cortex_var manual

Zamin Iqbal

Wellcome Trust Centre for Human Genetics, Roosevelt Drive, Oxford, UK

Revised on 11 October 2012

Version 1.0.5.13 of this document, corresponds to release 1.0.5.13 (and later) of cortex_var

ABSTRACT

`cortex_var` is a software suite for analysis of genetic variation in individuals and populations by *de novo* assembly, which offers (amongst other things) the following functionality:

- joint assembly of multiple samples/genomes (or references) using multicoloured de Bruijn graphs
- low memory use (at k=31, 10 human genomes in under 256Gb of RAM, 1000 yeasts in under 64Gb RAM)
- high quality variant calls from species with no reference genome assembly
- variant call quality improved by integration of information across samples - accurate classification of putative variants as polymorphism, repeat or error.
- calling indels, complex combinations of SNPs, indels and rearrangements (see our paper for nucleotide resolution validation with fully sequenced fosmids of entire alleles (not just breakpoint junctions)
- able to call phased haplotypes, consisting of groups of variants, potentially longer than either read-length or insert-size.
- power/sensitivity of variant calls predicted by simple mathematical model, validated both by simulations and with empirical data - enables the user to tailor experimental design to their needs. See our paper (Supplementary Material)
- supports arbitrarily large k-mer
- simple to parallelise on a cluster - vertebrate genomes can be assembled in less than a day.
- having built and cleaned a graph, can be dumped to a binary file for fast reloading
- reference-free calling of variants between species/strains
- alignment of a reference (or reads) to a graph, either to call variants, or to observe support/coverage in different samples/populations.
- speed, predictability and stability - memory use specified at the start
- Wrapper script `run_calls.pl` provides a method for a user to run an entire analysis in a single step, from fastq file all the way to VCF file.

See our paper (Z. Iqbal, M Caccamo, I Turner, P Flicek, G McVean, De novo assembly and genotyping of variants using colored de Bruijn graphs, (Nature Genetics)) for - description of our methods and model, simulations, a set of SNP, indel and structural variant calls on a HapMap/1000 Genomes human sample (NA12878); an assembly of 164 human individuals into a population graph, and determination of allele frequencies for 3Mb of novel sequence that is not accessible by mapping-based analysis, including much genic novel sequence that is of probable functional significance; reference-free variant calls on a population of 10 chimpanzees; the first demonstration of accurate typing of HLA-B from whole genome shotgun short-read data.

1 INTRODUCTION:

This is a comparatively long and detailed manual, and I imagine 99 percent of you won't read it from one end to another. I have tried to include both details of how to run Cortex and also typical use-cases. If you are new to Cortex, I always recommend trying the examples in the demo/ directory, which are very simple and quick, but give you an idea of how things work.

If you are interested in relatively small genomes, then the simplest way to get started is to try our new run_calls.pl wrapper script, that will build graphs, error-clean them, call variants and dump VCF files all very quickly and with just a single command-line instruction. One of the consequences of what we showed in our paper, was that some variants are visible at high kmers only (because of repeat content of the genome), and some are visible only at low kmers (because they happen to have low coverage). Therefore if we want to get a maximal set of variants, we should make calls at a variety of kmers and take a union. We have therefore now introduced with run_calls, two **workflows**, the ”joint workflow”, where discovery is done in a multicolour graph of all samples, and the ”independent workflow”, where discovery is done per sample, and then all samples are genotyped at the union of all discovered sites. In both cases, you can specify a range of kmers (and cleaning thresholds) and the machinery will then do everything for you, building, cleaning, calling, making union sets etc. These workflows should make using Cortex much easier for many people.

The majority of the manual is about using Cortex itself directly. If anything is unclear, please take a look at our Google group (groups.google.com/group/cortex_var), and/or contact me.

2 QUICK START FOR THE IMPATIENT

2.1 Using the run_calls wrapper

”I haven't got time to read all this, I have twenty samples (named after presidents) and and I have a rough reference genome assembly. I want SNP and indel calls which are not affected by all the assembly errors and artefacts. Just tell me how to detect all the polymorphisms between these samples, so I can discover what mutations make a president!"

- Install Cortex (see below)
- Build a tab separated "index file", one line per sample, detailing where your single and paired-end fastq are. Columns are sample_id, file **list** of single-ended files, file list of "left" paired-end files, file list of "right" paired-end files. If you have no single-ended data, put a dot, ".", in that field, and if you have no paired-end data, put a dot, ".", in those two fields. The lists can contain uncompressed or gripped fastq.

```
>cat INDEX
Reagan reagan_se reagan_pe1 reagan_pe2
Gorbachev gorbie_se gorbie_pe1 gorbie_pe2
...
>cat reagan_se
first_reagan_se_fastq.gz
second_reagan_se_fastq
etc
```

- Then build reference genome binaries at a couple of kmers (say 31,61). Each will take anywhere between 30 seconds (for a microbe) to 3 hours (for a large eukaryote) - this depends heavily on speed of disk/network access, and so will be faster if you use gripped fast)

```
cortex_var_31_c1 --kmer_size 31
--mem_height 17 --mem_width 100
--se_list file_listing_fasta
--max_read_len 10000
--dump_binary ref.k31.ctx --sample_id REF


cortex_var_63_c1 --kmer_size 61
--mem_height 17 --mem_width 100
--se_list file_listing_fasta
--max_read_len 10000
--dump_binary ref.k61.ctx --sample_id REF
```

- Build a Stampy hash of the genome

```
stampy.py -G refname ref.fa
stampy.py -g refname -H refname
```

- And now we can assemble variants, compare the two genomes at two different kmers, combine all the results and dump VCFs:

```
perl run_calls.pl --first_kmer 31
    --last_kmer 61
    --kmer_step 30
    --fastaq_index INDEX --auto_cleaning yes
    --bc yes --pd no
    --outdir dirname
    --outvcf NAME
    --ploidy 1
    --stampy_hash refname
    --stampy_bin /path/stampy.py
    --list_ref_fasta FILELIST
    --refbindir ref/
    --genome_size 2800000
    --max_read_len 100
    --qthresh 5
    --mem_height 17  --mem_width 100
    --vcftools_dir /path/vcftools_0.1.8a/
    --do_union yes
    --ref CoordinatesAndInCalling
    --workflow independent
    --logfile logfile log.txt
    --apply_pop_classifier
```

This will run, and coordinate everything for you, resulting in a pair of VCF files (both describing the **same** set of calls. See section 14.2 for a description of Cortex VCFs, and Section 15 for a detailed discussion of how run_calls works.

2.2 Calling Cortex directly

Read the manual! I've put a lot of time into it.

