CloudReg

Vikram Chandrashekhar

Jan 15, 2021

CONTENTS

CHAPTER

ONE

MOTIVATION

Quantifying terascale multi-modal human and animal imaging data requires scalable analysis tools. We developed CloudReg, an automated, terascale, cloud-based image analysis pipeline for preprocessing and cross-modal, non-linear registration between volumetric datasets with artifacts. CloudReg was developed using cleared murine brain light-sheet microscopy images, but is also accurate in registering the following datasets to their respective atlases: in vivo human and ex vivo macaque brain magnetic resonance imaging, ex vivo murine brain micro-computed tomography.

CHAPTER

TWO

RESULTS

CHAPTER

THREE

DOCUMENTATION

CloudReg is a pipeline for terascale image preprocessing and 3D nonlinear registration between two image volumes wth polynomial intensity correspondence.

3.1 Setup

CloudReg is designed to be used in the cloud. We chose to work with Amazon Web Services (AWS) and the below setup instructions are for that.

3.1.1 Requirements

- AWS account
- IAM Role and User with credentials to access EC2 and S3
- S3 Bucket to store raw data
- S3 Bucket to store processed data
- EC2 instance with Docker
- EC2 instance with MATLAB
- CloudFront CDN with HTTP/2 enabled for fast visualization
- Web Application Firewall for IP-address restriction on data access.

3.1.2 Create AWS account

Follow instructions here to create AWS account or use existing AWS account. All of the following AWS setup instructions should be performed within the same AWS account.

3.1.3 Create IAM Role

1. Log into AWS console
2. Navigate to IAM section of console
3. Click on *Roles* in the left sidebar
4. Click *Create Role*
5. Click *AWS Service* under *Type of Trusted Entity*
6. Click *EC2* as the AWS Service and click *Next*
7. Next to *Filter Policies*, search for *S3FullAccess* and *EC2FullAccess* and click the checkbox next to both to add them as policies to this role.
8. Click *Next*
9. Click *Next* on the *Add Tags* screen. Adding tags is optional.
10. On the *Review Role* screen, choose a role name, like *cloudreg_role*, and customize the description as you see fit.
11. Finally, click *Create Role*

3.1.4 Create IAM User

1. Log into AWS console
2. Navigate to IAM section of console
3. Click on *Users* in the left sidebar
4. Click *Add User*
5. Choose a User name like *cloudreg_user*, check *Programmatic Access*, and click *Next*
6. Click on *Attach existing policies directly* and search for and add *S3FullAccess* and *EC2FullAccess*, and click Next
7. Click *Next* on the *Add Tags* screen. Adding tags is optional. Then click *Next*
8. On the *Review* screen, verify the information is correct and click *Create User*
9. On the next screen, download the autogenerated password and key and keep them private and secure. We will need these credentials later

3.1.5 Create S3 Bucket

1. Log into AWS console
2. Navigate to S3 section of console
3. Click *Create Bucket*
4. Choose a bucket name and be sure to choose the bucket region carefully. You will want to pick the region that is geographically closest to you for optimal visualization speeds. Record the region you have chosen.
5. Uncheck *Block All Public Access*. We will restrict access to the data using CloudFront and a Firewall.
6. The remaining settings can be left as is. Click *Create Bucket*

3.1.6 Set up CORS on S3 Bucket

1. Log into AWS console
2. Navigate to S3 section of console
3. Click on the S3 Bucket you would like to add CORS to.
4. Click on the *Permissions* tab
5. Scroll to the bottom and click *Edit* under *Cross-origin resource sharing (CORS)*
6. Paste the following text:

```
[
    {
        "AllowedHeaders": [
            "Authorization"
        ],
        "AllowedMethods": [
            "GET"
        ],
        "AllowedOrigins": [
            "*"
        ],
        "ExposeHeaders": [],

        "MaxAgeSeconds": 3000
    }
]
```

7. click *Save Changes*

3.1.7 Set up CloudReg EC2 instance

1. Log into AWS console
2. Navigate to EC2 section of console
3. In the left sidebar, click *Instances*. Make sure you change the region (top right, middle drop-down menu) to match that of your raw data and processed data S3 buckets.
4. Click *Launch Instances*
5. In the search bar, enter the following: *ami-098555c9b343eb09c*. This is an Amazing Machine Image (AMI) called *Deep Learning AMI (Ubuntu 18.04) Version 38.0*. Click Select when this AMI shows up.
6. The default instance type should be *t2.micro*, if not choose change it to that type. Leave the remaining choices as their defaults and click *Review and Launch*.
7. Verify the EC2 instance information is correct and click *Launch*.
8. When the key pair pop-up appears, select *Choose an existing key pair* if you have already created one, or select *Create a new key pair* if you do not already have one. Follow the instructions on-screen to download and save the key pair.
9. Follow AWS tutorial to connect to this EC2 instance through the command line.
10. Once you have connected to the instance via SSH, create the cloud-volume credentials file on the instance using the CLI text editor of your choice.

11. After creating the cloud-volume credentials file, run the following command to turn off the EC2 instance: *sudo shutdown now*

3.1.8 Set up MATLAB EC2 instance

1. Follow instructions *here <https://github.com/mathworks-ref-arch/matlab-on-aws>* on setting up MATLAB on an EC2 instance. Be sure to create this instance in the same region as your S3 buckets.

3.1.9 Set up AWS Web Application Firewall

1. Log into AWS console
2. Navigate to CloudFront section of console

3.1.10 Set up AWS CloudFront

1. Log into AWS console
2. Navigate to CloudFront section of console

3.2 Run

3.3 Reference

3.3.1 COLM pipeline

`cloudreg.scripts.run_colm_pipeline_ec2.`**`run_colm_pipeline`**(*ssh_key_path*, *instance_id*, *input_s3_path*, *output_s3_path*, *num_channels*, *autofluorescence_channel*, *log_s3_path=None*, *instance_type='r5d.24xlarge'*)

Run COLM pipeline on EC2 instance

Parameters

- **`ssh_key_path`** (*`str`*) – Local path to ssh key needed for this server
- **`instance_id`** (*`str`*) – ID of the EC2 instance to run pipeline on
- **`input_s3_path`** (*`str`*) – S3 Path to raw data
- **`output_s3_path`** (*`str`*) – S3 path to store precomputed volume. Volume is stored at output_s3_path/channel for each channel.
- **`num_channels`** (*`int`*) – Number of channels in this volume
- **`autofluorescence_channel`** (*`int`*) – Autofluorescence channel number
- **`log_s3_path`** (*`str, optional`*) – S3 path to store intermediates including vignetting correction and Terastitcher files. Defaults to None.

- **instance_type** (*str, optional*) – AWS EC2 instance type. Defaults to "r5d.24xlarge".

cloudreg.scripts.colm_pipeline.**colm_pipeline**(*input_s3_path*, *output_s3_path*, *channel_of_interest*, *autofluorescence_channel*, *raw_data_path*, *stitched_data_path*, *log_s3_path=None*)

Run COLM pipeline including vignetting correction, stitching, illumination correction, and upload to S3 in Neuroglancer-compatible format

Parameters

- **input_s3_path** (*str*) – S3 path to raw COLM data. Should be of the form s3://<bucket>/<experiment>
- **output_s3_path** (*str*) – S3 path to store precomputed volume. Precomputed volumes for each channel will be stored under this path. Should be of the form s3://<bucket>/<path_to_precomputed>
- **channel_of_interest** (*int*) – Channel number to operate on. Should be a single integer.
- **autofluorescence_channel** (*int*) – Autofluorescence channel number. Should be a single integer.
- **raw_data_path** (*str*) – Local path where corrected raw data will be stored.
- **stitched_data_path** (*str*) – Local path where stitched slices will be stored.
- **log_s3_path** (*str, optional*) – S3 path at which pipeline intermediates can be stored including bias correction tile and xml files from Terastitcher. Defaults to None.

