Applying Cortex to Phase 2 1000Genomes data - the recipe

Zamin Iqbal (zam@well.ox.ac.uk)

12 October 2012 - version 10

Contents

1 Overview

This is the specification for how to apply Cortex to the 1000 Genomes populations; in other words this gives the nitty gritty details on how to build and error clean assembled graphs of the various populations, and finish up with a single VCF where every sample has been genotyped at every site (including sites that were only discovered in other populations).

2 People

We are currently running this collaboration between EBI, Michigan, Stanford, Einstein and Oxford, and so it is important we all apply exactly the same methodology/pipeline to the data.

3 Goal

The goal is to produce a highly accurate set of variant calls which access deeper into the mutation spectrum than any other callset on the 1000 Genomes.

However we do not expect it to be highly sensitive, as there is low power to detect singletons and doubletons. We would not want to draw inference afterwards on the basis of patterns specific to the low end of the frequency spectrum**.** For this iteration, we're all running with k=31.

4 What has changed since version 9 of this document?

Updated to use the fact that Cortex can now handle gzipped fastq. Updated the install instructions for Cortex - much simpler now. –format option removed from Cortex (autodetects if fasta, fastq or gzipped). –max_read_len not needed when loading fastq (but is needed with –gt).

5 Running the same pipeline at multiple centres

If we are to generate comparable callsets which really are products of the same pipeline, there are a few things we need to do.

1. Please use a clean install of Cortex **v1.0.5.13**, now available at

   http://sourceforge.net/projects/cortexassembler/files/cortex_var/latest/CORTEX_release_v1.0.5.13.tgz

1. Eirher use the wrapper-scripts I provide, or use exactly the command-lines documented below (except where the doc specifies to use a cleaning threshold dependent on your data). For example, in the genotyping step, I use a genome length of 3 billion. Please don't "fix" this and use a more accurate estimate of the human genome length - it's more important all of us have consistently calibrated genotype confidences.

2. A simple test case is to take a "pseudopopulation" of 5 samples, specifically NA19449 (6×), NA19436 (5×), NA19316 (9×), NA19443 (13×), NA19394 (6×), and run this population through the pipeline. At the end we should all get identical VCFs. I'm running the pipeline myself.

6 Pipeline Outline

The pipeline is as outlined in this section**. I have provided scripts to run all of these steps now**, available in the Cortex release under scripts/1000genomes

Each centre is allocated a set of populations to own. Then, for each population

1. **Build a binary graph file for each sample** (one command-line instruction). This requires ~20-50 GB of RAM (afraid it varies depending on how "dirty" the data is, 30Gb of disk, ~3 hours per sample. At Oxford I ran this on multiple processors on a single 512Gb RAM machine, but you could accelerate it further if you have a cluster with nodes with enough memory. Exact memory requirement per sample depends on depth of coverage; a high coverage (Illumina) sample would top out around 70GB of RAM.

2. **Error cleaning of the samples** is done by comparison with a pooled graph of all the samples in this population. This is the bottleneck as far as memory is concerned. All of the sample graphs are merged into one pool, which is then error-cleaned, and dumped as a file. For a high diversity (or low quality sequence) population, this would squeeze up against a 256Gb RAM limit. A pragmatic approach is to split the population into two halves, pool and error-clean these, and then merge those cleaned pools. This will halve the memory requirement and avoid going just over the 256 GB RAM limit, although it would mean we could lose some doubletons which had one sample in each half. This is what I did with the Luhya, as 40 samples contained 15 billion kmers (I was using k=55, so needed almost double the memory we need for this pipeline). At the end of this we have a clean pool graph. There is then a lower memory (64-72GB RAM) step, where each sample's uncleaned graph file is compared with the pool, cleaned, and then dumped as a cleaned graph. These cleaned graphs are a final product, which will be distributed to the 1000 Genomes project (and public).

3. **Site discovery** is now done on the pool. Memory requirement is 64-72Gb RAM, produces a callfile ~10Mb in size, and takes about 6-10 hours.

4. **Genotyping** requires a multicolour graph of all ~80 samples in the population. Although 80 whole genomes requires too much memory with the standard Cortex release (forthcoming release fixes this, but I don't want to use brand new code on a big production like this, I want heavily tested code), but 80 people's polymorphisms don't require that much space. Therefore we now cut out from each sample graph the polymorphisms that have been discovered, and dump that as a new binary graph file. Memory requirement is ~14Gb, so could be parallelised if you want.

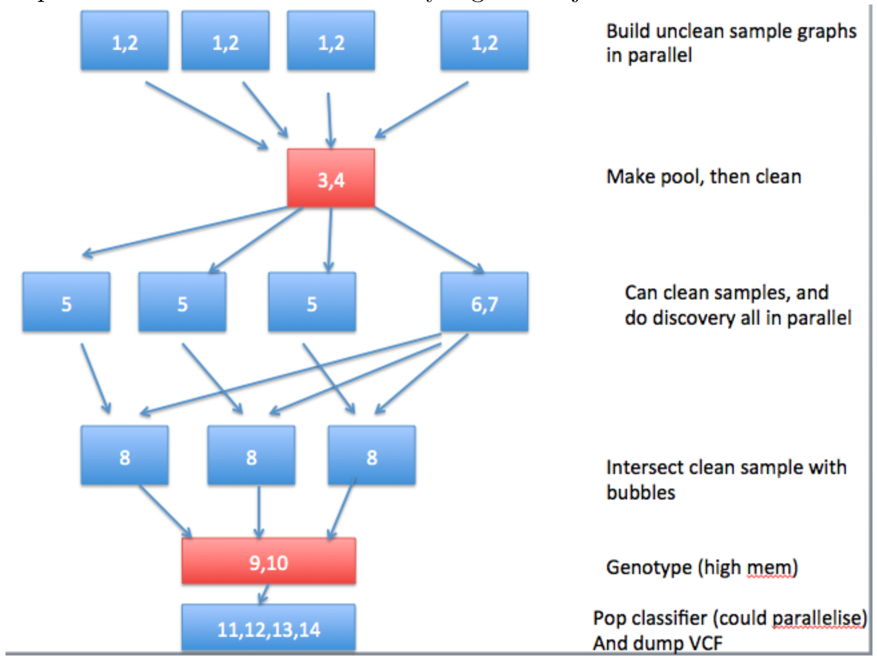
5. Load all these small graphs into one multicolour graph, and genotype everyone. Mem requirement:~256Gb of RAM. This depends on the level of diversity of the population. The Luhya would require ~260 for example. It would make sense to ensure the high diversity populations are done somewhere where there are servers with >256Gb of RAM, to be safest (eg I know the EBI have servers like this available).

