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

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, preform 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) });

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

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

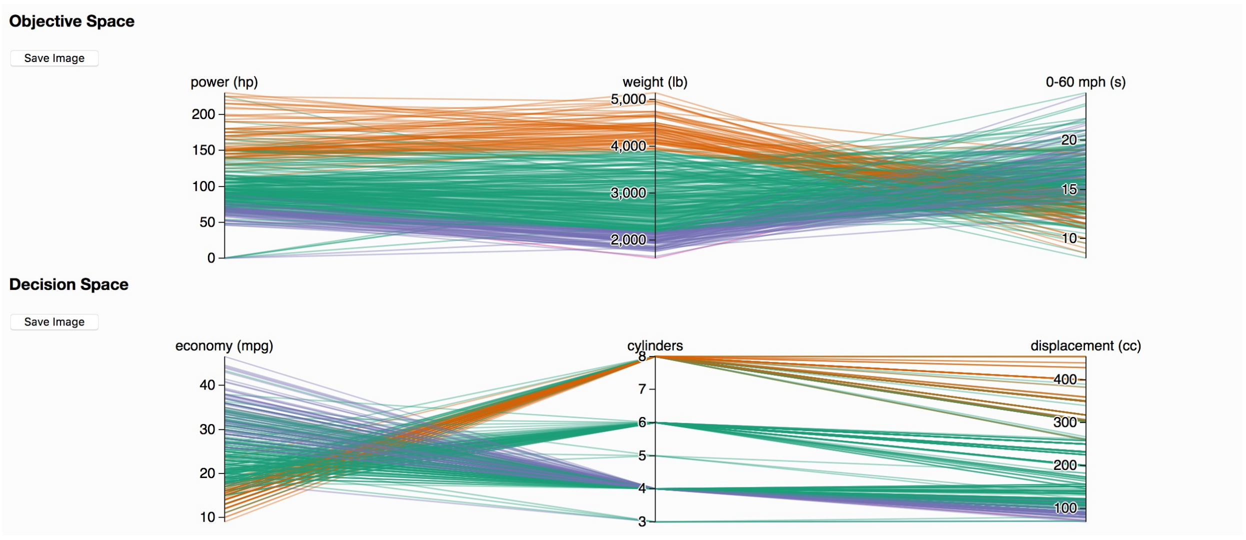
```
// choose default catagorical color scheme
var color_scheme = d3.scale.ordinal()
  .range(colorbrewer.Dark2[k])

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,

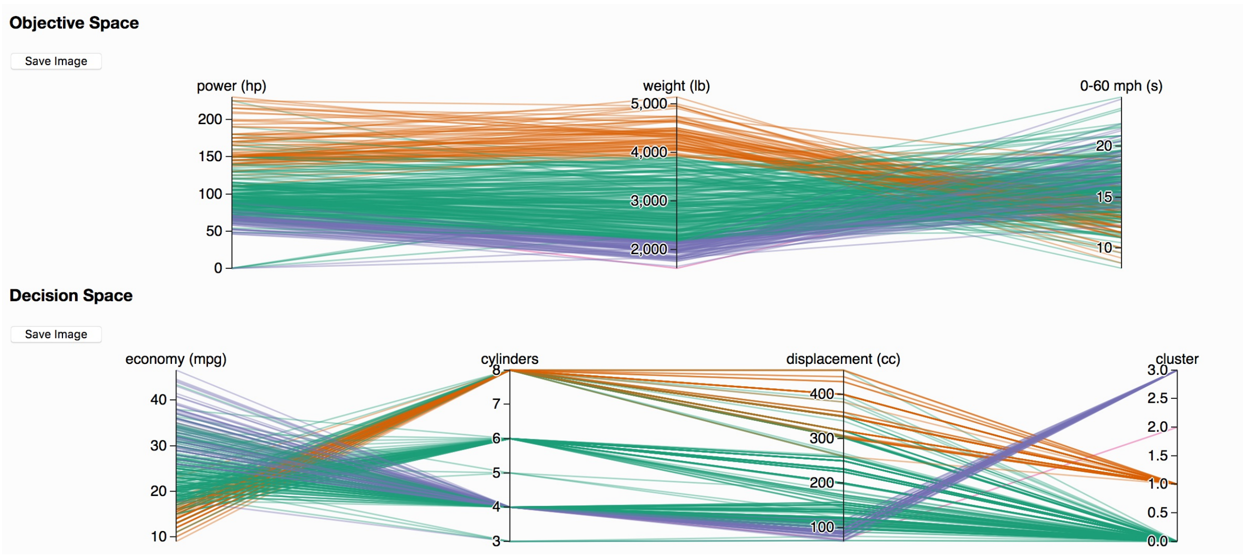
```
// objective space
var pc1 = d3.parcoords()("#plot01")
  .data(data)
  .hideAxis(decision_vars)
  .color(palette)
```

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 repetetive 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

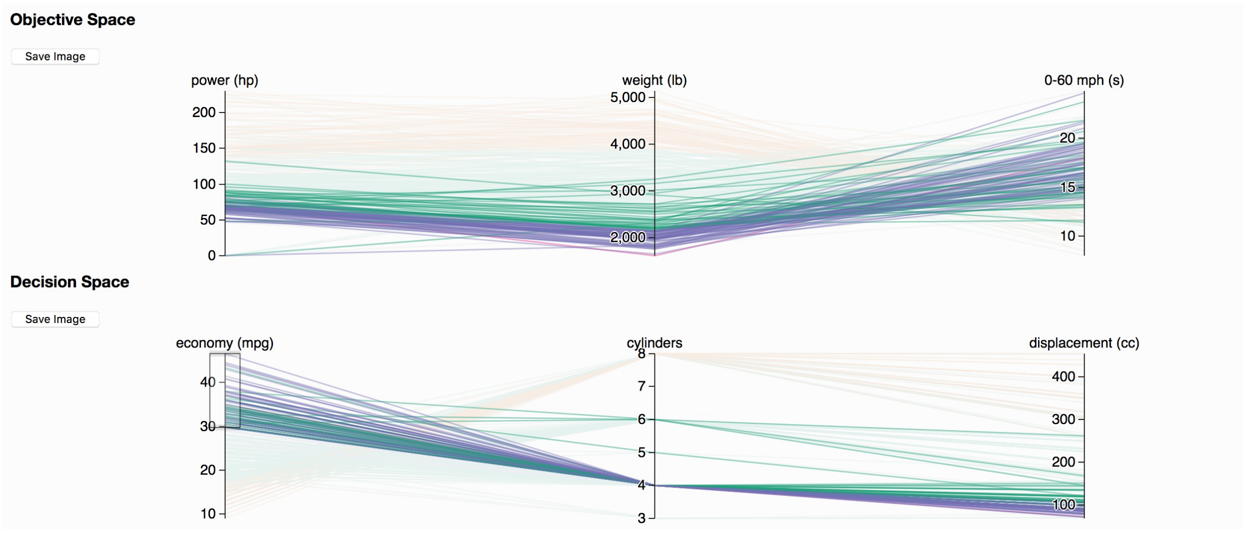
Defaults are listed below.

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

Linked brushing

Brushing is the primary tool for exploring solution subsets and refining the set of relevant data. The user simply drags the cursor over an axis range they are interested in, and all data outside the brush extents fade into the background &mdash though they remain slightly visible to maintain a frame of reference. This feature has been implemented in many visualization tools, and especially well in the *parallel-coordinates* library. The novel contribution made by *parasol* is that brushes can be linked between a pair of parallel coordinate plots. That is, if the data columns are split between two plots, as is often done in multiobjective optimization when representing decision and objective spaces, applying a brush to one plot will also update the other plot with the corresponding data. See the image below for reference. If a grid is present, it will also be updated to display only data within all brush extents.

Here a brush is applied to the economy axis in the decision space and the objective space is updated accordingly.



Current implementation details

Setting up the linked plots is currently achieved in a library-app hybrid method. In the call to the main `vizualize` function, the app developer specifies the last n columns that represent objective variables with *n_objs*:

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

Within the `visualize` function, we set up the objective and decision spaces as shown below. Note that "hidden" variables refer to functionality described in the Hide and show axes section.

