Supplement to Kroll et al., *MuCor: Mutation Aggregation and Correlation*

Correspondence:

James S. Blachly, M.D.
james.blachly@osumc.edu

Contents

Supplementary Text

Outline of workflow

MuCor is intended to be run in two steps. A companion utility, DepthGauge, is optional.

In the first step, **`mucor_config.py`** generates a configuration file either automatically (inferring certain values; see below), or as a syntactically-valid but blank template for editing. The configuration file is a JSON[1] format document that can be edited by hand if required.

In the second step, **`mucor.py`** reads the contents of the configuration file and executes the run. No additional parameters are required at this step.

Finally, to ensure the veracity of variant and wild type calls in the input space, **`depth_gauge.py`** can use the same configuration file as MuCor to query the original source BAM files, if available, and demonstrate adequate sequencing depth (or lack thereof) at points of interest: for example, at the site of one or more interesting variants in the MuCor results.

[1] JavaScript Object Notation. http://json.org.
http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

Usage and Description of Arguments: `mucor_config.py`

```
mucor_config.py [-h]
                [-ex]
                -g GFF
                -f FEATURETYPE
                [-db <dbname:/path/to/db.vcf.gz>]
                -s SAMPLES
                [-d PROJECT_DIRECTORY]
                [-vcff VCF_FILTERS]
                [-a ARCHIVE_DIRECTORY]
                [-r REGIONS]
                [-u]
                -jco JSON_CONFIG_OUTPUT
                -outd OUTPUT_DIRECTORY
                [-outt OUTPUT_TYPE]
```

Mandatory arguments:

Note that in `-ex` (example) mode, no other parameters are required.

`-g GFF, --gff GFF` Annotation GFF/GTF for feature binning

Specify a GTF/GFF3 format file with feature definitions. Each identified variant will be binned into and reported as belonging to one (or more, in case of genomic overlap) feature(s). Typically, this would be genes, but could be anything (operons, cytogenetic bands, etc.)

`-f FEATURETYPE, --featuretype FEATURETYPE`

Specify the feature type (a key from GTF or GFF3 column 9). For example, from the GENCODE annotations, choices include: `gene_name`, `gene_id`, `transcript_name`, `transcript_id`, etc.

`-s <sample_list.txt>, --samples <sample_list.txt>`

This parameter points to a file name containing sample names, with one sample name per line. An example is given below (Listing 1). Each sample name or ID read from this file will be matched against filenames in subdirectories of the directory specified in --`project_directory`. Note that this means *in rare cases, the substring matching algorithm might incorrectly associate a file with a sample ID*, particularly if sample IDs and filenames are not padded with zeros. For example, the ID `sample9` would match the [incorrect] file name `sample99`, but *not* the file name `sample09` ! The ID `sample1` would match filenames for both `sample1` and `sample10` . This problem can be avoided by using appropriately-padded sample identifiers and filenames, or by using a scheme such as UUIDs (practiced by the TCGA, among others).

```
lung_cancer_001
lung_cancer_002
lung_cancer_003
lung_cancer_004
lung_cancer_005
```

Listing 1. Example `sample_list.txt`

`-jco <filename.json>, --json_config_output <filename.json>`

This required parameter names the output configuration file, which is in JSON format.

`-outd <OUTPUT_DIRECTORY>, --output_directory <OUTPUT_DIRECTORY>`

This required parameter defines the directory (in relative or absolute terms) where output will be placed during and after a MuCor run.

Optional arguments:

`-h, --help` show a help message and exit

`-ex, --example` Write a valid, example JSON config file and exit.

`-db <dbname:/path/to/file.vcf.gz>,`

`--databases <dbname:/path/to/file.vcf.gz>`

This parameter is used to define lookup databases, and can be passed zero or more times. Databases are specified as a colon-delimited pair of the user-defined database name, and full path (no wildcard or ~ expansion) to the bgzipped, tabix-indexed (see Supplementary Table 1) VCF file. Passing this parameter multiple times will include multiple lookup databases simultaneously in the analysis.

Examples of lookup databases that might be used include dbSNP and COSMIC. One could also define databases with arbitrary identifiers associated with specific nucleotide changes of interest.

`-d <dirname>, --project_directory <dirname>`

This parameter specifies the working or project directory in which the configurator can find input variant call files. It defaults to the current working directory.

`-vcff <FILTERS>, --vcf_filters <FILTERS>`

This parameter specifies a comma-separated list of VCF filters for which data will be allowed to pass through. It defaults to the PASS filter.

`-a ARCHIVE_DIRECTORY, --archive_directory ARCHIVE_DIRECTORY`

This parameter specifies a directory in which MuCor can read and write archived annotations. The GTF/GFF files can take some time to process, and for very small input VCFs this parsing might consume a large fraction of total processing time. By specifying an archive directory, the parsed annotation file can serialized to disk and in future runs can be rapidly deserialized for faster startup.

`-r REGIONS, --regions REGIONS`

This parameter limits analysis to a list of regions specified either on the command line or in a BED[2] file. For example, in whole-genome or whole-exome sequencing, voluminous data make analysis difficult. Limiting MuCor output to a pre-specified list of “hotspots” can facilitate more rapid analysis.

