dcmstack Documentation

Release 0.6

Brendan Moloney

November 26, 2012

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

The *dcmstack* software allows series of DICOM images to be stacked into multi-dimensional arrays. These arrays can be written out as Nifti files with an optional header extension (the *DcmMeta* extension) containing a summary of all the meta data from the source DICOM files.

1.1 Dependencies

Either Python 2.6 or 2.7 is required. With Python 2.6 it is not possible to maintain the order of meta data keys when reading back the JSON.

DcmStack requires the packages pydicom (>=0.9.7) and NiBabel.

1.2 Installation

Download the latest release from github, and run easy_install on the downloaded .zip file.

1.3 Basic Conversion

The software consists of the python package (*dcmstack*) with two command line interfaces (*dcmstack* and *nitool*).

It is recommended that you sort your DICOM data into directories (at least per study, but perferably by series) before conversion.

To convert directories of DICOM data from the command line you generally just need to pass the directories to *dcmstack*:

```
$ dcmstack -v 032-MPRAGEAXTI900Pre/
Processing source directory 032-MPRAGEAXTI900Pre/
Found 64 source files in the directory
Created 1 stacks of DICOM images
Writing out stack to path 032-MPRAGEAXTI900Pre/032-MPRAGE_AX_TI900_Pre.nii.gz
```

Here we use the verbose flab (*-v*) to show what is going on behind the scenes. To embed the DcmMeta header extension we need to use the *–embed* option. For more information see *CLI Tutorial*.

Performing the conversion from Python code requires a few extra steps but is also much more flexible:

```
>>> import dcmstack
>>> from glob import glob
>>> src_dcms = glob('032-MPRAGEAXTI900Pre/*.dcm')
>>> stacks = dcmstack.parse_and_stack(src_dcms)
>>> stack = stacks.values[0]
>>> nii = stack.to_nifti()
>>> nii.to_filename('output.nii.gz')
```

The *parse_and_stack* function has many optional arguments that closely match the command line options for *dcmstack*. To embed the DcmMeta extension pass *embed_meta=True* to the *to_nifti* method. For more information see *Python Tutorial*.

1.4 Basic Meta Data Usage

To work with Nifti files containing the embedded DcmMeta extension on the command line, use the *nitool* command. The *nitool* command has several sub commands.

```
$ nitool lookup InversionTime 032-MPRAGE_AX_TI900_Pre.nii.gz
900.0
```

Here we use the *lookup* sub command to lookup up the value for 'InversionTime'. For more information about using *nitool* see *CLI Tutorial*.

To work with the extended Nifti files from Python, use the *NiftiWrapper* class.

```
>>> from dcmstack import dcmmeta
>>> nii_wrp = dcmmeta.NiftiWrapper.from_filename('032-MPRAGE_AX_TI900_Pre.nii.gz')
>>> nii_wrp.get_meta('InversionTime')
900.0
```

For more information on using the *NiftiWrapper* class see *Python Tutorial*.

For information on the DcmMeta extension see *DcmMeta Extension*.

CHAPTER

TWO

CLI TUTORIAL

The software has two command line interfaces: *dcmstack* and *nitool*. The *dcmstack* command is used for converting DICOM data to Nifti files with the optional DcmMeta extension. The *nitool* command is used to work with these exteneded Nifti files.

2.1 Advanced Conversion

While the *dcmstack* command has many options, the defaults should do the right thing in most scenarios. To see a complete list of the command line options (with brief descriptions) use the *-h* option.

2.1.1 Embedding Meta Data

If the *–embed* option is used, all of the meta data in the source DICOM files will be extracted and summarized into a DcmMeta extension, which is then embedded into the Nifti header. The meta data keys are the keywords from the DICOM standard. For details on the DcmMeta extension see *DcmMeta Extension*.

The meta data is filtered using regular expressions to reduce the chance of including PHI (Private Health Information). There are two types of regular expressions used for filtering: 'exclude' and 'include' expressions. Any meta data where the key matches an exclude expression will be excluded, **unless** it also matches an include expression. That is to say that the include expressions override the exclude expressions.

To see the list of the default regular expressions use the *–default-regexes* option. To add an additional exclude expression use *–exclude-regex* (*-e*) option and to add an additional include expression use the *–include-regex* (*-i*) option.

By default, any private DICOM elements are ignored unless there is a "translator" for that element. To see a list of available translators use the *–list-translators* (*-l*) option. To disable a specific translator use the *–disable-translator* option. To include private elements that don't have a translator use the *–extract-private* option.

IT IS YOUR RESPONSABILITY TO KNOW IF THERE IS PRIVATE HEALTH INFORMATION IN THE RESULTING FILE AND TREAT SUCH FILES APPROPRIATELY.

2.1.2 Output Names and Grouping

All DICOM files from the same series will grouped into a stack together. The output file name is determined by a Python format string that is formatted with the meta data. This can be specified with the *–output-format* option. By default the program will try to figure out an appropriate format string for the available meta data. Generally this will be the 'SeriesNumber' followed by the 'ProtocolName' or 'SeriesDescription' (or just the word "series").

2.1.3 Ordering Time and Vector Dimensions

In addition to the three spatial dimensions, Nifti images can have time and (less commonly) vector dimensions. By default, the software will try to guess the appropriate meta data key for sorting the time dimension. If you would like to specify the meta data key, or stack along the vector dimension, you can do so with the *–time-var* (*-t*) and *–vector-var* (*-v*) options. Both options take a meta data key as an argument.

If there isn't an attribute that can be used with a simple ascending order to sort along these dimensions, the *–time-order* or *–vector-order* options can be used. The argument to the option should be a text file with one value per line corresponding to the sorted order to use.

2.1.4 Creating Uncompressed Niftis

By default the output Nifti files will be compressed, and thus have the extension '.nii.gz'. Almost every program that can read Nifti files will still read them if they are compressed. To override this behavior you can use the *–output-ext* option.

2.1.5 Handling Bad Data

Valid DICOM files should have a specific preamble (an initial byte pattern) to identify them as a DICOM file. It is not uncommon to come across files that are missing this preamble but are otherwise valid (generally due to bad software). You can force dcmstack to try to read these files using the *–force-read* option.

