cortex_var User Manual

Zamin Iqbal

April 8, 2011

Contents

1 General Introduction

Cortex is a software suite for de Bruijn genome assembly and population/variation analysis.

- low memory use (at k=31, 10 human genomes in under 256Gb of RAM, 1000 yeasts in under 64Gb RAM)
- speed, predictability and stability - memory use specified at the start
- multicoloured be Bruijn graphs - allows joint assembly of multiple samples and/or reference genomes
- 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 variant calling on a sample by diploid assembly (SNPs to structural variants)
- 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.

Cortex consists of two executables: - cortex_con for consensus assembly (primary contact Mario - mario.caccamo@bbsrc.ac.uk) and cortex_var for variation analysis and population assembly (primary contact Zam - zam@well.ox.ac.uk). For details of cortex_con, please consult the documentation on our website, cortexassembler.sourceforge.net. This document is about cortex_var.

2 Command-line interface

2.1 cortex_var command-line interface

[-h | --help] = Help screen.

[--colour_list FILENAME] = File of filenames, one per colour. n-th file is a list of single-colour binaries to be loaded into colour n. Cannot be used with –se_list or –pe_list

[--multicolour_bin FILENAME] = Filename of a multicolour binary, will be loaded first, into colours 0..n. If using –colour_list also, those will be loaded into subsequent colours, after this.

[--se_list FILENAME] = List of single-end fasta/q to be loaded into a single-colour graph. Cannot be used with –colour_list

[--pe_list FILENAME] = Two filenames, comma-separated: each is a list of paired-end fasta/q to be loaded into a single-colour graph. Lists are assumed to ordered so that corresponding paired-end fasta/q files are at the same positions in their lists. Currently Cortex only use paired-end information to remove PCR duplicate reads (if that flag is set). Cannot be used with –colour_list

[--kmer_size INT] = Kmer size (default 21). Must be an odd number.

[--mem_width INT] = Size of hash table buckets (default 100).

[--mem_height INT] = Number of buckets in hash table in bits (default 10). Actual number of buckets will be 2^(the number you enter)

[--ref_colour INT] = Colour of reference genome.

[--remove_pcr_duplicates] = Removes PCR duplicate reads by ignoring read pairs if both reads start at the same k-mer as a previous read, and single-ended reads if they start at the same k-mer as a previous read

[--cut_homopolymers INT] = Breaks reads at homopolymers of length > this threshold. (Input of sequence restarts after homopolymer)

[--path_divergence_caller COMMA_SEP_COLOURS] = Make Path Divergence variant calls. Must specify colour of sample in which you want to find variants compared with the reference. This sample

colour can be a union of colours (comma-separated list). Must also specify --ref_colour and --list_ref_fasta

[--path_divergence_caller_output PATH_STUB] = Specifies the path and beginning of filenames of Path Divergence caller output files. One output file will be created per reference fasta listed in –list_ref_fasta

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

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

[--remove_seq_errors] = Remove tips + remove nodes if their coverage is more likely to be due to single-base sequencing error than sampling.

[--dump_binary FILENAME] = Dump a binary file, with this name (after applying error-cleaning, if specified).

[--output_supernodes FILENAME] = Dump a fasta file of all the supernodes (after applying all specified actions on graph).

[--detect_bubbles1 COMMA_SEP_COLOURS/COMMA_SEP_COLOURS] = Find all the bubbles in the graph where the two branches lie in the specified colours (after applying all specified actions on graph). Typical use would be --detect_bubbles1 1/1 to find hets in colour 1, or --detect_bubbles1 0/1 to find homozygous non-reference 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).

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

[--detect_bubbles2 COMMA_SEP_COLOURS/COMMA_SEP_COLOURS] = Exactly the same as detect_bubbles1, but allows you to make a second set of bubble calls immediately afterwards. This is to accomodate the common use-case where one loads a reference and an individual, and then wants to call homs, and hets.

[--output_bubbles2 FILENAME] = Bubbles called in detect_bubbles2 are dumped to this file.

[--format TYPE] = File format for input in se_list and pe_list. All files assumed to be of the same format. Type must be FASTQ, FASTA or CTX

[--max_read_len] = Maximum read length over all input files. (Mandatory if fastq or fasta files are input.)

[--print_colour_coverages] = Print coverages in all colours for supernodes and variants.

