Exercise Model Solution

Leighton Pritchard

March 16, 2014

1 Comparative Genomics and Visualisation Exercise Walkthrough

This directory contains the exercise - and some possible solutions - for the Comparative Genomics and Visualisation module.

This iPython notebook, when executed with `Cell -> Run All`, should complete one possible model answer to the exercise.

2 Question 1:

In the `data` subdirectory you will find the FASTA sequence file `draft_genome.fasta`, representing assembled contigs from a bacterial genome sequencing project. You will also find complete bacterial genome sequences representing the following bacterial genera:

- *Dickeya*: `NC_014500.fna`, `NC_013592.fna`, `NC_012880.fna`, `NC_012912.fna`
- *Klebsiella*: `NC_016612.fna`, `NC_011283.fna`, `NC_017540.fna`, `NC_013850.fna`
- *Pseudomonas*: `NC_022808.fna`, `NC_007492.fna`, `NC_004578.fna`, `NC_010322.fna`
- *Staphylococcus*: `NC_004661.fna`, `NC_007795.fna`, `NC_017337.fna`, `NC_013893`

Using bulk summary statistics of GC content and genome size, determine to which of the candidate bacterial genera the draft genome belongs.

Comment on the relationship between the draft genome's length, and the lengths of the genomes in the genus to which you think it belongs.

HINT: The draft genome may not only contain A, C, G and T.

HINT: There is some code in slide 49 of Part 1.

HINT: The iPython notebook `bacteria_size_gc.ipynb` may be useful.This question could be approached in several ways. The more insightful students might use BLAST to confirm (or obtain) their answer. One way would be to adapt the example code given in the PowerPoint slides, to give a table of GC content and length, and argue that the draft genome belongs to the genus with the closest representative(s).

There's a gotcha: the draft genome has `N`s in it.

In [1]:
```python
from Bio import SeqIO
import os

for f in os.listdir('data/'):
    s = SeqIO.read(os.path.join('data', f), 'fasta')
    a, c, g, t = (s.seq.count('A'), s.seq.count('C'),
                  s.seq.count('G'), s.seq.count('T'))
    gc = float(g + c)/sum([a, c, g, t])
    print s.id, gc, a + c + g + t
```

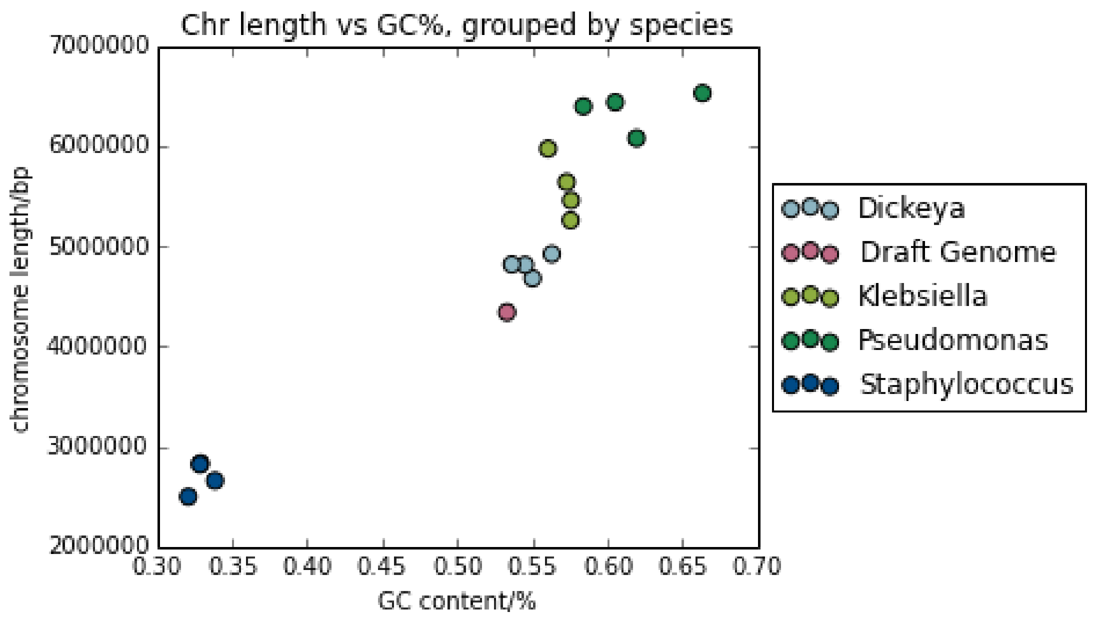
```
draft_genome 0.5334391098 4338079
gi|27466918|ref|NC_004461.1| 0.320951762488 2499279
gi|28867243|ref|NC_004578.1| 0.583988809975 6397126
gi|255961261|ref|NC_007492.2| 0.60520952006 6438405
gi|88193823|ref|NC_007795.1| 0.328682621147 2821360
gi|167031021|ref|NC_010322.1| 0.619422449547 6078430
gi|206575712|ref|NC_011283.1| 0.572861032833 5641239
gi|242237460|ref|NC_012880.1| 0.550150765581 4679450
gi|251787652|ref|NC_012912.1| 0.545172537431 4813854
gi|271498578|ref|NC_013592.1| 0.536390755924 4818394
gi|288932888|ref|NC_013850.1| 0.575780914371 5458505
gi|289549371|ref|NC_013893.1| 0.338681355389 2658366
gi|307128764|ref|NC_014500.1| 0.56296630194 4922776
gi|375256816|ref|NC_016612.1| 0.560485682549 5974108
gi|384546269|ref|NC_017337.1| 0.329162662517 2832478
gi|386032579|ref|NC_017540.1| 0.575516145445 5259564
gi|558672313|ref|NC_022808.1| 0.663376565373 6528877
```

Alternatively, reusing the code from `bacteria_size_gc.ipynb`, or the notebook itself, with modified input file locations will work.