With some data sets (generally EPI) slices can be missing their pixel data due to an error in the reconstruction. Using the *–allow-dummies* option will allow these files and fill the corresponding slice with the maximum possible value (i.e. 65535 for uint16).

2.1.6 Voxel Order

While the affine transform stored in the Nifti allows a mapping from voxel indices to patient space, some programs do not make use of the affine information. To provide a similar orientation in these programs we reorder voxels in the same manner as dcm2nii. This results in the positive row, column, and slice directions pointing toward the left, anterior, and superior (LAS) patient directions. This can be overridden with the *–voxel-order* option.

2.2 Working with Extended Nifti Files

The *nitool* command can be used to perform various tasks with the extended Nifti files (that is files with the the DcmMeta extension embedded). The *nitool* command exposes functionality through a number of sub commands. To see a list of sub commands with brief explanations use the *-h* option. To see detailed help for a specific subcommand use:

```
$ nitool <sub_command> -h
```

2.2.1 Looking Up Meta Data

To lookup meta data in an extended Nifti, use the *lookup* sub command. If you don't specify a voxel index (using *–index*) then only constant meta data will be considered.

```
$ nitool lookup InversionTime 032-MPRAGE_AX_TI900_Pre.nii.gz
900.0
$ nitool lookup InstanceNumber 032-MPRAGE_AX_TI900_Pre.nii.gz
$ nitool lookup InstanceNumber --index 0,0,0 032-MPRAGE_AX_TI900_Pre.nii.gz
1
$ nitool lookup InstanceNumber --index 0,0,1 032-MPRAGE_AX_TI900_Pre.nii.gz
2
```

In the above example 'InversionTime' is contant across the Nifti and so an index is not required. The 'InstanceNumber' is not constant (it varies over slices) and thus only returns a result if an index is provided.

2.2.2 Merging and Splitting

To merge or split extended Nifti files use the *merge* and *split* sub commands. This will automatically create appropriate DcmMeta extensions for the output Nifti file(s). Both sub commands take a *–dimension* (*-d*) option to specify the index (zero based) of the dimension to split or merge along.

If the dimension is not specified to the *split* command, it will use the last dimension (vector, time, or slice). By default each output will have the same name as the input only with the index prepended (zero padded to three spaces). A format string can be passed with the option *–output-format* (*-o*) to override this behavior.

If the dimension is not specified for the *merge* command, it will use the last singular or missing dimension (slice, time, or vector). By default the inputs will be merged in the order they are provided on the command line. To instead sort the inputs using some meta data key use the *–sort* (*-s*) option.

2.2.3 Dumping and Embedding

The DcmMeta extension can be dumped using the *dump* sub command. If no destination path is given the result will print to stdout. A DcmMeta extension can be embedded into a Nifti file using the *embed* sub command. If no input file is given it will be read from stdin. For details about the DcmMeta extension see *DcmMeta Extension*.

2.2.4 Injecting Meta Data

If you want to inject some new meta data into the header extension you can use the *inject* command. You need to specify the meta data classification, key, and values. For example, to set a globally constant element with the key 'PatientID' and the value 'Subject_001':

```
$ nitool inject 032-MPRAGE_AX_TI900_Pre.nii.gz global const PatientID Subject_001
```

CHAPTER

THREE

PYTHON TUTORIAL

This is a brief overview of how to use the *dcmstack* Python package. For details refer to *API Documentation*.

3.1 Creating DicomStack Objects

If you have an aquisition that you would like to turn into a single *DicomStack* object then you may want to do this directly.

```
>>> import dcmstack, dicom
>>> from glob import glob
>>> src_paths = glob('032-MPRAGEAXTI900Pre/*.dcm')
>>> my_stack = dcmstack.DicomStack()
>>> for src_path in src_paths:
...     src_dcm = dicom.read_file(src_path)
...     my_stack.add_dcm(src_dcm)
```

If you are unsure how many stacks you want from a collection of DICOM data sets then you should use the *parse_and_stack* function. This will group together data sets from the same DICOM series.

```
>>> import dcmstack
>>> from glob import glob
>>> src_paths = glob('dicom_data/*.dcm')
>>> stacks = dcmstack.parse_and_stack(src_paths)
```

Any keyword arguments for the *DicomStack* constructor can also be passed to *parse_and_stack*.

3.1.1 Specifying Time and Vector Order

By default, if there is more than one 3D volume in the stack the software will try to guess the meta key to sort the fourth (time) dimension. To specify the meta data key for the fourth dimension or stack along the fifth (vector) dimension, use the *time_order* and *vector_order* arguments to the *DicomStack* constructor.

3.1.2 Grouping Datasets

The *parse_and_stack* function groups data sets using a tuple of meta data keys provided as the argument *group_by*. The default values should group datasets from the same series into the same stack. The result is a dictionary where the keys are the matching tuples of meta data values, and the values are the are the corresponding stacks.

3.2 Using DicomStack Objects

Once you have created your *DicomStack* objects you will typically want to get the array of voxel data, get the affine transform, or create a Nifti1Image.

```
>>> stack_data = my_stack.get_data()
>>> stack_affine = my_stack.get_affine()
>>> nii = my_stack.to_nifti()
```

3.2.1 Embedding Meta Data

The meta data from the source DICOM data sets can be summarized into a *DcmMetaExtension* which is embeded into the Nifti header. To do this you can either pass True for the *embed_meta* parameter to *DicomStack.to_nifti* or you can immediately get a *NiftiWrapper* with *DicomStack.to_nifti_wrapper*.

By default the meta data is filtered to reduce the chance of including private health information. This filtering can be controlled with the *meta_filter* parameter to the *DicomStack* constructor.

IT IS YOUR RESPONSABILITY TO KNOW IF THERE IS PRIVATE HEALTH INFORMATION IN THE RESULTING FILE AND TREAT SUCH FILES APPROPRIATELY.

3.3 Creating NiftiWrapper Objects

The *NiftiWrapper* class can be used to work with extended Nifti files. It wraps a *Nifti1Image* from the *nibabel* package. As mentioned above, these can be created directly from a *DicomStack*.

```
>>> import dcmstack, dicom
>>> from glob import glob
>>> src_paths = glob('032-MPRAGEAXTI900Pre/*.dcm')
>>> my_stack = dcmstack.DicomStack()
>>> for src_path in src_paths:
...     src_dcm = dicom.read_file(src_path)
...     my_stack.add_dcm(src_dcm)
...
>>> nii_wrp = my_stack.to_nifti_wrapper()
>>> nii_wrp.get_meta('InversionTime')
900.0
```

They can also be created by passing a *Nifti1Image* to the *NiftiWrapper* constructor or by passing the path to a Nifti file to *NiftiWrapper.from_filename*.