3 COMMAND-LINE OPTIONS:

```
--help
   Help screen

--colour_list LIST_OF_LISTS
  When loading binaries, this is a list of
  filelists, one per colour, each containing
  a list of binaries to go into that colour.
  Cannot be used with --se_list, or --pe_list.
  Optionally, a second (tab-separated) column
  can specify sample identifiers for each colour.

--multicolour_bin BINARY_FILENAME
  If you are loading just one binary (1 or many
  colours), this is the easiest way to do it.

--se_list FILENAME
  List of single-ended fasta/q to be loaded.

--pe_list LIST1,LIST2
  Lists of "left" and "right", or \us 1 and \us 2
  paired-end fasta/q files, assumed to be in the
  same order, with corresponding files listed
  at the same positions in the two files.
  i.e. file1_1.fq and file1_2.fq should both be
  at the same position in LIST1 and LIST2
  respectively.

--kmer_size
  Odd integer. I don't recommend using values
  below 21. Cannot be larger than your
 read-length.

--mem_width
  One of two parameters that determine memory use.
  They specify a "rectangle" of memory, into which
  you try to fit all your sequencing data.
  Takes an integer value. If in doubt try 100.
  More details below.

--mem_height
   One of two parameters that determine memory use.
   They specify a "rectangle" of memory, into which
   you try to fit all your sequencing data.
   For a microbe try 17.
   For a human try 25 or 26.
   More details below.

--fastq_offset
   Default 33, for standard fastq.
   Some fastq directly from different
   versions of Illumina
   machines require different offsets.

--sample_id
   (Only) if loading fasta/q, you can
   use this option to set the sample
   identifier. This will be saved in
   any binary file you dump.

--dump_binary FILENAME
   Dump a binary file with this
   name. If you have loaded fast/q,
   then this always goes into colour 0,
   and Cortex will {\bf always} dump
   a single colour graph. If you have
   loaded binaries, this will dump a
   C coloured graph where C is the
   number of colours you compiled
   for. It's the same C in the executable
   name: e.g. cortex_var_31_C5 means
   it supports 5 colours.

--max_read_len
   Since version 1.0.5.12, there is no
   need to specify max_read_len
   when loading sequence data
   (FASTQ or FASTA). However
   it is still needed when using --gt
   to genotype a set of calls.


--quality_score_threshold INT
   Filter for quality scores in the
   input file (default 0).

--remove_pcr_duplicates
   Removes PCR duplicate reads by
   ignoring read pairs if both
   reads start at the same k-mer as a
   previous read.

--cut_homopolymers INT
   Breaks reads at homopolymers
   of length >= this threshold.
```

```
   (i.e. max homopolymer in filtered
    read==threshold-1, and new
    read starts after homopolymer.

--remove_low_coverage_supernodes INT
  This is the recommended way to
  remove errors. Do not use on a
   reference genome!
   Remove all supernodes where max
   coverage is <= the limit you set.

--remove_low_coverage_kmers INT
   Remove kmers with coverage
   less than or equal to  threshold.
   Not recommended, see manual
   and paper for why.

--load_colours_only_where_overlap_clean_colour INT
   Only load nodes from binary files
   in the colour-list when they overlap
   a specific colour (e.g. that contains
   a cleaned pooled graph);
   requires you to specify this particular
   colour. You must have loaded that
   colour beforehand, using
   --multicolour_bin

--successively_dump_cleaned_colours SUFFIX
  Used to allow error-correction of
  low-coverage data on large numbers
  of individuals with large genomes.
  Only to be used when also using
  --load_colours_only_where_overlap_clean_colour
  and --multicolour_bin.

--dump_covg_distribution FILENAME
   Print k-mer coverage distribution
   to the file specified

--dump_filtered_readlen_distribution FILENAME
  Dump to file the distribution of
   "effective" read lengths after
   quality/homopolymer/
   PCR dup filters.

--output_supernodes FILENAME
   Dump a fasta file of all the supernodes.

--max_var_len INT
  Maximum variant size searched
  for. Default 10kb. This is interpreted
  internally as an upper bound
  on how big super nodes can get.
  For relatively unrepetitive
  genomes (e.g. many microbes)
  it is not uncommon to get super nodes
  which are 10 or 20kb long.
```

```
--detect_bubbles1 arg1/arg2
  arg1 and arg2 are comma-separated
  lists of colours (numbers from 0
  to C-1).
  Find all the bubbles in the graph
  where the two branches/alleles
  lie in the specified colours.
  Typical use would be
  --detect_bubbles1 1/1
  to find hets in colour 1,
  or --detect_bubbles1 0/1
  to find bubbles where one branch
  is in colour 0 (and not colour1)
  and the other branch is in
  colour1 (but not colour 0).
  However, one can do more
  complex things:
  e.g.  --detect_bubbles1 1,2,3/4,5,6
  to find bubbles where one branch
  is in 1,2 or 3 (and not 4,5 or 6)
  and the other branch in colour
  4,5 or 6 (but not 1,2, or 3).
  See below for more details.
  Use -1 to specify all colours.
  Use *3 to mean all colours except 3.
  The "1" in detect_bubbles1 is legacy
  and will be removed in future.

--output_bubbles1 FILENAME
  Bubbles called in detect_bubbles1
  are dumped to this file.

--print_colour_coverages
  Print coverages in all colours
  for supernodes and variants.
  Mandatory if you want to dump
  VCF.

--exclude_ref_bubbles
  If you have specified --ref_colour,
  this will exclude any bubble in
  that colour from being called
  by the Bubble Caller.

--path_divergence_caller [ARGS]
  Make Path Divergence variant calls.
  Arguments can be specified in 2 ways.
  Option 1. Calls once, comparing
  reference and one colour (or union).
  e.g. --path_divergence_caller 1,2
         --ref_colour 0
  will look for differences between the
  union of colours 1,2 and the
  reference in colour 0
  Option2. Make several successive
  independent runs of the PD caller,
  each time against a different colour.
  To do this, use a square open
   bracket [ PRECEDED AND
   SEPARATED list.
   For example
   --path_divergence_caller [2[3[10
   --ref_colour 0
   will make calls on samples 2
   then 3 then 10, all output to the
   same file, with globally unique
   variant names. The caller will
   call against each colour in turn.
   You must also specify
   --ref_colour and --list_ref_fasta.


--path_divergence_caller_output PATH_STUB
   PD calls will go to a file called
   PATH_STUB_pd_calls.

--ref_colour INT
   Colour of reference genome.

--list_ref_fasta FILENAME
   File listing reference chromosome
   fasta file(s). One chromosome per file.
   Needed for path-divergence calls.

----gt INPUT,OUTPUT,{BC|PD}
   Given an input  file of calls in Cortex
   format (5p, br1, br2, 3p) genotype
   all colours in the graph and output
   to specified filename. All calls must
   be either from the BubbleCaller
   or PathDivergence (not a mixture),
   and you specify this with either BC
   or PD. eg --gt infile,outfile,BC.
   You need to specify --max_read_len
   (max length of any read in the Cortex
   call file, probably a flank),
   --genome_size and
   --experiment_type to do this.

 --experiment_type
 The statistical models for determining
 genotype likelihoods, and for
 deciding if bubbles are repeat or
 variants,   require knowledge of
 whether each sample is a
 separate diploid/haploid individual.
 Enter type of experiment - valid values
 are:
 EachColourADiploidSample,
 EachColourADiploidSampleExceptTheRefColour,
 EachColourAHaploidSample,
 EachColourAHaploidSampleExceptTheRefColour.
 This is only needed for determining
 likelihoods, so ignore this is you are
 pooling samples within a colour.

 --estimated_error_rate DECIMAL
   If you have some idea of the
   sequencing error rate (per
   base-pair), enter it here. eg 0.01.
   The default value is 0.01. This is
   stored in the metadata in the header
   of the graph binary file, if you
   use --dump_binary.

 --genome_size INT
   If you specify --experiment_type,
   and therefore want to calculate genotypes,
   you must also specify the (estimated)
   genome size in bp.

--align FILENAME,{output binary name|no}
   Aligns a LIST of fasta/q files to
    the graph,
   and prints coverage of each kmer in
   each read in each colour.
   Takes two arguments. First, a LIST
    of fasta/q.
   Second, either an output filename
    (if you want it to dump a binary of
    the part of the graph touched by
    the alignment) OR just "no"
    Must also specify --align_input_format,
    and --max_read_len.

--align_input_format TYPE
    --align requires a list of fasta or fastq.
    This option specifies the input format
    as LIST_OF_FASTQ or
    LIST_OF_FASTA.

--colour_overlaps arg1/arg2
   Compares each coloured subgraph
   in the first list with all of the
   coloured subgraphs in the second list.
   Outputs a matrix to stdout;
   (i,j)-element is the number of nodes/kmers
   in both colour-i (on first list)
   and colour-j (on second list).

--genotype_site
  (Beta code, soon to be upgraded)
   Genotype a single (typically
   multiallelic) site.
   Syntax is slightly complex.
   Requires an argument of the form
   x,y[z[N[A,B[fasta[
       <CLEANED|UNCLEANED>
       [p[q[<yes|no>[MIN.
   x,y is a comma-sep list of
    colours to genotype.
   z is the reference-minus-site
   colour.
   N is the number of alleles for
   this site
   (which cortex assumes are loaded
   in a multicolour_bin containing
   those allele first, one per colour).
    Cortex will genotype combinations
    A through B of the N choose
    2 possible genotypes (allows
    parallelisation);  fasta is the file
    listing one read per allele.
    CLEANED or UNCLEANED allows
    Cortex to tailor its genotyping model.
    p,q are two free/unused colours that
    Cortex will use internally.
    yes/no specifies whether to use the
     more sophisticated error
    model, which is still in development.
    I recommend you stick with "no" for
    now. The final argument, MIN, is
     optional and allows performance
     speedup by discarding any
     genotype with log-likelihood<MIN.

--print_novel_contigs args
    Allows printing of novel sequence
    absent from a reference
    (or more generally, absent
    from a set of colours).
    Takes arguments in this format
     a,b,../c,d,../x/y/<output filename>
     Cortex will find supernodes in
     the union graph of colours a,b,..
     Typically the list c,d,.. of colours
     is just one colour  - that of
     the reference.
     Cortex will print contigs
     (supernodes) to the output
      file which satisfy the following
      criteria.
      Contigs must be at least x bp long.
      The percentage (as integer) of
      kmers in the contig which are
      present in ANY of the colours c,d,...
      must be at most 1-y.
      i.e. y is the minimum proportion
     of novel kmers in a contig.
     Typically this is 100.
     We ignore the first and last kmer
     of the contig, as these will
     typically connect to the reference
```

4 COMPILING AND INSTALLING

The clearest and briefest instructions are in the INSTALL file in the Cortex release. Here I try to explain things, rather than tell you what to do.

4.1 Run the install script - only needs to be done once

Cortex now comes bundled with the GNU Scientific Library, samtools, and two libraries from Isaac Turner. These are all compiled once only, using the bundled shell script, install.sh. You can run it as follows:

```
bash install.sh
```

4.2 Compiling Cortex itself

Cortex produces executable files which are labelled (in the filename) with what kmer sizes and numbers of colours they support. **A file named cortex_var_X_cY supports kmers between** $X - 31$ **and** X **(inclusive), and colours between** 1 **and** Y. X is always one less than a multiple of 32. So if you want to work with k=41, you need to work out the nearest multiple of 32 greater than 41 - this is 64 - and subtract 1. Your compile command is then:

```
make MAXK=63 cortex_var
```

If you want to support up to 247 colours for k=41, then the command is

```
make MAXK=63 NUM_COLS=247 cortex_var
```

Remember MAXK must be one less than a multiple of 32. Default is MAXK=31, NUM_COLS=1. Another example

```
make NUM_COLS=3 MAXK=95 cortex_var
```

- this supports k between 65 and 95, and up to 3 colours. More colours and higher MAXK both increase memory use (explained below).

We have not implemented any error-checking in the Makefile, so negative, fractional or non-numeric values of MAXK or NUM_COLS, or a MAXK which is not 1 less than a multiple of 32, will give unpredictable results.

4.3 Install/setup needed for post-processing of Cortex calls

There are two external dependencies, which you will need in order to be able to dump VCFs. In both cases my script will need you to tell it the path to these packages.

- Get Stampy from http://www.well.ox.ac.uk/project-stampy. This is needed for placing of variants on a reference. Stampy needs you to have Python 2.6 or 2.7, and only supports x86_64. Download it, unzip it, change into that directory and type ”make”
- Get a tar ball of VCFTools from https://sourceforge.net/projects/vcftools/files/ (I specifically require a tar ball, not a subversion repository, as they seem to have different directory hierarchies, and I need to be able to find things). I can verify that version 0.1.8 and 0.1.9 work fine

Add the following two directories to your PERL5LIB

```
scripts/analyse_variants/bioinf-perl/lib
scripts/calling/
```

Add the following directory to your PATH environment variable

```
scripts/analyse_variants/needleman_wunsch-0.3.0
```

In both cases give the FULL path, not just the one I showed above,

WARNING - I understand the temptation to just compile Cortex once, for 100 (or some large number) of colours, and then use that for everything, but that will increase your memory usage. Wherever possible I recommend using the minimum number of colours for the specific command you are entering.

5 FILE INPUT

cortex_var accepts the following as input

1. Fasta (gzipped or uncompressed). These will always be loaded into a single colour graph, and will be dumped as a single-colour binary (these can subsequently be mixed into whatever colours you like)
2. Fastq (gripped or uncompressed). These will always be loaded into a single colour graph, and will be dumped as a single-colour binary (these can subsequently be mixed into whatever colours you like)
3. A list of lists (in Cortex jargon, a ”colourlist”) - one list of binaries per colour
4. One multicolour binary.