Format: a comma-separated list of any combination of

1. Regions in the format `<chr>[:<start>-<end>]`
2. BED filenames

[2] BED file format is documented at UCSC and Ensembl:
https://genome.ucsc.edu/FAQ/FAQformat.html#format1
http://www.ensembl.org/info/website/upload/bed.html

For example:

```
-r chr1:10230-10240,chr7,hotspots.bed,other_regions.bed
```

-u, --union Join all items with same ID for feature_type

This parameter joins all features (from the annotation passed using `-g`) with the same feature ID (specified with `-f`) into a single, continuous bin. For example, if you want intronic variants counted in your genes, use this option. However, be warned that this may lead to some spurious results: many annotations have entries (often small RNAs) annotated at multiple locations throughout the genome; when two or more entries are located on the same chromosome or contig, this can result in one huge unified bin. For example, MIR4283-2 exists twice on the + and – strand of the same chromosome over 1 megabase apart. MuCor contains some code to detect and eliminate wrongly unified bins, but not all cases are currently handled, nor could they be.

-outt OUTPUT_TYPE, --output_type OUTPUT_TYPE

This parameter specifies a comma-separated list of desired output types. It defaults to ‘all’. See Supplementary Table 2 for a list of acceptable values.

Usage and description of arguments: `mucor.py`

MuCor is dependent only on the JSON formatted configuration file which may be entirely hand created, entirely automatically generated by `mucor_config.py`, or automatically generated and hand-tuned.

```
mucor.py config_file.json
```

Usage and description of arguments: `depth_gauge.py`

Like MuCor, DepthGauge is dependent only on the JSON formatted configuration file which may be entirely hand created, entirely automatically generated by `mucor_config.py`, or automatically generated and hand-tuned. DepthGauge has three additional options, the latter of which can override the regions, if any, specified in the JSON configuration file.

```
depth_gauge.py config_file.json [-p] [–c] [-r <regions>]
```

`-p, --point`

Instead of reporting an average depth for every location within a window (default), take the middle coordinate within the range and calculate the depth at that point as a surrogate for the entire region

`-c, --count`

Instead of reporting an average depth for every location within a window (default), or taking a point estimate from the middle (-p), instead count the total number of reads mapped within the region.

`-r <regions>`

This option specifies a list of regions to query for depth. If present, it overrides the region(s) specified in the JSON configuration file. As an alternative, the JSON configuration file could be edited before running DepthGauge.

Example Workflow 1: Compare calls in multiple samples

In this example, suppose we have many independent samples for which variants have been called with a single variant caller. Perhaps these variant calls came directly from an Ion Torrent PGM or Miseq instrument as VCF files. Suppose further that the sample names consist of `sample` + a four digit number. For example, they might be called `sample0107` or `sample1234`.

Step 1. Create a text file with each sample name/id on a separate line.

If the sample names are serial and there are many, you may wish to create the list programmatically. In bash:

```
for ((i=1;i<=10;i++); do
     printf "sample%04d\n" ${i} >> samples.txt
done
```

This example will save a file `samples.txt` that contains ten serial sample IDs.

Step 2. Ensure all variant files are stored under a single project directory.

The files needn't be all together in the same parent directory or same subdirectory, so long as they are accessible in the file system tree underneath the project directory. For example, either of the following is compatible with the sampleid->filename auto-detection:

```
projA/
├── sample01.vcf
├── sample02.vcf
└── sample03.vcf

projB/
├── sample01/
│   └── sample01.vcf
├── sample02/
│   └── sample02.vcf
└── sample03/
    └── sample03.vcf
```

Step 3. Run mucor_config.py with appropriate arguments.

```
mucor_config.py -g /data/ref/gencode.gtf -f gene_id -s            \
samples.txt -jco example1.json -outd example1/ -outt all -db      \
dbsnp:/data/ref/dbsnp.vcf.gz
```

Step 4. Run MuCor and examine output files of interest.

```
mucor.py example1.json
```

Output files will reside in `example1/`. For this study, outputs txt/xlsx and featXsamp may be of interest.

Example Workflow 2: Compare calls from multiple callers in multiple samples

This workflow would proceed exactly as in Example Workflow 1, except that any sample might (but is not required to) have multiple variant files associated with it. For example, suppose that MuTect were used to call somatic mutations, and Varscan to detect INDELs. Each subdirectory in the tree above might have multiple files:

```
proj/
├── sample01/
│   ├── sample01_mutect.vcf
│   └── sample01_varscan.vcf
├── sample02/
│   ├── sample02_mutect.vcf
│   └── sample02_varscan.vcf
└── sample03/
    ├── sample03_mutect.vcf
    └── sample03_varscan.vcf
```

As before, the files needn't be separated into individual subdirectories and will be correctly auto-detected, even if they are all in the same directory. Note that filename-based auto-detection is based solely on sample id, and has nothing to do with the individual variant caller or the sub-type of VCF file. The filenames in the example above included `_mutect` or `_varscan` only for clarity's sake.

No other changes to the work flow in Example Workflow 1 are necessary; multiple inputs per sample will be merged internally.

Example Workflow 3: Compare two or more variant callers in a single sample

For this example, suppose we wish to compare the results of three different variant calling pipelines or packages when run on a single sample. Now, unlike in example workflow 2, wherein we wanted multiple callers within a single sample to be merged internally, here we aim to keep them separate. To do this, we can either build the JSON formatted config file manually, specifying which VCF files belong to which test conditions, or we can try to build it automatically with some trickery: in order to prevent the sample ID autodetection routine, which operates based on filename, from merging all pipeline outputs into a single sample, we'll need to rename files, removing reference to sample ID in the filename.

Step 1. Create a sample list text file consisting of a single sample ID.

```
cat sample01 > samples.txt
```

Step 2. Rename files, removing reference to sample ID in the filename.

```
proj/                                        proj/
└── sample01/                                └── sample01/
    ├── sample01_pipelineA.vcf   rename          ├── pipelineA.vcf
    ├── sample01_pipelineB.vcf     →             ├── pipelineB.vcf
    └── sample01_pipelineC.vcf                   └── pipelineC.vcf
```

Step 3. Run mucor_config.py with appropriate arguments.