3.3.2 Intensity Correction

cloudreg.scripts.correct_raw_data.**correct_raw_data**(*raw_data_path*, *channel*, *subsample_factor=2*, *log_s3_path=None*, *background_correction=True*)

Correct vignetting artifact in raw data

Parameters

- **raw_data_path** (*str*) – Path to raw data
- **channel** (*int*) – Channel number to prcess
- **subsample_factor** (*int, optional*) – Factor to subsample the raw data by to compute vignetting correction. Defaults to 2.
- **log_s3_path** (*str, optional*) – S3 path to store intermediates at. Defaults to None.
- **background_correction** (*bool, optional*) – If True, subtract estimated background value from all tiles. Defaults to True.

cloudreg.scripts.correct_raw_data.**correct_tile**(*raw_tile_path*, *bias*, *background_value=None*)

Apply vignetting corrrection to single tile

Parameters

- **raw_tile_path** (*str*) – Path to raw data image
- **bias** (*np.ndarray*) – Vignetting correction that is multiplied by image
- **background_value** (*float, optional*) – Background value. Defaults to None.

cloudreg.scripts.correct_raw_data.**correct_tiles**(*tiles*, *bias*, *background_value=None*)
Correct a list of tiles

Parameters

- **tiles** (*list of str*) – Paths to raw data images to correct
- **bias** (*np.ndarray*) – Vignetting correction to multiply by raw data
- **background_value** (*float, optional*) – Background value to subtract from raw data. Defaults to None.

cloudreg.scripts.correct_raw_data.**get_background_value**(*raw_data_path*)
Estimate background value for COLM data

Parameters **raw_data_path** (*str*) – Path to raw data

Returns Estimated value of background in image

Return type float

cloudreg.scripts.correct_raw_data.**sum_tiles**(*files*)
Sum the images in files together

Parameters **files** (*list of str*) – Local Paths to images to sum

Returns Sum of the images in files

Return type np.ndarray

cloudreg.scripts.correct_stitched_data.**correct_stitched_data**(*data_s3_path*, *out_s3_path*, *resolution=15*, *num_procs=12*)
Correct illumination inhomogeneity in stitched precomputed data on S3 and upload result back to S3 as precomputed

Parameters

- **data_s3_path** (*str*) – S3 path to precomputed volume that needs to be illumination corrected
- **out_s3_path** (*str*) – S3 path to store corrected precomputed volume
- **resolution** (*int, optional*) – Resolution in microns at which illumination correction is computed. Defaults to 15.
- **num_procs** (*int, optional*) – Number of proceses to use when uploading data to S3. Defaults to 12.

cloudreg.scripts.correct_stitched_data.**process_slice**(*bias_slice*, *z*, *data_orig_path*, *data_bc_path*)
Correct and upload a single slice of data

Parameters

- **bias_slice** (*sitk.Image*) – Slice of illumination correction
- **z** (*int*) – Z slice of data to apply correction to
- **data_orig_path** (*str*) – S3 path to source data that needs to be corrected
- **data_bc_path** (*str*) – S3 path where corrected data will be stored

3.3.3 Cloud Storage Input/Output

cloudreg.scripts.download_raw_data.**download_raw_data**(*in_bucket_path*, *channel*, *outdir*)

Download COLM raw data from S3 to local storage

Parameters

- **in_bucket_path** (*str*) – Name of S3 bucket where raw dadta live at
- **channel** (*int*) – Channel number to process
- **outdir** (*str*) – Local path to store raw data

cloudreg.scripts.download_raw_data.**download_tile**(*s3*, *raw_tile_bucket*, *raw_tile_path*, *outdir*, *bias=None*)

Download single raw data image file from S3 to local directory

Parameters

- **s3** (*S3.Resource*) – A Boto3 S3 resource
- **raw_tile_bucket** (*str*) – Name of bucket with raw data
- **raw_tile_path** (*str*) – Path to raw data file in S3 bucket
- **outdir** (*str*) – Local path to store raw data
- **bias** (*np.ndarray, optional*) – Bias correction multiplied by image before saving. Must be same size as image Defaults to None.

cloudreg.scripts.download_raw_data.**download_tiles**(*tiles*, *raw_tile_bucket*, *outdir*)

Download a chunk of tiles from S3 to local storage

Parameters

- **tiles** (*list of str*) – S3 paths to raw data files to download
- **raw_tile_bucket** (*str*) – Name of bucket where raw data live
- **outdir** (*str*) – Local path to store raw data at

cloudreg.scripts.download_raw_data.**get_all_s3_objects**(*s3*, ***base_kwargs*)

Get all s3 objects with base_kwargs

Parameters **s3** (*boto3.S3.client*) – an active S3 Client.

Yields *dict* – Response object with keys to objects if there are any.

cloudreg.scripts.download_raw_data.**get_list_of_files_to_process**(*in_bucket_name*, *prefix*, *channel*)

Get paths of all raw data files for a given channel.

Parameters

- **in_bucket_name** (*str*) – S3 bucket in which raw data live
- **prefix** (*str*) – Prefix for the S3 path at which raw data live
- **channel** (*int*) – Channel number to process

Returns List of S3 paths for all raw data files

Return type list of str

cloudreg.scripts.download_raw_data.**get_out_path**(*in_path*, *outdir*)

Get output path for given tile, maintaining folder structure for Terastitcher

Parameters

- **in_path** (*str*) – S3 key to raw tile
- **outdir** (*str*) – Path to local directory to store raw data

Returns Path to store raw tile at.

Return type str

cloudreg.scripts.create_precomputed_volume.**create_cloud_volume**(*precomputed_path*, *img_size*, *voxel_size*, *num_mips*, *chunk_size*, *parallel=False*, *layer_type='image'*, *dtype='uint16'*)

Create Neuroglancer precomputed volume S3

Parameters

- **precomputed_path** (*str*) – S3 Path to location where precomputed layer will be stored
- **img_size** (*list of int*) – Size of the image (in 3D) to be uploaded
- **voxel_size** (*[type]*) – Voxel size in nanometers
- **num_mips** (*int, optional*) – Number of downsampling levels in X and Y. Defaults to 6.
- **chunk_size** (*list, optional*) – Size of each chunk stored on S3. Defaults to [1024, 1024, 1].
- **parallel** (*bool, optional*) – Whether or not the returned CloudVlue object will use parallel threads. Defaults to False.
- **layer_type** (*str, optional*) – Neuroglancer type of layer. Can be image or segmentation. Defaults to "image".
- **dtype** (*str, optional*) – Datatype of precomputed volume. Defaults to "uint16".