```
// capture variable names
var variables = d3.keys(data[0]).slice(0,-2);

var id_col = "id"
var k_col = "cluster";
// NOTE: place clusters in decision space, but hide by default
pc2_hidden.push(k_col)

var decision_vars = _.union(variables.slice(0, variables.length - n_objs));
var objective_vars = _.difference(variables, decision_vars);
decision_vars = _.union(decision_vars, [k_col], [id_col]);
objective_vars = _.union(objective_vars, [id_col]);

// include any already hidden vars (for recursive aspect of keep/remove)
objective_vars = _.union(objective_vars, pc2_hidden);
decision_vars = _.union(decision_vars, pc1_hidden);
```

We then set up functionally-global variables to keep track of data between brushes:

```
// keep track of brushed data
var brushed_pc1 = [];
var brushed_pc2 = [];
var isBrushed_pc1 = false;
var isBrushed_pc2 = false;
```

The linked brushing functionality is then completely implemented in the process of building the plotting variables. With the exception of plot number, the code is the same for both variables, though we include both for clarity.

```
// objective space
var pc1 = d3.parcoords()("#plot01")
  .data(data)
  .hideAxis(decision_vars)
  .color(palette)
  .height(200)
  .alpha(0.4)
  .mode("queue")
  .render()
  .shadows()
  .reorderable()
  .brushMode("1D-axes")
  .on("brushend", function(brushed) {

    if (isBrushed_pc2) {
      // pc2 has brushes, keep only the intersection of arrays
      brushed_pc2 = _.intersection(brushed_pc2, brushed);
    } else {
      brushed_pc2 = brushed;
    }

    brushed_pc1 = brushed_pc2;
    isBrushed_pc1 = true;

    pc2.brushed(brushed_pc2);
    pc2.render();

    pc1.brushed(brushed_pc1);
    pc1.render();

    gridUpdate(brushed_pc2);
  });

// decision space
var pc2 = d3.parcoords()("#plot02")
  .data(data)
  .hideAxis(objective_vars)
  .color(palette)
  .height(200)
  .alpha(0.4)
  .mode("queue")
  .render()
  .shadows()
  .reorderable()
  .brushMode("1D-axes")
  .on("brushend", function(brushed) {

    if (isBrushed_pc1) {
      // pc1 has brushes, keep only the intersection of arrays
      brushed_pc1 = _.intersection(brushed_pc1, brushed);
    } else {
      brushed_pc1 = brushed;
    }

    brushed_pc2 = brushed_pc1;
    isBrushed_pc2 = true;

    pc2.brushed(brushed_pc1);
    pc2.render();

    pc1.brushed(brushed_pc2);
    pc1.render();

    gridUpdate(brushed_pc1);
  });
```

Above we define a new function that is called after user has defined brush extents by dragging their cursor along an axis. The argument of this function is then all data that lies within these extents alone. This data is matched with the functionally-global variables and each plotting variable is update with the new data accordingly. Because the funtion is essentially the same for both plotting variables, it lends itself to a straightforward library implementation.

Currently, only one brush is allowed per axis. In the future, we hope to allow for the use of multiple brush extents using *d3.multibrush*, though this adds a lot more complexity to a linked plot implementation.

Proposed library api

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

- num_objectives: last n columns that represent objective variables
- brushMode: {"1D-axes", "1D-axes-multi"} one brush per axis or multiple

Defaults are listed below.

