CGView

CGView is a circular genome graphing utility. You can give it features encoded in XML files, and it generates a display of those features along a circular genome fragment.

Basic Structure

CGView accepts three different file formats, but we'll focus on XML here. The basic structure is this:

- CG View settings
  - Legend (optional)
    - LegendItem (text that you want in a legend)
  - FeatureSlot 1 (this is a circular track)
    - Feature in FeatureSlot 1
  - FeatureSlot 2
    - Feature in FeatureSlot 2
  - FeatureSlot N
    - Feature in FeatureSlot N
  ...

In XML, this has the basic form:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<cgview backboneRadius="160" sequenceLength="10000" height="600" width="600">
    <legend position="upper-right">
        <legendItem text="Promoter" drawSwatch="true" swatchColor="green" />
        <legendItem text="Terminator" drawSwatch="true" swatchColor="red" />
    </legend>
    <featureSlot strand="direct">
        <feature color="green" decoration="clockwise-arrow" label="a promoter">
            <featureRange start="500" stop="900" />
        </feature>
    </featureSlot>
    <featureSlot strand="reverse">
        <feature color="red" decoration="arc" label="a terminator">
            <featureRange start="8500" stop="8800" />
        </feature>
    </featureSlot>
</cgview>
```

Basic Attributes

Here are some quick summaries of some useful attributes. Complete information is available from the CGView documentation.

featureSlot
Define a circular track in the graph. 

strand **direct** or **reverse** affects the direction of arrows, and whether features are drawn on the inside or outside of the backbone circle.

**feature**

Define a feature in the graph.

**color** red, blue, green, purple are easy choices, but you can also use RGB values with something like rgb(0,225, 0).

**decoration** arc, clockwise-arrow, counterclockwise-arrow

**proportionOfThickness** real number between 0 and 1.0 changes the thickness of the feature.

**radiusAdjustment** real number between 0 and 1.0 adjusts the radius at which the feature is drawn. 0 draws the feature as close to the backbone circle as possible, 0.5 draws the feature in the middle of the featureSlot.

**featureRange**

Where feature begins and ends (and there can be more than one **featureRange** per feature).

start, stop **integers between 0 and the sequence length**. Each **featureRange** needs start and stop coordinates.

**color**, **decoration**, **proportionOfThickness**, **radiusAdjustment** have the same meaning as for a **feature**, and setting these values for a **featureRange** overrides the values in the encompassing **feature**.

### Helper Scripts

Here's a quick Bash script that takes files in XML format representing featureSlot elements, and combines them into an XML file for CGView.

```bash
#!/bin/bash
cp header.xml cg3.xml
cp header_nogenelabels.xml cg3.xml
cat "$@" >> cg3.xml
echo "</cgview>" >> cg3.xml
cgview -i cg3.xml -o map.jpg -f jpg
```

This code depends on the following header file (which should be modified to fit the current use):

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<cgview backboneRadius="600" backboneColor="rgb(102,102,102)"
backboneThickness="8" featureSlotSpacing="small" labelLineLength="100"
lablePlacementQuality="good" labelLineThickness="2" rulerPadding="10"
tickThickness="6" arrowheadLength="6" rulerFont="SansSerif, plain, 20"
rule\r
rFontColor="rgb(0,0,0)" labelFont="SansSerif, plain, 15" isLinear="true"
minimumFeatureLength="0.2" sequenceLength="1255523" height="3000"
width="3000" glabelLabel="true" moveInnerLabelsToOuter="false"
featureThickness="30" tickLength="15" useInnerLabels="true"
shortTickColor="rgb(0,51,0)"
longTickColor="rgb(0,51,0)" zeroTickColor="rgb(0,51,0)"
showBorder="true" borderColor="black" backgroundColor="white" tickDensity="0.166666666666667">
```
Using makeMap.sh

makeMap.sh FeatureA.xml FeatureB.xml

generates the graph as map.jpg, with Feature A and Feature B as separate tracks, where (for example) FeatureA.xml and FeatureB.xml are files of the form

```
<featureSlot showShading="false" strand="direct">
<feature color="green" decoration="arc">
<featureRange start="1" stop="1184" />
<featureRange start="1198" stop="1367" />
<featureRange start="1416" stop="1607" />
<featureRange start="1614" stop="1853" />
</feature>
</featureSlot>
```

Note that the input files here aren't technically XML files themselves, since they don't begin with an XML declaration, but this is a minor technical point.

Graphing a histogram

Graphing a histogram in CGView using each position as a datapoint will probably take lots of computational time, with marginal benefits to resolution. Averaging over some region of positions is probably a more efficient way of getting a histogram. The following script takes a table of position counts (as a text file), and outputs an XML fragment of the counts averaged over regions of length `window_size` which can be used as input to `makeMap.sh` above.

The position count table must have the form

```
<column 1> <column 2, probably position> <int> <int> ... <int>
<column 1> <column 2, probably position> <int> <int> ... <int>
...
```

where each `<int>` is a count.

Script to average a count table

```
#! /usr/bin/env python
import sys

try:
    import argparse
except:
    print 'Try typing "module load python" and then running this again.'
    sys.exit(1)
```
def main():
    parser = argparse.ArgumentParser(description='Averages read depth along chromosome positions from a number of samples.')
    parser.add_argument('count_table')
    args = parser.parse_args()

    window_table = []
    window_size = 250
    window_sum = 0
    line_count = 0

    with open(args.count_table, 'r') as count_table:
        for line in count_table:
            fields = line.split()
            count_fields = [int(x) for x in fields[2:]]
            window_sum += sum(count_fields)

            line_count += 1
            if (line_count % window_size) == 0:
                window_table.append([line_count-window_size+1, line_count, window_sum])
                window_sum = 0

            if (line_count % window_size) != 0:
                window_table.append([line_count - (line_count % window_size) + 1, line_count, window_sum])

            max_count = 0
            for window in window_table:
                if max_count < window[2]:
                    max_count = window[2]

            for window in window_table:
                window[2] = float(window[2])/max_count

            print '<featureSlot strand="direct">'
            print '  <feature color="purple" decoration="arc">'
            for window in window_table:
                print '    <featureRange start="{}" stop="{}" proportionOfThickness="{}" />'
            print '</feature>
            print '</featureSlot>'
if __name__ == '__main__':
    main()