Returns CloudVolume object associated with this precomputed volume

Return type cloudvolume.CloudVolume

cloudreg.scripts.create_precomputed_volume.**create_precomputed_volume**(*input_path*, *voxel_size*, *precomputed_path*, *extension='tif'*)

Create precomputed volume on S3 from 2D TIF series

Parameters

- **input_path** (*str*) – Local path to 2D TIF series
- **voxel_size** (*np.ndarray*) – Voxel size of image in X,Y,Z in microns
- **precomputed_path** (*str*) – S3 path where precomputed volume will be stored
- **extension** (*str, optional*) – Extension for image files. Defaults to "tif".

cloudreg.scripts.create_precomputed_volume.**get_image_dims**(*files*)
Get X,Y,Z dimensions of images based on list of files

Parameters **files** (*list of str*) – Path to 2D tif series

Returns X,Y,Z size of image in files

Return type list of int

cloudreg.scripts.create_precomputed_volume.**process**(*z*, *file_path*, *layer_path*, *num_mips*)
Upload single slice to S3 as precomputed

Parameters

- **z** (*int*) – Z slice number to upload
- **file_path** (*str*) – Path to z-th slice
- **layer_path** (*str*) – S3 path to store data at
- **num_mips** (*int*) – Number of 2x2 downsampling levels in X,Y

cloudreg.scripts.download_data.**download_data**(*s3_path*, *outfile*, *desired_resolution=15000*, *resample_isotropic=False*, *return_size=False*)
Download whole precomputed volume from S3 at desired resolution and optionally resample data to be isotropic

Parameters

- **s3_path** (*str*) – S3 path to precomputed volume
- **outfile** (*str*) – Path to output file
- **desired_resolution** (*int, optional*) – Lowest resolution (in nanometers) at which to download data if desired res isnt available. Defaults to 15000.
- **resample_isotropic** (*bool, optional*) – If true, resample data to be isotropic.

Returns Resoluton of downloaded data in microns

Return type resolution

cloudreg.scripts.download_data.**get_mip_at_res**(*vol*, *resolution*)
Find the mip that is at least a given resolution

Parameters

- **vol** (*cloudvolume.CloudVoluem*) – CloudVolume object for desired precomputed volume
- **resolution** (*int*) – Desired resolution in nanometers

Returns mip and resolution at that mip

Return type tuple

3.3.4 Stitching

cloudreg.scripts.stitching.**generate_stitching_commands**(*stitched_dir*, *stack_dir*, *metadata_s3_bucket*, *metadata_s3_path*, *do_steps=2*)

Generate Terastitcher stitching commands given COLM metadata files.

Parameters

- **stitched_dir** (*str*) – Path to store stitched data at.
- **stack_dir** (*str*) – Path to unstiched raw data.
- **metadata_s3_bucket** (*str*) – Name of S3 bucket in which metdata is located.
- **metadata_s3_path** (*str*) – Specific path to metadata files in the bucket
- **do_steps** (*int, optional*) – Represents which Terastitcher steps to run. Defaults to ALL_STEPS (2).

Returns Metadata and list of Terastitcher commands

Return type tuple (dict, list of str)

cloudreg.scripts.stitching.**get_metadata**(*path_to_config*)

Get metadata from COLM config file.

Parameters **path_to_config** (*str*) – Path to Experiment.ini file (COLM config file)

Returns Metadata information.

Return type dict

cloudreg.scripts.stitching.**get_scanned_cells**(*fname_scanned_cells*)

Read Scanned Cells.txt file from COLM into list

Parameters **fname_scanned_cells** (*str*) – Path to scanned cells file.

Returns Indicates whether or not a given location has been imaged on the COLM

Return type list of lists

cloudreg.scripts.stitching.**run_terastitcher**(*raw_data_path*, *stitched_data_path*, *input_s3_path*, *log_s3_path=None*, *stitch_only=False*, *compute_only=False*)

Run Terastitcher commands to fully stitch raw data.

Parameters

- **raw_data_path** (*str*) – Path to raw data (VW0 folder for COLM data)
- **stitched_data_path** (*str*) – Path to where stitched data will be stored
- **input_s3_path** (*str*) – S3 Path to where raw data and metadata live
- **log_s3_path** (*str, optional*) – S3 path to store intermediates and XML files for Terastitcher. Defaults to None.
- **stitch_only** (*bool, optional*) – Do stitching only if True. Defaults to False.
- **compute_only** (*bool, optional*) – Compute alignments only if True. Defaults to False.

Returns Metadata associated with this sample from Experiment.ini file (COLM data)

Return type dict

cloudreg.scripts.stitching.**write_import_xml**(*fname_importxml*, *scanned_matrix*, *metadata*)

Write xml_import file for Terastitcher based on COLM metadata

Parameters

- **fname_importxml** (*str*) – Path to wheer xml_import.xml should be stored
- **scanned_matrix** (*list of lists*) – List of locations that have been imaged by the microscope
- **metadata** (*dict*) – Metadata assocated with this COLM experiment

cloudreg.scripts.stitching.**write_terastitcher_commands**(*fname_ts*, *metadata*, *stitched_dir*, *do_steps*)

Generate Terastitcher commands from metadata

Parameters

- **fname_ts** (*str*) – Path to bash file to store Terastitcher commands
- **metadata** (*dict*) – Metadata information about experiment
- **stitched_dir** (*str*) – Path to where stitched data will be stored
- **do_steps** (*int*) – Indicator of which steps to run

Returns List of Terastitcher commands to run

Return type list of str

This program uses a main subordinate approach to consume a queue of elaborations using teraconverter Copyright (c) 2016: Massimiliano Guarrasi (1), Giulio Iannello (2), Alessandro Bria (2) (1): CINECA (2): University Campus Bio-Medico of Rome The program was made in the framework of the HUMAN BRAIN PROJECT. All rights reserved.

EXAMPLE of usage (X is the major version, Y is the minor version, Z is the patch): mpirun -np XX python paraconverterX.Y.Z.py -s=source_volume -d=destination_path –depth=DD –height=HH –width=WW –sfmt=source_format –dfmt=destinatiopn_format –resolutions=RR

where: - XX is the desided level of parallelism plus 1 (for the main process) - DD, HH, WW are the values used to partition the image for parallel execution - source and destination format are allowed formats for teraconverter - RR are the requested resolutions (according to the convention used by teraconverter) See teraconverter documentation for more details

* Change Log *

v2.3.2 2017-10-07 - added management of –isotropic option in the partition algorithm - corrected a bug in function ‘collect_instructions’

v2.2.2 2017-10-07 - revisted platform dependent instructions

v2.2.1 2017-09-19 - added option –info to display the memory needed in GBytes without performing any

conversion

v2.2 2017-03-12 - the suspend/resume mechanism can be disabled by changing the value of variable

‘suspend_resume_enabled’ (the mechanism is enebled if True, disabled if False

- changed the policy to manage dataset partition and eliminated additional parameter to specify the desired degree of parallelism which is now directly passed by the main

v2.1 2017-02-06 - implemented a suspend/resume mechanism

the mechanism can slow down parallel execution if the dataset chunks are relatively small to avoid this a ram disk can be used to save the status (substitute the name 'output_nae' at line 953 with the path of the ram disk)

v2.0 2016-12-10 - dataset partitioning takes into account the source format in order to avoid that the

same image region is read by different TeraConverter instances; requires an additional parameter in the command line (see EXAMPLE of usage above)

cloudreg.scripts.paraconverter.**check_double_quote**(*inpstring*)