```
visualize(data)
  .linked(num_objectives, brushMode="1D-axes")
```

Reset brushes

While brushes can be cleared on an individual basis by just clicking the respective axis outside of the extents, we find that it is also convenient to have a method for clearing them all simultaneously. This very straightforward since there is already a function `brushReset()` in the *parallel-coordinates* library. In addition to calling this function on both plot variables, we need only clear the functionally-global variables and restore the grid. This is implemented with a button click as follows:

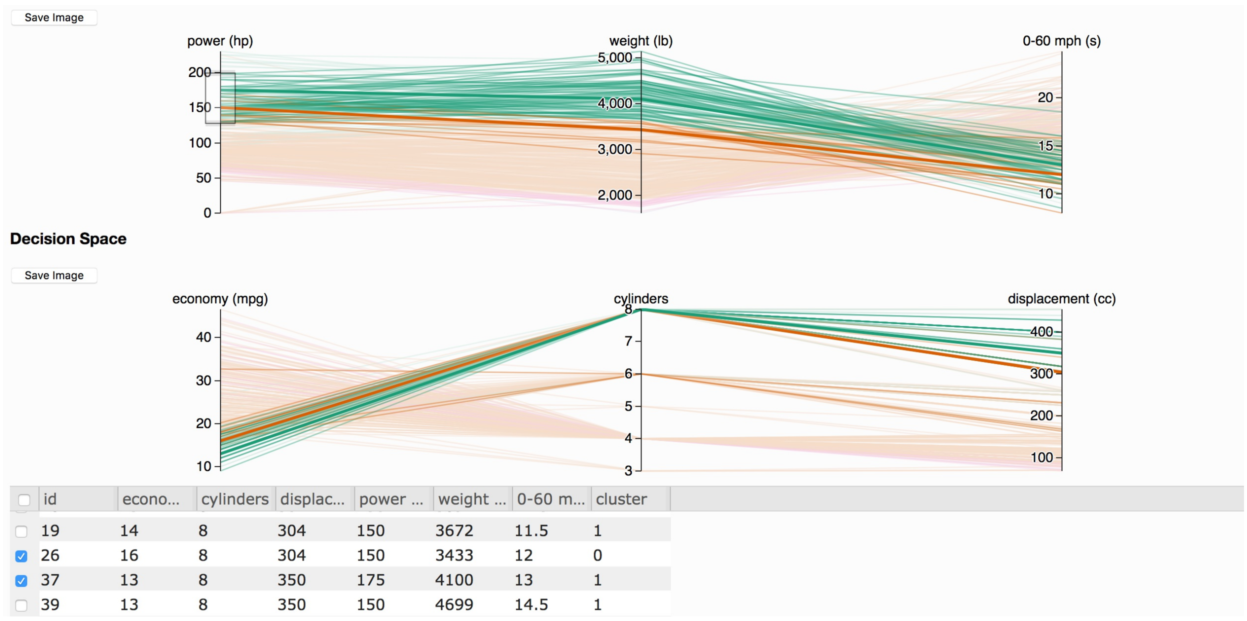
```
// create button
<button id="brush_reset">Reset Brushes</button>

// reset brushes
d3.select('#brush_reset').on('click', function() {
  // reset global vars
  brushed_pc1 = [];
  brushed_pc2 = [];
  isBrushed_pc1 = false;
  isBrushed_pc2 = false;

  pc1.brushReset();
  pc2.brushReset();

  gridUpdate(data);
});
```

In a library implementation, we would carry the over the intuitive terminology from the *parallel-coordinates* library:

```
parasol.brushReset()
```

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.



☐	id	econo...	cylinders	displac...	power ...	weight ...	0-60 m...	cluster
☐	19	14	8	304	150	3672	11.5	1
✓	26	16	8	304	150	3433	12	0
✓	37	13	8	350	175	4100	13	1
☐	39	13	8	350	150	4699	14.5	1

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 exits 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 preform 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 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 axes

Parallel coordinate plots can be a bit intimidating at first glance, especially when they feature many different variable axis. Upfront, the user can quickly simplify the visualization by hiding axes that are not of interest. Once the user has had a chance to narrow the set of relevant solutions, they can show any of these hidden axes to further assess the results.

Note: Currently, the hiding or showing of an axis completely resets all brushes and marked data. Progress has been made on maintaining these features, but there are still too many bugs to incorporate this full functionality at present.

Current implementation details

The hide and show axes functionality is currently implemented through buttons which act on the "input" variable defined by the web-app developer. Currently, this input variable is set to "cluster" as seen in the button functionality below. This is a proof of concept, but one can see how this hybrid implementation could easily be translated to a library function where the input variable is defined when the user checks a box in the GUI corresponding to the axis to act on.

The buttons are constructed in the usual manner:

```
<button id="show_axis">Show Clusters</button>
<button id="hide_axis">Hide Clusters</button>
```

We also establish a pair of global variables to keep track of hidden axes between recursive calls to the main `visualize` function, as is done in the Keep and Remove selection feature:

```
var pc1_hidden = [];
var pc2_hidden = [];
```

These global variables are referenced at the start of the `visualize` function when setting up the decision and objective spaces:

```
// include any already hidden vars (for recursive aspect of keep/remove)
objective_vars = _.union(objective_vars, pc2_hidden);
decision_vars = _.union(decision_vars, pc1_hidden);
```

When the hide (show) axis buttons is clicked, we check to see which plot the input variable belongs to and place (remove) it in the array of names for the partnering plot since these are hidden. If the variable is not found in either plot, we issue a warning to that effect. Otherwise, we reset the brushes and marked data by manually clicking the respective buttons for those features and re-render both plots to complete the update process.