[--max_var_len INT] = Maximum variant size searched for. Default 10kb.

[--list_ref_fasta FILENAME] = File listing reference chromosome fasta file(s); needed for path-divergence calls.

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

[--remove_low_coverage_kmers INT] = Filter for kmers with coverage less than or equal to threshold.

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

[--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 TEXT] = Only to be used when also using –load_colours_only_where_overlap_clean_colour and –multicolour_bin Used to allow error-correction of low-coverage data on large numbers of individuals with large genomes. Requires the user specify a suffix which will be added to the names of cleaned binaries. See manual for details.

[--align FILENAME] = Aligns a list of fasta/q files to the graph, and prints coverage of each kmer in each read in each colour. 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 for format as LIST_OF_FASTQ or LIST_OF_FASTA

3 Compilation

To build an executable that supports k≤31 and 1 colour:

make cortex_var

This creates a binary (in the bin directory) called cortex_var_31_c1. To build an executable that supports k≤31 and n colours:

make NUM_COLS=n cortex_var

produces an executable called cortex_var_31_cn. To build an executable that supports 33≤k≤63 and 17 colours (for example), type:

make NUM_COLS=17 MAXK=63 cortex_var

which creates an executable cortex_var_63_c17. etc. We have not implemented any error-checking in the Makefile, so negative, fractional or non-numeric values of MAXK or NUM_COLS will give unpredictable results.

4 Usage of Cortex

4.1 Introduction

Cortex is a framework for genome assembly and analysis; it has two families of executables, cortex_con_n and cortex_var_n_cm (where n=31,63,95..., and m=1,2,3...), which are developed along parallel tracks, sharing a common modular codebase. A detailed description of how it encodes de Bruijn graphs in a hash table, and the set of algorithms we provide for variant calling are given in our paper, "De novo assembly for variant calling and genotyping". cortex_con focuses on consensus assembly (hence the name), and cortex_var on assembly of variation and populations. Binaries built with one executable are completely compatible with the other, with the proviso that cortex_con only supports single-colour binaries.

4.2 Variation and population analysis with cortex_var

Summary: De Bruijn graphs can be built (from fasta or fastq) in almost the same way as for cortex_con, with a couple of small differences.

- Firstly cortex_var supports on-the-fly removal of PCR duplicates, and cutting reads at homopolymers (useful with 454 reads). In the future these may be merged into cortex_con also.
- Secondly, cortex_var can build (and dump) multi-colour binaries - each "colour" can represent a sample, a pool, a population - it depends on what data you give to Cortex.

cortex_var allows variant calling by two different algorithms - the Bubbler Caller (looking for certain motifs in the graph) and the Path Divergence Caller (aligning a reference genome to the graph and detecting breakpoints). The standard pattern of usage is to

1. build and error clean graphs of a single sample from fasta/q.
2. dump a single-colour binary (since we have removed many errors, the dumped binary contains fewer nodes, so next time we reload it requires less memory)

3. load binaries for different individuals into different colours for variation analysis.

4.2.1 File Input

cortex_con accepts the following as input

1. Fasta only. These will always be loaded into a single colour graph, and will be dumped as a single-colour binary (allowing mixing-and-matching of binaries into whatever colours you like)
2. Fastq only. These will always be loaded into a single colour graph, and will be dumped as a single-colour binary (allowing mixing-and-matching of binaries into whatever colours you like)
3. One list of single-colour 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

my_fastq2.fq

> cat pe_filelist1

fastq1_1.fq

fastq2_1.fq

fastq3_1.fq

> cat pe_filelist2

fastq1_2.fq

fastq2_2.fq

fastq3_2.fq

> cortex_var --se_list se_filelist --pe_list pe_filelist1,pe_filelist2 --mem_height <h> --mem_width <w> --max_read_len 100 --dump_binary somename.ctx

This will dump a single-colour binary called somename.ctx. 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 some lists of single-colour binaries (each list 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. 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

>ls list* > colour_filelist

[open colour_filelist with a text editor and ensure the order 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

>cortex_var --colour_list colour_filelist --kmer_size k --mem_height h --mem_width w --dump_binary trio_plus_ref.ctx

This will dump a 4-colour binary, with the mother, father, child, reference in colours 0,1,2,3. 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 do the following (we show this explicitly but it can easily be wrapped in bash or perl) - make a binary list for each individual, and then list these in the order you want them to go into colours:

> ls indiv_1.ctx > individual_1_binarylist

> ls indiv_2.ctx > individual_2_binarylist

.....