3.4 Using NiftiWrapper Objects

The *NiftiWrapper* objects have attribute *nii_img* pointing to the *Nifti1Image* being wrapped and the attribute *meta_ext* pointing to the *DcmMetaExtension*. There are also a number of methods for working with the image data and meta data together. For example merging or splitting the data set along the time axis.

3.4.1 Looking Up Meta Data

Meta data that is constant can be accessed with dict-style lookups. The more general access method is *get_meta* which can optionally take an index into the voxel array in order to provide access to varying meta data.

```
>>> nii_wrp = NiftiWrapper.from_filename('032-MPRAGEAXTI900Pre.nii.gz')
>>> nii_wrp['InversionTime']
900.0
>>> nii_wrp.get_meta('InversionTime')
900.0
>>> nii_wrp['InstanceNumber']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "build/bdist.linux-x86_64/egg/dcmstack/dcmmeta.py", line 1026, in __getitem__
KeyError: 'InstanceNumber'
>>> nii_wrp.get_meta('InstanceNumber')
>>> nii_wrp.get_meta('InstanceNumber', index=(0,0,0))
1
>>> nii_wrp.get_meta('InstanceNumber', index=(0,0,1))
2
```

3.4.2 Merging and Splitting Data Sets

We can create a *NiftiWrapper* by merging a sequence of *NiftiWrapper* objects using the class method *from_sequence*. Conversely, we can split a *NiftiWrapper* into a sequence if *NiftiWrapper* objects using the method *split*.

```
>>> from dcmstack.dcmmeta import NiftiWrapper
>>> nw1 = NiftiWrapper.from_filename('img1.nii.gz')
>>> nw2 = NiftiWrapper.from_filename('img2.nii.gz')
>>> print nw1.nii_img.get_shape()
(384, 512, 60)
>>> print nw2.nii_img.get_shape()
(384, 512, 60)
>>> print nw1.get_meta('EchoTime')
11.0
>>> print nw2.get_meta('EchoTime')
87.0
>>> merged = NiftiWrapper.from_sequence([nw1, nw2])
>>> print merged.nii_img.get_shape()
(384, 512, 60, 2)
>>> print merged.get_meta('EchoTime', index=(0,0,0,0)
11.0
>>> print merged.get_meta('EchoTime', index=(0,0,0,1)
87.0
>>> splits = list(merge.split())
>>> print splits[0].nii_img.get_shape()
(384, 512, 60)
>>> print splits[1].nii_img.get_shape()
(384, 512, 60)
>>> print splits[0].get_meta('EchoTime')
11.0
>>> print splits[1].get_meta('EchoTime')
87.0
```

3.5 Accessing the the DcmMetaExtension

It is generally recommended that meta data is accessed through the *NiftiWrapper* class since it can do some checks between the meta data and the image data. For example, it will make sure the dimensions and slice direction have not changed before using varying meta data.

However certain actions are much easier when accessing the meta data extension directly.

```
>>> from dcmstack.dcmmeta import NiftiWrapper
>>> nw1 = NiftiWrapper.from_filename('img.nii.gz')
>>> nw.meta_ext.shape
>>> (384, 512, 60, 2)
>>> print nw.meta_ext.get_values('EchoTime')
[11.0, 87.0]
>>> print nw.meta_ext.get_classification('EchoTime')
('time', 'samples')
```

CHAPTER

FOUR

DCMMETA EXTENSION

The DcmMeta extension is a complete but concise representation of the meta data in a series of source DICOM files.

The primary goals are:

1. Preserve as much meta data as possible
2. Make the meta data more accessible
3. Make the meta data human readable and editable

4.1 Extraction

The meta data is extracted from each DICOM input into a set of key/value pairs. Each non-private DICOM element uses the standard DICOM keyword as its key. Values are generally left unchanged (except for 'DS' and 'IS' value representations which are converted from strings to float and integer numbers respectively).

Translators are used to convert private elements into sets of key/value pairs. These are then added to the standard DICOM meta data with the translator name (followed by a dot) prepended to each of the keys it generates.

Private DICOM elements without translators are ignored by default, but this can be overridden. Any element with a value representation of 'OW' or 'OB' is ignored if it contains non-ASCII characters.

4.2 Summarizing

The meta data from individual input files is summarized over the dimensions of the Nifti file. Most of the meta data will be constant across all of the input files. Other meta data will be constant across each time/vector sample, or repeating for the slices in each time/vector sample. We summarize the meta data into one or more dictionaries as follows.

There will always be a dictionary 'global' with two nested dictionaries inside, 'const' and 'slices'. The meta data that is constant across all input files get stored under the 'const' dictionary. The meta data that varies across all slices will be stored under slices, where each value is a list of values (one for each slice).

If there is a time dimension there will also be 'time' dictionary containing two nested dictionaries, 'samples' and 'slices'. Meta data that is constant across a time sample will be stored in the 'samples' dictionary with each being value a list of values (one for each time sample). Values that repeat across the slices in a time sample (a single volume) will be stored in the 'slices' dictionary with each value being a list of values (one for each slice in a time point).

If there is a vector dimension there will be a 'vector' dictionary, handled in the same manner as the 'time' dictionary.

4.3 Encoding

The dictionaries of summarized meta data are encoded with JSON. A small amount of "meta meta" data that describes the DcmMeta extension is also included. This includes the affine ('dcmmeta_affine'), shape ('dcmmeta_shape'), any reorientation transform ('dcmmeta_reorient_transform'), and the slice dimension ('dcmmeta_slice_dim') of the data described by the meta data. A version number for the DcmMeta extension ('dcmmeta_version') is also included.

The affine, shape, and slice dimension are used to determine if varying meta data is still valid. For example, if the image affine no longer matches the meta data affine (i.e. the image has been coregistered) then we cannot directly match the per-slice meta data values to slices of the data array.