In [2]:
```
# IMPORTS
from Bio import SeqIO
from Bio.Graphics.ColorSpiral import get_color_dict
from IPython.display import Image

import pandas as pd

import os

# DATA
# Dictionary associating .fna chromosome sequence files to
# bacterial genus
bact_files = {"Dickeya": ("NC_012880.fna", "NC_012912.fna",
                          "NC_013592.fna", "NC_014500.fna"),
              "Klebsiella": ("NC_011283.fna", "NC_013850.fna",
                             "NC_016612.fna", "NC_017540.fna"),
              "Draft Genome": ("draft_genome.fasta",),
              "Pseudomonas": ("NC_004578.fna", "NC_007492.fna",
                              "NC_010322.fna", "NC_022808.fna"),
              "Staphylococcus": ("NC_004461.fna", "NC_013893.fna",
                                 "NC_017337.fna", "NC_007795.fna")}

# FUNCTIONS
# Function calculating chromosome length and GC content
def calc_size_gc(*names):
    """ When passed names corresponding to the bacteria
        listed in bact_files, returns a Pandas dataframe
        representing sequence length and GC content for
        each chromosome.
    """
    # Use a Pandas DataFrame to hold data. Dataframes are
    # useful objects/concepts, and support a number of
    # operations that we will exploit later.
    df = pd.DataFrame(columns=['species', 'length', 'GC', 'color'])
    # Get one colour for each species, from Biopython's
    # ColorSpiral module
    colors = get_color_dict(names, a=6, b=0.2)
```

```python
    # Loop over the passed species names, and collect data
    for name in names:
        try:
            for filename in bact_files[name]:
                ch = SeqIO.read(os.path.join('data', filename), 'fasta')
                ch_size = len(ch.seq)
                ch_gc = float(ch.seq.count('C') +
                              ch.seq.count('G')) / ch_size
                df = df.append(pd.DataFrame([dict(species=name,
                                                  length=ch_size,
                                                  GC=ch_gc,
                                                  color=colors[name]), ]),
                               ignore_index=True)
        except KeyError:
            print "Did not recognise species: %s" % name
            continue
    return df


# Plot chromosome size and GC data
def plot_data(dataframe, filename=None):
    """ When passed a dataframe corresponding to the output
        of calc_size_gc, renders a scatterplot of
        chromosome length against GC content.
    """
    # One advantage of using a Pandas dataframe is that we can
    # operate on the data by the content of the data. Here we're
    # treating the dataframe as a series of subsets on the basis
    # of named species. This allows us to label our scatterplot
    # by species, too.
    fig = figure(figsize=(8,4))
    ax = fig.add_subplot(111)
    ax.set_position([0.15, 0.15, 0.45, 0.75])
    for k, sub in dataframe.groupby("species"):
        ax.scatter(x=sub.GC, y=sub.length, c=list(sub.color),
                   label=k, s=50)
    ax.set_xlabel("GC content/%")
    ax.set_ylabel("chromosome length/bp")
    ax.set_title("Chr length vs GC%, grouped by species")
    leg = ax.legend(bbox_to_anchor=(1.0, 0.5), loc='center left')
    if filename is not None:
        fig.savefig(filename)
```

In [3]:
```python
plot_data(calc_size_gc('Dickeya', 'Draft Genome', 'Klebsiella',
                       'Pseudomonas', 'Staphylococcus'),
          filename="question1.png")
```



A third - more fiddly - method would be to modify the Shiny app to point to these genome files rather than those in the example. I don't think I'd have done this, but someone might choose to.

Answer

However they choose to do it, it should be clear that the draft genome is closer to `Dickeya` than to any of the other bacterial genera, and they should tentatively consider (given the limited options) that the draft genome is from a member of the *Dickeya* genus.

In the comment, they should say something sensible about the genome assembly possibly missing parts of the genome that are harder to sequence (e.g. collapsing repeats), so it should be expected to be shorter than the complete genomes.

3 Question 2

Using Average Nucleotide Identity (ANI) measures, determine whether the draft genome belongs to the same species as any of the other members of the genus.

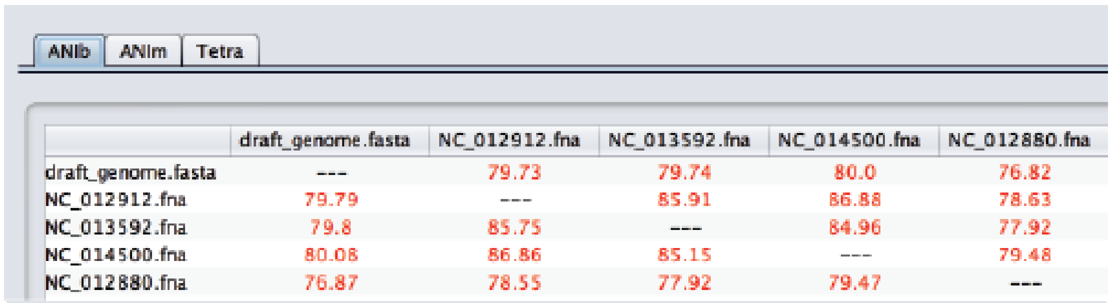
HINT: The `average_nucleotide_identity` activity may be useful.

CAUTION: Due to library incompatibilities in the VM, the `calculate_ani.py` script will not produce graphical output, so using the `-g` argument will have no effect.

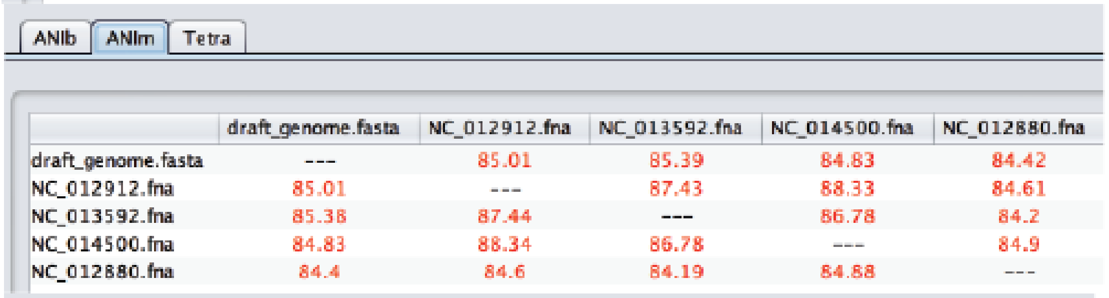
HINT: The `calculate_ani.py` script expects all input files (and no others) to lie in the same directory.Two ways to approach this are demonstrated in the `average_nucleotide_identity` activity. The first (and possibly more reliable, but **substantially** slower, and prone to cryptic permissions-related errors) is to use **JSpecies**. Applied correctly to the draft genome and *Dickeya* complete genomes, this should give the tabular output below:

In [4]:
```
Image("images/jspecies_output.png")
```

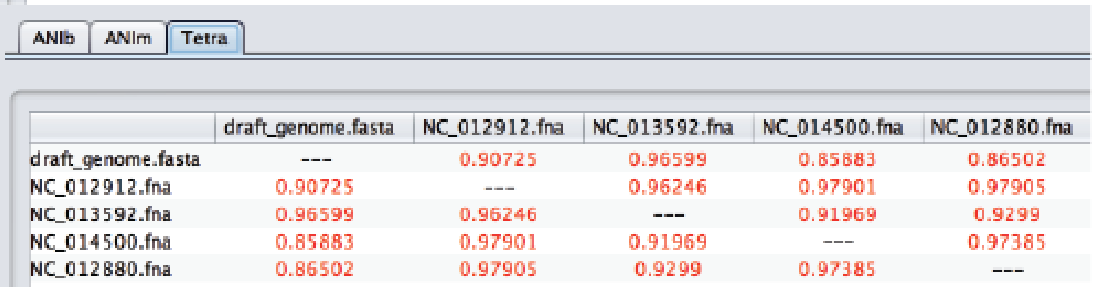
Out [4]:



	draft_genome.fasta	NC_012912.fna	NC_013592.fna	NC_014500.fna	NC_012880.fna
draft_genome.fasta	---	79.73	79.74	80.0	76.82
NC_012912.fna	79.79	---	85.91	86.88	78.63
NC_013592.fna	79.8	85.75	---	84.96	77.92
NC_014500.fna	80.08	86.86	85.15	---	79.48
NC_012880.fna	76.87	78.55	77.92	79.47	---



	draft_genome.fasta	NC_012912.fna	NC_013592.fna	NC_014500.fna	NC_012880.fna
draft_genome.fasta	---	85.01	85.39	84.83	84.42
NC_012912.fna	85.01	---	87.43	88.33	84.61
NC_013592.fna	85.38	87.44	---	86.78	84.2
NC_014500.fna	84.83	88.34	86.78	---	84.9
NC_012880.fna	84.4	84.6	84.19	84.88	---



	draft_genome.fasta	NC_012912.fna	NC_013592.fna	NC_014500.fna	NC_012880.fna
draft_genome.fasta	---	0.90725	0.96599	0.85883	0.86502
NC_012912.fna	0.90725	---	0.96246	0.97901	0.97905
NC_013592.fna	0.96599	0.96246	---	0.91969	0.9299
NC_014500.fna	0.85883	0.97901	0.91969	---	0.97385
NC_012880.fna	0.86502	0.97905	0.9299	0.97385	---

To use the `calculate_ani.py` script, the student will have to move, copy or symbolically link the draft and complete *Dickeya* genomes to a directory on their own (or delete the irrelevant genomes in `data`) in order to satisfy input requirements for the script without doing a number of unnecessary calculations, e.g.:

```
$ tree Dickeya/
Dickeya/
??? NC_012880.fna -> ../data/NC_012880.fna
??? NC_012912.fna -> ../data/NC_012912.fna
??? NC_013592.fna -> ../data/NC_013592.fna
??? NC_014500.fna -> ../data/NC_014500.fna
??? draft_genome.fasta -> ../data/draft_genome.fasta
```

The script will give the relevant tabular output:

```
$ more ANIb/perc_ids.tab
# calculate_ani.py Sat Mar 15 17:58:00 2014
# ANIb
        NC_012880       NC_012912       NC_013592       NC_014500       draft_genome
NC_012880       NA      0.8087  0.8022  0.8157  0.7934
NC_012912       0.8087  NA      0.8703  0.8797  0.8168
NC_013592       0.8022  0.8703  NA      0.8628  0.8168
NC_014500       0.8157  0.8797  0.8628  NA      0.8193
draft_genome    0.7934  0.8168  0.8168  0.8193  NA
```

```
$ more ANIm/perc_ids.tab
# calculate_ani.py Sat Mar 15 18:02:39 2014
# ANIm
```

```
        NC_012880       NC_012912       NC_013592       NC_014500       draft_genome
NC_012880       NA      0.8459  0.8418  0.8487  0.8441
NC_012912       0.8459  NA      0.8742  0.8833  0.8500
NC_013592       0.8418  0.8742  NA      0.8678  0.8539
NC_014500       0.8487  0.8833  0.8678  NA      0.8482
draft_genome    0.8441  0.8500  0.8539  0.8482  NA
```

```
$ more TETRA/tetra_corr.tab
# calculate_ani.py Sat Mar 15 18:06:59 2014
# TETRA
        NC_012880       NC_012912       NC_013592       NC_014500       draft_genome
NC_012880       NA      0.9790  0.9299  0.9739  0.8650
NC_012912       0.9790  NA      0.9625  0.9790  0.9072
NC_013592       0.9299  0.9625  NA      0.9197  0.9660
NC_014500       0.9739  0.9790  0.9197  NA      0.8588
draft_genome    0.8650  0.9072  0.9660  0.8588  NA
```

Answer

By every measure, ANIb, ANIm, or TETRA, the conclusion should be that the draft genome sequence is not of the same species as any of the other *Dickeya* genomes, as the similarity is less than 95% for ANI.

The scores for TETRA do not indicate species similarity - it was noted that TETRA can produce false positives and reflects bulk composition, rather than heredity. Also, more motivated students may follow up the references, where caution is advised with interpretation of TETRA scores, which are noted to sometimes be high for sequences that do not belong to the same species.

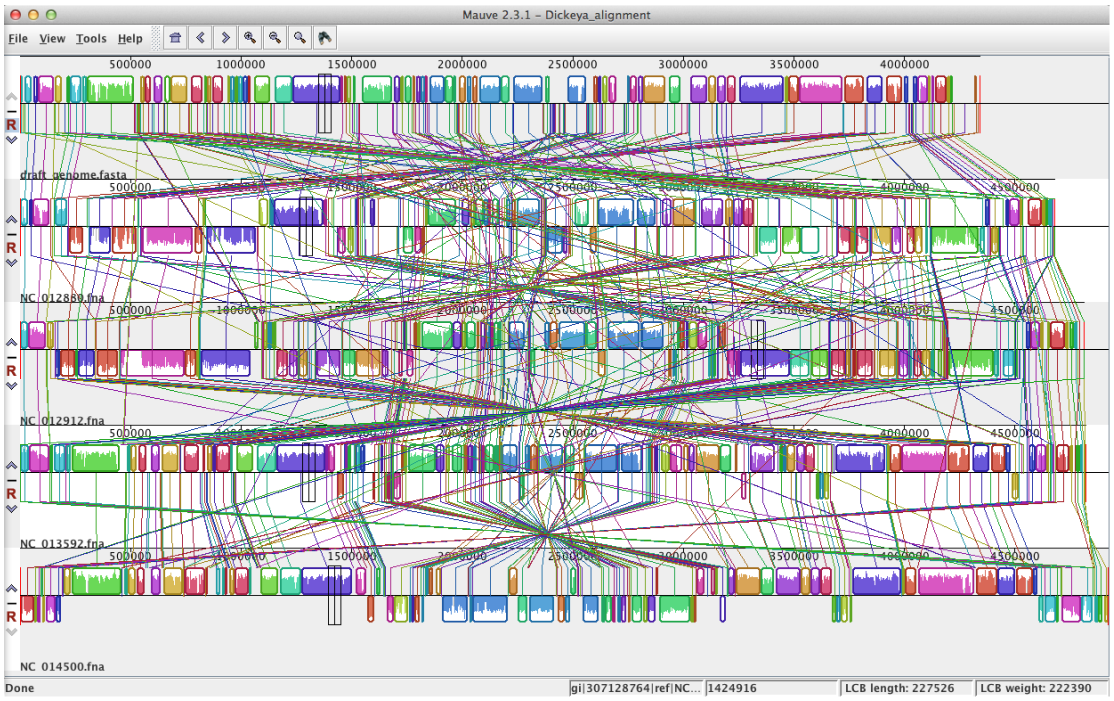
4 Question 3

Use suitable whole genome alignments to investigate whether the draft genome has undergone rearrangement with respect to any of the complete genomes in the genus. Produce graphical output to demonstrate the extent of rearrangement (or lack thereof), and comment on your figures.