No specific change to the configurator compared to example workflow 1 is required. Generate example3.json by running the configuration script with the file tree as above. Note that now when examining `example3.json` (see Listing 2, below, page 13), in contrast to example workflow 2, each pipeline is listed as a separate sample (Inote at the "`id`" field for each element in the `samples` array). By extension and in contrast to Listing 2, the samples file passed to `mucor_config.py` would have the entries `A, B, C` on three separate rows.

Step 4. Run MuCor and examine output files of interest.

```
mucor.py example3.json
```

Output files will reside in `example3/`. For this study, outputs mutXsamp and mutXsampVAF may be of interest, as they will provide a direct comparison on a variant-by-variant basis among the different pipelines. Supplementary Figure 1, below, page 14, demonstrates the mutXsampVAF report as a head-to-head comparison of two variant callers.

```
{
    "databases": {},
    "fast": "",
    "feature": "gene_name",
    "filters": [
        "PASS"
    ],
    "gff": "/home/references/annotation.gtf",
    "outputDir": "./example3",
    "outputFormats": [
        "all"
    ],
    "regions": [],
    "samples": [
        {
            "files": [
                {
                    "path": "/proj/sample01/pipelineA.vcf",
                    "snpeff": true,
                    "source": "pipelineA",
                    "type": "vcf"
                }
            ],
            "id": "A"
        },
        {
            "files": [
                {
                    "path": "/proj/sample01/pipelineB.vcf",
                    "snpeff": true,
                    "source": "pipelineB",
                    "type": "vcf"
                }
            ],
            "id": "B"
        },
        {
            "files": [
                {
                    "path": "/proj/sample01/pipelineC.vcf",
                    "snpeff": true,
                    "source": "pipelineC",
                    "type": "vcf"
                }
            ],
            "id": "C"
        }
    ],
    "union": false
}
```

Listing 2. `example3.json`

					CallerA	CallerB
feature	**chr**	**pos**	**ref**	**alt**		
BRAF	**chr7**	**140434647**	**G**	**A**	1	1
		140449071	**C**	**G**	0.602	0.6083
		140449150	**T**	**C**	0.571	0.6
CTNNB1	**chr3**	**41274764**	**C**	**A**	0	0.9966
		41280641	**T**	**C**	0	0.016
GSK3B	**chr3**	**119541539**	**A**	**G**	0.03	0
		119541588	**A**	**G**	0.015	0.0174
		119542377	**A**	**C**	0	0.0112
		119542716	**C**	**T**	0.977	0.9778
		119542765	**T**	**A**	0.098	0
		119543921	**T**	**C**	1	1
		119544144	**G**	**T**	1	1
		119545152	**C**	**T**	0.995	0.9949
		119595503	**A**	**G**	1	1
		119631814	**A**	**G**	0.996	0.9962
		119812422	**T**	**C**	0.013	0.0107
		119812450	**A**	**C**	0	0.0113
MAPK1	**chr22**	**22160384**	**T**	**C**	1	1
		22162145	**G**	**A**	0.02	0
MAPK3	**chr16**	**30125854**	**T**	**G**	0.208	0
		30128294	**A**	**G**	0.026	0
NRAS	**chr1**	**115256669**	**G**	**A**	1	1
PIK3CD	**chr1**	**9780659**	**T**	**G**	0.42	0
		9780667	**A**	**C**	0.353	0
		9780669	**A**	**G**	0.341	0
		9783937	**A**	**T**	0.009514	0

Supplementary Figure 1. Direct Comparison of two variant callers.

This direct comparison of two variant callers uses the *mutXsampVAF* reporting format. Any number of callers could be run simultaneously and would be displayed as additional columns.

Example Workflow 4: Compare two platforms in a single sample

This example does not differ conceptually from example workflow 3, except that the source of inputs may result from the same variant calling pipeline but different platforms. For example, the same library might be run on an Illumina HiSeq and an Illumina MiSeq. Or, identical DNA may be run on both an Illumina MiSeq and an Ion Torrent instrument.

Make sure to either construct the JSON config file correctly by hand or by naming files appropriately (i.e., removing reference to any sample ID listed in the sample list input) before running the configurator.

Example Workflow 5: Be confident about wild type calls with DepthGauge

This example begins with the same steps as in workflow 1 or 2.

Suppose that 100 cases were sequenced and then, using MuCor, that two interesting variants were revealed to be present in 20% of samples. To be confident that we are not making false negative wild type calls in any of the remaining 80% of cases, we may wish to ensure that there was adequate sequencing depth at the two loci of interest in the remaining 80% of cases.

Step 1. Create a BED file listing the hotspot(s)

```
chr16	85909136	85909137	rs397514710
chr16	85909052	85909053	rs397514711
```

Listing 3. `hotspots.bed`

Remember that BED file coordinates are zero-based, half-open. In addition, it may be helpful to expand coordinates around a hotspot ±3 to catch other changes within the same codon.

Step 2. Use DepthGauge to report the total depth at each position within the regions of interest.

```
depth_gauge.py example5.json –r hotspots.bed
```

Alternatively, `example5.json` could be modified to limit further analysis to the regions of interest; the `-r` command-line parameter would then not be necessary when running DepthGauge.

Step 3. Examine DepthGauge output and confirm or refute MuCor's "wild type" calls

chr	start	stop	name	S01	S02	S03	S04	S05	S06
chr16	85909136	85909137	rs397514710	5268	5266	5266	4410	3628	5268
chr16	85909052	85909053	rs397514711	34	10	32	32	16	36

Supplementary Figure 2. DepthGauge output with average coverage for 2 locations of interest.