We consider items 1 and 2 first, and then items 3 and 4. The release version of cortex_var does not support read-pairs (the internal development version does), but PCR duplicate removal algorithm does require knowledge of read-pairing (described below). Therefore cortex_var allows input of a list of single-ended fasta/q (--se_list), and a pair of lists for paired-end data (--pe_list filelist1,filelist2). For example:

```
> cat se_filelist
my_fastq1.fq.gz
my_fastq2.fq
> cat pe_filelist1
fastq1_1.fq
fastq2_1.fq.gz
fastq3_1.fq
> cat pe_filelist2
fastq1_2.fq
fastq2_2.fq.gz
fastq3_2.fq
> cortex_var --se_list se_filelist
  --pe_list pe_filelist1,pe_filelist2
  --mem_height  h
  --mem_width w
  --dump_binary somename.ctx
  --sample_id ZAM
```

This will dump a single-colour binary called somename.ctx (and will store the fact that this graph represents a sample called ZAM in the binary header metadata). However, with the current release of cortex_var, there is no benefit to using --pe_list unless also using --remove_pcr_duplicates.

Returning to items 3 and 4 above (loading binary files): if given both a multicolour binary, and a colourist (a list of sublists of single-colour binaries - each sublist for a different colour), then the multicolour binary is loaded first, into colours 0 to n, and then each of the sets of single-colour binaries are loaded into subsequent colours. Binary files contain a header specifying kmer, number of colours (and version), so there is also a quick check to ensure you are not trying to load more colours than the executable of cortex_var supports.

5.1 Example with a trio

Suppose we want to examine the genomes of two parents and a child, and have built single-colour binaries of each; assume that both Illumina and 454 data was available for each, requiring slightly different error-correction (see below), we build two binaries for each individual: - mum_illumina.ctx, mum_454.ctx, dad_illumina.ctx, dad_454.ctx, child_illumina.ctx and child_454.ctx. We then want to load the mother, father and child into colours 0,1,2 respectively. We have also built a binary of the reference genome ref.ctx, and want this in colour 4. All of these binaries must be built with the same kmer, k, and cortex_var must have been compiled to support at least 4 colours (make NUM_COLS=4 cortex_var, for example). We load the data as follows:

```
ls mum*.ctx > list_binaries_for_mum_colour
ls dad*.ctx > list_binaries_for_dad_colour
ls child*.ctx > list_binaries_for_child_colour
ls ref*.ctx > list_ref_binary
```

and then make a colour-list:

```
ls list* >  colour_filelist
```

Then open colour_filelist with a text editor and ensure the files are ordered mum,dad,child,ref:

```
> cat colour_filelist
list_binaries_for_mum_colour
list_binaries_for_dad_colour
list_binaries_for_child_colour
list_ref_binary
```

Now we can run:

```
cortex_var_31_c4 --kmer_size 31
 --colourlist colour_file
 --dump_binary family_an d_ref.ctx
```

This will dump a 4 colour binary, with the mother, father, child, reference in colours zero to three. If at some later date we want to compare these 3 individuals with 29 other individuals, each of whom has a single binary, indiv_n.ctx, then first we need to compile a version of cortex_var that can handle so many colours:

```
 make NUM_COLS=33 cortex_var
```

This will generate a binary cortex_var_31_c33 . We then make a new colourlist just of the new individuals, in an equivalent manner to above:

```
 ls indiv_1.ctx  >  individual_1_binarylist
 ls indiv_2.ctx  >  individual_2_binarylist
      ..
 ls indiv_29.ctx  >  individual_29_binarylist
 ls indiv*binary | sort  > list_new
```

and then run Cortex:

```
 cortex_var_31_c1
  --multicolour_bin family_plus_ref.ctx
  --colour_list list_new
  --kmer_size k
   --mem_height h --mem_width w
```

This will load the mother, father, child, reference into colours 0,1,2,3 and then individuals 1 to 29 into colours 4 to 33.

5.2 Relative and absolute paths in colourists

Absolute paths always work inside Cortex file lists. However at v1.0.5.13, we changed the convention on how Cortex interprets relative paths. **Prior** to v1.0.5.13, all relative paths were relative to the current working directory from which Cortex was being called. **From v1.0.5.13 onwards**, relative paths are interpreted as being relative to the file that lists that path. Here's an example (sorry for the cultural references), showing how you can now go to one directory containing binaries, and do ls to list them into one file, and then go somewhere else to make your colourist.

```
>pwd
/data/zam/output/binaries
>ls *.ctx
luke_skywalker.ctx
princess_leia.ctx
>ls *.ctx > list_bins
>cd /data/zam/some_other_dir
>echo /data/zam/output/binaries/list_bins
                > colour_list_vader_kin
>cortex_var_31_c1
       --colour_list colour_list_vader_kin
       --kmer_size 31..etc
```

Previously the paths in list_bins would have had to be relative to /data/zam/some_other_dir, now they are relative to whatever directory contains list_bins.

6 FILTERING INPUT DATA

6.1 Quality filter

Cortex allows reads to be filtered on-the-fly as they are loaded, by specifying --quality_score_threshold Q. Each time a read has any base with phred-scale base-quality less than or equal to Q, then the read is cut at that base. For example, if a 100 base read has a low-quality base at position 50, then this is split into two. With a kmer greater than 49, the entire read is effectively filtered, as after cutting the two remaining sequences are below the kmer length. If a 100 base read has low quality bases at positions 45, 70, 94 and 95, then with $k = 19$ the read is split into 3 chunks of sequence, each one of which contributes to the final de Bruijn graph. Some non-standard fastq use a different ASCII offset for quality - notably, some fastq as dumped by Illumina use an ASCII offset of 64 rather than the standard value of 33. Cortex allows you to specify the offset thus: --quality_offset 64; by default Cortex assumes the standard/official value of 33.

6.2 Quality filter as a memory-reduction device

Just to give you a rough idea: if you have relatively low coverage ($<$ 20x for a diploid, $<$ 10x for a haploid), then I would not use a quality threshold higher than 5. If you have much higher coverage, you can afford to raise this limit to 10, for example. A higher threshold will reduce your memory footprint, and speed up loading of data. However, I do not recommend using a high threshold of 40 as you end up throwing away too much good data - you can trust Cortex's error cleaning later on to improve your results.

6.3 PCR duplicate removal

A simple (and approximate) mechanism for removing PCR duplicate reads. As paired-end reads are loaded, the first kmers in each read are recorded (by annotating the graph). If a new read has starts with a kmer that was previously the first kmer of a read, and the mate read starts with a kmer that was previously the first kmer of a read, then both reads are discarded. PCR duplicate removal is specified by --remove_PCR_duplicates. This is an extremely fast method for duplicate removal compared with standard mechanisms requiring mapping and sorting, and we find that for some libraries removes as much as 5 percent of reads.

6.4 Homopolymer filter

Reads can be cut at homopolymers of a specified length. --cut_homopolymers will cut a read at a homopolymer longer than a specified value, starting a new read just after the homopolymer run. This can sometimes be useful with 454 data, both to reduce the number of errors in the graph, and to cut the memory usage. (In one case, with 454 data of a human, memory use was reduced by 70Gb of RAM by cutting homopolymers of length greater than 3, and the number of kmers dropped from over 7 billion to what one would expect for a human genome, around 2 billion).

7 CHOOSING HASH TABLE SIZE (I.E. SETTING YOUR MEMORY USE IN ADVANCE)

Cortex allocates memory once and for all at the start - if the available memory is not enough Cortex graciously stops with a message, rather than killing the server. The hash table can be thought of as a rectangular region of memory, and one must specify the height and width on the command-line --the area of the rectangle is the number of nodes in the largest possible graph. The units in which we measure height and width are nodes of the de Bruijn graph - i.e. the area of the rectangle is the number of nodes in the biggest supportable graph. Each node has a size that depends on the maximum kmer-size supported by the executable (specified at compile-time). A genome of size X bases will require at most X k-mers, plus a number of k-mers created by sequencing errors. The number of these depends on the quality of your data, the filters applied on loading data, and the coverage. A good initial guess might be to allocate double the number of k-mers in the genome. Choose h and w such that

$$2^h * w = \ 2 * (\text{length of genome}). \tag{1}$$

e.g. If the genome size is 2Mb, then we expect a maximum of 2 million kmers in the genome, plus a number due to sequencing errors, so we try 4 million as an overestimate. $2^{16} * 75 = $ 4.9 million. Thus we specify --mem_height=16 --mem_width=75. The memory-use M (in bytes) of a cortex_var single-colour hash table with N nodes, using an executable that supports a maximum kmer of K, can be calculated precisely, using this formula:

$$M = \lceil 8 \left\lceil \frac{K}{32} \right\rceil + 5 + 1 \rceil^{[8]} N \tag{2}$$

The formula is explained in our paper (basically it just contains contributions from storing kmer, coverage and edges); one thing we did not mention in the paper was the $\lceil\rceil^{[8]}$, which signifies that you round up everything within those brackets to the nearest multiple of 8 - this is because of OS-memory allocation preferring to give you memory in multiples of 8 bytes.