HINT: The `nucmer_to_crunch.py` script in `Part_1/scripts` will be useful in generating `.crunch` output from **NUCmer** comparisons.As ever, there are several ways to approach this. On the face of it, the simplest method might seem to be to use **Mauve**'s ProgressiveMauve alignment, which will conduct a multiple genome alignment, and indicate rearrangements. This would take a single input step, and not require the student to run multiple pairwise comparisons. The **Mauve** output will look something like the image below:

```
In [5]: Image("images/mauve_alignment.png")
```

```
Out [5]:
```



The disadvantage of this approach is that the output is messy and hard to interpret, and comparison of rearrangement needs to be considered transitively across the five genomes. This is difficult to do.The minimum number of **BLAST** and **NUCmer** (which should be preferred to **MUMmer**) alignments to generate suitable **ACT** output is four - the draft genome against each complete genome. Taking this approach will likely result in four distinct **ACT** or GenomeDiagram figures, rather than a more complex, and harder to interpret, interleaved alignment figure. This is, as it happens, the quicker way to produce the alignments - there appears to be a considerable amount of overhead in **JSpecies**.

For the **NUCmer** alignment, there is no need to go beyond the initial alignment, and production of `.coords` files, followed by conversion of these to `.crunch` files with the provided script.

Scripts to generate the alignments are given in `run_megablast.sh` and `run_nucmer.sh`.

```
In [6]: !cat run_megablast.sh
```

```
#!/usr/bin/env bash
#
# Script to run megaBLAST comparisons for Comparative Genomics and
Visualisation exercise.

for i in Dickeya/*.fna
do
    cmd="blastn -query Dickeya/draft_genome.fasta -subject ${i}
-outfmt 6 -out megablast_output/draft_v_${i#Dickeya/}.tab -task dc-
megablast"
    echo $cmd
    eval $cmd
done
```

```
In [7]: !cat run_nucmer.sh
```

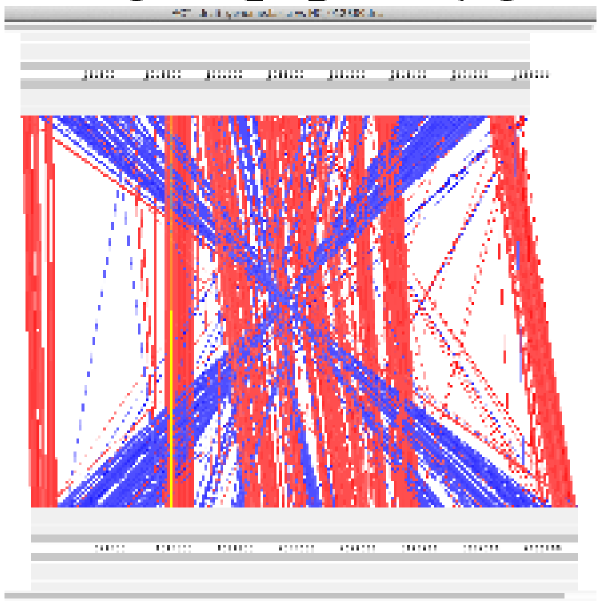
```
#!/usr/bin/env bash
#
# Script to run NUCmer comparisons for Comparative Genomics and
Visualisation exercise.

for i in Dickeya/*.fna
do
    prefix="nucmer_output/draft_v_${i#Dickeya/}"
    nucmer --maxgap=500 --mincluster=100 --prefix=${prefix} ${i}
Dickeya/draft_genome.fasta;
    show-coords -r ${prefix}.delta > ${prefix}.coords
    python ../Part_1/scripts/nucmer_to_crunch.py -i ${prefix}.coords
-o ${prefix}.crunch
done
```

Answer

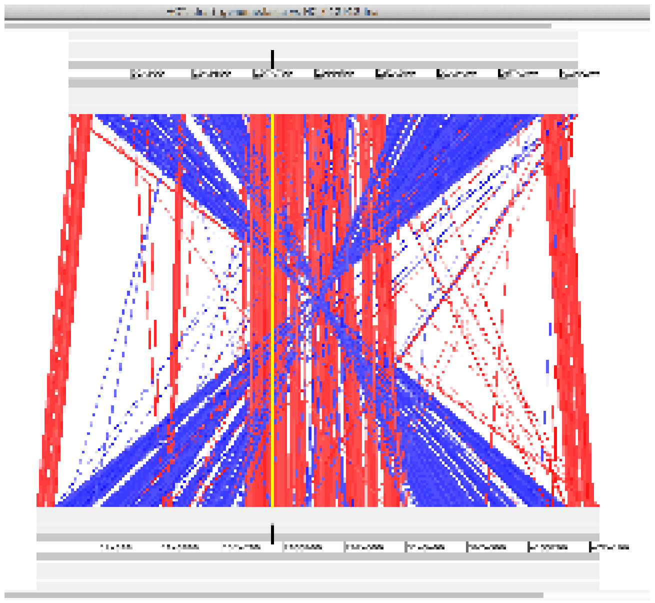
Example graphical output is shown below for **megaBLAST** and **NUCmer**:

In [8]:
```
# Four megaBLAST alignments
for ima, f in [(imread(os.path.join("images", f)), f) for f in
               os.listdir("images") if f.startswith("megablast")]:
    figure()
    imshow(ima)
    axis('off')
    title(f)
```