In Supplementary Figure 2, we see that we can be highly confident about wild type/variant status called for SNV rs397514710, but much less confident about the wild type/variant status of rs397514711. We may decide on the basis of these results to re-sequence, to eliminate that SNV from final analysis, or to analyze it as a ternary variable: *wild type*, *mutant*, and, for cases with fewer than some threshold number of reads, *unknown*.)

Example Workflow 6: Survey an amplicon panel for adequate coverage

It is not necessary to run MuCor *before* DepthGauge; this order can be reversed if required.

For this workflow, we have a targeted amplicon panel and wish to ensure that a run under consideration for evaluation (perhaps by MuCor) was successful in terms of amplification and sequencing. That is, in targeted amplicon sequencing, particular regions can occasionally fail to amplify; this may be due to GC-bias, genomic deletions, SNVs, or other technical factors. Before evaluating these data, it could be prudent to review the amplicons' coverage.

However, in targeted amplicon sequencing, coverage is generally uniform across a region of interest. Therefore, reporting individual depth at every point is redundant, and (naïvely) calculating the average depth is too time-consuming. Additionally, in paired-end sequencing of shorter amplicons, overlapping read pairs falsely increase (double) the sense of depth at the middle of the region, making a point-estimate (the `-p` option) misleading (and may differ between amplicons of different length). The `-c/--count` option was specifically designed for this scenario.

For this example, begin with the workflow from example 1 by organizing files (Step 1 and Step 2).

Modify Step 3 by limiting to regions specific to your amplicon design. This is usually supplied by your vendor. For Illumina TruSeq Custom Amplicon, the CAT Manifest file contains both targets and off-target regions; either can be parsed into a BED file. Ion Torrent Ampliseq designs may already include a BED file.

```
mucor_config.py […arguments from example 1, step 3] -r amplicons.bed
```

Step 4. Run DepthGauge in count mode

```
depth_gauge.py -c example6.json
```

Examine the output file, `Depth_of_Coverage-c.xlsx`. An example demonstrating an amplicon failure is shown in Supplementary Figure 3, below. Based on the results of this analysis, the user may wish to remove the suspect region from the region of interest BED file before running MuCor.

chr	start	stop	name	Sample_1	Sample_2	Sample_3
chrX	47428092	47428457	A-RAF	996	1,081	623
chr7	140453075	140453193	BRAF_Exon_2290058	3,182	3,735	4,939
chr19	33792516	33792965	CEBPA_Cds (35504885)_65295834	3	1	0
chr22	22160145	22160147	MAPK1_Exon_1839796	4,216	3,864	1,626
chr15	90631819	90631979	IDH2_Exon_2130155	1,902	287	346
chr5	170837490	170837895	NPM1_Cds (36346856)_65295889	2,990	2,277	952
chr12	25398280	25398285	KRAS:Exon 2 codons G12 and G13	4,948	4,949	4,761
chr12	25380275	25380278	KRAS:Exon 3, codon Q61	5,203	4,839	3,240

Supplementary Figure 3. DepthGauge output demonstrating an amplicon failure.

Writing New Input and Output Modules

The core functionality underlying MuCor's input, sorting, decoration, aggregation, and reporting is the excellent Python data analysis library Pandas (http://pandas.pydata.org/). Much of Pandas' core functionality is based around the DataFrame; MuCor uses `pandas.DataFrame` as its key internal data structure. All input flowing into MuCor must be written into a master data frame. Additional annotations from databases etc. are written as new columns to the data frame, and finally, getting output back out is a matter of slicing, dicing, aggregating, and reshaping the data frame. With this background in mind, we will briefly discuss the development of new input and output modules for MuCor.

New Input Modules

Step 1. Write Input Module

This input module will consist of a function that processes rows from a variant file and returns objects of class `variant` (see definition in `variant.py`). First, create a new file, `NewInputType.py`; it should at a minimum import the `Variant` class from the `variant` module, and the `HTSeq` module. In this file, define a function, `ParseNewType(self, Parser)`.

Now, within this function examine the Parser object; it has member variables `source`, `row`, `fieldId`, `header`, `fn`, `eff`, and `fc`, not all of which may be populated. For example, if a supported variant effect predictor has not decorated the file, `eff` (effect) and `fc` (functional consequence) will be empty; certainly, you may build your own variant effect prediction parsing into your function, instead of relying on what is being passed to the parser from `parseVariantFiles`; the SnpEff parsing code will eventually be moved from `parseVariantFiles` into the parsing modules where it belongs. Parse each row however appropriate, store the results in an instance of class `Variant`, and return it.

The calling function will deal with storing the returned `Variant` in the master data frame.

Place your new source file in the same directory as the rest of the MuCor code.

Step 2. Import Input Module

The first step in integration of your new input module is to import the new module and its attendant function(s). For this example, add `import NewInputType` to the top of `inputs.py`.

Step 3. Add Input Function to Supported Formats

The new format must be added to the list of supported input types. This is used to identify which parser should be used to generate the Variant object. The name given here must exactly match the 'source' field in the JSON configuration file (see point 2. d). These are typically named after the variant caller used, so we will refer to it as, "`NewCaller`." The following line would be added to the end of the Parser initialization function:

```
self.supported_formats["NewCaller"] = self.ParseNewType
```

Step 4. Bind Input Function to Parser

Next, the new function must be bound to the `Parser` object. Again following the example, the new function is called `ParseNewType`. One would need to add the following line to the end of the Parser initialization function:

```
self.ParseNewType = NewInputType.ParseNewType.__get__(self,Parser)
```

Step 5. Add New Format to Configurator Auto-Detector

Finally, one must modify `mucor_config.py` so it can identify variant files of this new type that should be parsed with our new function. This is done by adding some code to the function `DetectDataType` to return the format type string used in step 3, above ("`NewCaller`").

