Week 15 - Lecture 29

István Albert
Huck Institutes for the Life Sciences
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"

I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN!
IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?

I JUST TYPED
import antigravity
THAT'S IT?

... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.
Coordinate systems

- 0 based $\rightarrow$ first 10 $\rightarrow$ 0, 1, 2, ... 9
- 1 based $\rightarrow$ first 10 $\rightarrow$ 1, 2, 3, 4, ... 10

Typically

- 0 based are non inclusive 10:20 $\rightarrow$ [ 10, 20 )
- 1 based include both ends 10:20 $\rightarrow$ [ 10, 20 ]
Why do the two conventions exist?

• 1 based indexing makes more sense in everyday life

• 0 based indexing makes more sense when computing with intervals
## Comparing the indexing systems

<table>
<thead>
<tr>
<th>1 based indexing</th>
<th>0 based indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Third element</td>
<td>Third element</td>
</tr>
<tr>
<td>First ten:</td>
<td>First ten:</td>
</tr>
<tr>
<td>data[1:10]</td>
<td>data[0:10]</td>
</tr>
<tr>
<td>Second ten:</td>
<td>Second ten:</td>
</tr>
<tr>
<td>Third ten:</td>
<td>Third ten:</td>
</tr>
<tr>
<td>Size of the slice = 10 → end-start + 1</td>
<td>Size of the slice = 10 → end – start</td>
</tr>
<tr>
<td>Empty slice: data[?] – not sure</td>
<td>Empty slice data[10:10] → size=0</td>
</tr>
<tr>
<td>Five element long segment starting at 1000</td>
<td>Five element long segment starting at 1000</td>
</tr>
<tr>
<td>data[1000: 10000 + 4 ]</td>
<td>data[1000: 10000 + 5 ]</td>
</tr>
</tbody>
</table>
Genomic coordinates – brief overview

DNA two stranded $\rightarrow$ one coordinate system

in most formats use \textbf{start < end} even on the reverse strand

we need to remember that on reverse strand the upstream region is past of the 300 end.
Two commonly used formats

• BED – UCSC genome browser ➔ 0 based non inclusive ➔ also used to display tracks in the genome browser

• GFF – Sanger institute in Great Britain ➔ 1 based inclusive indexing system
Two commonly used interval standards BED and GFF

The first three required BED fields are:

1. **chrom** - The name of the chromosome (e.g. chr3, chrY, chr2_random) or scaffold (e.g. scaffold10671).
2. **chromStart** - The starting position of the feature in the chromosome or scaffold. The first base in a chromosome is numbered 0.
3. **chromEnd** - The ending position of the feature in the chromosome or scaffold. The chromEnd base is not included in the display of the feature. For example, the first 100 bases of a chromosome are defined as chromStart=0, chromEnd=100, and span the bases numbered 0-99.

The 9 additional optional BED fields are:

4. **name** - Defines the name of the BED line. This label is displayed to the left of the BED line in the Genome Browser window when the track is open to full display mode or directly to the left of the item in pack mode.
5. **score** - A score between 0 and 1000. If the track line useScore attribute is set to 1 for this annotation data set, the score value will determine the level of gray in which this feature is displayed (higher numbers = darker gray). This table shows the Genome Browser’s translation of BED score values into shades of gray:

<table>
<thead>
<tr>
<th>shade</th>
<th>score in range</th>
</tr>
</thead>
<tbody>
<tr>
<td>light</td>
<td>≤ 166</td>
</tr>
<tr>
<td>medium</td>
<td>167-277</td>
</tr>
<tr>
<td>medium</td>
<td>278-388</td>
</tr>
<tr>
<td>medium</td>
<td>389-499</td>
</tr>
<tr>
<td>dark</td>
<td>500-611</td>
</tr>
<tr>
<td>dark</td>
<td>612-722</td>
</tr>
<tr>
<td>dark</td>
<td>723-833</td>
</tr>
<tr>
<td>dark</td>
<td>834-944</td>
</tr>
<tr>
<td>dark</td>
<td>≥ 945</td>
</tr>
</tbody>
</table>

6. **strand** - Defines the strand - either ‘+’ or ‘-’.
7. **thickStart** - The starting position at which the feature is drawn thickly (for example, the start codon in gene displays).
8. **thickEnd** - The ending position at which the feature is drawn thickly (for example, the stop codon in gene displays).
9. **itemRgb** - An RGB value of the form R,G,B (e.g. 255,0,0). If the track line itemRgb attribute is set to "On", this RGB value will determine the display color of the data contained in this BED line. **NOTE**: It is recommended that a simple color scheme (eight colors or less) be used with this attribute to avoid overwhelming the color resources of the Genome Browser and your Internet browser.
10. **blockCount** - The number of blocks (exons) in the BED line.
11. **blockSizes** - A comma-separated list of the block sizes. The number of items in this list should correspond to blockCount.
12. **blockStarts** - A comma-separated list of block starts. All of the blockStart positions should be calculated relative to chromStart. The number of items in this list should correspond to blockCount.