Check if some strings needs of a double quote (if some space are inside the string, it will need to be inside two double quote). E.g.: –sfmt="TIFF (unstitched, 3D)" Input:

inpstring: input string or array of strings

Output: newstring = new string (or array of strings) corrected by quoting if necessary

cloudreg.scripts.paraconverter.**check_flag**(*params*, *string*, *delete*)

Check if a parameter (string) was beeen declared in the line of commands (params) and return the associated value. If delete is true the related string will be deleted If string is not present, return None Input:

params = list of parameters from original command line string = string to be searched delete = Boolean variable to check if the selected string must be deleted after copied in value variable

Output: value = parameter associated to the selected string

cloudreg.scripts.paraconverter.**collect_instructions**(*inst*)

Collect the remanent part of a list of strings in a unique string Input:

inst = Input list of strings

Output: results = String containing all the elements of inst

cloudreg.scripts.paraconverter.**create_commands**(*gi_np*, *info=False*)

Create commands to run in parallel Input: Output:

first_string = String to initialize parallel computation list_string = Dictionary of strings containing the command lines to process the data. E.G.: {i:command[i]} len_arr = Dictionary containing elements like {index:[size_width(i),size_height(i),size_depth(i)],.....} final_string = String to merge all metadadata

cloudreg.scripts.paraconverter.**create_sizes**(*size*, *wb*, *max_res*, *norest=False*)

Create a 3D array containing the size for each tile on the desidered direction Input:

start_wb = Start parameter for b size = size (in pixel) of the input immage wb = Rough depth for the tiles in the desidered direction max_res = Maximum level of resolution available (integer) norest = Boolean variable to chech if we need of the last array element (if it is different from the preavious one)

Output: arr = Array containing the size for each tile on the desidered direction

cloudreg.scripts.paraconverter.**create_starts_end**(*array*, *start_point=0*, *open_dx=True*)

Create arrays containing all the starting and ending indexes for the tiles on the desidered direction Input:

array = Array containing the size for each tile on the desidered direction start_point = Starting index for the input immage (optional) open_dx = If true (the default value) ==> ending indexes = subsequent starting indexes ==> Open end

Output: star_arr = Array containing all the starting indexes for the tiles on the desidered direction end_arr = Array containing all the ending indexes for the tiles on the desidered direction

cloudreg.scripts.paraconverter.**eliminate_double_quote**(*inpstring*)
Check if the string is already enclosed by quotes Input:

inpstring: input string or array of strings

Output: newstring = new string (or array of strings) corrected by eliminating enclosing quotes if any

cloudreg.scripts.paraconverter.**extract_params**()
Extract parameter from line of commands. Output:

params = list of parameters from original command line

cloudreg.scripts.paraconverter.**generate_final_command**(*input_name*, *output_name*, *wb1*, *wb2*, *wb3*, *sfmt*, *dfmt*, *iresolutions*, *max_res*, *params*, *last_string*)
Generate last command line to merge metadata Input:

input_name = Input file output_name = Standard output directory wb1 = Axprossimative depth for the tiles wb2 = Axprossimative height for the tiles wb3 = Axprossimative width for the tiles sfmt = Source format dfmt = Destination format iresolutions = List of integer values containing all the desidered values for level of resolution max_res = Maximum level of resolution available (integer) params = Array containing instruction derived from the remanent part of the imput string last_string = Remanent part of the input string

Output: final_string = Command line to merge metadata

cloudreg.scripts.paraconverter.**generate_first_command**(*input_name*, *output_name*, *wb1*, *wb2*, *wb3*, *sfmt*, *dfmt*, *iresolutions*, *max_res*, *params*, *last_string*)
Generate first command line Input:

input_name = Input file output_name = Standard output directory wb1 = Axprossimative depth for the tiles wb2 = Axprossimative height for the tiles wb3 = Axprossimative width for the tiles sfmt = Source format dfmt = Destination format iresolutions = List of integer values containing all the desidered values for level of resolution max_res = Maximum level of resolution available (integer) params = Array containing instruction derived from the remanent part of the imput string last_string = Remanent part of the input string

Output: first_string = Command line to preprocess the data

cloudreg.scripts.paraconverter.**generate_parallel_command**(*start_list*, *end_list*, *input_name*, *output_name*, *wb1*, *wb2*, *wb3*, *sfmt*, *dfmt*, *iresolutions*, *max_res*, *params*, *last_string*)
Generate the list of parallel command lines Input:

start_list = Ordered list of lists of starting points. E.g.: [[width_in[0], height_in[0], depth_in[0]], [width_in[1], height_in[1], depth_in[1]], ... ,[width_in[N], height_in[N], depth_in[N]]] end_list = Ordered list of lists of starting points. E.g.: [[width_fin[0], height_fin[0], depth_in[0]], [width_fin[1], height_fin[1], depth_fin[1]], ... ,[width_fin[N], height_fin[N], depth_fin[N]]] input_name = Input

file output_name = Standard output directory wb1 = Axprossimative depth for the tiles wb2 = Axprossimative height for the tiles wb3 = Axprossimative width for the tiles sfmt = Source format dfmt = Destination format iresolutions = List of integer values containing all the desidered values for level of resolution max_res = Maximum level of resolution available (integer) params = Array containing instruction derived from the remanent part of the imput string last_string = Remanent part of the input string

Output: list_string = Dictionary of strings containing the command lines to process the data. E.G.: {i:command[i]}

cloudreg.scripts.paraconverter.**main**(*queue*, *rs_fname*)

Dispatch the work among processors. Input:

queue = list of job inputs

cloudreg.scripts.paraconverter.**opt_algo**(*D*, *w*, *n*)

Solves the tiling problem patitioning the interval [0, D-1] into k subintervals of size 2^n b and one final subinterval of size r = D - k 2^n b Input:

D = dimension of the original array w = approximate estimation of value for b n = desideres level of refinement (e.g. : n = 0 => maximum level of refinement; n =1 => number of point divided by 2^1=2; n = 2 => number of point divided by 2^2=4;)

Output:

arr_sizes = [b, r, k, itera] b = normalized size of standard blocks (size of standard blocks = b * 2^n) r = rest (if not equal to 0, is the size of the last block) k = number of standard blocks itera = number of itarations to converge

cloudreg.scripts.paraconverter.**pop_left**(*dictionary*)

Cuts the first element of dictionary and returns its first element (key:value) Input/Output:

dictionary = Dictionary of string containing the command lines to use. After reading the dictionary the first element is deleted from the dictionary.