If we called the new input type "`NewCaller`," the `source` field in the config file must equal "`NewCaller`" for all samples of this type in order for the `ParseNewType` function to be called for this file type. Alternatively, if auto-detection is not possible or too difficult, a user may manually edit the JSON configuration file to enter the correct type in the "`source`" field.

New Output Modules

Step 1. Write Output Module

The first step in adding an output module to MuCor is to write the output module's core function. The function will eventually become a method bound to the `outputWriter` object; one may expect to have the global variant data frame (produced by the analysis core), output directory name, and the `Config` object available.

The main goal of the output module will be to transform the variant data frame into the desired output shape (for example, by filtering, aggregating, or pivoting), then write that output to one or more files. It would be good practice to limit output to one file per output mode or function; the file name is entirely of your choosing. More information about filtering and manipulating a Pandas data frame can be found in the Pandas documentation.

Once you've written a module with a function capable of reformatting the data frame and writing a file to the pre-specified output directory, save your python source file in the directory that contains the rest of the MuCor scripts.

Step 2. Integrate Output Module

Once the new output module can transform a dataframe into the desired output format, it can be integrated into the MuCor output writer object. This writer object is located in the 'output.py' python script included with MuCor.

Step 2A.

If the output module is in the same directory as the rest of the MuCor source, it can be imported in the output writer script `output.py` by adding a new line at the top of the file. For example, if the new output module is named, "`NewModule.py`",write `import NewModule`.

Step 2B.

Next, the function that writes the output file must be bound to the output Writer object. This is done so the method can be called without explicitly passing the Writer object. For example, if your new function is called `MyNewFunction`, it can be bound to the output writer by adding the following line to end of the Writer object initialization function:

```
self.MyNewFunction = NewModule.MyNewFunction.__get__(self,Writer)
```

Step 2C.

The output format must be named and added to the list of supported formats. The format name is a short descriptor that users may use to identify the output type. Existing format names include counts, featXsamp, and longtxt. Let us assume that the new output function will be identified as, "mnf-xls." In this case, it can be added to the list of supported formats using the line

```
self.supported_formats["mnf-xls"] = self.MyNewFunction.
```

This should be added to the bottom of the `Writer` object initialization function.

Step 2D.

Finally, the new output file name must be added to the list of known output file names. Continuing with the above example, one would add a new output file name with the following line:

```
self.file_names["mnf-xls"] = "my_new_function_output.xls"
```

As before, this should be added to the bottom of the Writer object initialization function.

Listing 4, below, is an example of how the modified `output.py` might look after adding the example detailed above. Added lines of code are bolded and highlighted for clarity. Notice that in this case `MyNewFunction` can write both excel files or text files, so there are two additions to the supported formats list and the output file types list. The rest of the changes follow the above example precisely.

With these modifications complete, users may run the new output module by passing the output types "`mnf-txt`" or "`mnf-xls`" to the `mucor_config.py --output_type` option. Furthermore, passing "`all`" will now include these two additional output types.

```
### from Step 2A in the text:
# import an output module from a separate file
import NewModule

class Writer(object):
    """Object that parses the dataframe and can write output in several different
formats"""

def __init__(self):
    self.data = pd.DataFrame()
    self.config = Config()
    self.outputDirName = ''
    self.attempted_formats = []  # used to prevent output modules from
                                 # being executed multiple times
    self.supported_formats = {  "default":   self.Default,
                                "counts":    self.Counts,
                                "txt":       self.VariantDetails,
                                "longtxt":   self.LongVariantDetails,
                                "xls":       self.VariantDetails,
                                "longxls":   self.LongVariantDetails,
                                "bed":       self.VariantBed,
                                "featXsamp": self.FeatureXSample,
                                "mutXsamp":  self.Feature_and_MutationXSample,
                                "vcf":       self.VCF,
                                "all":       self.All,
                                "runinfo":   self.RunInfo }

    self.file_names        = {  "counts":   "counts.txt",
                                "txt":      "variant_details.txt",
                                "longtxt":  "long_variant_details.txt",
                                "xls":      "variant_details.xlsx",
                                "longxls":  "long_variant_details.xlsx",
                                "bed":      "variant_locations.bed",
                                "featXsamp":"feature_by_sample.xlsx",
                                "mutXsamp": "feature_and_mutation_by_sample.xlsx",
                                "vcf":      "variant_locations.vcf",
                                "runinfo":  "run_info.txt" }

### from Step 2B in the text:
# first, make the function a bound method so it can be executed
# without explicitly passing the Writer object
self.MyNewFunction = NewModule.MyNewFunction.__get__(self,Writer)

### from Step 2C in the text:
# add the desired formats to the supported_formats dict
self.supported_formats["mnf-txt"] = self.MyNewFunction
self.supported_formats["mnf-xls"] = self.MyNewFunction

### from Step 2D in the text:
# add the desired output file names to the file_names dict
self.file_names["mnf-txt"] = "my_new_function_output.txt"
self.file_names["mnf-xls"] = "my_new_function_output.xls"
```

Listing 4. `output.py` in our example; added code bolded and highlighted for emphasis.