For the above example, if we create a hash table with 4.9million nodes, and k31, then memory use will be $(8 + 5 + 1) \times 4900000 = 68,600,000$. i.e 68.6 Megabytes of RAM. One final consideration is that of performance of the graph-building process - if we try to completely fill a hash table, performance will drop significantly towards the end, and so in general it is best to allocate a table slightly larger than the amount of data we expect to load. Each node in a multicolour cortex_var graph contains information about a given kmer (and its reverse complement) in multiple colours. If we have compiled cortex_var to support C colours, with a maximum kmer of K (using make NUM_COLS=C MAXK=K cortex_var), then memory use is specified thus:

$$M = \left\lceil (8 \left\lceil \frac{K}{32} \right\rceil + 5C + 1 \right\rceil^{[8]} N \tag{3}$$

For example, if we want to load sequence data for a deeply sequenced trio of humans into a graph with $k = 31$, we do the following. Firstly, we build one single colour binary for each individual. A human genome (length 3 Gigabases) should, to first approximation, contain at most 3 billion kmers . If we allow space for 3 billion sequencing errors also, then we notice that $2^{26} \times 90 \simeq 6$ billion. Since $8 + 5 + 1 = 14$, the nearest multiple of 8 above is 16 and so this should therefore require

$$\lceil 8{+}5{+}1 \rceil^{[8]} \times 6 \times 10^9 \text{ bytes} = 16 \times 6 \times 10^9 \text{bytes} = 84\text{Gb of RAM}. \tag{4}$$

In fact (for k around $20 - 50$), a human genome contains around 2.5 billion kmers (calculated by counting kmers in the human reference genome), and so after error correction the number of nodes in the graph drops to around 2.5 billion, which we dump to a binary. Finally, we now want to load 3 binaries into 3 colours in a graph that supports only 3 colours ($C = 3$). Most kmers will be shared (as the trio are from the same species), so we only need allocate around 3 billion nodes. Memory use, applying the formula, is $(8 + (5 \times 3) + 1) \times 3 \times 10^9 = 72$ Gb of RAM. Note that by judicious error-correction, we are able to load 3 humans into around the same amount of RAM as is needed for any individual prior to error correction. The precise amount of memory required depends on the quality of the sequencing data.

8 CHOOSING AN APPROPRIATE KMER

We go into considerable detail in our paper, explaining the inter-relation of read-length, depth of coverage, sequencing error rate, kmer size and genome repeat content. There are two main positions you may be in:

1. You have been given a data set to analyse. So coverage, read length (and the species) are predetermined. In that case all that remains is to ask - what do you want to achieve? Do you want a high sensitivity set of SNPs, a high specificity set of SNPs and indels, to explore the nature of larger structural variants? Is this in a single individual, or in a population? What do you know about the genome in question?
2. You have a scientific question you want to answer, and you are designing an experiment. Maybe you have a new species with no reference genome and you want to design a SNP chip, for which you need SNPs. Or maybe you suspect large insertions or deletions of begin important in your species, but noone has investigated them yet.

In the Supplementary Material of our paper we go through these issues in great detail. Here are some highlights: Larger kmers lead to greater ability to disentangle the genome graph at a cost in sensitivity. That cost in sensitivity can be offset by increasing coverage, up to a limit determined by the nature of the genome, and the kmer size. This can be quantified, thus - the power P to detect an allele of length t, given sequencing depth D and read length R , is given by

$$P = G(t,k)E(k,\varepsilon)(1-e^{-\lambda L})^2 e^{-\lambda t e^{-\lambda L}} \tag{5}$$

where $\lambda = D/R$, $L = R - k + 1$ and G is the Genome Complexity . E is the power loss due to sequencing errors and error-cleaning, dependent on k and the sequencing error rate ε. Things to notice:

1. The repeat content of the genome sets an upper bound on power, dependent on k. There are plots of G for the human genome in our paper, and we show how we estimate G.
2. As k increases towards R, L drops to zero,and therefore so does the power (because of the (1-exp)-squared)
3. The final terms give the probability, given a certain read-length and coverage, that the allele is present in the graph - in other words, that there is not a coverage gap in the middle of it. This last term basically comes down to our having determined the full probability distribution for a Lander-Waterman model on a de Bruijn graph (normally when you see the Lander-Waterman statistics, people deal only with mean and variance, not the full distribution).

We show in our paper how the model matches the results of simulations, as well as in empirical data, with analysis of read data from a high coverage human, and a population of chimpanzees. If you're applying this formula, remember that for heterozygous sites, you need to assemble two alleles, whereas for homozygous sites (where a reference genome gives you one allele) you only have to assemble one.

9 ERROR CLEANING

By error correcting and then dumping a binary just of the clean/correct nodes (and later reloading the clean binary), we reduce the number of nodes in the graph, and therefore also the memory requirement. **You only get a memory reduction from error-cleaning if you dump a binary after error-cleaning, and then load that binary**. That's the general paradigm in which you should use Cortex - don't multi colorise until after error-cleaning.

9.1 Error cleaning a single sample

cortex_var contains 2 means of error-cleaning:

1. --remove_low_coverage_supernodes N. This is the recommended option; it first removes tips and then removes supernodes where the maximum kmer coverage of all nodes in the interior of the supernode is at most N. This is described in Supplementary Methods Section 6 of our paper, and also in Supplementary Figure 3, which is well worth studying. We measured a 30% increase in discovery sensitivity compared with simple coverage cutoff for kmers, because it does not break up long contigs just because there is a brief coverage dip.
2. --remove_low_coverage_nodes. This is a simple method of error-cleaning, which can be useful when the volume of sequencing errors is such that the vast majority of nodes with low coverage are errors. However random sampling will also create nodes with low coverage, and deleting those will introduce gaps in an assembly. --remove_low_coverage_kmers N will remove all nodes with coverage $\leq N$. As we describe in our paper, we **do not recommend this method**, it's a blunt instrument and creates gaps in the assembly.

9.2 Error-cleaning low coverage samples when you have many samples from the same species/population

Standard error-cleaning methods for de Bruijn graphs all depend on having sufficiently high coverage ("things which happen rarely are more likely to be errors than real"). However recent projects, such as the 1000 Genomes Project, have pioneered a new design for sequencing experiments, where many individuals are sequenced to lower depth. cortex_var provides a means for error-correction by comparison with a population graph. The approach is:

- Build one uncleaned graph per individual.
- Merge all these graphs into one single-colour graph, and error-clean that
- Clean each individual graph by comparison with the cleaned pool - just take the intersection of the two

Here is a step-by-step example. Suppose you have 100 individuals, each sampled at low coverage, all from the same species/population:

1. Merge all of the individual binaries into one colour (use --colour_list FILE1, where FILE1 is a filelist containing just one file (FILE2), and where FILE2 is a list of

all the indiv_N.uncleaned.ctx) and error-clean using --remove_low_coverage_supernodes , and dump a cleaned population pooled graph clean_pool.ctx

2. Build a 2 colour version of Cortex, and tell it to load the cleaned pool into the first colour (colour 0), and then to load indiv_1.uncleaned.ctx into colour 1, and clean it by comparing it with the cleaned pool graph in colour 0, and then dump a cleaned individual graph, then wipe colour 1 clean, load indiv_2.uncleaned.ctx into colour 1, clean it by comparison with the pool, dump a cleaned individual graph,, wipe colour 1 clean, ... etc.
3. cortex_var_31_c2 --kmer_size 27 --mem_height h --mem_width h --multicolour_bin cleaned_pool.ctx --colour_list (list one colour, and that containing a list of all uncleaned individual binaries) --load_colours_only_where_overlap_clean_colour 0 --successively_dump_cleaned_colours SUFFIX

Each cleaned binary is dumped in the same directory as its corresponding unclean binary, with the SUFFIX added to its name to signify that it has been cleaned. The command line for the above is

```
cortex_var_31_c2 --kmer_size 27
--mem_height h --mem_width w
--multicolour_bin cleaned_pool.ctx
--colour_list COL_LIST
--load_colours_only_where_overlap_clean_colour 0
--successively_dump_cleaned_colours SUFFIX
```

10 VARIATION DISCOVERY USING THE BUBBLE CALLER

The Bubble Caller is described in detail in our paper. Essentially the idea is to look for motifs in the graph, which we call bubbles, which are created by both polymorphism and by repeats. We can build up an understanding of what this can do in stages:

- In a single-colour graph, built from sequence reads from a single diploid individual, bubbles are caused by differences between alleles, or paralogs, or sequencing errors. More generally, the same applies even in a multicolour graph, if we restrict to bubbles found in a specific colour. We do this with Cortex, supposing we are interested in colour i (for individual) - we look for bubbles in the graph where both branches/sides of the bubbles are present in colour i - here's the command-line

```
cortex_var_31_c1 --kmer_size k
--mem_height h --mem_width w
--multicolour_bin sample.ctx
--detect_bubbles1 i/i
--output_bubbles1 output_filename
--print_colour_coverages
```

- If we are lucky enough to have a reference genome for the species of interest, then we can do an approximate job of eliminating repeats by loading the reference genome into its own colour (say colour r), and ignoring bubbles that can be found in that colour. (I say this is approximate because reference genomes contain collapsed repeats - i.e. they are imperfect - see later on in this manual for the Population Filter, which does a much better job). We do this thus:

```
cortex_var_31_c100 --kmer_size 31
--mem_height h --mem_width w
--multicolour_bin sample_and_ref.ctx
--detect_bubbles1 i/i
--exclude_ref_bubbles
--ref_colour r
--output_bubbles1 output_filename
--print_colour_coverages
```

- Steps 1 and 2 above primarily find heterozygous sites, where the data from the individual (colour i) contains both alleles. (They may also found homozygous sites where sequencing errors have given ”false” coverage to the other allele - the genotyping step (see below) deals with this issue). If we have a reference genome (colour r) we can expand our discovery to allow both homozygous and heterozygous sites,**by looking for bubbles in the UNION of colours i and r**, thus:

```
cortex_var_31_c100 --kmer_size 31
--mem_height h --mem_width w
--multicolour_bin sample_and_ref.ctx
--detect_bubbles1 i,r/i,r
--exclude_ref_bubbles
--ref_colour r
--output_bubbles1 output_filename
--print_colour_coverages
```

Here, Cortex has treated colours i and r as a single colour, and looked for bubbles in that union-colour. If you are working with a diploid species, this is the right way to do it - get all the variant sites you can, and then decide if they are homozygous or heterozygous after calling, with the genotyping stage of Cortex. Cortex decides whether to genotype just after calling on the basis of whether you give it enough information. If you tell it whether the species is haploid or diploid, and you tell it the genome size (approximate is fine), then it will genotype each call as soon as it discovers it. I give details of this below.