>ls indiv_29.ctx > individual_29_binarylist

> ls individual*binary_list | sort > list_new_individuals

> cortex_var --multicolour_bin trio_plus_ref.ctx --colour_list list_new_individuals --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..29 into colours 4..33.

4.2.2 Filtering of input sequence data

Cortex allows reads to be filtered on-the-fly as they are loaded, by specifying –quality_score_threshold <value>. Each time a read has any base with phred-scale base-quality≤value, 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.

4.2.3 Choosing hash table size

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* (length of genome). 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 cingle-colour hash table with N nodes, using an executable that supports a maximum kmer of K, can be calculated precisely, using this formula (explanation given in our paper):

$$M = \left(8 \left\lceil \frac{K}{32} \right\rceil + 5 + 1\right) N$$

For the above example, if we create a hash table with 4.9million nodes, and k≤31, 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:

$$M = \left(8 \left\lceil \frac{K}{32} \right\rceil + 5C + 1\right) N$$

Note that this formula reduces to that for cortex_con if C=1. 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 3Gigabases) 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. This should therefore require $(8 + 5 + 1) \times 6 \times 10^9$bytes = 84Gigabytes of RAM. In fact (for k around 20-50), a human genome contains around 2.5 billion kmers (calculated by counting kmers in the human genome reference), 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 $(8 + (5 \times 3) + 1) \times 3 \times 10^9 = 72Gb$ 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.

4.2.4 Filtering of input sequence data

Input reads can be filtered by quality-value just as for cortex_con. In addition, Cortex has two extra filters:

1. 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% of reads.

2. Reads can be cut at homopolymers of a specified length. --cut_homopolymers <value> will cut a read at a homopolymer longer than 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).

4.2.5 Error Cleaning

cortex_var contains 3 means of error-cleaning:

1. Tip clipping - if sequence coverage is sufficiently high, and the genome in question is sufficiently un-repetitive, and the kmer value is large enough, then sequencing errors can create "tips" in the graph - short arcs which do not lead anywhere. However coverage gaps will look like tips, so this method is not appropriate with low coverage data. The option --tip_clip will remove these. We detail in our paper how it is possible to measure quantitatively the probability that a sequencing error will create a tip or clean bubble (the alternative would be to create a chimeric connection between two parts of the graph) - for example, for the human genome at k=21 with 36bp reads, only 51% of possible sequencing errors are unconfounded with the rest of the genome, rising to 75% at k=31. See the paper for more details.

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 <value> will remove all nodes with coverage $\leq$value.

3. Remove low-coverage nodes which are more likely to be created by sequencing errors than by random sampling. See our paper for details. --remove_seq_errors.

By error correcting and then dumping a binary, we reduce the number of nodes in the graph, and therefore also the memory requirement.

4.2.6 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 due to sampling"). 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 method for error-correction by comparison with a population graph. The approach is to build one uncleaned graph per individual, then to pool them into one graph and error-clean or correct that, and then finally to clean each individual graph by comparison with the cleaned pool. Here is a step-by-step example; suppose we have 100 individuals each sampled at low coverage,all from the same species/population:

1. Build individual uncleaned graphs, as described elsewhere in this manual. (Use --dump_binary to produce binaries, named indiv_N.uncleaned.ctx)
2. 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_seq_errors, and dump a cleaned population pooled graph clean_pool.ctx
3. 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.

The commandline for step 3 is:
cortex_var_31_c2 --kmer_size 27 --mem_height <h> --mem_width <w> --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_to_add_to_filename_to_signify_binary_is_clean>

4.2.7 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:

1. 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 multi-colour 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), thus: --detect_bubbles1 i/i --output_bubbles1 <output filename>. This means that we look for bubbles in the graph where both branches/sides of the bubbles are present in colour i.

2. If we are lucky enough to have a reference genome for the species of interest, then we can do a good 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. We do this thus: --detect_bubbles1 i/i --output_bubbles1 <output filename> --ref_colour r.