Supplementary Tables

Supplementary Table 1. Input formats recognized by MuCor

Parser	File format	Description and References
Database	VCF.GZ	Tabix-indexed, bgzipped VCF version ≥ 4.1 https://github.com/samtools/htslib http://samtools.sourceforge.net/tabix.shtml http://genome.ucsc.edu/goldenpath/help/vcf.html
Feature	GTF	Gene Transfer Format http://mblab.wustl.edu/GTF22.html
Feature	GFF3	General Feature Format version 3 http://www.sequenceontology.org/gff3.shtml
Region	BED	https://genome.ucsc.edu/FAQ/FAQformat.html#format1
Variant	OUT	MuTect .out format, basic and extended https://www.broadinstitute.org/cancer/cga/mutect_run
Variant	VCF	Variant Call Format version ≥ 4.1 http://samtools.github.io/hts-specs/VCFv4.2.pdf • Generic VCF • snpEff decorated • Ion Torrent PGM (default machine output) • Illumina Miseq (Miseq reporter output) • GATK • GATK SomaticIndelDetector • GATK HaplotypeCaller • MuTect • VarScan • FreeBayes • Samtools

Supplementary Table 2. Output report formats written by MuCor

Identifier	Description
all	Execute all output types
counts	Print counts of the number of variants per feature. Filename: `counts.txt`
txt	Print all information about each variant, one-per-row, irrespective of how many samples in which it appears. Useful for variant-centric studies. Identical to xls in layout. Filename: `variant_details.txt`
xls	Print all information about each variant, one-per-row, irrespective of how many samples in which it appears. Useful for variant-centric studies. Identical to txt in layout. Filename: `variant_details.xls/xlsx`
longtxt	Similar to txt above, but writes each instance of a variant to a new row. Each variant is written once per source file, instead of combining recurrent variants into one unique row. Identical to longxls in layout. Filename: `long_variant_details.txt`
longxls	Similar to xls above, but writes each instance of a variant to a new row. Each variant is written once per source file, instead of combining recurrent variants into one unique row. Identical to longtxt in layout. *NB*: The XLS format has a hard limit of 2^{16} rows; in long record format, a moderate sized study could exceed this (2,000 total variants/sample * 32 samples = 65,536 rows). MuCor can use Python's `xlwt` module to write .xls format, but it is preferable to have `XlsxWriter` or `openPyxl` installed for .xlsx support. Filename: `long_variant_details.xls/xlsx`
featXsamp	Print table of mutation counts per feature per sample. Samples are in columns, while features are in rows. The count of unique mutations per sample per feature are the table values. This output is useful for examining patterns in variation across samples, for example, to look at combinatoric mutation status for selected recurrently mutated genes. If used with data from a focused panel, or if the MuCor run output was limited by region (`-r`), this output could be used directly to make a heatmap. Filename: `feature_by_sample.xls/xlsx`

table continued next page

mutXsamp	Print table of mutations per sample. Unlike **featXsamp**, this differentiates among different variants within the same features. For example, in acute leukemia, the functional effect of mutations in *DNMT3A* depends on whether it is an R882 mutation or non-R882 mutation. As before, samples are in columns, with features in rows. However, rows 2-4 contain information about chromosome, position, ref, and alt. The table values are boolean: 1 for present mutation, 0 for missing mutation. This output could be used directly or with appropriate filtering to make an Oncoprint. Filename: `feature_and_mutation_by_sample.xls/xlsx`
mutXsampVAF	Identical to mutXsamp, but prints the variant allele frequency (VAF) in each cell, rather than 1 or 0. Filename: `feature_and_mutation_by_sample_vaf.xls/xlsx`

Supplementary Figures

For Supplementary Figure 1, see page 14.

For Supplementary Figure 2, see page 16.

For Supplementary Figure 3, see page 17.

Supplementary Figure 4. Example xls output.

chr	pos	ref	alt	vf	dp	feature	effect	fc
chr14	105235867	T	G	0.116	46	AKT1	?	UTR_3_PRIME
chr14	105236355	C	T	0.452, 0.4105	125, 95	AKT1	?	UTR_3_PRIME
chr2	32707554	A	G	0.0119	839	BIRC6	N2534D	NON_SYNONYMOUS_CODING
chr2	32707591	A	G	0.0112, 0.0108, 0.0111, 0.0132	1161, 927, 1174, 831	BIRC6	K2546R	NON_SYNONYMOUS_CODING
chr2	32707708	A	G	0.515, 0.5237	795, 1161	BIRC6	?	INTRON
chr2	32712738	A	C	0.033, 0.024, 0.0215, 0.028, 0.038, 0.03, 0.042, 0.026	272, 498, 465, 329, 265, 272, 192, 346	BIRC6	D2613A	NON_SYNONYMOUS_CODING
chr20	31022441	A	AG	0.0648, 0.0394, 0.06, 0.0659, 0.0581, 0.0588	247, 254, 200, 258, 172, 272	ASXL1	G643G?	FRAME_SHIFT
chrX	47424128	T	A	0.021	476	ARAF	?	INTRON
chrX	47424560	TG	T	0.0131, 0.0191	766, 628	ARAF	?	INTRON
chrX	47424611	GA	G	0.0423, 0.0462, 0.0266	379, 630, 568	ARAF	?	INTRON