The reorientation transform can be used to update directional meta data to match the image orientation. This transform encodes any reordering of the voxel data that occured during conversion. If the image affine does not match the meta data affine, then an additional transformation needs to be done after applying the reorientation transform (from the meta data space to the image space).

4.4 Example

Below is an example DcmMeta extension created from a data set with two slices and three time points (each with a different EchoTime). The meta data has been abridged (the "..." line) for clarity.


```
{
    "global": {
        "const": {
            "SpecificCharacterSet": "ISO_IR 100",
            "ImageType": [
                "ORIGINAL",
                "PRIMARY",
                "M",
                "ND"
            ],
            "StudyTime": 69244.484,
            "SeriesTime": 71405.562,
            "Modality": "MR",
            "Manufacturer": "SIEMENS",
            "SeriesDescription": "2D 16Echo qT2",
            "ManufacturerModelName": "TrioTim",
            "ScanningSequence": "SE",
            "SequenceVariant": "SP",
            "ScanOptions": "SAT1",
            "MRAcquisitionType": "2D",
            "SequenceName": "se2d16",
            "AngioFlag": "N",
            "SliceThickness": 7.0,
            "RepetitionTime": 3000.0,
            "NumberOfAverages": 1.0,
            "ImagingFrequency": 123.250392,
            "ImagedNucleus": "1H",
            "MagneticFieldStrength": 3.0,
            "SpacingBetweenSlices": 10.5,
            "NumberOfPhaseEncodingSteps": 96,
            "EchoTrainLength": 1,
            "PercentSampling": 50.0,
            "PercentPhaseFieldOfView": 100.0,
            "PixelBandwidth": 420.0,
            "SoftwareVersions": "syngo MR B17",
```

```
"ProtocolName": "2D 16Echo qT2",
"TransmitCoilName": "TxRx_Head",
"AcquisitionMatrix": [
    0,
    192,
    96,
    0
],
"InPlanePhaseEncodingDirection": "ROW",
"FlipAngle": 180.0,
"VariableFlipAngleFlag": "N",
"SAR": 0.11299714843984,
"dBdt": 0.0,
"StudyID": "1",
"SeriesNumber": 3,
"AcquisitionNumber": 1,
"ImageOrientationPatient": [
    1.0,
    -2.051034e-10,
    0.0,
    2.051034e-10,
    1.0,
    1.98754e-11
],
"SamplesPerPixel": 1,
"PhotometricInterpretation": "MONOCHROME2",
"Rows": 192,
"Columns": 192,
"PixelSpacing": [
    0.66666668653488,
    0.66666668653488
],
"BitsAllocated": 16,
"BitsStored": 12,
"HighBit": 11,
"PixelRepresentation": 0,
"SmallestImagePixelValue": 0,
"WindowCenterWidthExplanation": "Algo1",
"PerformedProcedureStepStartTime": 69244.546,
"CsaImage.EchoLinePosition": 48,
"CsaImage.UsedChannelMask": 1,
"CsaImage.MeasuredFourierLines": 0,
"CsaImage.SequenceMask": 134217728,
"CsaImage.RFSWDDataType": "predicted",
"CsaImage.RealDwellTime": 6200,
"CsaImage.ImaCoilString": "C:HE",
"CsaImage.EchoColumnPosition": 96,
"CsaImage.PhaseEncodingDirectionPositive": 1,
"CsaImage.GSWDDataType": "predicted",
"CsaImage.SliceMeasurementDuration": 286145.0,
"CsaImage.MultistepIndex": 0,
"CsaImage.ImaRelTablePosition": [
    0,
    0,
    0
],
"CsaImage.NonPlanarImage": 0,
"CsaImage.EchoPartitionPosition": 32,
```

```
"CsaImage.AcquisitionMatrixText": "96*192s",
"CsaImage.ImaAbsTablePosition": [
    0,
    0,
    -1630
],
"CsaSeries.TalesReferencePower": 334.36266914,
"CsaSeries.Operation_mode_flag": 2,
"CsaSeries.dBdt_thresh": 0.0,
"CsaSeries.ProtocolChangeHistory": 0,
"CsaSeries.GradientDelayTime": [
    12.0,
    14.0,
    10.0
],
"CsaSeries.SARMostCriticalAspect": [
    3.2,
    1.84627729,
    0.0
],
"CsaSeries.B1rms": [
    7.07106781,
    1.59132133
],
"CsaSeries.RelTablePosition": [
    0,
    0,
    0
],
"CsaSeries.NumberOfPrescans": 0,
"CsaSeries.dBdt_limit": 0.0,
"CsaSeries.Stim_lim": [
    45.73709869,
    27.64929962,
    31.94370079
],
"CsaSeries.PatReinPattern": "1;FFS;45.36;10.87;3;0;2;866892320",
"CsaSeries.B1rmsSupervision": "NO",
"CsaSeries.ReadoutGradientAmplitude": 0.0,
"CsaSeries.MrProtocolVersion": 21710006,
"CsaSeries.RFSWDMostCriticalAspect": "Head",
"CsaSeries.SequenceFileOwner": "SIEMENS",
"CsaSeries.GradientMode": "Fast",
"CsaSeries.SliceArrayConcatenations": 1,
"CsaSeries.FlowCompensation": "No",
"CsaSeries.TransmitterCalibration": 128.29875,
"CsaSeries.Isocentered": 0,
"CsaSeries.AbsTablePosition": -1630,
"CsaSeries.ReadoutOS": 2.0,
"CsaSeries.dBdt_max": 0.0,
"CsaSeries.RFSWDOperationMode": 0,
"CsaSeries.SelectionGradientAmplitude": 0.0,
"CsaSeries.PhaseGradientAmplitude": 0.0,
"CsaSeries.RfWatchdogMask": 0,
"CsaSeries.CoilForGradient2": "AS092",
"CsaSeries.Stim_mon_mode": 2,
"CsaSeries.CoilId": [
    255,
```

```
        196,
        238,
        238,
        238,
        238,
        238,
        238,
        238,
        238,
        238
    ],
    "CsaSeries.Stim_max_ges_norm_online": 0.62600064,
    "CsaSeries.CoilString": "C:HE",
    "CsaSeries.CoilForGradient": "void",
    "CsaSeries.TablePositionOrigin": [
        0,
        0,
        -1630
    ],
    "CsaSeries.MiscSequenceParam": [
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        0,
        93,
        0,
        0,
        0,
        0,
        0,
        0
```

```
        ],
        "CsaSeries.LongModelName": "NUMARIS/4",
        "CsaSeries.Stim_faktor": 1.0,
        "CsaSeries.SW_korr_faktor": 1.0,
        "CsaSeries.Sed": [
            1000000.0,
            156.13387238,
            156.13387238
        ],
        "CsaSeries.PositivePCSDirections": "+LPH",
        "CsaSeries.SliceResolution": 1.0,
        "CsaSeries.Stim_max_online": [
            0.22781265,
            17.30016327,
            0.5990392
        ],
        "CsaSeries.t_puls_max": 0.0,
        "CsaSeries.MrPhoenixProtocol.ulVersion": 21710006,
        "CsaSeries.MrPhoenixProtocol.tSequenceFileName": "%SiemensSeq%\\se_mc",
        "CsaSeries.MrPhoenixProtocol.tProtocolName": "2D 16Echo qT2",
        ...
        "CsaSeries.MrPhoenixProtocol.sAsl.ulMode": 1,
        "CsaSeries.MrPhoenixProtocol.ucAutoAlignInit": 1
    },
    "slices": {
        "InstanceCreationTime": [
            71405.671,
            71405.562,
            71405.671,
            71405.578,
            71405.671,
            71405.578
        ],
        "AcquisitionTime": [
            71118.2425,
            71116.7375,
            71118.2625,
            71116.7575,
            71118.2825,
            71116.7775
        ],
        "ContentTime": [
            71405.671,
            71405.562,
            71405.671,
            71405.578,
            71405.671,
            71405.578
        ],
        "InstanceNumber": [
            1,
            2,
            7,
            8,
            13,
            14
        ],
        "LargestImagePixelValue": [
```

```
                2772,
                2828,
                2077,
                2085,
                1470,
                1397
            ],
            "WindowCenter": [
                1585.0,
                1513.0,
                1495.0,
                1455.0,
                1100.0,
                1084.0
            ],
            "WindowWidth": [
                3191.0,
                3212.0,
                2750.0,
                2731.0,
                2120.0,
                2073.0
            ],
            "CsaImage.TimeAfterStart": [
                1.505,
                0.0,
                1.525,
                0.02,
                1.545,
                0.04
            ],
            "CsaImage.ICE_Dims": [
                "1_1_1_1_1_1_1_4_1_1_1_1_490",
                "1_1_1_1_1_1_1_1_1_1_2_1_490",
                "1_2_1_1_1_1_1_4_1_1_1_1_490",
                "1_2_1_1_1_1_1_1_1_1_2_1_490",
                "1_3_1_1_1_1_1_4_1_1_1_1_490",
                "1_3_1_1_1_1_1_1_1_1_2_1_490"
            ]
        }
    },
    "time": {
        "samples": {
            "EchoTime": [
                20.0,
                40.0,
                60.0
            ],
            "EchoNumbers": [
                1,
                2,
                3
            ]
        },
        "slices": {
            "ImagePositionPatient": [
                [
                    -64.000001919919,
```

```
                    -118.13729284881,
                    -33.707626344045
                ],
                [
                    -64.000001919919,
                    -118.13729284881,
                    -23.207628251394
                ]
            ],
            "SliceLocation": [
                -33.707626341697,
                -23.207628249046
            ],
            "CsaImage.ProtocolSliceNumber": [
                0,
                1
            ],
            "CsaImage.SlicePosition_PCS": [
                [
                    -64.00000192,
                    -118.13729285,
                    -33.70762634
                ],
                [
                    -64.00000192,
                    -118.13729285,
                    -23.20762825
                ]
            ]
        }
    },
    "dcmmeta_shape": [
        192,
        192,
        2,
        3
    ],
    "dcmmeta_affine": [
        [
            -0.6666666865348816,
            1.3673560894655878e-10,
            0.0,
            64.0
        ],
        [
            1.3673560894655878e-10,
            0.6666666865348816,
            0.0,
            -9.196043968200684
        ],
        [
            0.0,
            -1.325026720289113e-11,
            10.499998092651367,
            -33.70762634277344
        ],
        [
            0.0,
```

```
            0.0,
            0.0,
            1.0
        ]
    ],
    "dcmmeta_reorient_transform": [
        [
            -0.0,
            -1.0,
            -0.0,
            191.0
        ],
        [
            1.0,
            0.0,
            0.0,
            0.0
        ],
        [
            0.0,
            0.0,
            1.0,
            0.0
        ],
        [
            0.0,
            0.0,
            0.0,
            1.0
        ]
    ],
    "dcmmeta_slice_dim": 2,
    "dcmmeta_version": 0.6
}
```