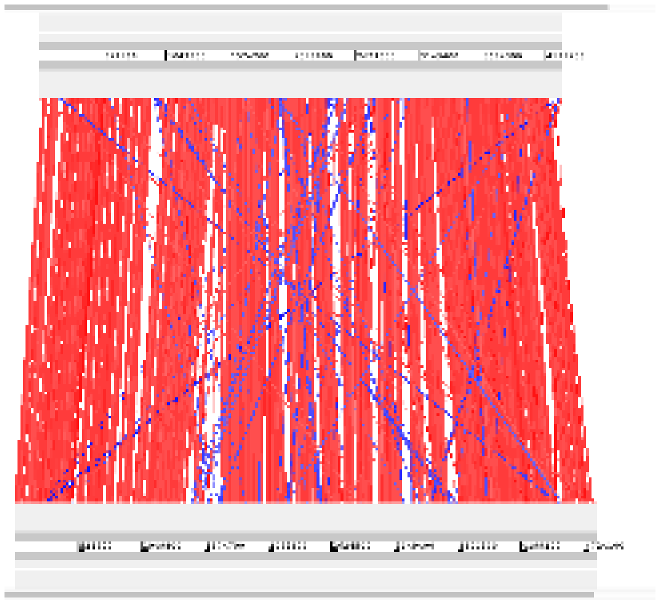
megablast_NC_012880.png



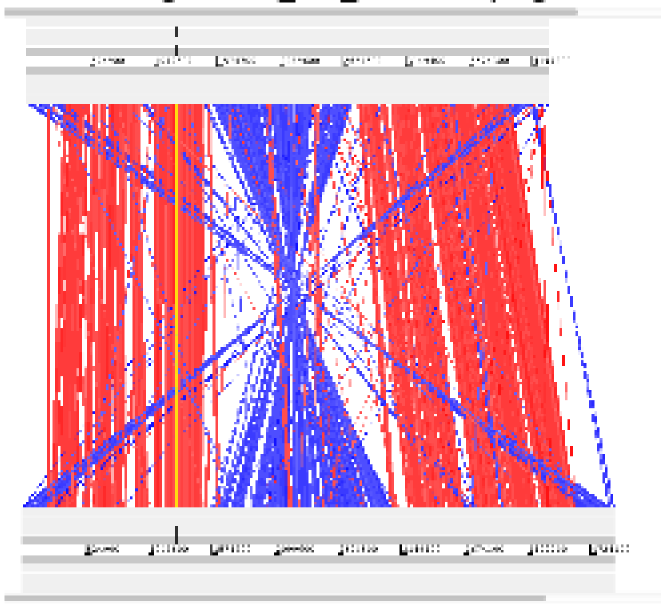
megablast_NC_012912.png



megablast_NC_013592.png



megablast_NC_014500.png



There are quite a few comments that can be made about the rearrangements with respect to each complete genome:

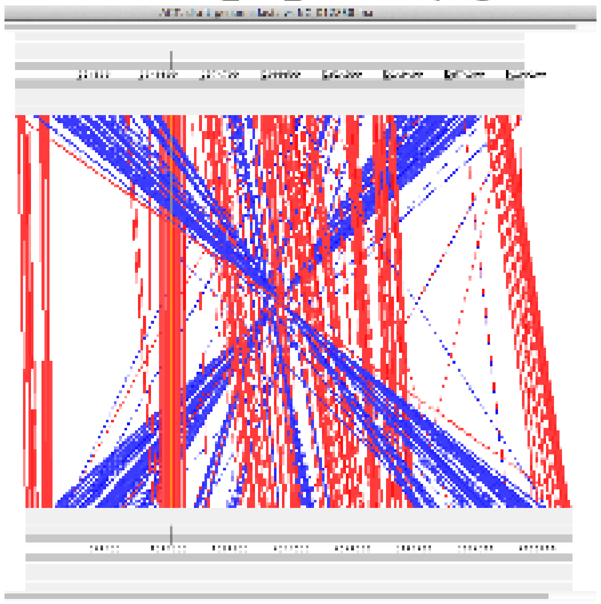
- NC_013592: The two genomes are largely syntenous. Sequence order is broadly conserved along the full length of each genome.

- NC_014500: There appear to have been at least two inversions. There is a large region in the 'middle' of the alignment, suggestive of at least one inversion. There is a second inversion which includes either end of the alignment; this one is probably a single inversion spanning the origin of replication.
- NC_012912: There are two large apparent inversion/translocations. A possible mechanism by which this could have occurred is a single inversion of the larger region, followed by a second inversion of the central region of that inverted region.
- NC_012880: This is similar to the NC_012912 alignment, and should have a similar explanation. There is the possibility of one or two further inversions in the middle of this larger inverted region, making three or four successive inversions a possible explanation for the observed alignment.

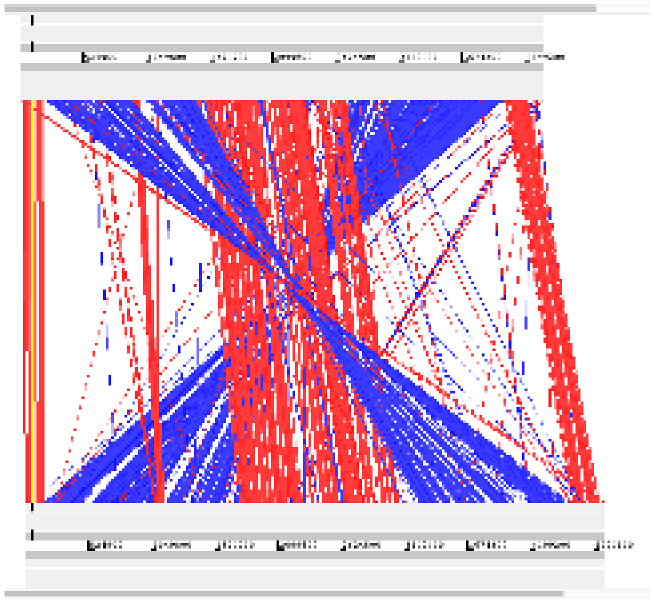
There are insertions/deletions in each genome comparison.The **NUCmer** alignments are very similar, and the interpretations should be pretty much identical.:

In [9]:
```
# Four NUCmer alignments
for ima, f in [(imread(os.path.join("images", f)), f) for f in
                   os.listdir("images") if f.startswith("nucmer")]:
    figure()
    imshow(ima)
    axis('off')
    title(f)
```

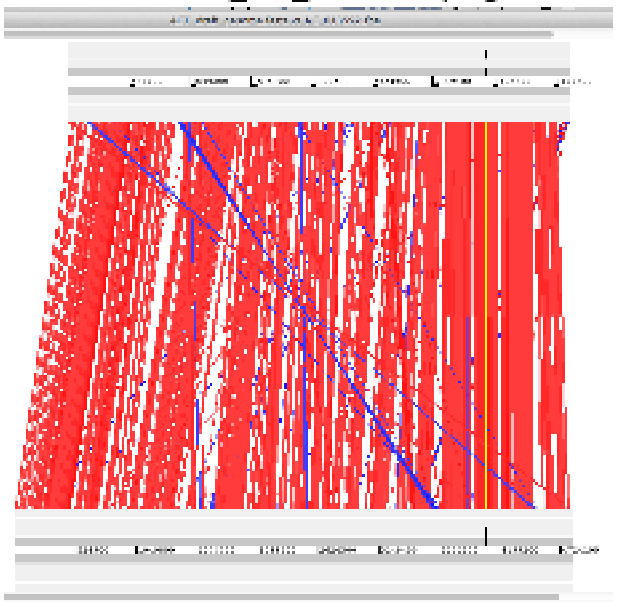
nucmer_NC_012880.png



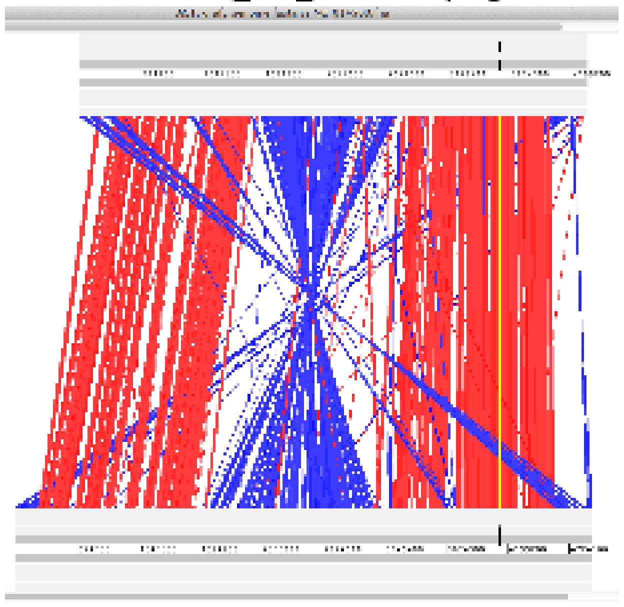
nucmer_NC_012912.png



nucmer_NC_013592.png



nucmer_NC_014500.png



5 Question 4

Use **Prodigal** to generate CDS predictions for the draft genome, placing protein sequence predictions in the `faa` subdirectory, and GFF output in the subdirectory `gff`.

