

cortex_var User Manual

Zamin Iqbal

October 31, 2011

Contents

1	General Introduction	2
2	Command-line interface	3
2.1	cortex_var command-line interface	3
3	Compilation	8
4	Usage of cortex_var	8
4.1	Introduction	8
4.2	Variation and population analysis with cortex_var	8
4.2.1	File Input	9
4.2.2	Filtering of input sequence data	11
4.2.3	Choosing hash table size	11
4.2.4	Filtering of input sequence data	13
4.2.5	Error Cleaning	13
4.2.6	Error-cleaning low coverage samples when you have many samples from the same species/population	14
4.2.7	Variation Discovery using the Bubble Caller	15
4.2.8	Variation discovery using the Path Divergence Caller . . .	17
4.2.9	Genotype calling	18
4.2.10	Analysing variant calls and converting to VCF format . .	19
4.2.11	Classification of bubbles as VARIANT, REPEAT or ER- ROR	20
5	Worked examples	20
5.1	Given a set of reads from a single individual: build a graph and get a consensus assembly (contigs)	20
5.2	Building binaries and applying error-cleaning on a per-library basis	21
5.3	Accelerating graph-building by using a cluster of servers	21
5.4	Call putative heterozygous variants in a single individual by de novo assembly	21
5.5	Call heterozygous variants in a single individual by de novo as- sembly, excluding repeats by using a reference genome	23

5.6	Given a trio of individuals, find variants present in the child but neither parent	23
5.7	Given sequencing data from 100 high coverage diploid eukaryotes from a single population, for which we do not have a reference genome, call high quality variants which will not be conflated with paralogs and pseudogenes	24
5.8	Given sequencing data from 100 LOW coverage diploid eukaryotes from a single population, for which we do not have a reference genome, call high quality variants which will not be conflated with paralogs and pseudogenes	24
6	Frequently Asked Questions (FAQ)	24
6.1	What does “too much rehashing” mean?	24
6.2	How do I work out how much coverage has been filtered away by PCR duplicate remove, quality filtering, homopolymer filtering? .	24
6.3	Can I find out what k-mer coverage distribution looks like? . . .	25
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?	25
6.5	When will Cortex get read-pair support?	25
7	Outstanding issues/bugs	25
8	Citing Cortex, and further reading	25
9	Contact Us	26

1 General Introduction

Cortex is a software suite for de Bruijn genome assembly, consisting 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. Some highlights of what cortex_var can do:

- **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.
- accesses higher proportion of genome than mapping-based approaches - the most accurate current method for 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) - a level of validation higher than on any other variant caller we are aware of, including the 1000 Genomes calls)
- 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

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.

2 Command-line interface

2.1 cortex_var command-line interface

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

DATA LOADING OPTIONS

- [--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
- [--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 $2^{\text{(the number you enter)}}$
- [--fastq_offset INT] = Default 33, for standard fastq. Some fastq directly from different versions of Illumina machines require different offsets.
- [--dump_binary FILENAME] = Dump a binary file, with this name (after applying error-cleaning, if specified).
- [--max_read_len] = For fastq, this is the Maximum read length over all input files. For fasta it is the size of chunk in which the reads are read (if reading a whole chromosome a typical value to use is 10000. Values above 20000 forbidden. (Mandatory if fastq or fasta files are input.)

FILTERING AND ERROR-CLEANING OPTIONS

- [--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. (Input of sequence restarts after homopolymer)

[--remove_low_coverage_supernodes INT] = Remove all supernodes where max coverage is <= the limit you set. Overrides --remove_seq_errors. Recommended method.

[--remove_seq_errors] = Remove tips + remove supernodes with coverage everywhere ==1.

[--remove_low_coverage_kmers INT] = Filter for kmers with coverage less than or equal to threshold. Not recommended. See manual and our 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 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.

OUTPUT STATISTICS AND SUPERNODES

[--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 (after applying all specified actions on graph).

VARIANT CALLING

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

[--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.

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

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

`[--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 accommodate 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.