Output: first_el = first element (values) of the dictionary

cloudreg.scripts.paraconverter.**prep_array**(*wb*, *r*, *k*)

Create a 1D array containing the number of elements per tile. Input:

wb = size of standard blocks r = rest (if not equal to 0, is the size of the last block) k = number of standard blocks

Output: array = A list containing the number of element for every tiles.

cloudreg.scripts.paraconverter.**read_item**(*input_arr*, *item*, *default*, *message=True*)

Read the value related to "item" from the list "input_arr" and if no item are present set it to "default". Please note: The function convert the output to the same type of "default" variable Input:

input_arr = List of strings from imput command line item = The item to search default = The default value if no item are present

Output: value = Output value for the selected item

cloudreg.scripts.paraconverter.**read_params**()

Read parameters from input string and from a file Input: Output:

input_name = Input file output_name = Standard output directory wb1 = Axprossimative depth for the tiles wb2 = Axprossimative height for the tiles wb3 = Axprossimative width for the tiles sfmt = Source format dfmt = Destination format iresolutions = List of integer values containing all the desidered values for level of resolution max_res = Maximum level of resolution available (integer) params = Array containing instruction derived from the remanent part of the imput string last_string = Remanent part of the input string height = Height of the input immage width = Width of the input immage depth = Depth of the input immage

cloudreg.scripts.paraconverter.**score_function**(*params*)

Assigns a score value with the formula: score = 100*N_of_voxel/max(N_of_voxel)

Input: params = dictionary containing {input_name : [Nx,Ny,Nz]}

Output: scores = dictionary containing {input_name : score}

cloudreg.scripts.paraconverter.**search_for_entry**(*string_2_serch*, *file_in*, *nline=0*)

Extract from the input file (file_in) up to the line number nline (if declared) the value assigned to string_2_serch. Input:

string_2_serch = string (or list of string) containing the variable to search (e.g. 'HEIGHT=') file_in = name of the file containing the information we neeed (e.g: prova.txt or /pico/home/prova.txt) nline = optional, number of the final row of the file we need to analyze

Output: Output = value or (list of values) assigned to the variable conteined in string_2_serch

cloudreg.scripts.paraconverter.**sort_elaborations**(*scores*)

Create a list of input_name sorted by score Input:

scores = dictionary of the form {input_name : score}

Output: scored = a list of input_name sorted by score

cloudreg.scripts.paraconverter.**sort_list**(*len_1*, *len_2*, *len_3*)

Create a list sorting the indexes along three directions: Input:

len_1 = Number of elements of the array for the first index len_2 = Number of elements of the array for the second index len_3 = Number of elements of the array for the third index

Output: order = An ordered list containig an a sequence of lists of 3 alements (one for each direction) that identify the position on the local index

cloudreg.scripts.paraconverter.**sort_start_end**(*start_1*, *start_2*, *start_3*, *end_1*, *end_2*, *end_3*, *size_1*, *size_2*, *size_3*)

Sort start points and edn point in two lists of elements Input:

start_1 = Array containing all the starting indexes for the tiles on the Depth direction start_2 = Array containing all the starting indexes for the tiles on the Height direction start_3 = Array containing all the starting indexes for the tiles on the Width direction end_1 = Array containing all the ending indexes for the tiles on the Depth direction end_2 = Array containing all the ending indexes for the tiles on the Height direction end_3 = Array containing all the ending indexes for the tiles on the Width direction size_1 = Array containing the size of the tile in the Depth direction size_2 = Array containing the size of the tile in the Height direction size_3 = Array containing the size of the tile in the Width direction

Output: order = An ordered list containig an a sequence of lists of 3 alements (one for each direction) that identify the position on the local index start_list = Ordered list of lists of starting points. E.g.: [[width_in[0], height_in[0], depth_in[0]], [width_in[1], height_in[1], depth_in[1]], ... ,[width_in[N], height_in[N], depth_in[N]]] end_list = Ordered list of lists of starting points. E.g.: [[width_fin[0], height_fin[0],

depth_in[0]], [width_fin[1], height_fin[1], depth_fin[1]], . . . ,[width_fin[N], height_fin[N], depth_fin[N]]]
len_arr = Dictionary containing elements like {index:[size_width(i),size_height(i),size_depth(i)],. . . ..}

cloudreg.scripts.paraconverter.**sort_work**(*params*, *priority*)
Returns a dictionary as params but ordered by score Input:

params = dictionary of the form {input_name : value} priority = the list of input_name ordered by score calculated by score_function

Output: sorted_dict = the same dictionary as params but ordered by score

cloudreg.scripts.paraconverter.**subordinate**()
Subordinate process.

cloudreg.scripts.paraconverter.**worker**(*input_file*)
Perform elaboration for each element of the queue. Input/Output

input_file = command to be executed

This program uses a main subordinate approach to consume a queue of elaborations using teraconverter Copyright (c) 2016: Massimiliano Guarrasi (1), Giulio Iannello (2), Alessandro Bria (2) (1): CINECA (2): University Campus Bio-Medico of Rome The program was made in the framework of the HUMAN BRAIN PROJECT. All rights reserved.

EXAMPLE of usage (X is the major version, Y is the minor version, Z is the patch):

For align step: mpirun -np XX python ParastitcherX.Y.Z.py -2 –projin=xml_import_file –projout=xml_displcomp_file [–sV=VV] [–sH=HH] [–sD=DD] [–imin_channel=C] [. . .]

where: - XX is the desided level of parallelism plus 1 (for the main process) - VV, HH, DD are the half size of the NCC map along V, H, and D directions, respectively - C is the input channel to be used for align computation

For fusion step: mpirun -np XX python ParastitcherX.Y.Z.py -6 –projin=xml_import_file –volout=destination_folder –volout_plugin=format_string [–slicewidth=WWW] [–sliceheight=HHH] [–slicedepth=DDD] [–resolutions=RRR] [. . .]

where: - format_string is one the formats: “TIFF (series, 2D)”, “TIFF, tiled, 2D”, “TIFF, tiled, 3D”, “TIFF, tiled, 4D”, - DDD, HHH, WWW are the values used to partition the image for parallel execution - RRR are the requested resolutions (according to the convention used by teraconverter) See teraconverter documentation for more details

* Change Log *

2018-09-05. Giulio. @CHSNGED on non-Windows platforms ‘prefix’ is automatically switched to ‘./’ if executables are not in the system path 2018-08-16. Giulio. @CHANGED command line interface: parameters for step 6 are the same than the sequential implementation 2018-08-16. Giulio. @ADDED debug control 2018-08-07. Giulio. @CREATED from parastitcher2.0.3.py and paraconverter2.3.2.py

---

terastitcher -2 –projin=/Users/iannello/Home/Windows/myTeraStitcher/TestData/Ailey/blending/test_00_01_02_03/xml_import_org.xm
–projout=/Users/iannello/Home/Windows/myTeraStitcher/TestData/Ailey/blending/test_00_01_02_03/xml_displcomp_seq.xml

mpirun -np 3 python /Users/iannello/Home/Windows/paratools/parastitcher2.0.3.py -2 –projin=/Users/iannello/Home/Windows/myTeraStitcher/TestData/Ailey/blending/test_00_01_02_03/xml_import_org.xml
–projout=/Users/iannello/Home/Windows/myTeraStitcher/TestData/Ailey/blending/test_00_01_02_03/xml_displcomp_par2.xml
mpirun -np 3 python /Users/iannello/Home/Windows/paratools/Parastitcher3.0.0.py -2 –projin=/Users/iannello/Home/Windows/myTeraStitcher/TestData/Ailey/blending/test_00_01_02_03/xml_import_org.xml
–projout=/Users/iannello/Home/Windows/myTeraStitcher/TestData/Ailey/blending/test_00_01_02_03/xml_displcomp_par2.xml

teraconverter –sfmt=”TIFF (unstitched, 3D)” -s=/Users/iannello/Home/Windows/myTeraStitcher/TestData/Ailey/blending/test_00_01_0
–dfmt=”TIFF (series, 2D)” -d=/Users/iannello/Home/Windows/myTeraStitcher/TestData/temp/result_p1 –resolutions=012 –depth=256 –width=256 –height=256

