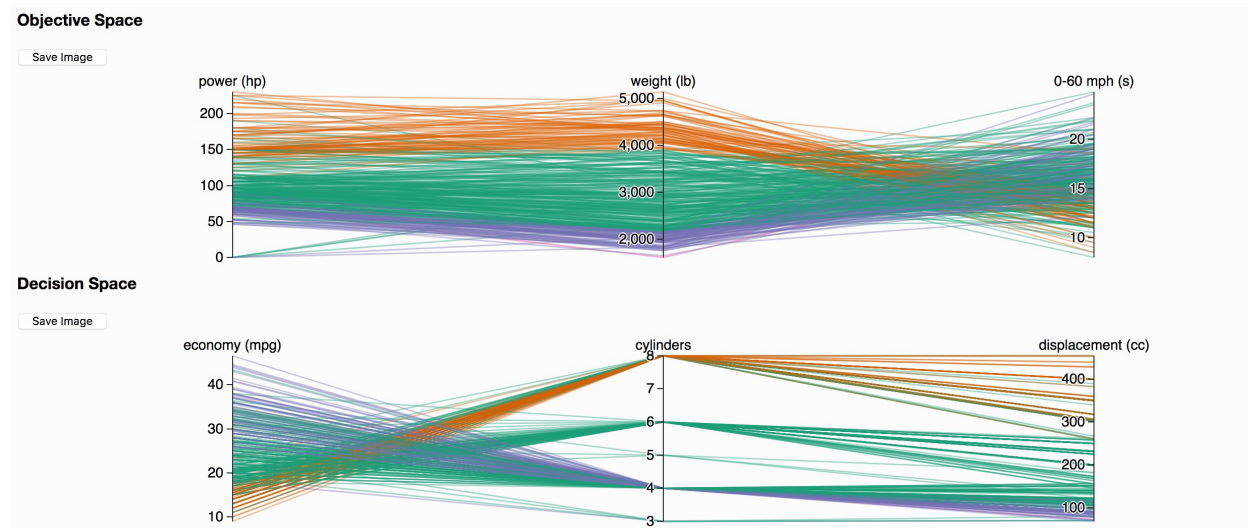
var palette = function(d) {
  return color_scheme(d['cluster']);
};
```

This `palette` variable is then passed to the parallel coordinate plot variables, so that each observation will be colored by its cluster id. For example,

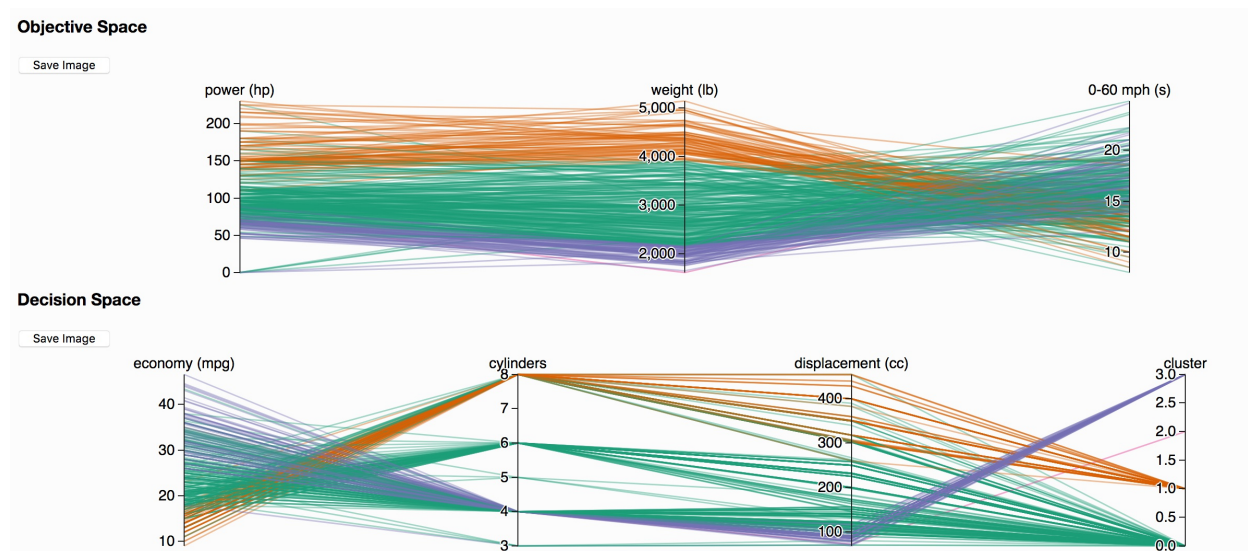
```
// objective space
var pc1 = d3.parcoords("#plot01")
  .data(data)
  .hideAxis(decision_vars)
  .color(palette)
```

In addition to representing clusters by color, users may also choose to view clusters on an axis for interactive [brushing](#). Because clusters id's are associated with each row of data, an axis is automatically created for them when plotting. By default, the cluster axis is currently hidden to avoid repetitive display of data. However, using the [Hide and Show axes](#) feature, they can easily be revealed. See the images below for reference.

Default: Clusters hidden on import.



After cluster axis selected in [Hide and Show axes](#).



Proposed library API

Clustering functionality should be an extension of the main `visualize` function. The user will specify the following:

- `k`: number of clusters
- `type`: {"k-means", "spectral"} the type of clustering to be preformed
- `group`: {"decisions", "objectives", "both"} the group on which clusters will be decided

Defaults are listed below.