Example:
Here’s an example of an annotation track that uses a complete BED definition:

```
track name=pairedReads description="Clone Paired Reads" useScore=1
chr22 1000 5000 cloneA 960 + 1000 5000 0 2 567,488, 0.3512
chr22 2000 6000 cloneB 960 + 2000 6000 0 2 453,398, 0.5601
```
This section describes the representation of a protein-coding gene in GFF3. To illustrate how a canonical gene is represented, consider Figure 1 (figure1.png). This indicates a gene named EDEN extending from position 1000 to position 9000. It encodes three alternatively-spliced transcripts named EDEN.1, EDEN.2 and EDEN.3, the last of which has two alternative translational start sites leading to the generation of two protein coding sequences.

There is also an identified transcriptional factor binding site located 50 bp upstream from the transcriptional start site of EDEN.1 and EDEN.2.

Here is how this gene should be described using GFF3:

```
0  #gff-version 3
1  #sequence-region ctg123 1 1497228
2  ctg123 . gene 1000 9000 . + . ID=gene00001;Name=EDEN
3  ctg123 . TF_binding_site 1000 1012 . + . ID=tfs00001;Parent=gene00001
4  ctg123 . mRNA 1050 9000 . + . ID=mRNA00001;Parent=gene00001
5  ctg123 . mRNA 1050 9000 . + . ID=mRNA00002;Parent=gene00001
6  ctg123 . mRNA 1300 9000 . + . ID=mRNA00003;Parent=gene00001
```
We may have data in different coordinate systems!

- **Biopython** → **Sequence** objects are 0 based whereas GFF files are 1 based!

Conversion from GFF to a BED based system

- **GFF start** → **BED start** - 1
- **GFF end** → **BED end**

Conversion from BED to GFF

- **BED start** → **GFF start** + 1
- **BED end** → **GFF end**
Today’s problem

• Relates to Hosung’, Megha’ and Liye’s project

• Compute distances between intervals

• For example: for each interval on one strand find the closest on the other strand
Important details

- What are the anchor points (the sections that represent the intervals)

- Which direction does the comparison proceed – upstream, downstream, any?
We start with the simplest problem

- Equal width intervals on both strands
- Compute the distance between midpoints
- Find the closest midpoint on the other strand
- We expect that the reverse strand is always shifted downstream from measurement on the forward strand:

  \[ \text{reverse} - \text{forward} > 0 \]
Now we start: investigate the data

```python
import bmmb

rows = bmmb.read_tabular('peak-data.txt')

print rows[0]
print rows[1]
```

Command Output:
```
['chrom', 'strand', 'start', 'end', 'value', 'chrm', 'dataid']
['chr04', '+', '1429696', '1429718', '16.35458111', '4', '1041429696']
```
Add a convenience function to bmmb for integers

```python
103 return values
104
105 def string_column(rows, colname):
106 return get_column(rows, colname, mapfunc=str)
107
108 def float_column(rows, colname):
109 return get_column(rows, colname, mapfunc=myfloat)
110
111 def int_column(rows, colname):
112 return get_column(rows, colname, mapfunc=myint)
```

If get are stuck adding this, see the bmmb.py module included with the data
Find midpoints

```python
import bmm

rows = bmm.read_tabular('peak-data.txt')
start = bmm.int_column(rows, 'start')
end = bmm.int_column(rows, 'end')

def midpoints(startx, endx):
    return (startx + endx) / 2

mids = map(midpoints, start, end)

print(mids[:3])
```

Output:

```
[1429707, 611924, 968288]
```
Extract the information we need to make distance measurements

1. Chromosome information (intervals are relative to a chromosome)
2. Midpoint information
3. Strand information
4. Data value (later we might want to filter by that)