6. Parse the output and **dump a VCF**. This will (obviously) be a VCF of calls made in this population, which will only really be useful for us to assess intermediate calls. Our final VCF would include all sites called in all populations

Once all the Phase 2 populations are complete, we make a union of all the calls across all populations, and then repeat steps 4,5,6 on each population using this new callset.

7 For the impatient - here's the pipeline in commandlines (using my scripts)

Note that all the arguments for all of these command-lines are predetermined - ie we know them now - except for one, the cleaning threshold T (see step 4 below). XYZ is the population identifier. This flow diagram summarises the job dependencies and order. Particularly high mem jobs are in red.


1,2
1,2
1,2
1,2
Build unclean sample graphs in parallel
3,4
Make pool, then clean
5
5
5
6,7
Can clean samples, and do discovery all in parallel
8
8
8
Intersect clean sample with bubbles
9,10
Genotype (high mem)
11,12,13,14
Pop classifier (could parallelise)
And dump VCF

1. perl script1_make_filelist_from_sequence_index.pl –sample NAxxxx (one process per sample)

2. perl script2_build_uncleaned_sample_graph.pl –sample NAxxxx (one process per sample

3. perl script3_build_unclean_pool.pl –list LIST –outdir OUTDIR –pop XYZ –mem_height 28 –mem_width 100 (just one process only, but this is the highest memory step, Ideally, alloc 430Gb RAM. On a 256 Gb RAM server, you will have to split into two pools) Once this is done, you can look at the output file, call Zam, and he will determine the threshold T, to be used below. This is the only point where a human needs to intervene.

4. cortex_var_31_c1 –kmer_size 31 –mem_height 28 –mem_width 100 –multicolour_bin XYZ_uncleaned_pool.q10.k31.ctx –remove_low_coverage_supernodes T –dump_binary XYZ_cleaned_pool.threshT.k31.ctx >& output_clean_pool (this also requires high memory, but is independent of script4 (could be run at the same time)

5. perl script4_clean_samples_against_clean_pool.pl –pop LWK –list list_uncleaned_binaries –cleaned_pool XYZ_cleaned_pool.threshT.k31.ctx –thresh T (Requires 125Gb RAM irrespective of how many samples you want to clean. This will clean each sample sequentially. Can be parallelised (pass it a smaller list, and run several instances)

6. perl script5_discovery.pl –ref /path/to/human_g1k_v37.proper_chroms.k31.ctx –pop XYZ –cleaned_pool XYZ_cleaned_pool.threshT.k31.ctx –thresh T –outdir XYZ_variant_calls/ (this is alsi independent of script4, but does need step6 above)

7. perl script6_dump_a_single_graph_of_bubble_branches_to_be_shared_later_by_parallel_processes.pl –bubble_calls XYZ_variant_calls/XYZ_bubbles_threshT –ref human_g1k_v37.proper_chroms.k31.ctx –thresh T (this is a very low memory and quick job)

8. perl script7_intersect_cleaned_sample_with_bubbles_PAR.pl –bubble_graph XYZ_bubbles_threshT.branches.k31.ctx –sample NA19316 –sample_cleaned_graph NAxxxx.uncleaned.q10.k31.cleanT.ctx –thresh T (this command dumps a small binary for one sample. Run many of these in parallel - will finish fast (<1 hour?)

9. cortex_var_63_c100 –colour_list ref_then_XYZ_clean_samples_overlapping_bubbles –kmer_size 31 –mem_height 23 –mem_width 100 –dump_binary multicol_ref_then_XYZ_samples_overlap_bubbles.k31.ctx (high memory step also (replace 100 with the number of samples+1), between 350 and 440 Gb RAM depending on how many samples).

10. cortex_var_63_c100 –kmer_size 31 –mem_height 23 –mem_width 100 –multicolour_bin multicol_ref_then_XYZ_samples_overlap_bubbles.k31.ctx –max_read_len 15000 –gt XYZ_bubbles_threshT,XYZ_bubbles_threshT.genotyped,BC –genome_size 3000000000 –experiment_type EachColourADiploidSampleExceptTheRefColour –print_colour_coverages –estimated_error_rate 0.01 –ref_colour 0 >& genotyping.log (do the genotyping)

11. perl scripts/analyse_variants/make_read_len_and_total_seq_table.pl genotyping.log >& genotyping.log.table (takes <1second)

12. perl scripts/analyse_variants/make_covg_file.plXYZ_bubbles_threshT.genotyped NUM_SAMPLES 0 (takes ~1 hour?)

