In this notebook we perform parallel calculations as in "complete" analysis, but use Robinson-Foulds distance rather than Pearson tip-to-tip distances to confirm that our conclusions are robust to the choice of distance metric.

In this notebook we make use of the trees that were computed in the complete notebook to avoid wasted compute time. Note that **this notebook will not work** if you have not previously run the complete notebook on the same instance.

We begin by defining the Robinson-Foulds (RF) distance as the `RFD` function.

```
def shear(tree, names):
    """Lop off tips until the tree just has the desired tip names"""

    tcopy = tree.deepcopy()
    all_tips = set([n.Name for n in tcopy.tips()])
    ids = set(names)

    if not ids.issubset(all_tips):
        raise ValueError, "ids are not a subset of the tree!"

    while len(tcopy.tips()) != len(ids):
        for n in tcopy.tips():
            if n.Name not in ids:
                n.Parent.removeNode(n)
        tcopy.prune()

    return tcopy

def RFD(tree1, tree2, proportion = False):
    """Calculates the Robinson and Foulds symmetric difference

    Implementation based off of code by Julia Goodrich
    """
    t1names = tree1.getTipNames()
    t2names = tree2.getTipNames()
    if set(t1names) != set(t2names):
        if len(t1names) < len(t2names):
            tree2 = shear(tree2, t1names)
        else:
            tree1 = shear(tree1, t2names)

    tree1_sets = tree1.subsets()
    tree2_sets = tree2.subsets()

    not_in_both = tree1_sets ^ tree2_sets

    total_subsets = len(tree1_sets) + len(tree2_sets)
    dist = len(not_in_both)

    if proportion:
        dist = dist/float(total_subsets)
    return dist
```

Next we generate the list of trees that we'll work on. Notice that we're reusing the trees computed in the 'complete' notebook here.

```
base_region_boundaries = [
      ('v2', 136, 1868), #27f-338r
      ('v2.v3', 136, 2232),
      ('v2.v4', 136, 4051),
      ('v2.v6', 136, 4932),
      ('v2.v8', 136, 6426),
      ('v2.v9', 136, 6791),
      ('v3', 1916, 2232), #349f-534r
      ('v3.v4', 1916, 4051),
      ('v3.v6', 1916, 4932),
      ('v3.v8', 1916, 6426),
      ('v3.v9', 1916, 6791),
      ('v4', 2263, 4051), #515f-806r
      ('v4.v6', 2263, 4932),
      ('v4.v8', 2263, 6426),
      ('v4.v9', 2263, 6791),
      ('v6', 4653, 4932), #967f-1048r
      ('v6.v8', 4653, 6426),
      ('v6.v9', 4653, 6791),
      ('v9', 6450, 6791), #1391f-1492r
      ('full.length', 0, 7682), # Start 150, 250, 400 base pair reads
      ('v2.150', 136, 702),
      ('v2.250', 136, 1752),
      ('v2.v3.400', 136, 2036), # Skips reads that are larger than amplicon size
      ('v3.v4.150', 1916, 2235),
      ('v3.v4.250', 1916, 2493),
      ('v3.v4.400', 1916, 4014),
      ('v4.150', 2263, 3794),
      ('v4.250', 2263, 4046),
      ('v4.v6.400', 2263, 4574),
      ('v6.v8.150', 4653, 5085),
      ('v6.v8.250', 4653, 5903),
      ('v6.v8.400', 4653, 6419)
]
percentages = range(76,98,3)

labels = []
trees = []
from os.path import exists
for p in percentages:
    for rb in base_region_boundaries:
        tree_path = '/home/ubuntu/data/gg_%s_otus_4feb2011_aligned_%s_pfiltered.tre' % (p
            ,rb[0])
        if exists(tree_path):
            labels.append("%i.%s" % (p,rb[0]))
            trees.append(tree_path)
print len(trees)
```

```
256
```

Set up the parallel IPython environment and the parallel map function.

```
from IPython import parallel
rc = parallel.Client(packer='pickle')
view = rc.load_balanced_view()
print "We have %i engines available" % len(rc.ids)
# skip cached results (for faster debugging)
rc[:]['skip_cache'] = True

import socket
ar = rc[:].apply_async(socket.gethostname)
mapping = ar.get_dict()
```

```
We have 111 engines available
```

```
rev_map = {}

per_node = 4
for eid, node in mapping.iteritems():
    if node not in rev_map:
        rev_map[node] = []
    if len(rev_map[node]) < per_node:
        rev_map[node].append(eid)
```

```
from IPython.utils.data import flatten
targets = flatten(rev_map.values())
len(targets)
```

```
56
```

```
dv = rc[targets]
dv.push(dict(shear=shear, RFD=RFD, trees=trees), block=False)
load_ar = dv.execute(
"""
from cogent.parse.tree import DndParser
ts = [DndParser(open(tree_fp)) for tree_fp in trees]
""", block=False)
load_ar
```

```
<AsyncResult: execute>
```

```
def compute_row(i, ts):
    """function to compute a whole row with RFD"""
    import numpy
    row = []
    t1 = ts[i]
    for t2 in ts:
```

```
        row.append(RFD(t1, t2, True))
    return numpy.array(row)
```