Let’s make some tuples that contain everything we need: (chrom, mid, value, strand)
import bmbb
from itertools import *

def midpoints(startx, endx):
    return (startx+endx)/2

rows = bmbb.read_tabular('peak-data.txt')
start = bmbb.int_column(rows, 'start')
end = 
mids = 
chrom = 
value = 
strand = 

rows = izip(chrom, mids, value, strand)

print rows.next()
Separate into strands

```python
rows = izip(chrom, mids, value, strand)
rows = list(rows)

def strand_filter(value):
    "Returns a function that checks last element"
    return condition

# separate into strands
fwd = filter( strand_filter('+'), rows )
rev = filter( strand_filter('-'), rows )

print len(fwd), len(rev)
```
We lay out the scaffolding on how the code might work later.
Divide the remaining problem into smaller tasks

• Each coordinate makes sense within a certain chromosome → create a chromosome filter

• Compute all differences on the matching chromosome

• Take all positive values

• Sort

• Take the smallest
Find the matching chromosome

```python
def find_closest(row):
    # unpack the tuple
    chrom, mid, value, strand = row

    # create the chromosome filter
    def chrom_filter(elem):
        return elem[0] == chrom

    # temporary forward strand that only contains the
    # matching chromosomes
    tempfwd = filter(chrom_filter, fwd)

    return (row, tempfwd[0])

# this will hold the results
res = imap(find_closest, rev)

print res.next()
print res.next()
```
Find all differences from current element to all other elements
Sort and filter for positive values
def positive(pair):
    diff, elem = pair
    return diff > 0

# it contains the distance and the match
decorated = filter(positive, decorated)

# let's rearrange the data a bit to look nicer
diff, match = decorated[0]

return diff, row, match

# this will hold the results
res = imap(find_closest, rev)

print res.next()
print res.next()
A quick benchmark – estimate runtime

```python
58 69
return diff, row, match

# this will hold the results
res = imap(find_closest, rev)

# a quick benchmark
import time
start = time.time()

for i in range(10):
  print res.next()

elapsed = time.time() - start
projected = elapsed * 6000/10

print 'Elapsed %4.2fs Projected %4.2fs' % (elapsed, projected)
```

```
(18, ('chr02', 682123, 4765.4937470000004, '-'), ('chr02', 682105, 500.4594430999999, '+'))
Elapsed 0.04s Projected 22.80s
```
Bugfix – it is possible to not have anything in positive range

```python
# it contains the distance and the match
decorated = filter(positive, decorated)

# let's rearrange the data a bit to look nicer
# it is possible to not get any positive result back
if decorated:
    diff, match = decorated[0]
else:
    diff, match = None, None

return diff, row, match

# this will hold the results
res = imap(find_closest, rev)

# slice it to smaller while testing
res = islice(res, 1000)
```
Create histogram of distances

```python
# slice it to smaller while testing
res = islice(res, 1000)

# generate a histogram
import pylab

def extract(row):
    # extract the distance only
    return row[0]

def toolarge(value):
    # keep only differences below a certain level
    return value < 50

data = map(extract, res)
data = filter(bmmb.missing, data)
data = filter(toolarge, data)

bins = range(1, 50)
pylab.hist(data, bins=bins)
pylab.show()
```
Average distance across strands
Scalability – not too good

It is a brute force method – for every single measurement on one strand it:

1. Filters for the chromosomes
2. Searches all data

N the number of data on forward strand
M the number of data on reverse strand

N * M operations – it scales quadratically

N = M = 100 $\rightarrow$ 10,000 operations

N = M = 200 (double the size) $\rightarrow$ 40,000 (quadruple the runtime)

We’ll look at optimizations next time.
Optional homework – one or more

1. Modify the code to only consider data that has a value of above 10

2. Save the results of the run into a file

3. Modify the program to also output the dataid columns with distances
Week 15 - Lecture 30

István Albert
Huck Institutes for the Life Sciences
About the course

- I hope it was interesting
- I hope it was useful
- Future plans ➔ expand on the subjects, do more difficult problems in a second lecture series
- This course **may or may not be offered** in the future.
- Depends on you the potential audience, advisors and administration.
Ask your questions there! We’ll try to build it into an extensive knowledge base!
Computation == Thought

Final advice

If you know what an object **IS**
then you will know what it **DOES**

Print it. Check its type. Check its content.
Today we’ll do performance analysis

• It is notoriously difficult to estimate what takes the longest to execute

• Profilers allow us to trace program execution

• Let’s trace our last program and see if we can speed it up (*spoiler: YES!*).
We’ll use the interval search program