CHAPTER

FIVE

API DOCUMENTATION

5.1 dcmstack Package

5.1.1 dcmstack Package

Package for stacking DICOM images into multi dimensional volumes, extracting the DICOM meta data, converting the result to Nifti files with the meta data stored in a header extension, and work with these extended Nifti files.

5.1.2 dcmstack Module

Stack DICOM datasets into volumes. The contents of this module are imported into the package namespace.

class dcmstack.dcmstack.**DicomOrdering**(*key*, *abs_ordering=None*, *abs_as_str=False*)

Bases: object

Object defining an ordering for a set of dicom datasets. Create a DicomOrdering with the given DICOM element keyword.

Parameters **key** : str

The DICOM keyword to use for ordering the datasets

abs_ordering : sequence

A sequence specifying the absolute order for values corresponding to the *key*. Instead of ordering by the value associated with the *key*, the index of the value in this sequence will be used.

abs_as_str : bool

If true, the values will be converted to strings before looking up the index in *abs_ordering*.

Methods

get_ordinate(ds)	Get the ordinate for the given DICOM data set.

get_ordinate(*ds*)

Get the ordinate for the given DICOM data set.

Parameters **ds** : dict like

The DICOM data set we want the ordinate of. Should allow dict like access where DICOM keywords return the corresponing value.

Returns **An ordinate for the data set. If 'abs_ordering' is None then this will** :

just be the value for the keyword 'key'. Otherwise it will be an :

integer. :

class dcmstack.dcmstack.**DicomStack**(*time_order=None*, *vector_order=None*, *allow_dummies=False*, *meta_filter=None*)

Bases: object

Defines a method for stacking together DICOM data sets into a multi dimensional volume.