1000_Genomes	1000_Genomes_VAF	dbSNP-137-Common	count	freq	sample
?	?	?	1	0.027777778	Sample_1
rs1130245	0.0165735	?	1	0.027777778	Sample_2, Sample_2
?	?	?	1	0.027777778	Sample_22
?	?	?	4	0.111111111	Sample_23, Sample_21, Sample_16, Sample_10
rs80071639	0.00519169	rs80071639	1	0.027777778	Sample_23, Sample_23
?	?	?	7	0.194444444	Sample_24, Sample_13, Sample_13, Sample_23, Sample_8, Sample_25, Sample_11, Sample_4
?	?	?	6	0.166666667	Sample_12, Sample_13, Sample_14, Sample_9, Sample_15, Sample_16
?	?	?	1	0.027777778	Sample_6
?	?	?	2	0.055555556	Sample_7, Sample_8
?	?	?	3	0.083333333	Sample_3, Sample_8, Sample_9

source
Sample_1.MuTect.vcf
Sample_2.MuTect.vcf, Sample_2.VarScan.vcf
Sample_22.VarScan.vcf
Sample_23.VarScan.vcf, Sample_21.VarScan.vcf, Sample_16.VarScan.vcf, Sample_10.VarScan.vcf
Sample_23.MuTect.vcf, Sample_23.VarScan.vcf
Sample_24.MuTect.vcf, Sample_13.MuTect.vcf, Sample_13.VarScan.vcf, Sample_23.MuTect.vcf, Sample_8.MuTect.vcf, Sample_25.MuTect.vcf, Sample_11.MuTect.vcf, Sample_4.MuTect.vcf
Sample_12.VarScan.vcf, Sample_13.VarScan.vcf, Sample_14.VarScan.vcf, Sample_9.VarScan.vcf, Sample_15.VarScan.vcf, Sample_16.VarScan.vcf
Sample_6.VarScan.vcf
Sample_7.VarScan.vcf, Sample_8.VarScan.vcf
Sample_3.VarScan.vcf, Sample_8.VarScan.vcf, Sample_9.VarScan.vcf

Note that this is variant-centric output; multiple samples exhibiting the same variant are collapsed into a single row. In these cases, the vf, dp, sample, and source fields are lists of comma-separated values.

In this example, the columns labeled “1000 Genomes”, “1000 Genomes VAF”, and “dbSNP-137-Common” are dynamically created from the databases specified with the –db option.

Supplementary Figure 5. Example longxls output.

chr	pos	ref	alt	vf	dp	feature	effect	fc	1000_Genomes	1000_Genomes_VAF	dbSNP-137-Common	count	freq	sample	source
chr14	105235867	T	G	0.116	46	AKT1	?	UTR_3_PRIME	?	?	?	1	0.027777778	Sample_1	Sample_1.MuTect.vcf
chr14	105236355	C	T	0.452	125	AKT1	?	UTR_3_PRIME	rs1130245	0.0165735	?	1	0.027777778	Sample_2	Sample_2.MuTect.vcf
chr14	105236355	C	T	0.4105	95	AKT1	?	UTR_3_PRIME	rs1130245	0.0165735	?	1	0.027777778	Sample_2	Sample_2.VarScan.vcf
chr14	105236377	G	A	0.484	160	AKT1	?	UTR_3_PRIME	rs41307094	0.0203674	rs41307094	1	0.027777778	Sample_3	Sample_3.MuTect.vcf
chr14	105236377	G	A	0.5082	122	AKT1	?	UTR_3_PRIME	rs41307094	0.0203674	rs41307094	1	0.027777778	Sample_3	Sample_3.VarScan.vcf
chr20	31022247	C	T	0.018	433	ASXL1	R578C	NON_SYNONYMOUS_CODING	?	?	?	1	0.027777778	Sample_10	Sample_10.MuTect.vcf
chr20	31022296	GC	G	0.012	832	ASXL1	C594	FRAME_SHIFT	?	?	?	1	0.027777778	Sample_11	Sample_11.VarScan.vcf
chr20	31022441	A	AG	0.0648	247	ASXL1	G643G?	FRAME_SHIFT	?	?	?	6	0.166666667	Sample_12	Sample_12.VarScan.vcf
chr20	31022441	AG	A	0.0674	282	ASXL1	G643	FRAME_SHIFT	?	?	?	8	0.222222222	Sample_13	Sample_13.VarScan.vcf
chr20	31022441	A	AG	0.0394	254	ASXL1	G643G?	FRAME_SHIFT	?	?	?	6	0.166666667	Sample_14	Sample_14.VarScan.vcf
chr20	31022441	AG	A	0.0446	315	ASXL1	G643	FRAME_SHIFT	?	?	?	8	0.222222222	Sample_7	Sample_7.VarScan.vcf
chr20	31022441	AG	A	0.0611	262	ASXL1	G643	FRAME_SHIFT	?	?	?	8	0.222222222	Sample_1	Sample_1.VarScan.vcf
chr20	31022441	A	AG	0.06	200	ASXL1	G643G?	FRAME_SHIFT	?	?	?	6	0.166666667	Sample_15	Sample_15.VarScan.vcf
chr20	31022441	A	AG	0.0659	258	ASXL1	G643G?	FRAME_SHIFT	?	?	?	6	0.166666667	Sample_9	Sample_9.VarScan.vcf
chr20	31022441	AG	A	0.0673	223	ASXL1	G643	FRAME_SHIFT	?	?	?	8	0.222222222	Sample_16	Sample_16.VarScan.vcf
chr20	31022441	AG	A	0.0652	230	ASXL1	G643	FRAME_SHIFT	?	?	?	8	0.222222222	Sample_17	Sample_17.VarScan.vcf
chr20	31022441	AG	A	0.0529	227	ASXL1	G643	FRAME_SHIFT	?	?	?	8	0.222222222	Sample_18	Sample_18.VarScan.vcf
chr20	31022441	A	AG	0.0581	172	ASXL1	G643G?	FRAME_SHIFT	?	?	?	6	0.166666667	Sample_19	Sample_19.VarScan.vcf
chr20	31022441	AG	A	0.0352	284	ASXL1	G643	FRAME_SHIFT	?	?	?	8	0.222222222	Sample_20	Sample_20.VarScan.vcf
chr20	31022441	A	AG	0.0588	272	ASXL1	G643G?	FRAME_SHIFT	?	?	?	6	0.166666667	Sample_21	Sample_21.VarScan.vcf
chr2	32707554	A	G	0.0119	839	BIRC6	N2534D	NON_SYNONYMOUS_CODING	?	?	?	1	0.027777778	Sample_22	Sample_22.VarScan.vcf
chr2	32707591	A	G	0.0112	1161	BIRC6	K2546R	NON_SYNONYMOUS_CODING	?	?	?	4	0.111111111	Sample_23	Sample_23.VarScan.vcf
chr2	32707591	A	G	0.0108	927	BIRC6	K2546R	NON_SYNONYMOUS_CODING	?	?	?	4	0.111111111	Sample_20	Sample_20.VarScan.vcf
chr2	32707591	A	G	0.0111	1174	BIRC6	K2546R	NON_SYNONYMOUS_CODING	?	?	?	4	0.111111111	Sample_21	Sample_21.VarScan.vcf
chr2	32707591	A	G	0.0132	831	BIRC6	K2546R	NON_SYNONYMOUS_CODING	?	?	?	4	0.111111111	Sample_10	Sample_10.VarScan.vcf