```
visualize(data, num_objectives)
    .cluster(k = 3, type="k-means", group="both")
```

Linked brushing

Coming soon.

Current implementation details

Proposed library api

Reset brushes

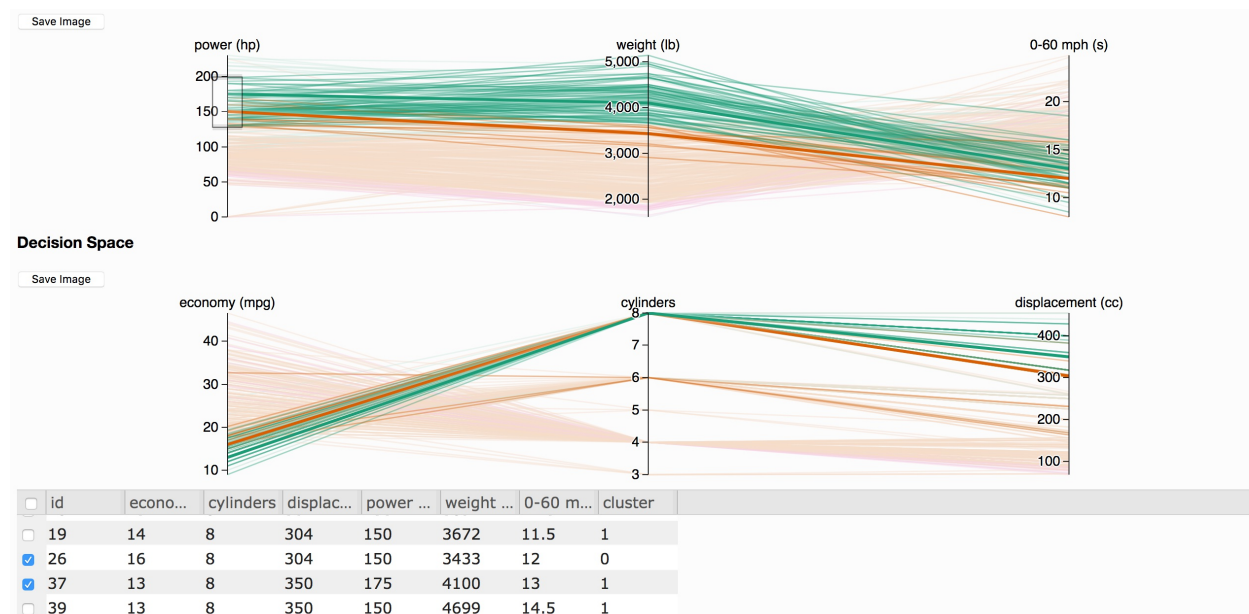
Coming soon.

Current implementation details

Proposed library api

Marking

Marking is a key feature in many visualization tools. It allows the user to view individual solution data and permanently highlight solutions of interest. See the image below for an example.



Current implementation details

Marking is currently implemented under the terminology "selections." This is because, similar to brushed data, markings are handled in their own canvas layer. In the stock version of the *parallel-coordinates* library, there already exists a *marks* layer which may have been included with plans for this feature in mind. However, to avoid potential conflict, we have temporarily

chosen to call this layer *selections* as shown below. We will soon attempt to use the *marks* layer alone however.