10.1 Finding bubbles that distinguish colours

Cortex does support discovery of variants that distinguish colours, which can be very useful. For example, if you want to find variants where one allele exists in colour 1, but has ZERO coverage in colour 2, and the other allele exists in colour 2, but has ZERO coverage in colour 1, then here's the command:

```
cortex_var_31_c2 --kmer_size 31
--mem_height h --mem_width w
--multicolour_bin sample_and_ref.ctx
--detect_bubbles1 1/2
--output_bubbles1 output_filename
--print_colour_coverages
```

Use cases:

- Comparing two reference genomes

- Comparing two stringently cleaned samples
- Looking for sequence that is definitely absent from some sample

My advice is - be careful. In real data, there are sequencing errors, so you don't always see ZERO reads on an allele, even when it is not present in the sample. If your colours are stringently cleaned, then this might be fine, or if they are pooled populations where you want to find highly differentiated variants, this might be fine. If these are reference genoes, this is **definitely** fine. But think carefully about whether you are better off just calling in the union and then genotyping, to allow for sequencing errors.

Note the command-line generalises simply. Suppose you had a set of cases (colours 1,3,4,5,6,7,8,9,10) and a set of controls (colours 20 to 29). You could look for variants that split the two groups thus:

```
cortex_var_31_c30 --kmer_size 31
--mem_height h --mem_width w
--multicolour_bin samples_and_ref.ctx
--detect_bubbles1 1,3,4,5,6,7,8,9,10/20,21,22,
        23,24,25,26,27,28,29
--output_bubbles1 output_filename
--print_colour_coverages
```

This is obviously not a very good example, in a normal case-control study you don't expect to find a massive signal like that.

10.2 Tricks

You can use an asterisk to denote all colours except a specific one. The following command looks for bubbles that distinguish 0 from the other colours

```
cortex_var_31_c30 --kmer_size 31
--mem_height h --mem_width w
--multicolour_bin samples_and_ref.ctx
--detect_bubbles1 *0/0
--output_bubbles1 output_filename
--print_colour_coverages
```

whereas this command looks for bubbles in the union of all colours except 0:

```
cortex_var_31_c30 --kmer_size 31
--mem_height h --mem_width w
--multicolour_bin samples_and_ref.ctx
--detect_bubbles1 *0/*0
--output_bubbles1 output_filename
--print_colour_coverages
```

Also, -1 signifies ALL colours (so you don't have to write enormous comma-separated lists). This example looks for bubbles in the union of all colours

```
cortex_var_31_c30 --kmer_size 31
--mem_height h --mem_width w
--multicolour_bin samples_and_ref.ctx
--detect_bubbles1 -1/-1
--output_bubbles1 output_filename
--print_colour_coverages
```

10.3 Options you must include if you want to build a VCF

1. You **must** use the --print_colour_coverages command when calling bubbles if you are going to dump a binary.
2. Save the output from Cortex (the stuff it prints to screen) as a log file. You will need it for the population filter (if you use it) and for building a VCF

10.4 What do Cortex calls look like?

Variants are printed in this format (this is an example for demonstration only, usually the flanks are much longer):

```
>var_1_5p_flank  length:42  INFO:KMER:31
CTGAGATAGGCTGGTCCTCACCTCCAGAGCCAGCCAGCCCCG
>branch_1_1
CGCCCTTGTTGAGTGTTCTTTGGAATTGTCGTTTTTTGAGCACAAC
TACAGCATTT
>branch_1_2
TGCCCTTGTTGAGTGTTCTTTGGAATTGTCGTTTTTTGAGCACAAC
TACAGCATTT
>var_1_3p_flank
TAGACTGCATGAAACCATGA
```

The format is fasta-like, with reads appearing in quartets. The first read is the 5prime flank, the next two are the two alternate alleles, and the final read is the 3prime flank. The first number after var_or branch_is the number of the variant. This example is a SNP, so the two branches (alleles) differ only in the first base.

If we had added --print_colour_coverages to the command-line, the output would be in this format, showing for each branch and for each colour the coverage of each kmer along the branch :

```
>var_1_5p_flank
CTGAGATAGGCTGGTCCTCACCTCCAGAGCCAGCCAGCCCCG
>branch_1_1
CGCCCTTGTTGAGTGTTCTTTGGAATTGTCGTTTTTTGAGCACAAC
TACAGCATTT
>branch_1_2
TGCCCTTGTTGAGTGTTCTTTGGAATTGTCGTTTTTTGAGCACAAC
TACAGCATTT
>var_1_3p_flank
TAGACTGCATGAAACCATGA
branch1 coverages
Covg in Colour 0:
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
Covg in Colour 1:
4 3 3 3 3 2 2 2 2 2 2 2 2 2 2 2 1 1 1 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 4
4 4 4 4 4 4 3 3 3 3 3 3 3 3
branch2 coverages
Covg in Colour 0:
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0
Covg in Colour 1:
4 3 3 3 3 3 4 4 4 3 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 3 3 3 3 3 2 2 2 2 2
```

```
3 4 3 3 3 4 5 5 5 5 5 4 4 4
```

Suppose we had specified the reference genome be loaded into colour 0. We see that branch2 (allele2) has zero coverage in colour 0, so this is not the reference allele. However branch1 has coverage 1 in colour 0, so is the reference allele (and has no paralogs in the reference). Finally, we see both alleles have coverage in colour 1 (the de Bruijn graph of the individual).

10.5 Haploid organisms

The Bubble Caller works perfectly well with haploid organisms. It has no knowledge of ploidy at the point of discovery, and just looks for bubbles. When you genotype, then you need to tell if if haploid or diploid.

11 VARIATION DISCOVERY USING THE PATH DIVERGENCE CALLER

11.1 Applying the PD caller to a union/pool of colours

The idea of the Path Divergence Caller is to build a 2-colour de Bruijn graph of a sample, and a reference genome, and then follow the path through the graph taken by a reference genome, detecting (primarily homozygous) variants via their breakpoints (where the path of the reference diverges from the graph of the sample). On human data, for example, the Path Divergence Caller successfully calls SNPs, indels, inversions, complex haplotypes consisting of phased SNPs and indels, and Alu retrotransposon indels. See our paper for a detailed analysis of its sensitivity and specificity, validated by comparison with fully sequenced and finished fosmid sequence. See for example Figure 3b in our paper. Note these important requirements: the ‘**PD requires one fasta per chromosome** in the reference **and requires that the entire reference be loaded into the graph**. If you try to use a reference which is not already loaded into the graph, Cortex will throw an error (cannot find such-and-such kmer) and exit.

For example, if the reference is in colour 0, and the sample in colour 1, then we invoke the caller thus:

```
--path_divergence_caller 1
--ref_colour 0
--list_ref_fasta list_fasta
--path_divergence_caller_output output_file
```

If, more generally, we had loaded 8 samples into colours 0,1,2...7, and we wanted to consider them as a pool, and wanted to look for variants between them and a reference in colour 8, then we would type:

```
--path_divergence_caller 0,1,2,3,4,5,6,7
--ref_colour 8
--list_ref_fasta list_fasta
--path_divergence_caller_output output_file
```

One output file is created. The output format is as for the Bubble Caller.

One detail worth noting - **Cortex has a global setting for the maximum variant length it looks for**, set by default to 10kb. If you are looking at a reference sequence smaller than that, Cortex won't be able to get a sliding window of the size it expects, and won't call anything. In such cases, set --max_var_len to something more appropriate. For example in one of the demo/ examples we look at a reference genome which is about 2kb long, and we set --max_var_len 500 to successfully call a variant which is the deletion of an Alu from within an Alu (a completely made-up example). Obviously, if you want to call larger things, and set max_var_len larger. The theoretical limit is half the length of the chromosome, though I have never tried with anything greater than 100kb.

11.2 Applying the PD caller to different colours consecutively

Suppose we have loaded 8 samples into colours 0,1,2...7, and the reference into colour 8, and we want to first call variants by comparing sample 0 with the reference, then sample 1, then sample 2, all the way up to sample 7. To do this, we use a slightly awkward syntax:

```
--path_divergence_caller [0[1[2[3[4[5[6[7
--ref_colour 8
--list_ref_fasta list_fasta
--path_divergence_caller_output output_file
```

12 GENOTYPE CALLING

Cortex will genotype calls using the model described in our paper. To do this, it needs to know depth of coverage, read length, sequencing error rate, and also what the colours represent - is each colour data from a diploid sample, or a haploid sample (the only two options we currently support (call me if you want more)).

In the process of building binaries from fastq, Cortex stores information about read length and total base pairs loaded into the graph, and this is preserved in the header of its binary files, so this information is available already. But to work out depth of coverage, it needs the length of the genome (--genome_size). If you can estimate the sequencing error rate (eg by comparing a small number of sites with other experimental data - for human HapMap samples, I look at sites that HapMap says are ref-ref in my sample, and count how many of those sites I have coverage on the alt-allele), then enter that in --estimated_error_rate (per base) - if you enter nothing, Cortex uses a default of 0.01. Finally, you need to use --experiment_type. Valid arguments for this are
EachColourADiploidSample,
EachColourADiploidSampleExceptTheRefColour,
EachColourAHaploidSample,
EachColourAHaploidSampleExceptTheRefColour.
If you do this, for example with the reference in colour 0, and diploid samples in colours 1 to 10, and run the Bubble Caller, your output will look like this

```
Colour/sample GT_call llk_hom_br1 llk_het llk_hom_br2
0=REF   NO_CALL 0       0
1 HOM2 -17.47 -3.68 -1.54
2 HOM1 -3.40 -8.20 -35.49
3 HOM2 -22.03 -4.50 -1.67
4 HOM1 -1.56 -2.89 -12.28
5 HET -11.49 -2.53 -6.26
6 HOM1 -1.55 -2.88 -12.26
```

```
7 HOM1 -1.75 -5.13 -28.10
8 HET -6.47 -2.56 -11.65
9 HOM1 -3.00 -7.79 -35.90
10 HOM1 -1.69 -4.41 -21.69
```

followed by the usual (flanks, branches, colour coverages). GT_call is the called maximum likelihood genotype. llk means log likelihood. Thus each row has a colour, a genotype call, and then the log likelihoods of the three possible genotypes. When generating a VCF from this file, our process_calls.pl script annotates a genotype confidence, as the difference between the maximum log likelihood, and the next biggest.