mpirun -np 3 python /Users/iannello/Home/Windows/paratools/paraconverter2.3.2.py –sfmt=”TIFF (unstitched, 3D)” -s=/Users/iannello/Home/Windows/myTeraStitcher/TestData/Ailey/blending/test_00_01_02_03/xml_merging.xml –dfmt=”TIFF (tiled, 3D)” -d=/Users/iannello/Home/Windows/myTeraStitcher/TestData/temp/result_p1 –resolutions=012 –depth=256 –width=256 –height=256 mpirun -np 3 python /Users/iannello/Home/Windows/paratools/Parastitcher3.0.0.py -6 –sfmt=”TIFF (unstitched, 3D)” -s=/Users/iannello/Home/Windows/myTeraStitcher/TestData/Ailey/blending/test_00_01_02_03/xml_merging.xml –dfmt=”TIFF (tiled, 3D)” -d=/Users/iannello/Home/Windows/myTeraStitcher/TestData/temp/result_p1 –resolutions=012 –depth=256 –width=256 –height=256

cloudreg.scripts.parastitcher.**check_double_quote**(*inpstring*)

Check if some strings needs of a double quote (if some space are inside the string, it will need to be inside two double quote). E.g.: –sfmt=”TIFF (unstitched, 3D)” Input:

inpstring: input string or array of strings

Output: newstring = new string (or array of strings) corrected by quoting if necessary

cloudreg.scripts.parastitcher.**check_flag**(*params*, *string*, *delete*)

Check if a parameter (string) was beeen declared in the line of commands (params) and return the associated value. If delete is true the related string will be deleted If string is not present, return None Input:

params = list of parameters from original command line string = string to be searched delete = Boolean variable to check if the selected string must be deleted after copied in value variable

Output: value = parameter associated to the selected string

cloudreg.scripts.parastitcher.**collect_instructions**(*inst*)

Collect the remanent part of a list of strings in a unique string Input:

inst = Input list of strings

Output: results = String containing all the elements of inst

cloudreg.scripts.parastitcher.**create_commands**(*gi_np*, *info=False*)

Create commands to run in parallel Input: Output:

first_string = String to initialize parallel computation list_string = Dictionary of strings containing the command lines to process the data. E.G.: {i:command[i]} len_arr = Dictionary containing elements like {index:[size_width(i),size_height(i),size_depth(i)],.....} final_string = String to merge all metadadata

cloudreg.scripts.parastitcher.**create_sizes**(*size*, *wb*, *max_res*, *norest=False*)

Create a 3D array containing the size for each tile on the desidered direction Input:

start_wb = Start parameter for b size = size (in pixel) of the input immage wb = Rough depth for the tiles in the desidered direction max_res = Maximum level of resolution available (integer) norest = Boolean variable to chech if we need of the last array element (if it is different from the preavious one)

Output: arr = Array containing the size for each tile on the desidered direction

cloudreg.scripts.parastitcher.**create_starts_end**(*array*, *start_point=0*, *open_dx=True*)

Create arrays containing all the starting and ending indexes for the tiles on the desidered direction Input:

array = Array containing the size for each tile on the desidered direction start_point = Starting index for the input immage (optional) open_dx = If true (the default value) ==> ending indexes = subsequent starting indexes ==> Open end

Output: star_arr = Array containing all the starting indexes for the tiles on the desidered direction end_arr = Array containing all the ending indexes for the tiles on the desidered direction

cloudreg.scripts.parastitcher.**do_additional_partition**(*nprocs*, *nrows*, *ncols*, *n_ss*)
All parameters should be float

cloudreg.scripts.parastitcher.**eliminate_double_quote**(*inpstring*)
Check if the string is already enclosed by quotes Input:

inpstring: input string or array of strings

Output: newstring = new string (or array of strings) corrected by eliminating enclosing quotes if any

cloudreg.scripts.parastitcher.**extract_np**(*inputf*)
extract the number of slices along z from the input xml file.

cloudreg.scripts.parastitcher.**extract_params**()
Extract parameter from line of commands. Output:

params = list of parameters from original command line

cloudreg.scripts.parastitcher.**find_last_slash**(*string*)
Search for / in a string. If one or more / was found, divide the string in a list of two string: the first containf all the character at left of the last / (included), and the second contains the remanent part of the text. If no / was found, the first element of the list will be set to ''

cloudreg.scripts.parastitcher.**generate_final_command**(*input_name*, *output_name*, *wb1*, *wb2*, *wb3*, *sfmt*, *dfmt*, *iresolutions*, *max_res*, *params*, *last_string*)
Generate last command line to merge metadata Input:

input_name = Input file output_name = Standard output directory wb1 = Axprossimative depth for the tiles wb2 = Axprossimative height for the tiles wb3 = Axprossimative width for the tiles sfmt = Source format dfmt = Destination format iresolutions = List of integer values containing all the desidered values for level of resolution max_res = Maximum level of resolution available (integer) params = Array containing instruction derived from the remanent part of the imput string last_string = Remanent part of the input string

Output: final_string = Command line to merge metadata

cloudreg.scripts.parastitcher.**generate_first_command**(*input_name*, *output_name*, *wb1*, *wb2*, *wb3*, *sfmt*, *dfmt*, *iresolutions*, *max_res*, *params*, *last_string*)
Generate first command line Input:

input_name = Input file output_name = Standard output directory wb1 = Axprossimative depth for the tiles wb2 = Axprossimative height for the tiles wb3 = Axprossimative width for the tiles sfmt = Source format dfmt = Destination format iresolutions = List of integer values containing all the desidered values for level of resolution max_res = Maximum level of resolution available (integer) params = Array containing instruction derived from the remanent part of the imput string last_string = Remanent part of the input string

Output: first_string = Command line to preprocess the data

cloudreg.scripts.parastitcher.**generate_parallel_command**(*start_list*, *end_list*, *input_name*, *output_name*, *wb1*, *wb2*, *wb3*, *sfmt*, *dfmt*, *iresolutions*, *max_res*, *params*, *last_string*)

Generate the list of parallel command lines Input:

start_list = Ordered list of lists of starting points. E.g.: [[width_in[0], height_in[0], depth_in[0]], [width_in[1], height_in[1], depth_in[1]], ... ,[width_in[N], height_in[N], depth_in[N]]] end_list = Ordered list of lists of starting points. E.g.: [[width_fin[0], height_fin[0], depth_in[0]], [width_fin[1], height_fin[1], depth_fin[1]], ... ,[width_fin[N], height_fin[N], depth_fin[N]]] input_name = Input file output_name = Standard output directory wb1 = Axprossimative depth for the tiles wb2 = Axprossimative height for the tiles wb3 = Axprossimative width for the tiles sfmt = Source format dfmt = Destination format iresolutions = List of integer values containing all the desidered values for level of resolution max_res = Maximum level of resolution available (integer) params = Array containing instruction derived from the remanent part of the imput string last_string = Remanent part of the input string

Output: list_string = Dictionary of strings containing the command lines to process the data. E.G.: {i:command[i]}

cloudreg.scripts.parastitcher.**main_step2**(*queue*)

dispatch the work among processors

queue is a list of job input

cloudreg.scripts.parastitcher.**main_step6**(*queue*, *rs_fname*)