- Also sent out via email in case you didn’t get it to work

- We can only profile functions, so we need to move the program into a separate function. (easy)
Move everything into a function

```python
import bmmb
from itertools import *

def run_search():
    """Put everything into a function so that the profiler can run it""

    def midpoints(startx, endx):
        return (startx+endx)/2

    rows = bmmb.read_tabular('peak-data.txt')
    start = bmmb.int_column(rows, 'start')
    end = bmmb.int_column(rows, 'end')
    mids = map(midpoints, start, end)
    chrom = bmmb.string_column(rows, 'chrom')
    value = bmmb.float_column(rows, 'value')
    strand = bmmb.string_column(rows, 'strand')
```

Here is how we run the function:

```python
import time

# this will hold the results
res = imap(find_closest, rev)

# slice it to 100 elements to save time
res = islice(res, 100)
return list(res)

# comment this out when importing it as a module
import time

start = time.time()
run_search()

end = time.time()
diff = end - start
print "Elapsed %.2f, projected=\%.2f" % (diff, diff*6000/100)
```

Elapsed 0.59, projected=35.16
How to profile a program

```python
import math, cProfile

def add(x):
    return x + x

def formula(x):
    return x + x + math.sin(x)

def compute():
    """We'll test this function""
    for num in xrange(10**5):
        x = add(num)
        y = formula(num)

    # this runs the compute function and collects
    # the statistics in the stats.bin file
    print '*** start'
    cProfile.run('compute()', 'stats.bin')
    print '*** end'
```

```
*** start
*** end
```
Display the statistics

```python
import pstats

ps = pstats.Stats('stats.bin')

# clean up filenames for the report
ps.strip_dirs()

# sort the statistics by cumulative time
ps.sort_stats('cumulative')

# print the stats
ps.print_stats()
```

```
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
1  0.000    0.000    1.079    1.079  <string>:1(<module>))
1  0.379    0.379    1.079    1.079  prof1.py:9(compute)
100000  0.357    0.000    0.545    0.000  prof1.py:6(formula)
100000  0.188    0.000    0.188    0.000  {math.sin}
100000  0.155    0.000    0.155    0.000  prof1.py:3(add)
1  0.000    0.000    0.000    0.000  {method 'disable' of '_lsprof.Profiler' objects}
```
Benchmark the `run_search` function

```python
from align1 import run_search
import cProfile

# this runs the align function and collects
# the statistics in the stats.bin file
print '*** start'

cProfile.run(('run_search()', 'stats.bin'))

print '*** end'
```
Questions

• How many times was the `chrom_filter()` function invoked?

• What percent of the runtime was spent inside the `chrom_filter()` function.

• What does the `chrom_filter()` function do?
Re-computing the matching chromosomes is a waste of time

• Why not store each chromosomal input as values of a dictionary

• See align2.py

• Profile this program. What speedup do you see? What is the most expensive function?
Benchmark align2 module - it is now twice as fast!

```python
from align2 import run_search
import cProfile

# this runs the align function and collects # the statistics in the stats.bin file
print '*** start'
cProfile.run(('run_search()', 'stats.bin'))
print '*** end'
```
New Questions

• What is the slowest function now?

• Tiny optimizations in functions that are called many many times will add up.
Adding binary search to the problem

• We have not covered this → instead of computing all differences what if we first found best candidates then used only those

• align3.py has this code
Benchmarking align3.py faster but not by much?

```python
from align3 import run_search
import cProfile

# this runs the align function and collects
# the statistics in the stats.bin file
print '*** start'
cProfile.run('run_search()', 'stats.bin')

print '*** end'
```
Runtime components

• There are fixed costs and there are variable compute costs.

• Reading in a file will add X seconds. The faster the rest of the code the more the fixed costs seem to dominate

• But the program may have different scaling properties
We can easily double/triple the data
Runtime for normal and tripled data

The graph shows the runtime variation for optimization and data sizes. The x-axis represents different methods (Method 1, Method 2, Method 3), and the y-axis represents runtime. Two data sizes are compared: original size and tripled size. The bar chart indicates that Method 1 has significantly higher runtime compared to the other methods.
```python
from numpy import *
import matplotlib.pyplot as plt

size = 0.35
index = array([[1, 2, 3]])
orig = array([[33, 16, 14]])
triple = array([[82, 30, 17]])

fig = plt.figure()
ax = fig.add_subplot(111)
rects1 = ax.bar(index, orig, size, color='b')
rects2 = ax.bar(index + size, triple, size, color='r')

ax.set_xlabel('Runtime')
ax.set_ylabel('Runtime variation for optimization and data')
ax.set_xticks(index + size)
ax.set_xticklabels(('Method 1', 'Method 2', 'Method 3'))
ax.legend((rects1[0], rects2[0]), ('Original size',))

plt.show()
```
Thank you!

Good Luck!