13. cat classifier.parallel.ploidy_aware.R | R –vanilla –args 1 5000027 XYZ_bubbles_threshT.genotyped.covg_for_classifier 5000027 81 1 genotyping.log.table 3000000000 31 2 XYZ_bubbles_threshT.genotyped.classified (Run the population filter - takes ~24 hours, but you can parallelise into as many chunks as you like - see below - inp principle can be done in minutes)

14. perl process_calls.pl –callfile bubbles.genotyped –callfile_log genotyping.log –outvcf populationname.bubbles –outdir OUTPTU_DIRECTORY –samplename_list <file, lists names of each colour/sample in the same order they are in the colours> –num_cols number_of_colours –stampy_hash /path/to/ref.stampy –vcftools_dir /path/to/vcftools –caller BC –kmer 31 –stampy_bin /path/to/stampy.py –refcol 0 –pop_classifier POP_FILTER_OUTPUT –ploidy 2 –ref_fasta grc37.fa_without_any_decoy_stuff

8 Compiling and installing the latest release of Cortex

Use a clean install of Cortex version 1.0.5.13. Follow the instructions in the INSTALL file. In essence, you need to

1. Run a bash script, "bash install.sh", which compiles all the auxiliary libraries (only needs to be done once)

2. Compile Cortex itself

Then there are a few things you need to do for the VCF-dumping tools

1. Obtain Stampy from http://www.well.ox.ac.uk/project-stampy. Unzip it somewhere, cd into it and type make. Stampy needs Python version 2.6 or 2.7. Only supports x86_64 and (experimental) Mac. The process_calls.pl script for dumping VCF will need as an argument the path to your Stampy.py.

2. Download this version of VCFTools https://sourceforge.net/projects/vcftools/files/vcftools_0.1.9.tar.gz

9 Details of each of the above pipeline steps

In the following steps, I always use XYZ to refer to the population abbreviation (eg LWK for Luhya).

9.1 Files you will need to download

The following files are all available from here:

ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/working/20120814_cortex_resources/

1. A standard reference genome binary human_g1k_v37.proper_chroms.k31.ctx
2. A Stampy hash of this reference binary, this is two files: human_g1k_v37.sthash, and human_g1k_v37.stidx
3. A common installation of Stampy to avoid version nonsense: stampy-1.0.13.tgz

9.2 Making a list of samples

Parse this 1000 Genomes sequence.index file:

ftp.1000genomes.ebi.ac.uk:/vol1/ftp/sequence_indices/20120522.sequence.index

Pull out those lines with your population identifier in column 10 (counting from 0 - has POPULATION as header). Exclude lines with 1 in column 20 (WITHDRAWN).

Script provided: I have provided a sample script to do just this: it reads the sequence index, and given a sample-id, gets the appropriate files from the index, checks they are present on the filesystem (unzipping them if necessary) and makes the filelists in the format Cortex expects them. Usage

```
perl script1_make_filelist_from_sequence_index.pl –sample NA19316
```

9.3 File naming convention

Please can we all use the same convention for file-naming. Use .ctx at the end of a filename to signify a Cortex binary graph file. I suggest, for uncleaned binaries, a format: NAxxxx.k31.uncleaned.q10.ctx, and for cleaned NAxxxx.k31.q10.cleanedT.ctx, where T is the threshold used for cleaning the **pool (**see below**),** and 5 is the quality score threshold used**.**

9.4 Building per-sample uncleaned graphs

The command-line is

```
cortex_var_31_c1 –sample_id NAxxxx –kmer_size 31 –mem_height 25 –mem_width 150 –se_list FILE_SE –pe_list FILE_PE1,FILE_PE2 –quality_score_threshold 10 –dump_binary sample_id.uncleaned.q10.k31.ctx –remove_pcd_duplicates >& logfile_sampleid_build_uncleaned_q10
```

FILE_SE1 lists all the single ended **unzipped** fastq. FILE_PE1 lists all the _1 paired end unzipped fastq, etc. **Note this change from the last version of this document: I have added a –sample_id argument, so the sample identifier is stored in the binary**.

Script provided: I've provided a script that does this - **script2_build_uncleaned_sample_graph.pl**. It takes the sample_id as its only argument - eg

```
perl script2_build_uncleaned_sample_graph.pl –sample NA12878
```

(but has hardcoded inside it the path to the root directory where you want all of this stuff to happen), so you can parallelise many instances of this by just calling this with different sample_id's as arguments. Each instance uses 75GB RAM (can be modified). Time taken to build a graph for one sample depends on speed of access to your disk, but could be anywhere between 2 and 6 hours.

9.5 Building a pool and error-cleaning it

We make a colourlist - a list containing one **tab-separated line with two fields. First, the name of a file listing all the uncleaned binaries. Second, the text that will go into the "sample-id" metadata in the header of this binary. In this case, use XYZ_pool (eg for the Luhya I would use LWK_pool). This just means that later on, if someone loads this graph, they will know**

```
>cat colourlist_for_merge
list_all_uncleaned_binaries XYZ_Pool
>cat list_all_uncleaned_binaries
sample_1.uncleaned.q10.k31.ctx
sample_2.uncleaned.q10.k31.ctx
sample_3.uncleaned.q10.k31.ctx
sample_4.uncleaned.q10.k31.ctx
..
sample_80.uncleaned.q10.k31.ctx
>cortex_var_31_c1 –kmer_size 31 –mem_height 27 –mem_width 110 –
colour_list colourlist_for_merge –dump_binary XYZ_uncleaned_pool.q10.k31.ctx
–dump_covg_distribution XYZ_uncleaned_pool.covg.txt >& output_merge_uncleaned
```

The mem height/width arguments I specified above will allow you to go up to 14.7 billion kmers, which is as many as you can fit on a 256Gb RAM server (a cleaned population should have 3-5 billion (eg Luhya is 3.5 billion) , my UNcleaned LWK had to be split into two pools of 15 billion (obviously a lot of overlap, the total is not 30 billion) - so a 256GB server would not have sufficed). If you have a server with at least 512Gb RAM, the **preferred option if just use –mem_height 28 –mem_width 100** , which will support 27 billion kmers, **and use 420Gb of RAM**. This is overkill, but it pretty much guarantees success, and you don't need to split the pool into two etc (see next section).