`[--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`

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

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

ADVANCED OPTIONS

`[--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 (`EachColourADiploidSample`, `EachColourADiploidSampleExceptTheRefColour`, `EachColourAHaploidSample`, `EachColourAHaploidSampleExceptTheRefColour`). This is only needed for determining likelihoods, so ignore this if you are pooling samples within a colour (support to be added for this later).

- [--estimated_error_rate] = If you have some idea of the sequencing error rate (per base-pair), enter it here. eg 0.01. Currently used in calculating likelihoods
- [--genome_size] = If you specify --experiment_type, and therefore want to calculate likelihoods, you must also specify the (estimated) genome size in bp.
- [--align FILENAME1,<FILENAME2|no>] = Aligns a list FILENAME1 of fasta/q files to the graph, and prints coverage of each kmer in each read in each colour. During alignment, any node that it touched by the aligned reads is marked. If FILENAME2 is specified, a Cortex binary graph is dumped of all marked nodes in the main graph - this allows the user to pull out subgraphs of regions of interest. 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

OPTIONS THAT I CREATED BUT IT'S POSSIBLE NOONE HAS A USE FOR

- [--colour_overlaps COMMA_SEP_COLOURS/COMMA_SEP_COLOURS] = 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 in both colour-i (on first list) and colour-j (on second list). This was intended to allow clustering of colours according to how much sequence they share.

EARLY ACCESS/BETA OPTIONS

- [--genotype_site] = (Beta code!) Genotype a single complex (typically multi-allelic) 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 pair 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 alleles 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 disarding any genotype with log-likelihood<MIN. See manual for details. Must also specify --max_var_len to give the length of the longest allele

3 Compilation

You will need to compile the GNU Scientific Library, which I have bundled with Cortex (both released under GPL). First go to the `gsl-1.15` directory and type `make`. If that fails with an error like this “`gsl-1.15/gsl/gsl_sf_gamma.h:25: fatal error: gsl/gsl_sf_result.h: No such file or directory`” then contact your system administrator, and ask them to install `gsl-bin`, `libgsl0ldbl`, `libgsl0-dev`. Then return to the root directory,

To build an executable that supports $k \leq 31$ 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 $k \leq 31$ and n colours:

```
make NUM_COLS=n cortex_var
```

produces an executable called `cortex_var_31_cn`. To build an executable that supports $33 \leq k \leq 63$ 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_var`

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 $n=31,63,95,\dots$, and $m=1,2,3,\dots$), 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 and genotyping are given in our paper, “De novo assembly and genotyping of variants using colored de Bruijn Graphs”, in *Nature Genetics* (in press). `cortex_con` focuses on consensus assembly (hence the name) and is not discussed here, 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_var` 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 $2^h * w \approx 2 * (\text{length of genome})$. 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. $2^{16} * 75 \approx 4.9$ 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 (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 $k \leq 31$, 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 $(8 + 5 + 1) \times 6 \times 10^9 \text{bytes} = 84 \text{Gigabytes}$ 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 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 2 means of error-cleaning:

1. 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. As we describe in our paper, we do **not** recommend this method, it's a blunt instrument and creates gaps in the assembly wh
2. `--remove_low_coverage_supernodes N`. This is the recommended option, removing 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.

3. This is not really a third option: `--remove_seq_errors` - does tip-clipping, and plus `--remove_low_coverage_supernodes` 1.