Dispatch the work among processors. Input:

queue = list of job inputs

cloudreg.scripts.parastitcher.**opt_algo**(*D*, *w*, *n*)

Solves the tiling problem patitioning the interval [0, D-1] into k subintervals of size 2^n b and one final subinterval of size r = D - k 2^n b Input:

D = dimension of the original array w = approximate estimation of value for b n = desideres level of refinement (e.g. : n = 0 => maximum level of refinement; n =1 => number of point divided by 2^1=2; n = 2 => number of point divided by 2^2=4;)

Output:

arr_sizes = [b, r, k, itera] b = normalized size of standard blocks (size of standard blocks = b * 2^n) r = rest (if not equal to 0, is the size of the last block) k = number of standard blocks itera = number of itarations to converge

cloudreg.scripts.parastitcher.**partition**(*m*, *n*, *N*)

return the number of partitions along V and H, respectively that are optimal to partition a block of size m_V x n_H in at least P sub-blocks

m: block size along V n: block size along H N: number of required partitions

return: p_m, p_n: the number of partitions along V and H, respectively

PRE:

cloudreg.scripts.parastitcher.**pop_left**(*dictionary*)

Cuts the first element of dictionary and returns its first element (key:value) Input/Output:

dictionary = Dictionary of string containing the command lines to use. After reading the dictionary the first element is deleted from the dictionary.

Output: first_el = first element (values) of the dictionary

cloudreg.scripts.parastitcher.**prep_array**(*wb*, *r*, *k*)
Create a 1D array containing the number of elements per tile. Input:

wb = size of standard blocks r = rest (if not equal to 0, is the size of the last block) k = number of standard blocks

Output: array = A list containing the number of element for every tiles.

cloudreg.scripts.parastitcher.**read_input**(*inputf*, *nline=0*)
Reads the file included in inputf at least up to line number nline (if declared).

cloudreg.scripts.parastitcher.**read_item**(*input_arr*, *item*, *default*, *message=True*)
Read the value related to "item" from the list "input_arr" and if no item are present set it to "default". Please note: The function convert the output to the same type of "default" variable Input:

input_arr = List of strings from imput command line item = The item to search default = The default value if no item are present

Output: value = Output value for the selected item

cloudreg.scripts.parastitcher.**read_params**()
Read parameters from input string and from a file Input: Output:

input_name = Input file output_name = Standard output directory wb1 = Axprossimative depth for the tiles wb2 = Axprossimative height for the tiles wb3 = Axprossimative width for the tiles sfmt = Source format dfmt = Destination format iresolutions = List of integer values containing all the desidered values for level of resolution max_res = Maximum level of resolution available (integer) params = Array containing instruction derived from the remanent part of the imput string last_string = Remanent part of the input string height = Height of the input immage width = Width of the input immage depth = Depth of the input immage

cloudreg.scripts.parastitcher.**score_function**(*params*)

Assigns a score value with the formula: score = 100*N_of_voxel/max(N_of_voxel)

Input: params = dictionary containing {input_name : [Nx,Ny,Nz]}

Output: scores = dictionary containing {input_name : score}

cloudreg.scripts.parastitcher.**search_for_entry**(*string_2_serch*, *file_in*, *nline=0*)
Extract from the input file (file_in) up to the line number nline (if declared) the value assigned to string_2_serch. Input:

string_2_serch = string (or list of string) containing the variable to search (e.g. 'HEIGHT=') file_in = name of the file containing the information we neeed (e.g: prova.txt or /pico/home/prova.txt) nline = optional, number of the final row of the file we need to analyze

Output: Output = value or (list of values) assigned to the variable conteined in string_2_serch

cloudreg.scripts.parastitcher.**sort_elaborations**(*scores*)
Create a list of input_name sorted by score Input:

scores = dictionary of the form {input_name : score}

Output: scored = a list of input_name sorted by score

cloudreg.scripts.parastitcher.**sort_list**(*len_1*, *len_2*, *len_3*)
Create a list sorting the indexes along three directions: Input:

len_1 = Number of elements of the array for the first index len_2 = Number of elements of the array for the second index len_3 = Number of elements of the array for the third index

Output: order = An ordered list containig an a sequence of lists of 3 alements (one for each direction) that identify the position on the local index

cloudreg.scripts.parastitcher.**sort_start_end**(*start_1*, *start_2*, *start_3*, *end_1*, *end_2*, *end_3*, *size_1*, *size_2*, *size_3*)
Sort start points and edn point in two lists of elements Input:

start_1 = Array containing all the starting indexes for the tiles on the Depth direction start_2 = Array containing all the starting indexes for the tiles on the Height direction start_3 = Array containing all the starting indexes for the tiles on the Width direction end_1 = Array containing all the ending indexes for the tiles on the Depth direction end_2 = Array containing all the ending indexes for the tiles on the Height direction end_3 = Array containing all the ending indexes for the tiles on the Width direction size_1 = Array containing the size of the tile in the Depth direction size_2 = Array containing the size of the tile in the Height direction size_3 = Array containing the size of the tile in the Width direction

Output: order = An ordered list containig an a sequence of lists of 3 alements (one for each direction) that identify the position on the local index start_list = Ordered list of lists of starting points. E.g.: [[width_in[0], height_in[0], depth_in[0]], [width_in[1], height_in[1], depth_in[1]], ... ,[width_in[N], height_in[N], depth_in[N]]] end_list = Ordered list of lists of starting points. E.g.: [[width_fin[0], height_fin[0], depth_in[0]], [width_fin[1], height_fin[1], depth_fin[1]], ... ,[width_fin[N], height_fin[N], depth_fin[N]]] len_arr = Dictionary containing elements like {index:[size_width(i),size_height(i),size_depth(i)],.....}

cloudreg.scripts.parastitcher.**sort_work**(*params*, *priority*)
Returns a dictionary as params but ordered by score Input:

params = dictionary of the form {input_name : value} priority = the list of input_name ordered by score calculated by score_function

Output: sorted_dict = the same dictionary as params but ordered by score

cloudreg.scripts.parastitcher.**subordinate**()
Subordinate process.

cloudreg.scripts.parastitcher.**worker**(*input_file*)
Perform elaboration for each element of the queue. Input/Output

input_file = command to be executed

3.3.5 Registration

cloudreg.scripts.run_registration_ec2.**run_registration**(*ssh_key_path*, *instance_id*, *instance_type*, *input_s3_path*, *atlas_s3_path*, *parcellation_s3_path*, *atlas_orientation*, *output_s3_path*, *log_s3_path*, *initial_translation*, *initial_rotation*, *orientation*, *fixed_scale*, *missing_data_correction*, *grid_correction*, *bias_correction*, *sigma_regularization*, *num_iterations*, *registration_resolution*)