This is a slow process, which may take up to 48 hours, depending on the load on your disk, and connection between disk and server. We've just added a significant speed improvement to Cortex though, so I'm not sure how long it will be. If CPU is the bottleneck, our performance boost will help, but if IO is the bottleneck, then it won't - depends on your compute infrastructure/usage at the time.

Script provided: I've provided ascript: **script3_build_unclean_pool.pl,**

```
perl script3_build_unclean_pool.pl –list LIST –outdir OUTDIR –pop XYZ
–mem_height 28 –mem_width 100
```

where LIST is a filelist of the uncleaned sample binaries, the OUTDIR is wherever you want to put the merged population binary. The script is very simple, but it will complain if you try to allocated too little memory.

Error Modes: there are two ways in which this script can fail

1. Your server does not have enough memory to give you as much as you have asked for, so right at the start (immediately), it will fail to allocate the memory it needs. You will get a message saying "Giving up - unable to allocate memory for the hash table". If you use script3, that error will be in this file OUTDIR/XYZ_merged_uncleaned_samples.k31.q10.ctx.log

2. Your server gives you as much memory as you have asked for, but it turns out you have asked for too little, and your sequencing data won't fit. You will get a message saying "Dear user - you have not allocated enough memory to contain your sequence data.". If you use script3, that error will be in this file: OUTDIR/XYZ_merged_uncleaned_samples.k31.q10.ctx.log

9.5.1 In case you completely run out of memory

If at all possible on your server, increase mem_height and width in order to be able to merge all the samples into one pool. If that is impossible, then follow the instructions in this section.

If this fails because even the full capacity of a 256Gb RAM server is not enough, then split the samples into two lists, and do this:

```
>cat colourlist_for_merge_first_half
list_first_half_uncleaned_binaries
>cat list_first_half_uncleaned_binaries
sample_1.uncleaned.q10.k31.ctx
sample_2.uncleaned.q10.k31.ctx
sample_3.uncleaned.q10.k31.ctx
sample_4.uncleaned.q10.k31.ctx
..
sample_40.uncleaned.q10.k31.ctx
>cat colourlist_for_merge_second_half
list_second_half_uncleaned_binaries
>cat list_second_half_uncleaned_binaries
sample_41.uncleaned.q10.k31.ctx
sample_42.uncleaned.q10.k31.ctx
sample_43.uncleaned.q10.k31.ctx
...
sample_80.uncleaned.q10.k31.ctx
>cortex_var_31_c1 –kmer_size 31 –mem_height 27 –mem_width 110 –
colour_list colourlist_for_merge_first_half –dump_binary XYZ_first_half_pool.clean1.q10.k31.ctx
–remove_low_coverage_supernodes 1 >& output_merge_first_half
>cortex_var_31_c1 –kmer_size 31 –mem_height 27 –mem_width 110 –
colour_list colourlist_for_merge_second_half –dump_binary XYZ_second_half_pool.clean1.q10.k31.ctx
–remove_low_coverage_supernodes 1 >& output_merge_second_half
```

ie break the pool in two, and do the absolute minimal cleaning, and then merge. So first make a new colourlist:

```
>cat colourlist_for_merging_subpools
list_both_subpools
>cat list_both_subpools
XYZ_first_half_pool.clean1.q10.k31.ctx
XYZ_second_half_pool.clean1.q10.k31.ctx
```

and then merge the two halves:

```
>cortex_var_31_c1 –kmer_size 31 –mem_height 27 –mem_width 110 –
colour_list colourlist_for_merging_subpools –dump_binary XYZ_uncleaned_pool.q10.k31.ctx
–dump_covg_distribution XYZ_uncleaned_pool.covg.txt –remove_low_coverage_supernodes
1 >& output_merge_first_half
```

If we can possibly avoid doing this, it is best to, as it will kill low frequency variants which are split between the two pools.

Now we are ready to choose the proper cleaning threshold for the whole pool.

9.5.2 Choosing the error cleaning threshold:

This point of the process does require a human decision - please contact me with the coverage distribution of the uncleaned pool to confirm the right threshold.

Here's how the choice is made:

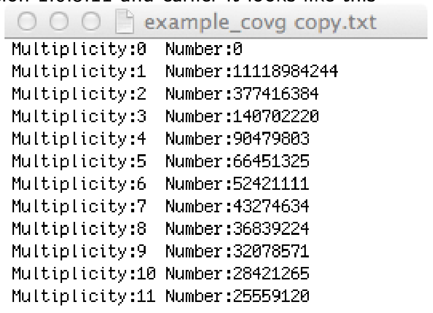
If we have a 1% error rate, and depth of coverage D (for example when I built the Luhya, I had 85 individuals at 6x=510x), then our rough expectation is 0.01D errors at a monomorphic site. ie 5.1 errors at every site for the LWK . So the default choice would be :

$$\text{Default cleaning threshold} = \left\lceil \frac{\text{TOTAL\_DEPTH\_REPORTED\_BY\_CORTEX}}{100} \right\rceil$$

Note I say reported by Cortex - given you have used a quality filter, you effectively lose coverage. Each sample had a log file called logfile_sampleid_build_uncleaned_q10 above, and in it, under some asterisks (**********) it reports the mean read length and total sequence loaded. **Sum the total sequence loaded from all samples, and divide by 3 billion**, to get TOTAL_DEPTH_REPORTED_BY_CORTEX.