Tailored towards creating NiftiImage output, but can also just create numpy arrays. Can summarize all of the meta data from the input DICOM data sets into a Nifti header extension (see *dcmmeta.DcmMetaExtension*).

Parameters **time_order** : str or DicomOrdering

The DICOM keyword or DicomOrdering object specifying how to order the DICOM data sets along the time dimension.

vector_order : str or DicomOrdering

The DICOM keyword or DicomOrdering object specifying how to order the DICOM data sets along the vector dimension.

allow_dummies : bool

If True then data sets without pixel data can be added to the stack. The "dummy" voxels will have the maximum representable value for the datatype.

meta_filter : callable

A callable that takes a meta data key and value, and returns True if that meta data element should be excluded from the DcmMeta extension.

Notes

If both time_order and vector_order are None, the time_order will be guessed based off the data sets.

Methods

add_dcm(dcm[, meta])	Add a pydicom dataset to the stack.
clear()	Remove any DICOM datasets from the stack.
get_affine()	Get the affine transform for mapping row/column/slice indices
get_data()	Get an array of the voxel values.
get_shape()	Get the shape of the stack.
to_nifti([voxel_order, embed_meta])	Returns a NiftiImage with the data and affine from the stack.
to_nifti_wrapper([voxel_order])	Convienance method.

add_dcm(*dcm*, *meta=None*)

Add a pydicom dataset to the stack.

Parameters **dcm** : dicom.dataset.Dataset

The data set being added to the stack

meta : dict

The extracted meta data for the DICOM data set *dcm*. If None extract.default_extractor will be used.

Raises **IncongruentImageError** :

The provided *dcm* does not match the orientation or dimensions of those already in the stack.

ImageCollisionError :

The provided *dcm* has the same slice location and time/vector values.

clear()
Remove any DICOM datasets from the stack.

get_affine()
Get the affine transform for mapping row/column/slice indices to Nifti (RAS) patient space.

Returns **A 4x4 numpy array containing the affine transform.** :

Raises **InvalidStackError** :

The stack is incomplete or invalid.

get_data()
Get an array of the voxel values.

Returns **A numpy array filled with values from the DICOM data sets' pixels.** :

Raises **InvalidStackError** :

The stack is incomplete or invalid.

get_shape()
Get the shape of the stack.

Returns **A tuple of integers giving the size of the dimensions of the stack.** :

Raises **InvalidStackError** :

The stack is incomplete or invalid.

sort_guesses = ['EchoTime', 'InversionTime', 'RepetitionTime', 'FlipAngle', 'TriggerTime', 'AcquisitionTime', 'Con
The meta data keywords used when trying to guess the sorting order. Keys that come earlier in the list are given higher priority.

to_nifti(*voxel_order='LAS'*, *embed_meta=False*)
Returns a NiftiImage with the data and affine from the stack.

Parameters **voxel_order** : str

A three character string repsenting the voxel order in patient space (see the function *reorder_voxels*).

embed_meta : bool

If true a dcmmeta.DcmMetaExtension will be embedded in the Nifti header.

Returns **A nibabel.nifti1.Nifti1Image created with the stack's data and affine.** :

to_nifti_wrapper(*voxel_order=''*)
Convienance method. Calls *to_nifti* and returns a *NiftiWrapper* generated from the result.

exception dcmstack.dcmstack.**ImageCollisionError**
Bases: exceptions.Exception

An exception denoting that a DICOM which collides with one already in the stack was passed to a *DicomStack.add_dcm*.

dcmstack.dcmstack.**axcodes2ornt**(*axcodes*, *labels=None*)
Convert axis codes *axcodes* to an orientation

Parameters **axcodes** : (N,) tuple

axis codes - see ornt2axcodes docstring

labels : optional, None or sequence of (2,) sequences

(2,) sequences are labels for (beginning, end) of output axis. That is, if the first element in *axcodes* is `front`, and the second (2,) sequence in *labels* is ('back', 'front') then the first row of *ornt* will be `[1, 1]`. If None, equivalent to `(('L','R'),('P','A'),('I','S'))` - that is - RAS axes.

Returns **ornt** : (N,2) array-like

oritation array - see io_orientation docstring

Examples

```
>>> axcodes2ornt(('F', 'L', 'U'), (('L','R'),('B','F'),('D','U')))
[[1, 1],[0,-1],[2,1]]
```

dcmstack.dcmstack.**dcm_time_to_sec**(*time_str*)
Convert a DICOM time value (value representation of 'TM') to the number of seconds past midnight.

Parameters **time_str** : str

The DICOM time value string

Returns **A floating point representing the number of seconds past midnight** :

dcmstack.dcmstack.**default_group_keys = ('SeriesInstanceUID', 'SeriesNumber', 'ProtocolName')**
Default keys for grouping DICOM files from the same acquisition together.

dcmstack.dcmstack.**default_key_excl_res = ['Patient', 'Physician', 'Operator', 'Date', 'Birth', 'Address', 'Institutio**
A list of regexes passed to *make_key_regex_filter* as *exclude_res* to create the *default_meta_filter*.

dcmstack.dcmstack.**default_key_incl_res = ['ImageOrientationPatient', 'ImagePositionPatient']**
A list of regexes passed to *make_key_regex_filter* as *force_include_res* to create the *default_meta_filter*.

dcmstack.dcmstack.**default_meta_filter**(*key*, *value*)
Default meta_filter for *DicomStack*.

dcmstack.dcmstack.**make_key_regex_filter**(*exclude_res*, *force_include_res=None*)
Make a meta data filter using regular expressions.

Parameters **exclude_res** : sequence

Sequence of regular expression strings. Any meta data where the key matches one of these expressions will be excluded, unless it matches one of the *force_include_res*.

force_include_res : sequence

Sequence of regular expression strings. Any meta data where the key matches one of these expressions will be included.

Returns **A callable which can be passed to 'DicomStack' as the 'meta_filter'.** :

dcmstack.dcmstack.**ornt_transform**(*start_ornt*, *end_ornt*)

Return the orientation that transforms from *start_ornt* to *end_ornt*.