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 - you only keep kmers in the individual if they are also in the 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 reasonable 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 --exclude_ref_bubbles`.
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 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>`. I'm not very keen on this, as it conflates discovery with genotyping. A bubble can appear to be homozygous, but actually this is due to a coverage gap (or a small amount of coverage which was removed by error-cleaning). So I recommend instead you just do this: `--detect_bubbles1 r,i/r,i --output_bubbles1 <output filename>`. This will call all bubbles in the union graph of the reference and individual. You can leave genotyping to the next step (described below).
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
```


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 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. However for fasta this is not preserved currently (essentially because fasta "reads" are often full chromosomes in typical usage) - the bottom line is that **Cortex will currently only genotype calls made from FASTQ, not from FASTA**. 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 HOM2 -nan -nan -nan
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.

4.2.10 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 occurred 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 are 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. 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.

Unfortunately the current version of process_calls.pl requires you to manually edit it to contain the path to your Stampy binary and Stampy hash.

4.2.11 Classification of bubbles as VARIANT, REPEAT or ERROR

We describe in our paper our statistical model for classifying structures in the graph. We provide an R script, classify.R, for classifying variant calls - this will be integrated into the main executable in future releases. How to apply the filter

1. First, run a perl script which parses the Cortex callfile, and makes a single tab-separated file of calls with information on coverage on each allele in each sample. If one of your colours is the reference, the script takes this as the final argument (enter -1 if none of the colours is the reference). Command-line looks like this: perl /path/to/cortex release/scripts/analyse_variants/make_covg_file <output of Bubble Caller> <number of colours in graph> <number of colour which is reference>. This makes a file with filetype : .covg_for_classifier .
2. Open the script /path/to/cortex release/scripts/analyse_variants/classifier.R, and modify the lines marked with "<<<<MODIFY", to contain the relevant coverage, read-length, kmer information etc. Then run the R script.

The output of the classifier looks like this: tabe 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
```

As mentioned above, this will be integrated into the main body of Cortex in the future, along with the means to take the results of the classifier and merge into your VCF.

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 putative heterozygous variants in a single individual by de novo assembly

It is important to distinguish between **discovery** and **genotyping**. Looking for bubbles is discovery; if we give Cortex relevant information (estimated genome length, whether or not our species is haploid or diploid), then it will apply our

statistical model to genotype a site. Why is this important ? Sometimes a bubble, which is putatively heterozygous, will be caused by a sequencing error - the model allows us to determine likelihoods of the various possible genotypes. Similarly, if one were calling bubbles in a graph with a sample and a reference, one could call a putatively homozygous bubble (sample only has one allele) - but that might be due to a coverage gap or due to error cleaning.

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.

If we want genotype calls, then we need to add extra information:

```
--detect_bubbles1 0/0 --output_bubbles1 <output filename> --experiment_type
EachColourADiploidSample --genome_size 3000000000 --estimated_error_rate
0.005
```

If you do not enter an estimated error rate, a default of 0.01 is used.

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.

If we want genotype calls too, then we need to add the extra commands

```
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> --experiment_type EachColourADiploidSam-
pleExceptTheRefColour --genome_size 3000000000 --estimated_error_rate
0.005
```

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. If you in addition genotype the call, you can use genotype confidence as a means for stratifying these calls and seeing if you believe whether they are true novel mutations or just sequencing errors.

5.7 Given sequencing data from 100 high coverage diploid eukaryotes from a single population, for which we do not have a reference genome, call high quality variants which will not be conflated with paralogs and pseudogenes

For each sample, build a graph and clean it individually. Collect them into a single multicolour graph. Call bubbles on the joint graph, call genotypes, make a VCF. Apply the probabilistic classifier to determine if bubbles and polymorphisms, repeats or errors, and integrate these results into your VCF.

5.8 Given sequencing data from 100 LOW coverage diploid eukaryotes from a single population, for which we do not have a reference genome, call high quality variants which will not be conflated with paralogs and pseudogenes

For each sample, build a graph (no cleaning). Merge all of these into a single-colour pool and clean it. Then clean all the sample graphs against the pool. Then make a single multicolour graph containing all the cleaned sample graphs each in their own colour. Call bubbles on the joint graph, call genotypes, make a VCF. Apply the probabilistic classifier to determine if bubbles and polymorphisms, repeats or errors, and integrate these results into your VCF.

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 - coming soon! Please do not email me asking if it is worth waiting for read-pair support before you apply Cortex - it's not worth waiting. Even without read-pairs, Cortex is able to make calls that other callers cannot, and has an extremely low FDR - so it already has a value. When I release the read-pair code that should increase sensitivity, but you can get value out of this straight-away.

7 Outstanding issues/bugs

1. Cortex 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 colored de Bruijn graphs” Z. Iqbal(*), M. Caccamo(*), I. Turner, P. Flicek, G. McVean (Nature Genetics) (in press)

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.