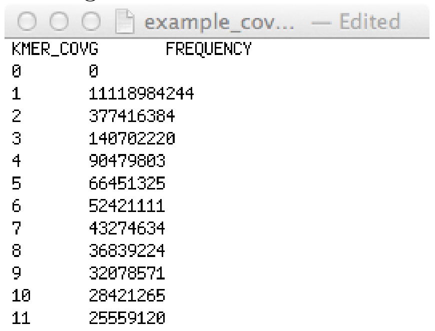
In the case of the Luhya, this is 510/100=5.1. **However that's just your prior expectation. You need to do two more things.**

First; Look at the file XYZ_uncleaned_pool.covg.tx**t** produced at the last step. With version 1.0.5.11 and earlier it looks like this



```
Multiplicity:0  Number:0
Multiplicity:1  Number:11118984244
Multiplicity:2  Number:377416384
Multiplicity:3  Number:140702220
Multiplicity:4  Number:90479803
Multiplicity:5  Number:66451325
Multiplicity:6  Number:52421111
Multiplicity:7  Number:43274634
Multiplicity:8  Number:36839224
Multiplicity:9  Number:32078571
Multiplicity:10 Number:28421265
Multiplicity:11 Number:25559120
```

but actually that confuses everyone, "multiplicity" is actually Kmer coverage, and "Number" is frequency (the number of kmers with that kmer coverage), so I'm moving to this format

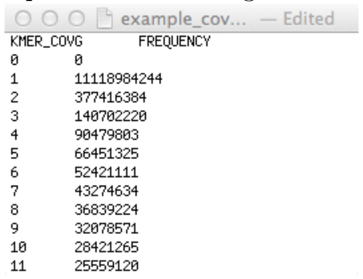


```
KMER_COVG       FREQUENCY
0       0
1       11118984244
2       377416384
3       140702220
4       90479803
5       66451325
6       52421111
7       43274634
8       36839224
9       32078571
10      28421265
11      25559120
```

Anyway, clean the file up so there is no "Multiplicity:" or "Number:" or header. Plot the first column (kmer coverage) as X axis (running from 0 to 400 or so) and the second column (frequency) as Y axis (on a log scale). eg in R the command would be

```
data<-read.table("XYZ_uncleaned_pool.covg.txt", header=FALSE)
plot(data[,1], data[,2], log="y", xlim=c(0,500), main="Kmer covg
distribution", xlab="Kmer covg", ylab="Frequency")
```

This produces something like this:

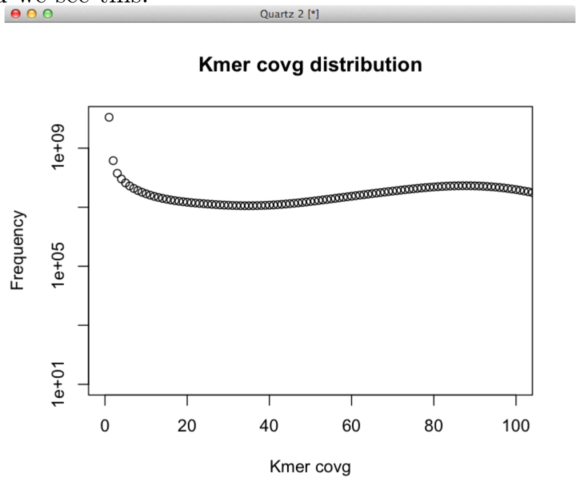


```
KMER_COVG       FREQUENCY
0       0
1       11118984244
2       377416384
3       140702220
4       90479803
5       66451325
6       52421111
7       43274634
8       36839224
9       32078571
10      28421265
11      25559120
```

Now you see a peak somewhere just under 100, and a minimum somewhere on the left. Now ideally/simplistically we would pick T=the x-coordinate of the minimum. But we don't want T too large or it will remove low frequency variants, so it is a tradeoff. This example is in fact the LWK data I used for my k=55 calls for 1000 Genomes. Zoom in to work out where the minimum is

plot(data[,1], data[,2], log="y", xlim=c(0,100), main="Kmer covg distribution", xlab="Kmer covg", ylab="Frequency")

and we see this:



OK, it's very flat at the bottom. We remember that we just want to kill the majority of the errors. Cortex is very effective at distinguishing false positives when calling in a population - we just need to clean the graph up enough that the errors don't mess up all our bubbles.

Now, we compare our default choice mentioned above, 5.1, and check it lies lies to the right of the spike at 1, and lies inside the minimum. In this case, it looks fine. I actually chose a value of T=6 for this dataset, as I wanted to be conservative.

Second: double check that our choice is not killing far too many low frequency variants. How do we do this? Well, we have 510x coverage on 85 samples=170 chromosomes. So an allele can have allele count between 0 and

170. Each copy of the allele has coverage=510/170=3x. However coverage drops in a de Bruijn graph by a factor (R-k+1)/R. For the LWK dataset I had mean read length 90bp and k=55, so this factor is (90-55+1)/90=0.4. Thus each allele copy has a coverage=0.4*3 =1.2. So if we use T=6, we remove alleles with count<=6/1.2=5. That corresponds to a frequency of 5/170=3%. That's a rough estimate (and later analysis suggests that was a bit of an underestimate)

Having made the choice, then error-clean with threshold T.

```
cortex_var_31_c1 --kmer_size 31 --mem_height 28 --mem_width 100 --multicolour_bin
XYZ_uncleaned_pool.q10.k31.ctx --remove_low_coverage_supernodes T --dump_binary
XYZ_cleaned_pool.threshT.k31.ctx >& output_clean_pool
```

Error cleaning can take a day or two, don't panic if it is running silently. If you have to tell your scheduler how long to expect it to take, tell it it will take a week (it won't, but it will really annoy you if it kills it too early).

No script provided! This is just a single command-line. **Later scripts will assume you stick to the naming convention.**

9.6 Cleaning the per-sample graphs by comparing with the cleaned pool

Make a filelist for each sample; in general (outside this Phase 2 plan) this would list all the binaries that correspond to that sample (eg different from libraries, which might require different error-cleaning), but in our case, we just have one binary per sample, so this filelist has one line only, giving the full path to the uncleaned binary. *Please make sure there are no empty lines at the end of the filelist*, **and that there is a carriage return at the end of each line**. Then make a list, LIST of these filelists. So LIST contains one line per sample, and each line is a filelist.