Initialize the calculations.

```
ntasks = len(trees)
tree_ref = parallel.Reference('ts')
rf_view = rc.load_balanced_view(targets=targets)
row_ars = [ rf_view.apply_async(compute_row, i, tree_ref) for i in range(ntasks) ]
```

Poll for completion of the jobs, providing updates each time another job has completed. When the computation has completed, write the distance matrix to file.

```
import time
ready = [ ar.ready() for ar in row_ars ]
while not all(ready):
    ready = [ ar.ready() for ar in row_ars ]
    sys.stdout.write('\r%3i / %3i' % (sum(ready), len(ready)))
    sys.stdout.flush()
    time.sleep(5)

rfrows = [ ar.get() for ar in row_ars ]

rf_dist_mat = numpy.array(rfrows)
```

```
0 / 256
0 / 256
0 / 256
...
[ Elided for brevity ]
...
252 / 256
253 / 256
255 / 256
256 / 256
```

```
from qiime.format import format_distance_matrix
rf_parallel_fp = '/home/ubuntu/data/rf_distance_matrix.txt'
rf_parallel_f = open(rf_parallel_fp,'w')
rf_parallel_f.write(format_distance_matrix(labels,rf_dist_mat))
rf_parallel_f.close()
```

We now have a Robinson-Foulds distance matrix which we can compare against our original Pearson distance matrix to confirm that our conclusions are robust to the choice of distance metric.

We'll begin by comparing these with the Mantel test.

```
!compare_distance_matrices.py -i /home/ubuntu/data/distance_matrix_complete.txt,/home/
    ubuntu/data/rf_distance_matrix.txt -o /home/ubuntu/data/pearson_v_rf --method mantel
    -n 1000
!cat /home/ubuntu/data/pearson_v_rf/mantel_results.txt
```

```
# Number of entries refers to the number of rows (or cols) retained in each
# distance matrix after filtering the distance matrices to include only those
# samples that were in both distance matrices. p-value contains the correct
# number of significant digits.
DM1 DM2 Number of entries Mantel r statistic p-value Number of permutations Tail type
/home/ubuntu/data/distance_matrix_complete.txt /home/ubuntu/data/rf_distance_matrix.txt
256 0.76938 0.001 1000 two sided
```

We see that the Mantel test gives a significant p-value ($p = 0.001$), telling us that the Pearson distances are significantly correlated with the Robinson-Foulds distances.

As we based our original conclusions on our interpretation of a Principal Coordinates (PCoA) plot, it's also useful to perform a Procrustes analysis to tell us whether we would be likely to derive the same conclusion from the PCoA plot generated from Robinson-Foulds distances, based on low sum-of-squares distances between the paired points in the two principal coordinate plots.

```
!principal_coordinates.py -i /home/ubuntu/data/rf_distance_matrix.txt -o /home/ubuntu/
    data/rf_pc.txt
```

```
!transform_coordinate_matrices.py -i /home/ubuntu/data/rf_pc.txt,/home/ubuntu/data/pc.
    txt -o /home/ubuntu/data/pearson_v_rf/ -r 1000
```

```
!cat /home/ubuntu/data/pearson_v_rf/rf_pc_pc_procrustes_results.txt
```

```
FP1 FP2 Included_dimensions MC_p_value Count_better M^2
/home/ubuntu/data/rf_pc.txt /home/ubuntu/data/pc.txt 3 0.000 0 0.686
```

We see that the Procrustes test also gives a significant p-value ($p < 0.001$). We can visualize the Procrustes plot in three dimensions as well.

```
!compare_3d_plots.py -i /home/ubuntu/data/pearson_v_rf/pc1_transformed.txt,/home/ubuntu/
    data/pearson_v_rf/pc2_transformed.txt -o /home/ubuntu/data/pearson_v_rf/plots/ -m
    tree_metadata.txt
```

We can view the resulting Procrustes plot here

NOTE: The above link is not static: to view the plot, you must run the notebook.