12.1 Genotype calling on calls that have already been made

Cortex also allows you to genotype calls at a later stage than discovery. This allows you to make calls individually on a set of samples, make a union set of all the variants, and then go back with this bigger set of variants, to genotype all the samples. All you need to do is pass in a set of Cortex calls (**without –print_colour_coverages output** and specify the output filename, and whether the caller was the Bubble Caller of Path Divergence caller. The simplest way to do this is call on all your samples, concatenate the output files, and then pass that in. That's not ideal, as you end up with non-unique variant names (each sample call set has variants called var_1, var_2, etc). So we have a script to make it easier:

```
perl analyse_variants/
        make_union_varset_at_single_kmer.pl
  --kmer 31
  --index INDEX
  --varname_stub SOME_STRING
  --outfile FILENAME_OF_UNION_CALLSET
```

The index file is tab separated with 3 files. These are: filename of a BubbleCaller or PD caller output, kmer size, and cleaning threshold used. You can just enter 0 for the cleaning threshold if you want, that argument is only really used by run_calls.pl. All the files must have been called by the BC, or all called by the PD. You can't pass in a mixture.

You're then ready to genotype your calls - if you use a multicolour graph of all your samples, each of them will be genotyped at each of these sites. First work out the longest read in the file FILENAME_OF_UNION_CALLSET (which includes both flanks and alleles), call it MAX, and then run:

```
 cortex_var_31_c137
  --multicolour_bin all_samples_and_ref.ctx
  --kmer_size 31
  --gt UNION_CALLSET,UNION_CALLSET.genotyped,BC
  --experiment_type
     EachColourADiploidSampleExceptTheRefColour
  --genome_size 3000000000
  --max_read_len MAX
  --print_colour_coverages
```

The output is then ready to go straight into process_calls.pl (see below).

12.2 Making all of that simpler

This whole business of calling on different samples and running scripts to make union call sets and working out maximum read lengths and then running –gt all seems a lot of bother, and a bit of a pain for the user. We therefore provide a wrapper script that does **everything**, from building binaries, cleaning them, calling, making union sets, genotyping and then dumping VCFs. See the section below of run_calls.pl.

13 CLASSIFICATION OF VARIANTS AS VARIANT, REPEAT OR ERROR

We describe in our paper our statistical model for classifying structures in the graph - essentially polymorphisms, repeats and errors all have different allele-balance and coverage statistics in a population, so we can use this information to classify putative variants.

13.1 When should I use the population filter/classifier?

For diploid species: Use the filter when you have several samples - we have got good results with s few as 10 chimpanzees, but the more the better. Do not use it on a single sample. We have used it on humans and chimps so far.

For haploid species: In the case of a haploid species, because there is no such thing as a heterozygous site, the signal from repeats is very strong. As a result, we've used the population filter to remove repeats even when looking at just two samples. We have used this on *P. falciparum* and *S. aureus* for example, and got good results.

13.2 How to run the population filter

We provide an R script (in the scripts/analyse_variants directory),called classifier.parallel.ploidy_aware.R, for classifying putative variants - this will be integrated into the main executable in future releases.

Here is how to apply the filter (all of the scripts below are in scripts/analyse_variants):

13.3 Make some auxiliary files needed by the R script

- Make a table file (takes < 1 second). Takes as input, the log file you stored from running the Bubble or PD caller - i.e. whatever Cortex printed to screen - most importantly, the table of mean read lengths and total sequence in each colour.

```
perl make_read_len_and_total_seq_table.pl
      genotyping.log
 >& genotyping.log.table
```

- Make a coverage file. The script takes as arguments a call file (the output file from Bubble or PD caller), the number of colours for which Cortex was compiled when those calls were made (e.g. if you used cortex_var _31_217 to analyse 3 samples, then use 217), and the final argument is the reference colour (enter -1 if there was no reference)

```
perl make_covg_file.pl
```

```
            BC_or_PD_callfile
            NUM_COLOURS
            REF_COLOUR
```

This creates a file called BC_or_PD_callfile_covg_for_classifier.

13.4 Run the population filter/site classifier

Suppose you have N calls. The classifier takes these arguments (in this order)

1. number of the first variant to use (if the first one is var_1, then enter 1).
2. how many variants to process/classify
3. input covg_for_classifier file.
4. Number of rows/lines in the covg_for_classifier file = N
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.
6. was there a reference colour? 1 for yes and 0 for no (doesn't matter which colour it was
7. Table of read lengths and covgs
8. Estimated genome size. (Don't panic if not exact)
9. kmer size
10. ploidy. 1 for haploid, 2 for diploid, no other value acceptable.
11. output file name

You can therefore run the classifier in one process or in parallel. Roughly speaking, I would not bother parallelising unless I had millions of calls.

Here's how to run the classifer - unfortunately I don't know how to make R take command-line arguments in a user-friendly way, so you just pass in one argument after another, space-separated:

```
cat classifier.parallel.ploidy_aware.R | R
    --vanilla
    arg1 arg2 arg3 arg4 arg5
    arg6 arg7 arg8 arg9 arg10 arg 11
```

13.5 Running in one process

To give a concrete example, supposing I have N calls over num_colours colours/samples, I used kmer=31, there is a reference, and the genome is diploid and 4Gb in size:

```
cat classifier.parallel.ploidy_aware.R | R
  --vanilla
  --args
         1
         N
         callfile.covg_for_classifier
         N
         num_colours
         1
         genotype.log.table
         4000000000
         31
         2
         callfile.classified
```

Please note **this will not work** if you use 4e9 instead of 4000000000. The output of the classifier looks like this: tab separated columns are variant number, classification (variant, repeat or error), and confidence (difference between maximum log likelihood, and next biggest)

```
1 variant 15.46981
2 repeat 1.228099
...
```

13.6 Running in parallel

Exactly as above, just change the first and second arguments to specify which variants you want to classify (1 to 1000, 1001 to 1999, etc), and the final argument to specify the output file. When you are finished, concatenate the output files in the correct order (i.e. the final file should have first column which is all the variants in numerical order of variants).

14 CONVERTING CORTEX CALLS INTO VCF FORMAT

Cortex provides a script (process_calls.pl) which takes a BC or PD call file, plus various arguments, and dumps a pair of VCFs. The raw (.raw.vcf) file contains the actual calls. The decomposed (.decomp.vcf) file breaks down composite variants into sub-SNPs and indels where possible. Why is this necessary?

A great strength of Cortex is that it looks for variants in a manner completely agnostic to variant type. It does not look for SNPs, or deletions or insertions - it looks for any allelic differences. However, as a result, Cortex variant calls can often consist of clusters of nearby SNPs, or SNPs and indels, or large deletions with a small insertion at the breakpoint, etc, and it can be non-trivial to classify the type of variant found. Of course, in many cases there is no canonical decomposition into subvariants, and the final truth depends on whether the subvariants occured at the same time, by the same mechanism, which can only be inferred by looking at how they segregate in a population. We have therefore found it useful to be able to do a full Needleman-Wunsch alignment between the two branches (alleles). Since version 1.0.5.6 Cortex has used a C implementation of Needleman-Wunsch from Isaac Turner, which is bundled into the release (and which is much faster than the old process_bubbles.pl script Cortex used to use).

14.1 process_calls command-line arguments

process_calls.pl takes the following mandatory arguments

```
--callfile FILENAME
```

- this is the file output by the Bubble or Path Divergence caller (which **must** have been called with --print_colour_coverages.

```
 --callfile_log FILENAME
```

- this is a file containing the text Cortex printed to screen (”stdout” output) - it is generally simplest if you pipe consolee output when calling to a file for this reason.

```
 --outvcf STRING
```

The VCFs output will have filenames which start with this string.

```
  --outdir DIRNAME
```

Output directory name

```
  --samplename_list FILENAME
```

A file containing one line per colour, and on each line, a sample identifier - these end up in the header line of the VCF. Use "REF" for the reference colour (if there is one). This option will be removed in future releases - Cortex now supports an option --sample_id, so Cortex binaries, and Cortex console/stdout output have this information. In the future I'll fix the script to simply read the names from the callfile_log above. But for now, sorry, you need to make this file. If you use the run_calls.pl script, which wraps everything from binary building, cleaning, calling all the way to VCF dumping (see below), then this is done for you.

```
--num_cols
```

Specify the number of colours in the graph. If you used cortex_var_31_c2419 but only loaded 127 samples, you should enter 2419 here.

```
  --stampy_bin
```

Full path to stampy.py (which you can obtain from
http://www.well.ox.ac.uk/project-stampy).

```
--stampy_hash PATH
```

Cortex calls variants completely without use of a reference, but inevitably one needs to place these on an assembly. process_calls.pl will map the flanks of your calls to an assembly using Stampy (it must be Stampy, you cannot replace it with a mapper of your choice - note this is not a standard operation for a mapper). You need to first build a Stampy hash. Suppose your reference was ref.fa, then you do this as follows

```
stampy.py -G /path/to/foo ref.fa
stampy.py -g /path/to/foo -H /path/to/foo
```

This will create two files, /path/to/foo.stidx, and /path/to/foo.sthash.You should enter --stampy_hash /path/to/foo. (Replace the string foo with the name of your species or reference or whatever). If you have no reference, and don't care about coordinates, but want to know what the variants are (SNPs, indels, complex), and who has what allele (genotypes), then we abuse VCF slightly. Suppose you have N variant calls. First create a pseudo-reference which has N chromosomes. Chromosome i is the 5prime flank, branch1 and then 3prime flank of variant i:

```
  perl make_fake_reference.pl --callfile FILE
       --outfile OUTFILE
```

Then build a stampy hash of this pseudo-reference and pass it in with the –stampy_hash argument above. Back to arguments of process_calls.pl:

```
  --vcftools_dir
```

This is needed in the VCF dumping process. Give the name of the root VCFTools directory - this should have subdirectories called: bin, cpp, lib etc.

```
  --caller STRING
```

Valid arguments are BC or PD (signifying Bubble Caller or Path Divergence Caller).

```
  --kmer INT
```

Self-explanatory - the kmer used to call these variants.