Parameters **start_ornt** : (n,2) orientation array

Initial orientation.

end_ornt : (n,2) orientation array

Final orientation.

Returns **orientations** : (p, 2) ndarray

The orientation that will transform the *start_ornt* to the *end_ornt*.

dcmstack.dcmstack.**parse_and_group**(*src_paths*, *group_by=('SeriesInstanceUID', 'SeriesNumber', 'ProtocolName')*, *extractor=None*, *force=False*, *warn_on_except=False*)

Parse the given dicom files and group them together. Each group is stored as a (list) value in a dict where the key is a tuple of values corresponding to the keys in 'group_by'

Parameters **src_paths** : sequence

A list of paths to the source DICOM files.

group_by : tuple

Meta data keys to group data sets with. Any data set with the same values for these keys will be grouped together. This tuple of values will also be the key in the result dictionary.

extractor : callable

Should take a dicom.dataset.Dataset and return a dictionary of the extracted meta data.

force : bool

Force reading source files even if they do not appear to be DICOM.

warn_on_except : bool

Convert exceptions into warnings, possibly allowing some results to be returned.

Returns **groups** : dict

A dict mapping tuples of values (corresponding to 'group_by') to groups of data sets. Each element in the list is a tuple containing the dicom object, the parsed meta data, and the filename.

dcmstack.dcmstack.**parse_and_stack**(*src_paths*, *group_by=('SeriesInstanceUID', 'SeriesNumber', 'ProtocolName')*, *extractor=None*, *force=False*, *warn_on_except=False*, ***stack_args*)

Parse the given dicom files into a dictionary containing one or more DicomStack objects.

Parameters **src_paths** : sequence

A list of paths to the source DICOM files.

group_by : tuple

Meta data keys to group data sets with. Any data set with the same values for these keys will be grouped together. This tuple of values will also be the key in the result dictionary.

extractor : callable

Should take a dicom.dataset.Dataset and return a dictionary of the extracted meta data.

force : bool

Force reading source files even if they do not appear to be DICOM.

warn_on_except : bool

Convert exceptions into warnings, possibly allowing some results to be returned.

stack_args : kwargs

Keyword arguments to pass to the DicomStack constructor.

`dcmstack.dcmstack.`**`reorder_voxels`**(*vox_array*, *affine*, *voxel_order*)

Reorder the given voxel array and corresponding affine.

Parameters **vox_array** : array

The array of voxel data

affine : array

The affine for mapping voxel indices to Nifti patient space

voxel_order : str

A three character code specifing the desired starting point for rows, columns, and slices in terms of the orthogonal axes of patient space: (l)eft, (r)ight, (a)nterior, (p)osterior, (s)uperior, and (i)nferior.

Returns **out_vox** : array

An updated view of vox_array.

out_aff : array

A new array with the updated affine

reorient_transform : array

The transform used to update the affine.

ornt_trans : tuple

The orientation transform used to update the orientation.

5.1.3 `dcmmeta` Module

DcmMeta header extension and NiftiWrapper for working with extended Niftis.

exception `dcmstack.dcmmeta.`**`MissingExtensionError`**

Bases: `exceptions.Exception`

Exception denoting that there is no DcmMetaExtension in the Nifti header.

class `dcmstack.dcmmeta.`**`NiftiWrapper`**(*nii_img*, *make_empty=False*)

Bases: `object`

Wraps a Nifti1Image object containing a DcmMeta header extension. Provides access to the meta data and the ability to split or merge the data array while updating the meta data.

Parameters **nii_img** : nibabel.nifti1.Nifti1Image

The Nifti1Image to wrap.

make_empty : bool

If True an empty DcmMetaExtension will be created if none is found.

Raises **MissingExtensionError** :

No valid DcmMetaExtension was found.

ValueError :

More than one valid DcmMetaExtension was found.

Methods

from_dicom(klass, dcm_data[, meta_dict])	Create a NiftiWrapper from a single DICOM dataset.
from_dicom_wrapper(klass, dcm_wrp[, meta_dict])	Create a NiftiWrapper from a nibabel DicomWrapper.
from_filename(klass, path)	Create a NiftiWrapper from a file.
from_sequence(klass, seq[, dim])	Create a NiftiWrapper by joining a sequence of NiftiWrapper objects
get_meta(key[, index, default])	Return the meta data value for the provided *key*.
meta_valid(classification)	Return true if the meta data with the given classification appears
remove_extension()	Remove the DcmMetaExtension from the header of nii_img. The
replace_extension(dcmmeta_ext)	Replace the DcmMetaExtension.
split([dim])	Generate splits of the array and meta data along the specified dimension.
to_filename(out_path)	Write out the wrapped Nifti to a file

classmethod **from_dicom**(*klass*, *dcm_data*, *meta_dict=None*)
Create a NiftiWrapper from a single DICOM dataset.

Parameters **dcm_data** : dicom.dataset.Dataset

The DICOM dataset to convert into a NiftiWrapper.

meta_dict : dict

An optional dictionary of meta data extracted from *dcm_data*. See the *extract* module for generating this dict.

classmethod **from_dicom_wrapper**(*klass*, *dcm_wrp*, *meta_dict=None*)
Create a NiftiWrapper from a nibabel DicomWrapper.

Parameters **dcm_wrap** : nicom.dicomwrappers.DicomWrapper

The dataset to convert into a NiftiWrapper.

meta_dict : dict

An optional dictionary of meta data extracted from *dcm_data*. See the *extract* module for generating this dict.

classmethod **from_filename**(*klass*, *path*)
Create a NiftiWrapper from a file.

Parameters **path** : str

The path to the Nifti file to load.

classmethod **from_sequence**(*klass*, *seq*, *dim=None*)
Create a NiftiWrapper by joining a sequence of NiftiWrapper objects along the given dimension.

Parameters **seq** : sequence

The sequence of NiftiWrapper objects.

dim : int

The dimension to join the NiftiWrapper objects along. If None, 2D inputs will become 3D, 3D inputs will become 4D, and 4D inputs will become 5D.

Returns **result** : NiftiWrapper

The merged NiftiWrapper with updated meta data.

get_meta(*key*, *index=None*, *default=None*)
Return the meta data value for the provided *key*.

Parameters **key** : str

The meta data key.

index : tuple

The voxel index we are interested in.

default :

This will be returned if the meta data for *key* is not found.