Check out how many kmers were in your cleaned pool. The output from cleaning the pool should end with text like this:

```
Dump multicolour binary with 1 colours (compile-time setting)
3949783396 kmers dumped
```

In a diverse African population, you expect 4 billion. In a less diverse population you might only have 3.5 billion or less . See Appendix 1 for details. Here I use mem_height 25 and mem_width 150 to support 5 billion.

Then you clean as follows

```
cortex_var_31_c2 --kmer_size 31 --mem_height 25 --mem_width 150 --multicolour_bin
XYZ_cleaned_pool.threshT.k31.ctx --colour_list LIST --load_colours_only_where_overlap_clean_colour
0 --successively_dump_cleaned_colours cleanT
```

This will dump one cleaned binary per sample, in the same directory that it currently resides, and will add the suffix "cleanT", where T is the threshold you used for cleaning the pool.

Script provided: I've provided a script, **script4_clean_samples_against_clean_pool.pl**, that will run this. All you need to do is give it a list of all the sample uncleaned binaries, and the cleaned-pool-binary, and it will make all the filelists Cortex needs and run the error cleaning. It will default to the mem height/width arguments in the command-above unless you tell it otherwise. So you run it like this

```
perl script4_clean_samples_against_clean_pool.pl --pop LWK --list list_uncleaned_binaries
--cleaned_pool XYZ_cleaned_pool.threshT.k31.ctx --thresh T
```

This will require about 125Gb of RAM. If you have multiple high RAM servers, you can parallelise this - instead of having one long list of all the uncleaned binaries, split the list up. eg if you have a 512Gb RAM machine, you could do 4 instances in parallel - so split the list into 4 and run 4 instances in parallel

```
perl script4_clean_samples_against_clean_pool.pl –pop LWK –list list_uncleaned_binaries_1
–cleaned_pool XYZ_cleaned_pool.threshT.k31.ctx –thresh T
perl script4_clean_samples_against_clean_pool.pl –pop LWK –list list_uncleaned_binaries_2
–cleaned_pool XYZ_cleaned_pool.threshT.k31.ctx –thresh T
perl script4_clean_samples_against_clean_pool.pl –pop LWK –list list_uncleaned_binaries_3
–cleaned_pool XYZ_cleaned_pool.threshT.k31.ctx –thresh T
perl script4_clean_samples_against_clean_pool.pl –pop LWK –list list_uncleaned_binaries_4
–cleaned_pool XYZ_cleaned_pool.threshT.k31.ctx –thresh T
```

9.7 Site discovery

Straightforward, no decisions to make. We include the reference genome purely for annotation purposes. To ensure we all use the same one, **I have provided a pre-built reference binary human_g1k_v37.proper_chroms.k31.ctx** (built from the official 1000g reference, excluding all the decoy stuff), which I have sent to the DCC (Holly and Laura).

So we need to make 3 filelists - filelist_ref_binary, filelist_clean_pool, and colour_list, such that:

```
>cat discovery_colour_list
filelist_ref_binary
filelist_clean_pool
>cat ref_binary
/path/to/grc37.ctx
>cat filelist_clean_pool
/path/to/XYZ_cleaned_pool.threshT.k31.ctx
cortex_var_31_c2 –kmer_size 31 –mem_height 25 –mem_width 150 –colour_list
discovery_colour_list –detect_bubbles1 1/1 –output_bubbles1 XYZ_bubbles_threshT
–exclude_ref_bubbles –ref_colour 0 >& XYZ_bubbles_threshT.log
```

This will require about 120GB of RAM (mem_height and width are no bigger, ut this time there are two colours). Note I added _threshT to the name of the output file. That's just future-proofing, in case we decide/have time to clean at another threshold and re-call. And note that the reference genome colour is EXCLUDED from discovery, discovery is done entirely in colour 1.

Script provided: I have provided a script to do this, which you run as follows:

```
perl script5_discovery.pl –ref /path/to/human_g1k_v37.proper_chroms.k31.ctx
–pop XYZ –cleaned_pool XYZ_cleaned_pool.threshT.k31.ctx –thresh T –outdir
XYZ_variant_calls/
```

This will detect bubbles and dump a file called XYZ_variant_calls/XYZ_bubbles_threshT (which you will need to pass as an argument into subsequent scripts/steps).

NOTE: this only needs the cleaned pool to call, so it can run in parallel while you simultaneously clean the samples. i.e. script5 and script4 can run at the same time.

9.8 Genotyping

Normally we just load all the samples into a multicolour graph and genotype. Since we want to deal with a whole population of humans, we have to employ a trick to reduce the RAM requirements by only looking at polymorphisms. Having done bubble discovery in the cleaned pool graph, we cut out the small part of each sample graph that overlaps these bubbles, creating a small binary file (~2GB), and then make a multicolour graph from those for genotyping with. For the Luhya this dropped the number of kmers down to 700 million, from 5 billion (i.e. we can drop mem_height and width).

9.8.1 First make a small binary file, just of the bubbles (no sample information, just telling us what the bubbles are)

The commandline is

```
perl scripts/analyse_variants/make_branch_fasta.pl –callfile XYZ_bubbles_threshT
–kmer 31
```

This creates a fasta file called XYZ_bubbles_threshT.branches.fasta - this just adds k+1 bases from the 5-prime flank on the front of each of the two branches, so you get two reads for each call. Then make a binary file of this (should take a few minutes)

```
ls XYZ_bubbles_threshT.branches.fasta > list_XYZ_bubbles
cortex_var_31_c1 –se_list list_XYZ_bubbles –mem_height 23 –mem_width 100 –max_read_length
15000 –format FASTA –dump_binary XYZ_bubbles.k31.ctx >& output_dump_bubble_binary
```

This leaves us with a binary file which tells us what the sequence of the bubbles is: XYZ_bubbles.k31.ctx.