3. Steps 1 and 2 above only find heterozygous sites, where the data from the individual (colour i) contains both alleles. If we have a reference genome (colour r) we can find homozygous non-reference sites thus: --detect_bubbles1 i/r --output_bubbles1 <output filename>. This looks for bubbles where one branch/allele is present in colour r (the reference), and *not* in colour i (the sample/individual) and the other is branch is present in colour i but not in colour r. There is therefore no benefit to specifying --ref_colour in this case.

4. It is commonly desirable to call both homozygous and heterozygous sites, and this is enabled thus: --detect_bubbles1 i/i --output_bubbles1 <output filename> --detect_bubbles2 r/i --output_bubbles2 <output filename>.

5. Suppose we had data from 10 haploid samples, each sequenced separately from isolates, and we want to find variants. We could load each into a different colour, and then look for bubbles: --detect_bubbles1 0,1,2,3,4,5,6,7,8,9/0,1,2,3,4,5,6,7,8,9 --output_bubbles1 <output filename> --print_colour_coverages. By adding the option --print_colour_coverages, the output also shows coverage of each allele in all colours, allowing one to process the output to see which variants are present in which individuals. One could get the same information with a slightly more elegant command-line by pooling *all* the data from the 10 samples in colour 0, and then having colours 1 to 10 for each sample individually, and then use the command-line: --detect_bubbles1 0/0 --output_bubbles1 <output filename> --print_colour_coverages. In other words - look for bubbles in the union of all our samples (colour 0), but then once found, print out how much information there is in each individual to support these variants.

6. Suppose we wanted to do a crude search for variants that distinguish two groups of samples - .e.g to variants distinguishing colours 0,1,2,3,4 from colours 5,6,7,8,9), then we would type: --detect_bubbles1 0,1,2,3,4/5,6,7,8,9 --output_bubbles1 <output filename> --print_colour_coverages. In general we expect to have to be more sophisticated than this, and look for a difference in allele frequencies between the two populations rather than complete presence/absence, and would do this by applying Item 5 above.

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

```
>var_1_5p_flank
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).

4.2.8 Variation discovery using the Path Divergence Caller

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.

If we have a list of fasta files (generally we have one fasta per chromosome in the reference), 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 <name of file listing the reference chromosome fasta>

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 <name of file listing the reference chromosome fasta>

One output file is created for each chromosome, and Cortex numbers the chromosomes 1...n in the order in which they are listed in the input list. 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).

4.2.9 Analysing variant calls and converting to VCF format

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 inversions - 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). We provide a handyscript to do just this, in the scripts/analyse_variants directory: process_bubbles.pl. This script has one dependency, for Algorithm::NeedlemanWunsch, which can be downloaded from CPAN. Performance can be slow when applied to Path Divergence Calls, which can have alleles which a tens of kilobases long. Typical usage would be

perl process_bubbles.pl file_of_variants

This goes through the file of called variants, ignores the flanks, and aligns the two alleles in each call. It also aligns one allele with the reverse complement of the other, but only prints the result if there is a significant alignment, to allow detection of inversions.

We also provide a second, more complicated and scrappy script for converting calls to VCF format (note VCF format by definition requires a reference genome to exist - variant calls cannot be put into VCF format without a reference against which to log coordinates) - called process_calls.pl. This script does the following:

1. Take a file of Cortex variant calls as input, specifying which caller made the calls. Requirement: must have used --print_colour_coverages
2. Map the 5prime flanks to a reference genome (internally, we use Stampy), filter out calls where the mapping has quality <30.
3. For each call, align the two branches (alleles) against each other, and parse the alignment to try to classify the call. This step has a tendency to falsely classify a small number of indels as inversions, and will be improved in future.
4. Filter all calls to ensure median coverage on both branches (for het calls) or the non-reference branch (for hom non-ref calls) is >=2.
5. Filter Path Divergence Calls as follows: nodes which are on the ref allele but not alt allele must have median covg=0. This is a very strict condition - remove it to allow much longer calls to pass the filter (up to ~100kb), with a somewhat increased chance of false positives.
6. Dump two vcf's. One (the "raw" vcf) just gives the two alleles as called by Cortex in the VCF file (apart from trimming off the end of both alleles if they are identical). The second (the "decomposed" vcf), attempts to split each call into it's constituent SNPs, indels etc.

4.3 Brief outline of graph building with cortex_con to highlight different cmd-line input

Summary: De Bruijn graphs can be built (from fasta or fastq), cleaned, and dumped as binary files. Binaries can be merged. Polymorphism can be removed

and contigs can be printed. File input specification is *different* to that for cortex_var, which is the reason we are describing cortex_con input here. *For reliable and up-to-date information on cortex_ con, consult its documentation at the project website.*

4.3.1 File input

cortex_con takes a list of fasta, fastq or Cortex binary graph files as input (the list must be homogeneous, not a mixture of fasta and fastq). The released version does not support read-pairs (the internal development version does), and so input is by a single list:

cortex_con --input_list <filename of list> --input_format <fasta|fastq|binary>

If inputting fasta/q, then it is mandatory to also specify --max_read_len (the length of the longest read to be read in - it is perfectly acceptable for this to be an overestimate, cortex just needs an upper-bound). Since sequencing machines can now dump extremely large fastq files, and since one may want to split jobs over a cluster, cortex_con supports building a binary just from a subset of the reads within a file. Therefore, instead of listing fastq files, one is permitted to list "fastq start_read end_read" (tab-separated).

5 Worked examples

We give some worked examples here. See also the demo/ directory within the release which contains several concrete examples for you to play with. Each example has its own directory, with a README within explaining what you should do, what you will see and how to interpret it,

5.1 Given a set of reads from a single individual: build a graph and get a consensus assembly (contigs)

Consult the cortex_con manual!

5.2 Building binaries and applying error-cleaning on a per-library basis

Suppose we have data from multiple libraries from a single individual. For best results, these should not be treated as one homogeneous set of reads, but each library should be processed separately (they are likely to have different characteristics and potentially require different error-correction). To build one graph per library, and then merge them: First make one fastq filelist per library and build a graph, as above, and then dump it as a binary.

cortex_con_31 --input_format fastq --input_file <fastq in library 1> --kmer_size <k> --mem_height <h> --mem_width <w> --dump_binary library1.ctx

Repeat for the other libraries. Apply error-correction to each library as you see fit. Then to merge them, make a list of all the library.ctx files, and merge:

```
cortex_con_31 --input_format binary --input_file <filelist of ctx>
    --kmer_size 29 --mem_height <h> --mem_width <w> --dump_binary
    merged.ctx
```

5.3 Accelerating graph-building by using a cluster of servers

The graph building process can be accelerated by parallelising across a cluster of servers. Simply divide the fastq into a subsets, and process each subset on a single node, dumping a binary from each one; then merge all of these binaries into a final graph. The latest versions of sequencing machines can produce enormous fastq files, and so cortex_con also supports splitting a single fastq - you can specify the index of the first and last reads within a fastq to load. For example, this might be a filelist for use on a single node of the cluster (tab separated):

```
first_fastq.fq 1 1000
```

and this for the second node:

```
first_fastq 1001 2000
```

etc. Each of these processes can be tailored (by choice of volume of input data) to use an amount of memory within the capacity of a cluster node. Finally, we merge these binaries on a single machine (core) which has sufficient RAM to support the genome plus sequencing errors.

5.4 Call heterozygous variants in a single individual by de novo assembly

We need first to have a de Bruijn graph generated from the sequence data of this individual. Suppose for concreteness we are interested in k=55. We might do this directly from the reads (suppose maximum read length is 100, again for concreteness):

```
cortex_var_63_c1 --se_list list_of_fastq --kmer_size 55 --mem_height
    h --mem_width w --max_read_len 100 --remove_seq_errors
```

Or we might have already built a single-colour binary somename.ctx, which we could load :