Returns **value** :

The meta data value for the given *key* (and optionally *index*)

Notes

The per-sample and per-slice meta data will only be considered if the *samples_valid* and *slices_valid* methods return True (respectively), and an *index* is specified.

meta_valid(*classification*)
Return true if the meta data with the given classification appears to be valid for the wrapped Nifti image. Considers the shape and orientation of the image and the meta data extension.

remove_extension()
Remove the DcmMetaExtension from the header of nii_img. The attribute *meta_ext* will still point to the extension.

replace_extension(*dcmmeta_ext*)
Replace the DcmMetaExtension.

Parameters **dcmmeta_ext** : DcmMetaExtension

The new DcmMetaExtension.

split(*dim=None*)
Generate splits of the array and meta data along the specified dimension.

Parameters **dim** : int

The dimension to split the voxel array along. If None it will prefer the vector, then time, then slice dimensions.

Returns **result** :

Generator which yields a NiftiWrapper result for each index along *dim*.

to_filename(*out_path*)
Write out the wrapped Nifti to a file

Parameters **out_path** : str

The path to write out the file to

Notes

Will check that the DcmMetaExtension is valid before writing the file.

dcmstack.dcmmeta.**is_constant**(*sequence*, *period=None*)
Returns true if all elements in (each period of) the sequence are equal.

Parameters **sequence** : sequence

The sequence of elements to check.

period : int

If not None then each subsequence of that length is checked.

dcmstack.dcmmeta.**is_repeating**(*sequence*, *period*)
Returns true if the elements in the sequence repeat with the given period.

Parameters **sequence** : sequence

The sequence of elements to check.

period : int

The period over which the elements should repeat.

dcmstack.dcmmeta.**patch_dcm_ds_is**(*dcm*)
Convert all elements in *dcm* with VR of 'DS' or 'IS' to floats and ints. This is a hackish work around for the backwards incompatability of pydicom 0.9.7 and should not be needed once nibabel is updated.

5.1.4 extract Module

Extract meta data from a DICOM data set.

class dcmstack.extract.**MetaExtractor**(*ignore_rules=None*, *translators=None*, *conversions=None*, *warn_on_trans_except=True*)
Bases: object

Callable object for extracting meta data from a dicom dataset. Initialize with a set of ignore rules, translators, and type conversions.

Parameters **ignore_rules** : sequence

A sequence of callables, each of which should take a DICOM element and return True if it should be ignored. If None the module default is used.

translators : sequence

A sequence of *Translator* objects each of which can convert a DICOM element into a dictionary. Overrides any ignore rules. If None the module default is used.

conversions : dict

Mapping of DICOM value representation (VR) strings to callables that perform some conversion on the value

warn_on_trans_except : bool

Convert any exceptions from translators into warnings.

class dcmstack.extract.**Translator**
Bases: tuple

A namedtuple for storing the four elements of a translator: a name, the dicom.tag.Tag that can be translated, the private creator string (optional), and the function which takes the DICOM element and returns a dictionary.

Methods

`count`(...)	
`index`((value, [start, ...)	Raises ValueError if the value is not present.

name
Alias for field number 0

priv_creator
Alias for field number 2

tag
Alias for field number 1

trans_func
Alias for field number 3

dcmstack.extract.**csa_image_trans = Translator(name='CsaImage', tag=(0029, 1010), priv_creator='SIEMENS CSA**
Translator for the CSA image sub header.

dcmstack.extract.**csa_image_trans_func**(*elem*)
Function for translating the CSA image sub header.

dcmstack.extract.**csa_series_trans = Translator(name='CsaSeries', tag=(0029, 1020), priv_creator='SIEMENS CSA**
Translator for parsing the CSA series sub header.

dcmstack.extract.**csa_series_trans_func**(*elem*)
Function for parsing the CSA series sub header.

dcmstack.extract.**default_extractor = <dcmstack.extract.MetaExtractor object at 0x2489250>**
The default *MetaExtractor*.

dcmstack.extract.**default_ignore_rules = (<function ignore_private at 0x24835f0>, <function ignore_non_ascii_byt**
The default tuple of ignore rules for *MetaExtractor*.

dcmstack.extract.**default_translators = (Translator(name='CsaImage', tag=(0029, 1010), priv_creator='SIEMENS**
Default translators for MetaExtractor.

dcmstack.extract.**ignore_non_ascii_bytes**(*elem*)
Ignore rule for *MetaExtractor* to skip elements with VR of 'OW', 'OB', or 'UN' if the byte string contains non ASCII characters.

dcmstack.extract.**ignore_private**(*elem*)
Ignore rule for *MetaExtractor* to skip private DICOM elements (odd group number).

dcmstack.extract.**parse_phoenix_prot**(*prot_key*, *prot_val*)
Parse the MrPheonixProtocol string.

Parameters **prot_str** : str

The 'MrPheonixProtocol' string from the CSA Series sub header.

Returns **prot_dict** : OrderedDict

Meta data pulled from the ASCCONV section.

Raises **PhoenixParseError** : A line of the ASCCONV section could not be parsed.

dcmstack.extract.**simplify_csa_dict**(*csa_dict*)
Simplify the result of nibabel.nicom.csareader.

Parameters **csa_dict** : dict

The result from nibabel.nicom.csareader

Returns **result** : OrderedDict

Result where the keys come from the 'tags' sub dictionary of *csa_dict*. The values come from the 'items' within that tags sub sub dictionary. If items has only one element it will be unpacked from the list.

dcmstack.extract.**tag_to_str**(*tag*)
Convert a DICOM tag to a string representation using the group and element hex values seprated by an underscore.

dcmstack.extract.**tm_to_seconds**(*time_str*)
Convert a DICOM time value (value representation of 'TM') to the number of seconds past midnight.

Parameters **time_str** : str

The DICOM time value string

Returns **A floating point representing the number of seconds past midnight** :

dcmstack.extract.**unpack_vr_map = {'SS': 'h', 'US': 'H', 'UL': 'I', 'FD': 'd', 'SL': 'i', 'US or SS': 'H', 'FL': 'f'}**
Dictionary mapping value representations to corresponding format strings for the struct.unpack function.

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

d

PYTHON MODULE INDEX

d