Script provided: I've provided a tiny script to do this script6_dump_a_single_graph_of_bubble_branches_to_be_shared

```
perl script6_dump_a_single_graph_of_bubble_branches_to_be_shared_later_by_parallel_processes.pl
–bubble_calls XYZ_variant_calls/XYZ_bubbles_threshT –ref human_g1k_v37.proper_chroms.k31.ctx
–thresh T
```

This will dump a bubble binary called XYZ_variant_calls/XYZ_bubbles_threshT.branches.k31.ctx, **AND** it will dump a binary of the reference genome intersected with the bubbles

9.8.2 Then dump a bubbles binary for each sample (overlap of cleaned binary and bubbles)

Make a colour list for the CLEANED sample binaries:

```
>cat colour_list_samples_cleaned
sample_1_list
sample_2_list
...
>cat sample_1_list
sample_1_clean.ctx
>cat sample_2_list
sample_2_clean.ctx
```

And then we dump the intersection of each cleaned sample graph, with the bubbles. This is equivalent to what we did before when we cleaned each sample against the cleaned pool. Here we "clean" each of the sample binaries against the bubbles graph, and dump the overlap/intersection:

```
cortex_var_63_c2 –kmer_size 31 –mem_height 23 –mem_width 100 –multicolour_bin
XYZ_bubbles.k31.ctx –colour_list colour_list_sample_bubbles –load_colours_only_where_overlap_clean_colour
0 –successively_dump_cleaned_colours OVERLAP_BUBBLES >&output_overlapping_samples_with_bubbles
```

This will first of all load the bubbles graph into colour 0, and then it will go through the colour_list, and for each line (sample), it will load that sample into the second colour in the graph (colour 1), only taking the intersection with the bubbles graph that is already there, then dump a single-colour graph of the intersection with filename sample_N_list_OVERLAP_BUBBLES.ctx , then wipe that colour clean and move onto the next sample. It is **important** that you use cortex_var_31**_c2** for this.

Should take about 1 hour per individual. **If you want, you can parallelise this step** (give each node a colour list containing only one sample). Time taken =~ 1 hour if one process per sample, or ~80 hours for 80 samples on one core.

Script provided:

```
perl script7_intersect_cleaned_sample_with_bubbles_PAR.pl –bubble_graph
XYZ_variant_calls/XYZ_bubbles_threshT.branches.k31.ctx –sample NA19316
–sample_cleaned_graph NA19316/NA19316.uncleaned.q10.k31.cleanT.ctx –thresh
T
```

This leaves an intersected (small) graph in the same directory as the original cleaned binary.

9.8.3 Same for the reference - overlap the reference genome with the bubbles

```
>cat colourlist_ref
list_grc37_ref_binary
>cat list_grc37_ref_binary
grc37_ref_binary.ctx
```

cortex_var_63_c2 –multicolour_bin XYZ_bubbles.k31.ctx –kmer_size 31 –mem_height 23 –mem_width 100 –load_colours_only_where_overlap_clean_colour 0 –successively_dump_cleaned_colours OVERLAP_BUBBLES –colour_list colourlist_ref >& output_dump_overlap_of_ref_with_bubbles

Script provided: already done by script 6 above

9.8.4 Then make a multicolour graph of all samples at all sites/bubbles

Make a colourlist (called pop_ref_then_samples_overlapping_bubbles) as normal:

```
>cat ref_then_XYZ_clean_samples_overlapping_bubbles
list_grc37_ref_binary_overlap_bubbles
list_sample1_overlap_bubbles
list_sample2_overlap_bubbles
...
list_sample_80_overlap_bubbles
>cat list_grc37_ref_binary_overlap_bubbles
grc37_ref_binary.ctx.OVERLAP_BUBBLES
>cat list_sample1_overlap_bubbles
sample_1_clean.ctx.OVERLAP_BUBBLES
..etc..
```

Supposing we have 80 samples then the commandline is:

```
cortex_var_63_c81 –colour_list ref_then_XYZ_clean_samples_overlapping_bubbles
–kmer_size 31 –mem_height 23 –mem_width 100 –dump_binary multicol_ref_then_XYZ_samples_overlap_bubbles.k31.ctx
```

This should take approximately 1 hour.

No script provided

9.8.5 Run the genotyping

```
cortex_var_63_c81 –kmer_size 31 –mem_height 23 –mem_width 100 –multicolour_bin
multicol_ref_then_XYZ_samples_overlap_bubbles.k31.ctx –max_read_len 15000
–gt XYZ_bubbles_threshT,XYZ_bubbles_threshT.genotyped,BC –genome_size 3000000000
–experiment_type EachColourADiploidSampleExceptTheRefColour –print_colour_coverages
–estimated_error_rate 0.01 –ref_colour 0 >& genotyping.log
```

This will take of the order of 1 day, and to give you an idea, **this will use 430GB of RAM for 100 samples, or 370GB of RAM for 85 samples**.

No script provided

9.9 Apply the population filter and dump a VCF

9.9.1 Make some auxiliary files needed for running the filter

Make a table file (takes <1 second)

```
perl scripts/analyse_variants/make_read_len_and_total_seq_table.pl geno-
typing.log >& genotyping.log.table
```

Make a covg file (takes ~30 mins - produces an output file with one line per variant, so very easy to monitor progress if you are concerned). If you have 80 samples (plus one reference) then the commandline is

```
perl scripts/analyse_variants/make_covg_file.plXYZ_bubbles_threshT.genotyped
81 0
```

This creates XYZ_bubbles_threshT.genotyped.covg_for_classifier.