The following arguments are optional

```
  --refcol INT
```

Which colour was the reference (if any). Default is -1 (meaning no reference present)

```
  --pop_classifier FILENAME
```

If you used the population filter/classifier, then pass in here the name of the classifier output file.

```
  --ploidy INT
```

Valid values are 1 (haploid) and 2 (diploid). Default is 2.

```
  --prefix STRING
```

String prefix will go on the start of all variant names. e.g --prefix ZAM will produce variants ZAM_var_1, ZAM_var_2, etc

```
  --ref_fasta FILENAME
```

Stampy maps calls to a reference with a mapping quality. We use a threshold of 40 by default, so 1 in 10000 are wrongly placed on the reference. If you pass in the name of the reference fasta here, this script will check the VCF and remove misplaced variants.

14.2 Interpreting Cortex VCFs

The main Cortex VCF file is the "raw.vcf". Here is a typical line, which I'll have to print over a few lines to fit into the width of this column in the manual:

```
5       12087   var_278 TA      T       .
PASS     PV=3;SVLEN=-1;SVTYPE=DEL
GT:COV:GT_CONF  0/0:14,0:40.52  1/1:1,6:16.86
```

This is a variant at position 12087 on chromosome 5. The reference allele is TA and the alternate allele is T, so it is a 1 base deletion. The PASS means it has passed all the Cortex filters. PV=3 means that there is an ambiguity of 3bp in where you could decide to ”place” this variant. Here’s the intuition. If one allele is XYX and the other is X, we could consider this a deletion of the first XY, or of YX. Those two options would imply different positions for the variant. Cortex left-aligns. This text: GT:COV:GT_CONF tells you that Cortex has done genotyping (GT), and for each sample you will get a genotype call, the coverages on the two alleles, and the genotype confidence (log probability of the max likelihood genotype - log probability of the 2nd most likely). In this case the first sample is called homozygous reference (0/0), with coverage 14 on the reference allele and 0 on the alternate, and with genotype confidence of 40.52. Larger confidences mean you are more confident of the call.

Here is another example

```
17      732281  var_134
ATCCA   ACCCC
 .        PASS
 SVLEN=2;SVTYPE=PH_SNPS
 GT:COV:GT_CONF
 0/0:18,0:51.93  1/1:1,13:32.69
```

This time the reference allele is longer, as is the alternate allele, and they are quite similar. It’s a little difficult to decide what we think of it, and so process_calls does an alignment of the two alleles (look in the .aligned_branches file in the output directory). SVTYPE is set to COMPLEX, so process_calls thinks, on the basis of the alignment, that it is a cluster of phased SNPs. If we look in the decomp VCF, this splits these into separate SNP calls, but loses the phasing information.

```
 17       732282  var_4_sub_snp_1
  T        C        .        PASS
  SVLEN=0;SVTYPE=SNP_FROM_COMPLEX
  GT:COV:GT_CONF
 0/0:18,0:51.93   1/1:1,13:32.69

17       732285  var_4_sub_snp_2
A        C        .        PASS
SVLEN=0;SVTYPE=SNP_FROM_COMPLEX
GT:COV:GT_CONF
 0/0:18,0:51.93   1/1:1,13:32.69
```

If you used the population filter, then each site has a site confidence as well as a genotype confidence.

14.3 Missing calls

Calls which cannot be mapped to the reference you are using cannot go into the VCF, as you don’t have chr or position. Those which are discarded are mentioned in the output of process_calls (”Ignore this var_9 - due to this error Did not map”). In future I’ll dump them to an unmapped.vcf.

14.4 Making a high confidence set

Look at the distribution of site confidences of your sites (if you have used the population filter), and choose a threshold for minimum site confidence. Look at the minimum or median genotype confidence for each site (across samples), and set a minimum threshold on that. Remember these confidences are in log space, so a confidence of 10 means this is e^{10} times more likely than the alternative, so no need to set massive thresholds.

15 SIMPLIFYING CORTEX ANALYSES - WORKFLOWS AND RUN_CALLS.PL

15.1 Standard workflows

The latest release of Cortex introduces two standard workflows, allowing the user to run a complete analysis, all the way from fastq file to VCF file of variants, with a single command-line. The pipeline builds uncleaned graphs of each sample and examines the coverage distribution of kmers to choose a per-sample cleaning threshold. If desired, additional stringent or relaxed thresholds are used bracketing the automated choice, or it is possible to override automated error-cleaning with a specified threshold across all samples. Multiple simultaneous instances of the building, cleaning and discovery phases can be run (all based on the same directory structure), allowing parallelisation across kmers or samples. Both workflows are controlled by a Perl script called run_calls.pl. Both workflows create ”binaries”, ”calls” and ”vcfs” subdirectories within the specified output-directory, allowing the user to re-use binaries, and also do post-mortem analysis of any of the call sets.

15.2 Workflow 1 - joint discovery

The joint discovery workflow is the most direct and simple workflow. Suppose for simplicity that one cleaning threshold has been used for each sample. These cleaned graphs are loaded into a single multi-colour graph. If (one or more) reference genomes are available, these can be loaded into further colours. The user specifies if a reference is Absent, used for CoordinatesOnly or CoordinatesAndCalling. The user is also allowed to specify the number of cleaning thresholds to use above and below that chosen by the pipeline. Each sample is given a list of cleaning thresholds (ordered numerically), and all samples have the same number of cleaning thresholds in this list. For each kmer, and for each index in the list of cleaning thresholds, a multicolour graph is built, the bubble caller is applied to find variants, and each call is immediately genotyped for all samples. A VCF is built at each kmer, and then these are merged at the end.

As the number of samples increases, it becomes more likely that either a site becomes multiallelic, or that sequencing errors can confound a variant, which may reduce sensitivity at a fraction of sites. These can can be overcome by more stringent error-cleaning, and by use of the independent workflow (below).

15.3 Workflow 2 - independent discovery

This workflow is used to maximise sensitivity, and requires the use of a reference for both calling and coordinates. Each sample has graphs built for different k-mer values, and cleaned to different levels just as for the joint workflow. However in this case, discovery is done repeatedly for each sample, for each k-mer, for each cleaning threshold, in a 2-colour graph containing the sample and the reference genome. At each kmer, a union set of calls is collated from all the callsets, and then all the samples are genotyped on this set of sites, in the joint (multicolour) graph of all samples, using the lowest specified cleaning threshold for each sample. Since a reference genome is available, this workflow allows the user to specify that the Path Divergence caller is also used, allowing the user to access a range of larger variants. On combining VCFs, run_calls may find multiple variants at the same site, or overlapping. These sites are marked in the FILTER field, as MULTIALLELIC, or OVERLAPPING.

This workflow is only available if a reference genome is available, although in principle, one could generalise it by designating a specific sample to take the role of the reference here.

15.4 Reference genomes can be incorporated at different levels - run_calls terminology

In the joint workflow, the user is given three choices for how to include a reference genome (**”Coordinates Only”, ”Coordinates And In Calling”, and ”Absent”**). The independent workflow always uses ”Coordinates And In Calling”. These approaches are described thus:

- **Coordinates Only**: A reference genome is loaded into the graph in its own colour, but is completely ignored during variant discovery. For each discovered variant, the two alleles are compared post-hoc with the reference colour, and he flank is mapped to the reference to get coordinates.
- **Coordinates And Calling**: The reference genome is loaded into its own colour, and is included in variant discovery. For each discovered variant, the two alleles are compared post-hoc with the reference colour, and he flank is mapped to the reference to get coordinates
- **Absent**: There is no reference loaded into the graph, and so once variants are called, they are placed in a VCF with dummy chromosome/position fields. A fake reference genome is used, with one ”chromosome” per variant, so all variants should lie on their own chromosome.

If a reference genome is used, the user is required to pre-build binary graphs of the reference at all kmer-sizes which are to be used. To give a sense of scale, for *S.aureus* a k=31 reference genome binary takes 13 seconds to build, and a k=61 binary takes 20 seconds.

The reasons to prefer the ”Coordinates Only” (excluding the reference colour from discovery) are:

1. Ensures complete freedom from reference-bias - equal power for discovery of both alleles.
2. Discovers differences between samples, and not between the samples and the reference - i.e. avoid unnecessary and irrelevant variant calls created where all samples differ from the reference.
3. Avoids situations where the reference graph confounds the joint graph of the samples.

The reasons to prefer ”Coordinates And Calling” (including the reference colour when discovering variants) are:

1. The reference is known to be of high quality and not very divergent from the samples
2. There is a need to know sites where all samples have the same genotype, which differs from the reference genome