```
// hide axis
d3.select('#hide_axis').on('click', function() {
  var found = false;
  var input = ["cluster"];

  if ( ! _.difference(input, objective_vars).length ) {
    found = true;
    // in objectives so add it to decisions which are hidden
    pc1_hidden.push(input[0]); // keep list of hidden
    decision_vars = _.union(decision_vars, input);
    pc1.hideAxis(decision_vars);
    pc1.render().updateAxes(500); // animationTime (ms)
  }
  else if ( ! _.difference(input, decision_vars).length ) {
    found = true;
    // in decisions so add it to objectives which are hidden
    pc2_hidden.push(input[0]); // keep list of hidden
    objective_vars = _.union(objective_vars, input);
    pc2.hideAxis(objective_vars);
    pc2.render().updateAxes(500); // animationTime (ms)
  }
  if (found == false) {
    // not in dataset
    throw new Error("Variable not found.");
  }
  else {
    // for now: RESET brushes and selections
    // deselect all elements in grid (fires event)
    document.getElementById('brush_reset').click();
    document.getElementById('clear_selected').click();

    pc1.render();
    pc2.render();
  }
});

// show axis
d3.select('#show_axis').on('click', function() {
  var found = false;
  var input = ["cluster"];

  if ( ! _.difference(input, pc1_hidden).length ) {
    found = true;
    // in hidden objectives so remove from decisions which are hidden
    pc1_hidden = _.difference(pc1_hidden, input); // remove from list
    decision_vars = _.difference(decision_vars, input);
    pc1.hideAxis(decision_vars);
    pc1.render().updateAxes(500); // animationTime (ms)
  }
  else if ( ! _.difference(input, pc2_hidden).length ) {
    found = true;
    // in hidden decisions so remove from objectives which are hidden
    pc2_hidden = _.difference(pc2_hidden, input); // remove from list
    objective_vars = _.difference(objective_vars, input);
    pc2.hideAxis(objective_vars);
    pc2.render().updateAxes(500); // animationTime (ms)
  }
  if (found == false) {
    // not in dataset
    throw new Error("Variable not hidden.");
  }
  else {
    // for now: RESET brushes and selections
    document.getElementById('brush_reset').click();
    document.getElementById('clear_selected').click();

    pc1.render();
    pc2.render();
  }
});
```

Proposed library api

This feature is most intuitively implemented as a GUI with checkboxes next to each of the axes variable names. Once clicked, the corresponding axis will be hidden. A second click clears the box and shows the axis. Next to the hide and show checkbox will be another checkbox to flip the axis so that it aligns with the *direction of preference* arrow built into the linked plot display. This will be easily incorporated into the GUI using the `flipAxes` function in the *parallel-coordinates* library.

The library function to include this GUI should be as straightforward as possible. We propose the following:

```
parasol.dimensions()
```

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

Set axes limits

To overcome the bias of drastic spread for relatively small axes ranges, we find that it is important to allow the user to edit the axes limits as necessay. That is, when the range of a single axis is relatively small, solution lines will be spread across the full extent of that axis, emphasizing small differences in solutions. However, if the user deems that these differences are not as significant as their spread may imply, they should be able to increase the axis limits so that the solutions are better grouped together. Decreasing the limits &mdash bounded by the maximum and minimum values of the data &mdash would have the inverse effect.

Proposed library api

Once the functionality has been developed, this feature can easily be included with the GUI for Hide and show axes. Alongside each variable's checkbox, numeric input boxes will be provided for minimum and maximum axis limits to be specified. In the upper-right corner of the GUI, a button will be provided to restore default limits.

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 begining 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 slected 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 possilbe.

Note: There are three possilbe 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 in 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:

- selection: {"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, selection="all", filename="pareto_solutions.csv")
```

Explore selection

Once the user has arrived at a set of narrowed solutions, they may seek to gain further insight into specific relationships within that data. To that end, we seek to extend the interactive functionality of the library with quick access to other basic visualizations. For example, the user may wish to use a scatterplot to better understand the relationship between two specific variables, a histogram to analyze solution density for a single variable, or even a heatmap for multiple variables.

Proposed library api

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

- vars: variables to be visualized
- visualization: {"scatter", "hist", "heatmap", etc.} type of visualization
- selection: {"brushed", "marked", "both"} the data field to be explored

The default implementation is provided below.

```
parasol.explore(data, vars, visualization, selection="both")
```

Last Updated: May 16, 2018