9.9.2 Run the population filter

Either run the population filter in one process (slow - takes 24 hours). Suppose you have 5,000,027 calls:

```
cat classifier.parallel.ploidy_aware.R | R –vanilla –args 1 5000027 XYZ_bubbles_threshT.genotyped.covg_for_classifier
5000027 81 1 genotyping.log.table 3000000000 31 2 XYZ_bubbles_threshT.genotyped.classified
```

These are what the arguments are, set up to allow parallelisation. I'm afraid this is the best UI i can manage in R:

- ### arg 1 - number of the first variant to use (if the first one is var_1, then enter 1).
- ### arg 2 - how many variants to process/classify
- ### arg 3 - input covg_for_classifier file. You should generate this in advance
- ### arg 4 - number of rows/lines in the covg_for_classifier file = number of variants overall
- ### arg 5 - number of colours in the graph. For example you use cortex_var_31_c7 to get your calls, then this argument should be 7, even if you only loaded data from one sample in
- ### arg 6 - was there a reference colour? 1 for yes and 0 for no (doesn't matter which colour it was)
- ### arg 7 - table of read lengths and covgs
- ### arg 8 - estimated genome size. (Don't panic if not exact)
- ### arg 9 - kmer size
- ### arg 10 - ploidy. 1 for haploid, 2 for diploid, no other value acceptable.
- ### arg 11 - output file name

Or run the population filter in parallel, eg in 1000 processes each containing 5000 variants. The N-th process will be (note the final argument is an output file, so must have N in its name)

```
cat classifier.parallel.ploidy_aware.R | R –vanilla –args N*5000 5000 genotypes.covg_for_classifier
5000027 81 1 genotyping.log.table 3000000000 31 2 XYZ_bubbles_threshT.genotyped.split_N.classified
```

If you do run in parallel, cat the resulting files together into one file; the first column of these files is the variant number, so please ensure you cat them in order. Suppose at the end of this you have a file called POP_FILTER_OUTPUT.

9.9.3 Dump a VCF

Build a stampy hash f the reference genome

Call process_calls.

```
perl process_calls.pl –callfile bubbles.genotyped –callfile_log genotyping.log –
outvcf populationname.bubbles –outdir OUTPTU_DIRECTORY –
samplename_list <file, lists names of each colour/sample in the same or-
der they are in the colours> –num_cols number_of_colours –
stampy_hash /path/to/ref.stampy –require_one_allele_is_ref yes –
vcftools_dir /path/to/vcftools –caller BC –kmer 31 –
stampy_bin /path/to/stampy.py –refcol 0 –
pop_classifier POP_FILTER_OUTPUT –ploidy 2 –ref_fasta grc37.fa
```

10 Appendices

10.1 How Cortex specifies memory use

Cortex specifies memory use up-front. You tell it how much to use, and that's how much it will use. If you specify too little, it won't be able to fit all your data into that space, and it will give up. Memory use is specified by the command-line parameters –mem_height H and –mem_width W. Cortex then allocated a table of height 2^H and width W. The table contains $2^H \times W$ cells, each of which represents a kmer/word. A human genome contains about 2.5 billion kmers, which corresponds to H=25, W=75. Every time you increment H by 1, you double the size of the table. So H=26, W=75 supports 5 billion kmers, which is a good estimate for the maximum number you should get in one of your samples.

Each cell in the table takes a fixed amount of memory, depending on the number of colours in the graph. When building per_sample graphs and the pool, this is just 1, but in later steps it becomes 2, and then the number of samples +1 (for the reference). So the formula is: memory use = number of cells x memory use per cell = $2^H \times W \times (8 + 5c + 1)$. So one uncleaned sample should take about 2.5 billion times (8+5+1) bytes = 40GB, but because of sequencing errors, this could go up to 70 or so. Generally you find one odd sample which has a disproportionately huge number of errors, eg because it was sequenced to double the depth for some reason.

You therefore have a pipeline choice, depending on your compute availability

1. Allocate 90GB per sample and let them run safe in the knowledge all the processes will complete cleanly, because there is plenty of memory. Downside: you have to use 80Gb per process, so you can parallelise less. This is what I did. I ran 3 parallel processes on a big ram machine, and sent 80 samples through that. It took a couple of weeks as I recall.

2. Try and squeeze the mem limit down, run through the majority of samples quickly, then go back and redo the failures. Faster, but more manual intervention required (or a fancy job tracking system)

I've given good estimates for what W and H should be in the above steps, but you may want to modify slightly. These choices make NO difference to the content of the final graph. They just affect the speed of processing.

10.2 How to monitor the output of Cortex and see how much memory is actually used/needed

Cortex output when loading fastq will include the following lines

```
Total kmers in table: 1340096415
****************************************
SUMMARY:
```

```
Colour: MeanReadLen TotalSeq
0 97 17484335969
```

This tells you it loaded all the data, and it contained 1.3 billion kmers. These went into Colour 0 of the graph, and had mean read length (after breaking reads at Ns and cutting reads at low quality bases) of 97, and in total 17Gb of sequence was parsed/loaded.

So if all your samples are being loaded up and contain only 3 billion kmers, there's no need for you to allocate 7 billion, for example.

If Cortex is unable to load the data, the traditional error message is "too much rehashing", but in the next release I'll make that more user-comprehensible.

10.3 More details on error-cleaning and loss of low frequency variants

I'm not going to explain here the error-cleaning technique Cortex uses - it's not as crude as removing low coverage words/kmer, but that's a good first approximation. So, if we remove events that are seen 4 times or less in the graph, what frequency of variants do we lose? If there

Having done this, take the coverage distribution file, and plot the data within (x-axis=column1 of the file, might as well set the limits on the x axis to be between 0 and 300, and y-axis=second column, on a log scale). You will see a plot with a sharp leak at 1 - lots of unique kmers due to errors, and another peak to the right, at the mean coverage across the genome (which is roughly $D(R-k+1)/R$, where D is the total depth in your population (eg 80 times 4x=320), R is the read length (say 90) and k=31. Thus you expect a peak roughly at 213. We need to choose a threshold T for the Cortex error cleaning, and the choice must lie between the peak at 1 and the second peak. For the Luhya, this worked out at T=6. Please email me (Zam) at this stage if you are at all hesitant or surprised at what you see