```
var pc = function(selection) {
  selection = pc.selection = d3.select(selection);

  __.width = selection[0][0].clientWidth;
  __.height = selection[0][0].clientHeight;

  // canvas data layers
  ["marks", "foreground", "brushed", "highlight", "selections"]
  .forEach(function(layer) {
    canvas[layer] = selection
      .append("canvas")
      .attr("class", layer)[0][0];
    ctx[layer] = canvas[layer].getContext("2d");
  });

  // svg tick and brush layers
  pc.svg = selection
    .append("svg")
    .attr("width", __.width)
    .attr("height", __.height)
    .style("font", "14px sans-serif")
    .style("position", "absolute")

    .append("svg:g")
    .attr("transform", "translate(" + __.margin.left +
      ", " + __.margin.top + ")");

  return pc;
};
```

The other key edit to the *parallel-coordinates* library involves styling the *selections* layer. In the CSS stylesheet, we add a *dimmed* state to the canvas layer options as follows (file: `d3.parcoords_ucb.css`):

```
/* custom "dimmed" class for selections layer */
.parcoords canvas.dimmed {
  opacity: 0.85;
}
```

This is applied to the foreground and brushed layers when selections exist to increase contrast between layers, making the selections more prominent. This design aspect is handled in the custom `select` and `deselect` functions we've added to the *parallel-coordinates* library (file: `d3.parcoords_ucb.js`):

```

// select an array of data
pc.select = function(data) {
  if (arguments.length === 0) {
    return __.selected;
  }

  // add row to already selected data
  __.selected = _.union( __.selected, data);
  data.forEach(path_selections);
  d3.selectAll([canvas.foreground, canvas.brushed]).classed("dimmed", true);
  return this;
};

// clear selections layer
pc.deselect = function() {
  __.selected = [];
  pc.clear("selections");
  d3.selectAll([canvas.foreground, canvas.brushed]).classed("dimmed", false);
  return this;
};

```

Wrapping this feature into *parasol* involves a few more additions to the *parallel-coordinates* library. These are listed below (file: `parcoords_ucb.js`):

- add the array element *selected* to the `d3.parcoords` function
- add default styles to selections layer:

```

ctx.selections.lineWidth = 3;
ctx.selections.shadowColor = "#ffffff";
ctx.selections.shadowBlur = 10;
ctx.selections.scale(devicePixelRatio, devicePixelRatio);

```

- create `renderSelections` function

```

pc.renderSelections = function() {
  if (!d3.keys(__.dimensions).length) pc.detectDimensions();

  events.render.call(this);
  return this;
};

```

- default


```
pc.renderSelections.default = function() {
pc.clear('selections');

if (__.selected.length) {
  __.selected.forEach(path_selections);
}
};
```

- selectedQueue

```
var selectedQueue = d3.renderQueue(path_selections)
  .rate(50)
  .clear(function() {
    pc.clear('selections');
  })

pc.renderSelections.queue = function() {
  if (__.selected.length) {
    selectedQueue(__.selected);
  } else {
    selectedQueue([]);
    // This is needed to clear the currently selected items
  }
};
```

- path function

```
function path_selections(d, i) {
  ctx.selections.strokeStyle = d3.functor(__.color)(d, i);
  return color_path(d, ctx.selections);
};
```

Note: The exact construction details of this functionality parallel the *highlighting* feature, with the exception of the rendering component which parallels *brushing*. That is, for line-for-line implementation detail, simply search the terms "highlight" and "renderBrushed" in the `parcoords_ucb.js` file, and the additions for *selections* can be found and traced in this way.

Now that we have added *selections* capability to the *parallel-coordinates* library, we can include this feature in *parasol*. A selection is made when the user clicks the checkbox for a row in the data grid. This calls the `pc.select` function which increases line thickness and contrast. In the future, we plan to give the user the freedom to assign a new color to the selection as well. We include this checkbox functionality as follows:

```

var checkboxSelector = new Slick.CheckboxSelectColumn({
  cssClass: "slick-cell-checkboxsel"
});
// add checkboxes to left grid
columns.unshift(checkboxSelector.getColumnDefinition());

var dataView = new Slick.Data.DataView();
var grid = new Slick.Grid("#grid", dataView, columns, options);
grid.setSelectionModel(new Slick.RowSelectionModel({selectActiveRow: false}));
grid.registerPlugin(checkboxSelector);

// keep checkboxes matched with row on filter/brush
dataView.syncGridSelection(grid, preserveHidden=false);

```

When a selection is added or removed, the following code registers the change and updates the grid and plots as necessary. Note that we assess whether brushed data exists as an extra measure of security, ensuring that nothing outside of the brush extents will be selected.

```

grid.onSelectedRowsChanged.subscribe(function (e, args) {
  // reset and update selected rows
  var selected_row_ids = grid.getSelectedRows();
  var brush_union = _.union(pc1.brushed(), pc2.brushed());
  var d;
  if (brush_union.length) {
    d = brush_union;
  } else {
    d = data;
  }
  pc1.deselect();
  pc2.deselect();
  selected_row_ids.forEach(function(i) {
    pc1.select([d[i]]);
    pc2.select([d[i]]);
  });
});

// fill grid with data
gridUpdate(data);

```

Proposed library api

Marking functionality is an integral aspect of *parasol* and will be included by default. Because it is inherently interactive, it will not be a function. Instead, when a user calls the main `visualize` function they will automatically be able to perform interactive marking. An exception may need to be made in order to allow the user to specify the marking color however.

Clear markings

While markings can be cleared on an individual basis, we find that it is also convenient to have a method for clearing all markings instantly. This is very straightforward since all we need to do is deselect all rows. This is implemented with a button click as follows:

```
<button id="clear_selected">Clear Selections</button>

d3.select('#clear_selected').on('click', function() {
  // deselect all elements in grid (fires event)
  grid.setSelectedRows([]);
});
```

In a library implementation, this should be as simple as

```
parasol.clearMarkings()
```

Reorderable axes

The reorderable axes functionality is not a novel feature of this project. It is completely implemented in Kai Chang's *parallel-coordinates* library. Even still, it is included as a feature due to its significance for *parasol*. Whether implemented as a web-application or library, it is critical that the user have the ability to interact with axes of the produced plots in order to overcome the bias of the static relationships between variables. The implementation is simple, one need only extend the plotting variables as follows:

```
// objective space
var pc1 = d3.parcoords()("#plot01")
  .data(data)
  .hideAxis(decision_vars)
  .reorderable()
```

Hide and Show axis

Coming soon.

Current implementation details

Proposed library api

Set axes limits

Coming soon.

Proposed library api

Keep and Remove selection

One of the primary methods for quickly identifying relevant solutions is the ability to narrow down the set of visualized data by removing unnecessary data or keeping only a small subset of relevant data.

Note: There are three possible fields of data the user may seek to keep (or remove): brushed data, marked data, or the union of both. For the `keep` feature, the current default is to keep only the union of the brushed and marked data. For the `remove` feature, the current default is to remove all brushed data that is not marked.

Current implementation details

This feature pair is currently implemented with buttons which we create at the beginning of the script in the widgets div.

```
<button id="keep_selected">Keep</button>
<button id="remove_selected">Remove</button>
```

The critical detail for these features is that the entire plotting div is stored in a single variable so that it can be referenced with the *jQuery* library when we need to reinitialize the plots with reduced data.