HINT: The `predict_CDS` activity will be useful.

Answer

This is a straightforward use of `prodigal` as in the activity, and can be completed with the single command ‘

6 Question 5

Perform reciprocal best BLAST hit analysis on the draft genome, using the annotations from part 4 above, and all sequenced members of the genus to which it belongs.

How many reciprocal best BLAST matches does the draft genome make with each other genome from that genus? Comment on what you find.

Write the RBBH data to `.crunch` output files.

HINT: The `find_rbbh` activity will be useful. In particular, you may want to adapt the `run_rbbh.sh` script to perform the BLASTP analyses with your files.

NOTE: The BLASTP comparisons will take a while to run. You may want to make a cup of tea. Or have a long walk.

NOTE: A GenBank (`.gbk`) format file has been provided for the draft genome. The features in that GenBank file will correspond to the **Prodigal** prediction you made in **Question 4**, if default settings were used.**!!NOTE!!**: You will need to use the alternative `write_crunch()` function below to generate output suitable for **Question 6** below.

In [10]:
```python
# Function to split a full sequence reference ID into only the last value
def split_seqid(seqid):
    if '|' not in seqid:
        return seqid
    return seqid.split('|')[-2]

# Function to write a single line of a Pandas RBBH dataframe to file
def write_line(line, features, fh):
    ft1 = features[split_seqid(line['query_id_x'])]
    ft2 = features[split_seqid(line['subject_id_x'])]
    fh.write(' '.join([str(line['bitscore_x']),
                       str(line['percentage_identity_x']),
                       str(ft1[2]) if ft1[3] < 0 else str(ft1[1]),
                       str(ft1[1]) if ft1[3] < 0 else str(ft1[1]),
                       str(line['query_id_x']),
                       str(ft2[2]) if ft2[3] < 0 else str(ft2[1]),
                       str(ft2[1]) if ft2[3] < 0 else str(ft2[1]),
                       str(line['subject_id_x'])
                       ]) + '\n')

# Function to write .crunch output for ACT visualisation, from the
# RBBH identified above
def write_crunch(rbbh, features, outdir="rbbh_output",
                 filename="rbbh.crunch"):
    """ Writes .crunch output in outdir, for those RBBH stored in the
        passed dataframe
    """
    with open(os.path.join(outdir, filename), 'w') as fh:
        rbbh.apply(write_line, axis=1, args=(features, fh))
    print "Wrote file to %s" % os.path.join(outdir, filename)
```

If the student follows the hint above, they should be able to substitute the files and directories they need to work with, in the `run_rbbh_blast.sh` script, to generate the required BLAST output. A working example is given in the `run_rbbh_blast.sh` script in this directory. Running this BLASTP analysis will take a while - about 40min on my machine.

```
In [11]: !cat run_rbbh_blast.sh
```

```
#!/usr/bin/env bash
#
# run_rbbh_blast.sh
#
# Short script for Comparative Genomics and Visualisation exercise
walkthrough.
#
# (c) The James Hutton Institute 2014
# Author: Leighton Pritchard

# Define input files and input/output directories
filenames="NC_012880.faa NC_012912.faa NC_014500.faa NC_013592.faa
draft_genome.faa"
indir="faa"
outdir="rbbh_output"

# Make the output directory, if needed
mkdir -p ${outdir}

# Make BLAST databases
echo "Making databases"
for f in ${filenames}
do
    cmd="time makeblastdb -in ${indir}/${f} -dbtype prot -out
${outdir}/${f}"
    echo ${cmd}
    eval ${cmd}
done

# Loop over files and run BLASTP on all pairs of files in all
combinations
for f in ${filenames}
do
    for g in ${filenames}
    do
        if [ "${f}" != "${g}" ]; then
            echo "Running BLASTP..."
            cmd="time blastp -query ${indir}/${f} -db ${outdir}/${g}
-outfmt \"6 qseqid sseqid qlen slen length nident pident evalue
bitscore\" -out ${outdir}/${f%.faa}_vs_${g%.faa}.tab -max_target_seqs
1"
            echo ${cmd}
            eval ${cmd}
        fi
    done
done
```

Once this has been done, code from the `find_rbbh.ipynb` notebook can be used to analyse the output, either by running the notebook with appropriate changes, or as shown below:

In [12]:
```
# IMPORTS
from matplotlib.colors import LogNorm
import pandas as pd
import os

# GLOBALS
datadir = "rbbh_output"

# Function to read BLAST data
def read_data(s1, s2):
    """ Reads BLASTP tabular output data from the file corresponding
        to passed accessions, returning a Pandas dataframe.
    """
    # We specify no index for these dataframes
    df = pd.DataFrame.from_csv(os.path.join(datadir,
                                            "%s_vs_%s.tab" % (s1, s2)),
                               sep="\t", index_col=None)
    df.columns=['query_id', 'subject_id', 'query_length',
                'subject_length', 'alignment_length',
                'identical_sites', 'percentage_identity',
                'Evalue', 'bitscore']
    return df

# Function to calculate query and subject coverage for a BLAST dataset
def calculate_coverage(*df):
    """ For the passed dataframe, calculates query and subject coverage,
        and returns the original dataframe with two new columns,
        query_coverage and subject_coverage.
    """
    for d in df:
        d['query_coverage'] = 100 * d.alignment_length/d.query_length
        d['subject_coverage'] = 100 * d.alignment_length/d.subject_length
    return df


# Function to filter dataframe on percentage identity and coverage
def filter_cov_id(pid, cov, *df):
    """ Filters the passed dataframe, returning only rows that meet
        passed percentage identity and coverage criteria.
    """
    return tuple([d[(d.percentage_identity > pid) &
                    (d.query_coverage > cov) &
                  (d.subject_coverage > cov)] for d in df])


# Function to filter dataframe to best HSP match only, for any query
def filter_matches(*df):
    """ Filters rows duplicated by query_id with a
        Pandas DataFrame method: drop_duplicates. By default, this
        keeps the first encountered row, which is also the "best"
        HSP in BLAST tabular output.
    """
    return tuple([d.drop_duplicates(cols='query_id') for d in df])

# Function to parse the appropriate BLASTP data and return a
# dataframe of RBBH
def find_rbbh(s1, s2, pid=0, cov=0):
    """ Takes the accessions for two organisms, and
        1. parses the appropriate BLASTP output into two dataframes, and
           calculates coverage, discarding all but the "best" HSP, on the
           basis of bitscore, where there are multiple HSPs for a hit.
        2. cuts out rows that do not meet minimum percentage identity and
           coverage criteria.
        3. identifies RBBH from the remaining BLAST matches.
    """
    assert s1 != s2, "Accessions should not match! %s=%s" % (s1, s2)
    assert 0 <= pid <= 100, \
            "Percentage identity should be in [0,100], got %s" % str(pid)
    assert 0 <= cov <= 100, \
            "Minimum coverage should be in [0,100], got %s" % str(cov)
    # Read in tabular BLAST output
    df1, df2 = read_data(s1, s2), read_data(s2, s1)
    # Filter to best HSP match only
    df1, df2 = filter_matches(df1, df2)
    # Calculate query and subject coverage for each dataset
    df1, df2 = calculate_coverage(df1, df2)
    # Filter datasets on minimum percentage identity and minimum coverage
    df1, df2 = filter_cov_id(pid, cov, df1, df2)
    # Identify RBBH from the two datasets on the basis of matching query and
    # subject sequence ID
    matches = df1.merge(df2, left_on=("query_id", "subject_id"),
                        right_on=("subject_id", "query_id"))
    rbbh = matches[["query_id_x", "subject_id_x", "percentage_identity_x",
                    "percentage_identity_y", "query_coverage_x",
                    "query_coverage_y", "subject_coverage_x",
                    "subject_coverage_y", "bitscore_x", "bitscore_y",
                    "Evalue_x", "Evalue_y"]]
    rbbh.to_csv(os.path.join(datadir, "rbbh_%s_vs_%s.tab" % (s1, s2)),
                sep='\t')
    return df1, df2, rbbh

# Function to process GenBank files into a dictionary of CDS features,
# keyed by protein ID, where the values are a tuple of (source, start,
# end, strand) information.
def read_genbank(*filenames):
    """ Returns a dictionary of CDS annotations, where the dictionary
        keys are the CDS protein ID accession numbers, and the values
        are (source, start, end, strand, id) information about CDS
        location on the chromosome.

        - *filenames, the organisms' GenBank annotation files
    """
    ft_dict = {}
    for filename in filenames:
        with open(filename, 'rU') as fh:
            record = SeqIO.read(fh, 'genbank')
            # Reconstruct the name in the corresponding .fna file
            record_name = '|'.join(["gi", record.annotations['gi'],
                                    "ref", record.id])
            for ft in [f for f in record.features if f.type == "CDS"]:
                ft_dict[ft.qualifiers['protein_id'][0]] = \
                    (record_name, int(ft.location.start),
                     int(ft.location.end), ft.location.strand)
    print "Loaded %d features" % len(ft_dict)
    return ft_dict
```

Answer

Example use of the functions above is shown below. We can process all four in a loop - which the student may notice:

In [13]:
```
for prefix in ('NC_012880', 'NC_012912', 'NC_013592', 'NC_014500'):
    print "Processing %s" % prefix
    df1, df2, rbbh = find_rbbh('draft_genome', prefix, pid=0, cov=0)
    print "RBBH count vs %s: %d" % (prefix, len(rbbh))
    features = read_genbank("gbk/draft_genome.gbk",
                            "gbk/%s.gbk" % prefix)
    write_crunch(rbbh, features,
                 filename="draft_genome_vs_%s.crunch" % prefix)
```

```
Processing NC_012880
RBBH count vs NC_012880: 2889
Loaded 7931 features
Wrote file to rbbh_output/draft_genome_vs_NC_012880.crunch
Processing NC_012912
RBBH count vs NC_012912: 3074
Loaded 8124 features
Wrote file to rbbh_output/draft_genome_vs_NC_012912.crunch
Processing NC_013592
RBBH count vs NC_013592: 3113
Loaded 8105 features
Wrote file to rbbh_output/draft_genome_vs_NC_013592.crunch
Processing NC_014500
RBBH count vs NC_014500: 3078
Loaded 8510 features
Wrote file to rbbh_output/draft_genome_vs_NC_014500.crunch
```

So the answer is 2889 RBBH with `NC_012880`, 3074 with `NC_012912`, 3113 with `NC_013592` and 3078 with `NC_014500`.

For the comment, any sensible observation will do. Possibilities include noting that the number of RBBH is fairly constant and about 75% of the total CDS complement, or that there's no obvious association between RBBH count and the extent of genome reorganisation.

7 Question 6

Use **i-ADHoRe** to identify collinear regions in the draft genome, and **one** other member of the genus.

Choose one of the sequenced members of the genus. Generate a visualisation of the collinear regions in that genome and the draft genome, and comment on the output.

HINT: You will find the `i-ADHoRe` activity useful - particularly if you modify the `generate_config.py` script and copy `iadhore.py` to this directory, or if you reuse the `i-ADHoRe.ipynb` notebook.

HINT: You might find the command '

Answer

We work through with `NC_013592` as the chosen example.

Firstly, the `.crunch` files will need to be processed, using the hint above: `$ cat rbbh_output/draft_genome_vs_NC_013592.crunch | cut -d ' ' -f 5,8 > i-adhore/rbbh_data.tab`.

A local copy of `generate_config.py` can be adapted to refer to the relevant analysis files, as in the current directory. The only changes necessary are to point to different files and directories.

```
$ diff generate_config.py ../Part_2/i-ADHore/generate_config.py
41c41
< gbkdir = 'gbk'
---
> gbkdir = 'data'
44,48c44,46
< genbank_files = {'draft_genome': os.path.join(gbkdir, 'draft_genome.gbk'),
<                    #'NC_012880': os.path.join(gbkdir, 'NC_012880.gbk'),
<                    #'NC_012912': os.path.join(gbkdir, 'NC_012912.gbk'),
<                    'NC_013592': os.path.join(gbkdir, 'NC_013592.gbk'),
<                    #'NC_014500': os.path.join(gbkdir, 'NC_014500.gbk')
---
> genbank_files = {'ECA': os.path.join(gbkdir, 'NC_004547.gbk'),
>                  'Pecwa': os.path.join(gbkdir, 'NC_013421.gbk'),
>                  'ETA': os.path.join(gbkdir, 'NC_010694.gbk')
52c50
< rbbh_data = 'i-adhore/rbbh_data.tab'
---
> rbbh_data = 'data/rbbh_data.tab'
```

Then this can be run, followed by `i-adhore`, as in the activity:

```
$ python generate_config.py

$ i-adhore i-ADHoRe_config.ini
```

placing the output in `i-ADHoRe_activity`.

Next, code from the `i-ADHoRe.ipynb` notebook can be reused to generate the visualisation. The students may find it easier just to reuse the notebook directly, changing the location of the input files.