15.5 Command-line options for run_calls.pl

```
--first_kmer
   This script allows you to run
   across a range of kmer sizes.
   This is the lowest.
   It must be odd.

--last_kmer
   Ignore this if you want to
   run for one kmer only.

--kmer_step
   If you run for many kmers,
   this is the increment.
   Make sure this does not imply use
   of odd kmer values. Currently
   there is not enough checking
   in run_calls.pl for people
   entering bad arguments for
   this.

--auto_cleaning   {yes|no}
   Takes values "yes" or "no".
   Default "no".
   This looks at covg distribution
   and chooses a cleaning threshold.
   This makes a big difference to the
   success of calling, it's much better
   than just choosing one threshold
   for all samples.

--auto_below INT
   You can also ask it to make extra
   cleaned binaries for, say 2 thresholds
   below the auto-chosen one.
   By default it wont do this.

--auto_above INT
   You can ask it to make extra
   cleaned binaries for, say 3 threshold
   values above the auto-chosen
   one (will stay below the expected
   depth).

--user_cleaning {yes|no}
   Takes values "yes" or "no".
   Default "no".
   Make your own cleaning choices

--user_min_clean INT
   Specify cleaning threshold.
   Use this for either specifying just
   one  threshold or a range.

--user_max_clean INT
   If you want to try a range.
   Ignore this if you only want to
    use one threshold

--user_clean_step INT
   Increment between
   user-specified cleaning
   thresholds.

--bc {yes|no}
   Make Bubble Calls.
   You must enter yes or no.
   Default (if you don't use --bc)
   is no.

--pd {yes|no}
   Make PD calls.
   You must enter yes or no.
   Default (if you don't use --bc)
   is no.

--outdir DIRNAME
   Output directory. Everything
   will go into subdirectories
   of this directory


--outvcf NAME
   VCFs generated will have
   names that start with the
   text you enter here.

--ref
  Specify if you are using a
  reference, and if so, how.
  Valid values are:
  CoordinatesOnly,
  CoordinatesAndInCalling,
  Absent

--ploidy
  Must be 1 or 2. Default 2.

--prefix STRING
   If you want your variant calls
   to have names with a specific
   prefix, use this.


--stampy_hash PATH_STUB
  MANDATORY. Build a hash of
  your reference genome, and
  specify here the path to it.
  If stampy makes blah.stdidx etc
  then specify blah.
  See below for what to do if there is
  no reference.

--stampy_bin /path/stampy.py
  Specify the path to your Stampy binary.
  Or manually edit this at the top of the
  file (it's marked out for you).

--fastaq_index FILENAME
   MANDATORY. Tab separated file has
   columns:
   SAMPLE_NAM
   se_list
   pe_list1
   pe_list2.
   (One line per sample)

--qthresh INT
  Quality score threshold

--dups
  Remove PCR duplicates

--homopol  INT
   Cut homopolymer threshold.

--mem_height
   for Cortex

--mem_width
   for Cortex

--max_read_len
   Max read length. If you are passing
   in reference genomes,
   use 10000.

--gt_assemblies
   Valid arguments, "yes" and "no".
   Default is "no".
   If "yes", run_calls assumes the input data
   are whole genome assemblies, and sets
   sequencing error rate to a tiny value,
   to allow "genotyping" of differences between
   the assemblies.

--max_var_len
   max var length to look for.
   Default value 40000 (bp)

--genome_size
   Genome length in base pairs -
   needed for genotyping.

--refbindir
```

```
   Directory containing binaries
   built of the reference at all
   the kmers you want to use.
   The binary filename should
   contain the kmer value,
   eg refbinary.k31.ctx

--list_ref_fasta FILE
   File listing the fasta files
    (one per chromosome)
    for the reference.
    Needed for the PD caller.

--vcftools_dir DIRNAME
   VCFtools is used to generate
   VCFs - mandatory to either
   specify this on cmd-line,
   or manually edit the path
   at the top of this script.
   This should be the VCFtools
   root dir, which has
   subdirectories called:
   bin,  cpp,  lib ...

--do_union {yes|no}
   Having made many
    callsets (per kmer and cleaning),
    should we combine all calls into
     a union set, and genotype all
     samples? Valid values are yes
     and no. Default is no.
     If you want a VCF, type yes.
     If you just want to build binaries
     for now, type no.

--manual_override_cleaning FILE
   You can specify specific
   thresholds for specific samples
   by giving a file here, each line
   has three (tab sep) columns:
   sample name, kmer, and
   comma-separated thresholds.
   Don't use this unless you know
   what you are doing.

--logfile
   Output always goes to a logfile,
   not to stdout/screen.
  If you do not specify a name here,
  it goes to a file called
   "default_logfile".
   So, enter a filename for your
   logfile here.
   Use filename,f to force overwriting
   of that file even if it already exists.
   Otherwise run_calls will abort to
   prevent overwriting.
```

```
--workflow
    Mandatory to specify this.
    Valid arguments are
    "joint" and "independent".

--apply_pop_classifier
   Apply the Cortex population filter,
   to classify putative sites as repeat,
   variant or error.
   This is a very powerful method
   of removing false calls.
   but it requires population information
    to do so - ie only use it if you have at
    least 10 samples.
   This is just a flag (takes no argument)

 --squeeze_mem
   You need to set mem_height and
   mem_width large enough
   that a single uncleaned sample
   graph can be held.
   If this flag is set, once all samples are
   build and cleaned, Cortex will
   count how many kmers there are
   in the cleaned graphs and reduce
   memory use (mem_height
   and mem_width) to a smaller
   value that is sufficient to hold
   the cleaned data,.
```

16 TYPICAL USE-CASES FOR RUN_CALLS AND THE CORTEX WORKFLOWS

16.1 Comparing two strains of microbe

Suppose we have sequence data from two strains of some microbe that we want to compare. The most direct way to do this is to use the joint workflow (load them both into a graph and compare). If we have a reference, which may be slightly diverged, we do the following. First, choose a kmer-range to try - we know that different variants will be accessible at high kmer (genome repeat content/genome complexity) and at low kmer (relatively low coverage at these sites means they are lost at high kmer). So we could do an immense parameter sweep, but in my experience all you need is one low and one high kmer - in the example below I use k=31 and 61. We first build k=31,61 binaries of the reference genome:

```
cortex_var_31_c1 --kmer_size 31
--mem_height 17 --mem_width 100
--se_list file_listing_fasta
--format FASTA
--dump_binary ref.k31.ctx --sample_id REF


cortex_var_63_c1 --kmer_size 61
```

```
--mem_height 17 --mem_width 100
--se_list file_listing_fasta
 --format FASTA
--dump_binary ref.k61.ctx --sample_id REF
```

So build an ”index file”, which has two rows (one per sample) of tab-separated text, where the columns are sample identifier, se_list, pe_list1 and pe_list2 (as used by Cortex itself). Then we are ready to run the analysis:

```
perl run_calls.pl --first_kmer 31
    --last_kmer 61
    --kmer_step 30
    --fastaq_index INDEX --auto_cleaning yes
    --bc yes --pd no
    --outdir dirname
    --outvcf NAME
    --ploidy 1
    --stampy_hash ref/species_name
    --stampy_bin /path/stampy.py
    --list_ref_fasta FILELIST
    --refbindir ref/
    --genome_size 2800000
    --format FASTQ --qthresh 5
    --mem_height 17  --mem_width 100
    --vcftools_dir /path/vcftools_0.1.8a/
    --do_union yes
    --ref CoordinatesOnly --workflow joint
    --logfile logfile log.txt
    --apply_pop_classifier
```

This will:

1. Build unclean k=31, k=61 binaries of both samples
2. Look at the coverage distribution of both of these, choose the most appropriate error-cleaning threshold, clean both binaries.
3. Make a 3-colour graph (reference, sample1, sample2), and then call variants in the union of colours 1 and 2 (--detect_bubbles1 1,2/1,2)
4. Apply the population filter - in the case of two haploids, this will remove repeats.
5. Build a k=31 and a k=61 VCF.
6. Combine the vcfs from different kmers to produce two final VCFs (raw and decomp)

This is the first thing I would try, if there was a reference. I would also try the independent workflow, and include the Path Divergence caller:

```
perl run_calls.pl --first_kmer 31
    --last_kmer 61
    --kmer_step 30
    --fastaq_index INDEX --auto_cleaning yes
    --bc yes --pd yes
    --outdir dirname
    --outvcf NAME
    --ploidy 1
    --stampy_hash ref/species_name
    --stampy_bin /path/stampy.py
    --list_ref_fasta FILELIST
    --refbindir ref/
    --genome_size 2800000
    --format FASTQ --qthresh 5
    --mem_height 17  --mem_width 100
    --vcftools_dir /path/vcftools_0.1.8a/
    --do_union yes
    --ref CoordinatesAndInCalling
    --workflow independent
    --logfile logfile log.txt
    --apply_pop_classifier
```

If in fact there was no reference available, I would use --ref Absent (and leave out --list_ref_fasta and --refbindir).

16.1.1 Extending to more strains Cortex memory use M (in bytes) scales with number c of samples/colours, k-mer k, and the total number N of kmers according to this formula:

$$M = N \left\lceil 8\lceil \frac{k}{32} \rceil + 5c + 1 \right\rceil^{[8]} \tag{6}$$

where $\lceil n \rceil^{[8]}$ means round n up to the nearest multiple of 8. For typical microbial genomic use-cases, each isolate or strain is sequenced to relatively high depth ($> 20\times$), and therefore each sample graph can be error-cleaned independently - both the joint and independent workflows assume this is the case. **This ensures that sequencing errors impact memory use only at the per-sample level**. By the time a multicolour graph is built, memory requirements depend on the genome size and sample diversity, but not on total sequencing depth or sequencing error-rate. Applying this formula to an experiment using 100/1000/10000 samples of a 3Mb genome, we can estimate the memory requirement at k=31 by using an upper bound of 5 million true unique kmers amongst the samples. This results in an estimated memory use of 2.5/25/250 Gb RAM.

16.2 Looking at a human family trio for de novo mutations

Actually there’s nothing special about microbes. The above independent workflow would also be ideal for looking at a human trio. With 3 diploids there would essentially be no value to the population filter, so the command-line would be:

```
perl run_calls.pl --first_kmer 31
    --fastaq_index INDEX
    --auto_cleaning yes
    --bc yes --pd yes
    --outdir dirname
    --outvcf NAME
    --ploidy 2
    --stampy_hash grc37
    --stampy_bin /path/stampy.py
    --list_ref_fasta FILELIST_GRC37
    --refbindir ref/
    --genome_size 3000000000
    --format FASTQ --qthresh 5
    --mem_height 26  --mem_width 100
    --vcftools_dir /path/vcftools_0.1.8a/
    --do_union yes
    --ref CoordinatesAndInCalling
    --workflow independent
    --logfile logfile log.txt
    --squeeze_mem yes
```

The mem_height and mem_width options specify how much memory you want to use when building the graph. However there are two very different phases to the process. First, one build and cleans per-sample graphs. This requires more nodes (due to sequencing errors), but only one colour. The second phase needs all the sample cleaned graphs in a multicolour graph. By default run_calls will use the same mem_height and mem'us width parameters all the way through. This means for the second phase, you are using a lot more memory than you need, and for human genomes, this will be profligate. The --squeeze_men option tells run_calls to reduce memory use after error cleaning.

17 TROUBLESHOOTING RUN_CALLS

Some typical issues which have arisen when people use run_calls:

17.1 I've got really high coverage of two microbial strains where I know the answer, but run_calls is not calling them

If you do have very high coverage (hundreds of x), you can afford to filter your data quite conservatively. Using

```
--qthresh 10 --auto_cleaning stringent
```

is a good start, or you might try

```
--qthresh 10 --auto_cleaning yes --auto_above 2
```

telling run_calls to use a couple of extra cleaning thresholds above the default.