```
// capture both plots in variable so both can be completely
// redone using keep and remove functionality
var plot = `<h3>Objective Space</h3>
<button id="svg01">Save Image</button>
<div id="plot01" class="parcoords" style="height:200px; width=100%;"></div>

<h3>Decision Space</h3>
<button id="svg02">Save Image</button>
<div id="plot02" class="parcoords" style="height:200px; width=100%;"></div>
`;

// initialize plot
$("#div#plots").html(plot);
```

Once clicked, both features build a new data variable based on the chosen region, and check that it is not empty. The plot variable is then referenced and cleared using *jQuery* and then reassigned by a call to the main `visualize` function with the new data variable as an argument. Clearing the plot canvas is necessary so that plots do not overlap upon reassignment.

Note: Clusters are recomputed in the process of plotting the reduced data.

The current implementation of `keep` is as follows:

```
d3.select('#keep_selected').on('click', function() {
  // delete all data not selected and do complete refresh
  // NOTE: selected/brushed data equivalent between plots at this stage
  data = _.union(pc1.selected(), pc1.brushed());
  if (data.length >= k ) {
    // clear canvas layers
    $("#div#plot01").html("");
    $("#div#plot02").html("");
    $("#div#plot").html(plot);
    // for complete reset, clusters will be recomputed
    visualize(data, n_objs = n_objs, k = k);
  } else {
    throw new Error("Not enough data selected to perform clustering.");
  }
});
```

Similarly, `remove` is handled as follows:

```
d3.select('#remove_selected').on('click', function() {
  // delete all brushed data (unless selected) and do complete refresh
  if (pc1.brushed().length) {
    // NOTE: selected/brushed data equivalent between plots at this stage
    var brushed_not_selected = _.difference(pc1.brushed(), pc1.selected());
    data = _.difference(data, brushed_not_selected);
    if (data.length >= k) {
      // clear canvas layers
      $("#div#plot01").html("");
      $("#div#plot02").html("");
      $("#div#plot").html(plot);
      // for complete reset, clusters will be recomputed
      visualize(data, n_objs = n_objs, k = k);
    }
    else {
      throw new Error("Not enough data remaining to perform clustering.");
    }
  }
  else {
    throw new Error("No data chosen to remove.");
  }
});
```

```
}  
});
```

Proposed library api

Because this pair of features requires the user to have made selections, it is inherently interactive. It is then intuitive that even in a library implementation, these features would be set up as a GUI in which the user checks boxes to either keep or remove brushed and/or marked data. Still, the library function to include this GUI should be as straightforward as possible. We propose the following:

```
parasol.reduce()
```

This function will append a button GUI to the interface built by the user.

Export selection as csv

Once the user has identified a set of relevant solutions, they will likely require the access to the solution data for further analysis. Currently, the user can download the data as a csv file, and in future development other formats may be possible.

Note: There are three possible fields of data the user may seek to extract: brushed, marked, or all remaining data. The current default is to export all data since this option is the most intuitive for a one-click button. In a library implementation, it will be straightforward to provide a choice.

Current implementation details

The export feature is currently implemented as a button. We must create that at the beginning of the script, inside the widgets div.

```
// create button  
<button href='#' id='export_selected'>Export</button>
```

Once clicked, we begin by noting the data field to export and checking that the chosen field is not empty. We strip the irrelevant row id information and use the *d3* library to format the data as a csv. We create a blob (JavaScript data type) from this csv data and use the `saveAs` function in the *FileSaver* library to build and initiate the download.

```

d3.select('#export_selected').on('click', function() {

    // export all remaining data to new csv and download
    var data_exp = data;

    if (data_exp == null || !data_exp.length) {
        throw new Error("No data selected.");
        return;
    }

    //remove id column
    data_exp.forEach(function(d) { delete d.id; });

    // format data as csv
    var columns = d3.keys(data_exp[0]);
    var csv = d3.csvFormat(data_exp, columns);

    // create url for download
    var file = new Blob([csv], {type: 'text/csv'});
    saveAs(file, "pareto_solutions.csv");

});

```

Proposed library api

The export feature should be a standalone function. The user will specify the following:

- group: {"brushed", "marked", "all"} the data field to be exported
- filename: file name with file type extension

The default implementation is provided below.

```

parasol.export(data, group="all", filename="pareto_solutions.csv")

```

Explore selection

Coming soon.

Proposed library api

Discussion

Coming soon.