Note that this is raw output; variants common across multiple samples are *not* collapsed into single rows. The “count” and “crequency” columns tally the number and commonality of individual variants across samples in this long format.

As in the previous example, the database columns are dynamically generated according to the config.

Supplementary Figure 6. Example featXsamp output.

feature	S001	S005	S010	S011	S012	S013	S015	S020	S021	S022	S023
C11orf65	52	70	58	30	35	71	39	61	41	59	37
CEACAMP10	0	13	0	10	8	9	8	0	11	6	11
GALNT8	21	22	24	22	20	22	20	15	18	25	17
GS1-179L18.1	0	1	0	0	0	0	0	0	0	0	0
HLA-A	0	0	0	0	0	1	0	0	0	0	0
RN7SL15P	0	0	0	1	0	0	0	0	0	0	0
WRAP53	2	2	2	0	2	2	2	2	2	2	2

In this figure, we see a matrix with cases across the X axis (top) and features (defined by parameters `-g` and `-f`) down the Y axis (left). In each cell is a count of the number of variants corresponding to each case and feature. The totals above are reflective of many SNVs within whole genes. Recall that analysis space can be constrained by region; by running MuCor with region definitions (e.g., of hotspots) the featXsamp output format is suitable for direct translation to a mutation heatmap or oncoprint.

Supplementary Figure 7. Example mutXsamp and mutXsampVAF output.

					X027	X391	X538	X584	X825	X136	X107
feature	**chr**	**pos**	**ref**	**alt**							
ARAF	**chrX**	**47424615**	**C**	**T**	0	0	0	1	1	0	0
BRAF	**chr7**	**140482908**	**T**	**C**	0	0	1	0	0	0	0
KRAS	**chr12**	**25380285**	**G**	**A**	0	0	0	0	1	0	0
NRAS	**chr1**	**115258744**	**C**	**A**	0	0	0	0	0	0	1
		115258748	**C**	**T**	0	0	0	0	1	0	0
PIK3CD	**chr1**	**9776339**	**A**	**G**	0	0	1	1	0	1	0
		9784423	**C**	**T**	0	0	1	1	0	1	0
		9784861	**G**	**A**	0	0	0	0	1	0	0
TP53	**chr17**	**7577027**	**G**	**A**	0	0	0	0	1	0	0
		7578115	**T**	**C**	1	1	1	1	1	1	1
		7578146	**T**	**C**	0	0	0	1	0	1	0
		7578645	**C**	**T**	1	1	1	1	1	0	1
		7578671	**C**	**T**	0	0	1	0	0	0	0
		7579472	**G**	**C**	1	1	0	1	1	0	1
		7579801	**G**	**C**	1	1	0	1	1	0	1

					X027	X391	X538	X584	X825	X136	X107
feature	**chr**	**pos**	**ref**	**alt**							
ARAF	**chrX**	**47424615**	**C**	**T**	0	0	0	0.5366	0.5597	0	0
BRAF	**chr7**	**140482908**	**T**	**C**	0	0	0.4444	0	0	0	0
KRAS	**chr12**	**25380285**	**G**	**A**	0	0	0	0	0.0644	0	0
NRAS	**chr1**	**115258744**	**C**	**A**	0	0	0	0	0	0	0.4282
		115258748	**C**	**T**	0	0	0	0	0.0592	0	0
PIK3CD	**chr1**	**9776339**	**A**	**G**	0	0	0.4798	0.4902	0	0.5171	0
		9784423	**C**	**T**	0	0	0.4592	0.5556	0	0.5	0
		9784861	**G**	**A**	0	0	0	0	0.0334	0	0
TP53	**chr17**	**7577027**	**G**	**A**	0	0	0	0	0.0131	0	0
		7578115	**T**	**C**	0.4455	0.9975	0.5241	0.5144	0.5045	0.5066	0.9976
		7578146	**T**	**C**	0	0	0	0.4725	0	0.4831	0
		7578645	**C**	**T**	0.5583	0.9916	0.0699	0.4679	0.5265	0	0.9903
		7578671	**C**	**T**	0	0	0.0938	0	0	0	0
		7579472	**G**	**C**	0.5652	1	0	0.5263	0.6897	0	1
		7579801	**G**	**C**	0.4853	1	0	0.5351	0.491	0	1

The only difference between these tables is whether the cell contains a binary 0/1, or floating-point VAF.