In [14]:
```python
# IMPORTS
import os
from itertools import chain

# Biopython
from Bio.Graphics import GenomeDiagram as gd
from Bio.Graphics.ColorSpiral import get_color_dict, get_colors
from Bio import SeqIO
from Bio.SeqFeature import SeqFeature, FeatureLocation

# Reportlab
from reportlab.lib.units import cm
from reportlab.lib import colors

# local
from iadhore import IadhoreData

# IPython-specific
from IPython.display import Image

gbkdir = "gbk"
gbkdata = {'draft_genome': os.path.join(gbkdir, 'draft_genome.gbk'),
            #'NC_012880': os.path.join(gbkdir, 'NC_012880.gbk'),
            #'NC_012912': os.path.join(gbkdir, 'NC_012912.gbk'),
            'NC_013592': os.path.join(gbkdir, 'NC_013592.gbk'),
            #'NC_014500': os.path.join(gbkdir, 'NC_014500.gbk')
            }
multiplicon_file = "i-ADHoRe_activity/multiplicons.txt"
segment_file = "i-ADHoRe_activity/segments.txt"
outdir = "gd_output"

# For visualisation reasons, we must have a stable ordered list
# of organisms
orgs = tuple(sorted(gbkdata.keys()))

# Make an output directory if it doesn't exist
if not os.path.isdir(outdir):
    os.mkdir(outdir)

gdd = gd.Diagram("i-ADHoRe exercise")  # Instantiate diagram
```

```python
# Dictionaries to hold records and tracks for later reference
records, tracks, featuresets, regionsets = {}, {}, {}, {}

# Get a colour for each organism, in a dictionary keyed by organism name
org_colours = get_color_dict(gbkdata.keys(), a=4)

# Reset levels for first track
tracklevel = 0

# Load each GenBank file, and create a track each time
for org in orgs:
    print "Loading %s" % gbkdata[org]
    records[org] = SeqIO.read(gbkdata[org], "genbank")
    tracks[org] = gdd.new_track((8 * tracklevel) + 1, name=org,
                                greytrack=True, greytrack_labels=10,
                                height=1, start=0, end=len(records[org]))
    regionsets[org] = tracks[org].new_set(name="collinear regions")
    print "Adding features for %s" % org
    featuresets[org] = tracks[org].new_set(name="CDS features")
    label_state = True
    for feature in [f for f in records[org].features if f.type == 'CDS']:
        label_state = not label_state  # Alternate labels
        featuresets[org].add_feature(feature, color=org_colours[org],
                                     label=False, sigil="ARROW",
                                     label_size=3)
    tracklevel += 1  # increment track level for visualisation

# Function to render circular and linear versions of a GenomeDiagram
def render_diagram(diagram, outdir, filestem, circular=True, linear=True):
    if linear:
        print "Rendering linear diagram"
        diagram.draw(format='linear', orientation='landscape',
                     pagesize=(300 * cm,
                               (len(diagram.get_tracks())*91.4/15)*cm),
                     fragments=1)
        diagram.write(os.path.join(outdir, filestem + '_l.pdf'), 'PDF')
        diagram.write(os.path.join(outdir, filestem + '_l.png'), 'PNG')
    if circular:
        print "Rendering circular diagram"
        diagram.draw(format='circular', orientation='landscape',
                 pagesize=(100*cm, 100*cm))
        diagram.write(os.path.join(outdir, filestem + '_c.pdf'), 'PDF')
        diagram.write(os.path.join(outdir, filestem + '_c.png'), 'PNG')

# Convenience function for getting feature locations
def get_ft_loc(org, ft):
    for f in records[org].features:
        if f.type == 'CDS' and f.qualifiers['locus_tag'][0] == str(ft):
            return f.location.nofuzzy_start, f.location.nofuzzy_end

# Load i-ADHoRe output
data = IadhoreData(multiplicon_file, segment_file)

# Select only the multiplicons at level 3 - these are our syntenous
# core regions
full_leaves = data.get_multiplicons_at_level(2)

# For rendering, we want to have a different colour for each region
region_colours = list(get_colors(len(full_leaves), a=5, b=0.33,
                                 jitter=0.25))

# How many syntenous core regions are there?
print "We found %d syntenous core regions." % len(full_leaves)

# We loop over our syntenous core regions, and add crosslinks
# to the diagram
for midx, m in enumerate(full_leaves):
    segments = data.get_multiplicon_segments(m)
    # Loop over pairs of consecutive genomes in the table, and add
    # crosslinks for multiplicons
    for idx in range(1, len(orgs)):
        try:  # There's a bug in here I've not fixed, yet
            org1, org2 = orgs[idx-1], orgs[idx]
            org1loc = \
                list(chain.from_iterable([get_ft_loc(org1, f) for f in
                                                        segments[org1]]))
            org2loc = \
                list(chain.from_iterable([get_ft_loc(org2, f) for f in
                                                        segments[org2]]))
            org1ft = (tracks[org1], min(org1loc), max(org1loc))
            org2ft = (tracks[org2], min(org2loc), max(org2loc))
            # Need to create a colour here rather than pass a tuple
            # - unlike with features.
            # I still need to raise this bug in Biopython!
            c = colors.Color(region_colours[midx][0], region_colours[midx][1],
                             region_colours[midx][2])
            crosslink = gd.CrossLink(org1ft, org2ft, c)
            gdd.cross_track_links.append(crosslink)
            # Add feature to track (no transparency)
            # We add org1 here, then the final org when the
            # looping's done
            f = SeqFeature(FeatureLocation(min(org1loc),
                                           max(org1loc)),
                               strand=None)
            regionsets[org1].add_feature(f, label=False,
                        color=colors.Color(region_colours[midx][0],
                                           region_colours[midx][1],
                                           region_colours[midx][2]))
        except ValueError:
            continue
    # Finish off the cross-link features
    try:  # This follows from the bug I've not yet squashed, above
        f = SeqFeature(FeatureLocation(min(org2loc), max(org2loc)),
                           strand=None)
    except ValueError:
        print "Problem with location: ", org2loc
        continue
    regionsets[org2].add_feature(f, label=False,
                           color=colors.Color(region_colours[midx][0],
                                              region_colours[midx][1],
                                              region_colours[midx][2]))

render_diagram(gdd, outdir, "exercise", circular=False)
```

```
Loading gbk/NC_013592.gbk
Adding features for NC_013592
Loading gbk/draft_genome.gbk
Adding features for draft_genome
We found 104 syntenous core regions.
Rendering linear diagram
```

Rendering the output gives a usable figure.

As before, any sensible comment will do. Possibilities include noting the extent of rearrangement, and ordered, syntenous regions - noting the inversions between the draft and other genomes, and comparing to the **ACT** visualisations of the **megaBLAST** or **MUMmer** output.

```
In [15]: Image("gd_output/exercise_l.png")
```

```
Out [15]:
```