```
cortex_var_63_c1 --multicolour_bin somename.ctx --kmer_size 55
    --mem_height h --mem_width w --max_read_len 100
```

the –multicolour_bin option accepts any binary and can determine from the binary header the number of colours within. Despite the word “multicolour” in the option name “–multicolour_bin”, it also supports single colour binaries - apologies for any confusion. Finally, one might load that same binary as follows:

cortex_var_63_c1 --colour_list one_individual --kmer_size 55 --mem_height h--mem_width w --max_read_len 100

where

> cat one_individual

sample_name

>cat sample_name

somename.ctx

This is somewhat cumbersome for a single binary, –colour_list needs a list of colours, each of which contains a list of binaries.

From here (and in the same commandline), it is straightforward to call variants in the graph - just add the following to the end

--detect_bubbles1 0/0 --output_bubbles1 <output filename>

This looks for bubbles in the colour 0 graph, and prints them to the output file.

5.5 Call heterozygous variants in a single individual by de novo assembly, excluding repeats by using a reference genome

Build a binary ref.ctx of the reference genome, and a binary of the individual indiv.ctx. Load them into a two-colour de Bruijn graph as follows

> ls ref.ctx > reference_colour

> ls indiv.ctx > individual_colour

> ls *_colour > list_of_colours

[open list_of_colours in text editor and check the files are listed in the order you want.]

> cat list_of_colours

reference_colour

individual_colour

>cortex_var_31_c2 --colour_list list_of_colours --mem_height h --mem_width w --ref_colour 0 --detect_bubbles1 1/1 --output_bubbles1 <output filename>

This tells Cortex that colour 0 is the reference, so when --detect_bubbles1 is called, it first detects bubbles in colour 0, and excludes them as repeats. It then detects bubbles in colour 1, and prints them to the output file.

5.6 Given a trio of individuals, find variants present in the child but neither parent

Load the mother, father, child into colours 0,1,2 as described above. Then call bubbles as follows:

--detect_bubbles1 0,1/2 --output_bubbles1 <output filename>

This looks for bubbles in the graph, completely ignoring colours (ie it searches in the union of all colours). It then requires that one branch of a bubble be in colour 0 or 1 and have zero coverage in colour 2, and the other branch be there in colour 2 and have zero coverage in colour 0 and 1.

6 Frequently Asked Questions (FAQ)

6.1 What does "too much rehashing" mean?

It means you have specified too small a hash table, and so there is insufficient memory to hold all your data. Rerun with a bigger combination of mem_height and mem_width (but check this will fit in the memory available to your server). *Be warned that if working on a shared server, Linux is capable of allowing you and another user to allocate between you more memory than is available on the machine. Linux assumes people do not really use as much as they allocate.*

6.2 How do I work out how much coverage has been filtered away by PCR duplicate remove, quality filtering, homopolymer filtering?

Cortex does this for you, but only does so for fastq files (since it has to support fasta files where a single read may be 200Mb long, for reference chromosomes, and since real data which requires filtering is always fastq, we only support getting these statistics for fastq). Firstly, when you load a set of fastq files cortex prints out something like this:

```
**************************************
SUMMARY:
Colour: MeanReadLen TotalSeq
0 94 63198788
**************************************
```

This is telling you the mean read length after filtering/cutting reads, and the total number of base pairs loaded after filtering. If you need to know the full distribution of filtered read lengths, use --dump_filtered_readlen_distribution when loading the data.

6.3 Can I find out what k-mer coverage distribution looks like?

Yes, use --dump_covg_distribution when you load the data.

6.4 I want to use fastq as dumped directly by my Illumina machine, and I know there is something different about their fastq format. What do I do?

Find out what ASCII offset is used by your version of the Illumina pipeline. Often the appropriate setting for Illumina data is --quality_offset 64; by default Cortex assumes the standard/official/Sanger value of 33.

6.5 When will Cortex get read-pair support?

Read-pair support for cortex_var is in development. cortex_con already supports read-pairs - see the cortex_con manual for details of how to use it for consensus assembly.

7 Outstanding issues/bugs

1. Cortex apparently does not compile for the Intel compiler on IA64. We'll fix this a.s.a.p. It compiles and is tested on gcc, on Linux 64-bit and Mac OS X.

2. The scripts for parsing Cortex variant calls and turning them into VCF are not as polished as we'd like - stay tuned for updates soon

8 Citing Cortex, and further reading

If you publish results dependent on use of Cortex, please cite our paper

> "De novo assembly and genotyping of variants using coloured de Bruijn graphs" Iqbal(*), Caccamo(*), Flicek, McVean

If you want to read further details about Cortex and the algorithms it uses, see that paper.

9 Contact Us

For any questions about cortex_var, please contact me (Zam Iqbal) at zam@well.ox.ac.uk.
For questions regarding cortex_con please consult the cortex_con documentation at cortexassembler.sourceforge.net, or contact Mario Caccamo at mario.caccamo@bbsrc.ac.uk.
For questions about Cortex in general, feel free to contact either/both of us.