Run EM-LDDMM registration on an AWS EC2 instance

Parameters

- **ssh_key_path** (*str*) – Local path to ssh key for this server
- **instance_id** (*str*) – ID of EC2 instance to use
- **instance_type** (*str*) – AWS EC2 instance type. Recommended is r5.8xlarge
- **input_s3_path** (*str*) – S3 path to precomputed data to be registered
- **atlas_s3_path** (*str*) – S3 path to atlas data to register to
- **parcellation_s3_path** (*str*) – S3 path to corresponding atlas parcellations
- **output_s3_path** (*str*) – S3 path to store precomputed volume of atlas transformed to input data
- **log_s3_path** (*str*) – S3 path to store intermediates at
- **initial_translation** (*list of float*) – Initial translations in x,y,z of input data
- **initial_rotation** (*list*) – Initial rotation in x,y,z for input data
- **orientation** (*str*) – 3-letter orientation of input data
- **fixed_scale** (*float*) – Isotropic scale factor on input data
- **missing_data_correction** (*bool*) – Perform missing data correction to ignore zeros in image
- **grid_correction** (*bool*) – Perform grid correction (for COLM data)
- **bias_correction** (*bool*) – Perform illumination correction
- **sigma_regularization** (*float*) – Regularization constat in cost function. Higher regularization constant means less regularization
- **num_iterations** (*int*) – Number of iterations of EM-LDDMM to run
- **registration_resolution** (*int*) – Minimum resolution at which the registration is run.

cloudreg.scripts.registration.**get_affine_matrix**(*translation*, *rotation*, *from_orientation*, *to_orientation*, *fixed_scale*, *s3_path*, *center=False*)

Get Neuroglancer-compatible affine matrix transfrming precomputed volume given set of translations and rotations

Parameters

- **translation** (*list of float*) – x,y,z translations respectively in microns
- **rotation** (*list of float*) – x,y,z rotations respectively in degrees
- **from_orientation** (*str*) – 3-letter orientation of source data
- **to_orientation** (*str*) – 3-letter orientation of target data
- **fixed_scale** (*float*) – Isotropic scale factor
- **s3_path** (*str*) – S3 path to precomputed volume for source data
- **center** (*bool, optional*) – If true, center image at it's origin. Defaults to False.

Returns Returns 4x4 affine matrix representing the given translations and rotations of source data at S3 path

Return type np.ndarray

cloudreg.scripts.registration.**register**(*input_s3_path*, *atlas_s3_path*, *parcellation_s3_path*, *atlas_orientation*, *output_s3_path*, *log_s3_path*, *orientation*, *fixed_scale*, *translation*, *rotation*, *missing_data_correction*, *grid_correction*, *bias_correction*, *regularization*, *num_iterations*, *registration_resolution*)

Run EM-LDDMM registration on precomputed volume at input_s3_path

Parameters

- **input_s3_path** (*str*) – S3 path to precomputed data to be registered
- **atlas_s3_path** (*str*) – S3 path to atlas to register to.
- **parcellation_s3_path** (*str*) – S3 path to atlas to register to.
- **atlas_orientation** (*str*) – 3-letter orientation of atlas
- **output_s3_path** (*str*) – S3 path to store precomputed volume of atlas transformed to input data
- **log_s3_path** (*str*) – S3 path to store intermediates at
- **orientation** (*str*) – 3-letter orientation of input data
- **fixed_scale** (*float*) – Isotropic scale factor on input data
- **translation** (*list of float*) – Initial translations in x,y,z of input data
- **rotation** (*list*) – Initial rotation in x,y,z for input data
- **missing_data_correction** (*bool*) – Perform missing data correction to ignore zeros in image
- **grid_correction** (*bool*) – Perform grid correction (for COLM data)
- **bias_correction** (*bool*) – Perform illumination correction
- **regularization** (*float*) – Regularization constat in cost function. Higher regularization constant means less regularization

- **num_iterations** (*int*) – Number of iterations of EM-LDDMM to run
- **registration_resolution** (*int*) – Minimum resolution at which the registration is run.

3.3.6 Utility Functions

class cloudreg.scripts.util.**S3Url**(*url*)

```
>>> s = S3Url("s3://bucket/hello/world")
>>> s.bucket
'bucket'
>>> s.key
'hello/world'
>>> s.url
's3://bucket/hello/world'
```

```
>>> s = S3Url("s3://bucket/hello/world?qwe1=3#ddd")
>>> s.bucket
'bucket'
>>> s.key
'hello/world?qwe1=3#ddd'
>>> s.url
's3://bucket/hello/world?qwe1=3#ddd'
```

```
>>> s = S3Url("s3://bucket/hello/world#foo?bar=2")
>>> s.key
'hello/world#foo?bar=2'
>>> s.url
's3://bucket/hello/world#foo?bar=2'
```

Attributes

bucket

key

url

cloudreg.scripts.util.**aws_cli**(**cmd*)
Run an AWS CLI command

Raises **RuntimeError** – Error running aws cli command.

cloudreg.scripts.util.**calc_hierarchy_levels**(*img_size*, *lowest_res=1024*)
Compute max number of mips for given chunk size

Parameters

- **img_size** (*list*) – Size of image in x,y,z
- **lowest_res** (*int, optional*) – minimum chunk size in XY. Defaults to 1024.

Returns Number of mips

Return type int

cloudreg.scripts.util.**chunks**(*l*, *n*)
Convert a list into n-size chunks (last chunk may have less than n elements)

Parameters

- **l** (*list*) – List to chunk
- **n** (*int*) – Elements per chunk

Yields *list* – n-size chunk from l (last chunk may have fewer than n elements)

cloudreg.scripts.util.**download_terastitcher_files**(*s3_path*, *local_path*)
Download terastitcher files from S3

Parameters

- **s3_path** (*str*) – S3 path where Terastitcher files might live
- **local_path** (*str*) – Local path to save Terastitcher files

Returns True if files exist at s3 path, else False

Return type bool

cloudreg.scripts.util.**get_bias_field**(*img*, *mask=None*, *scale=1.0*, *niters=[50, 50, 50, 50]*)
Correct bias field in image using the N4ITK algorithm (http://bit.ly/2oFwAun)

Parameters

- **img** (*SimpleITK.Image*) – Input image with bias field.
- **mask** (*SimpleITK.Image, optional*) – If used, the bias field will only be corrected within the mask. (the default is None, which results in the whole image being corrected.)
- **scale** (*float, optional*) – Scale at which to compute the bias correction. (the default is 0.25, which results in bias correction computed on an image downsampled to 1/4 of it's original size)
- **niters** (*list, optional*) – Number of iterations per resolution. Each additional entry in the list adds an additional resolution at which the bias is estimated. (the default is [50, 50, 50, 50] which results in 50 iterations per resolution at 4 resolutions)

Returns Bias-corrected image that has the same size and spacing as the input image.

Return type SimpleITK.Image

cloudreg.scripts.util.**get_matching_s3_keys**(*bucket*, *prefix=''*, *suffix=''*)
Generate the keys in an S3 bucket.

Parameters

- **bucket** (*str*) – Name of the S3 bucket.
- **prefix** (*str*) – Only fetch keys that start with this prefix (optional).
- **suffix** (*str*) – Only fetch keys that end with this suffix (optional).

Yields *str* – S3 keys if they exist with given prefix and suffix

cloudreg.scripts.util.**get_reorientations**(*in_orient*, *out_orient*)
Generates a list of axes flips and swaps to convert from in_orient to out_orient

Parameters

- **in_orient** (*str*) – 3-letter input orientation
- **out_orient** (*str*) – 3-letter output orientation

Raises **Exception** – Exception raised if in_orient or out_orient not valid

Returns New axis order and whether or not each axis needs to be flipped

Return type tuple of lists

cloudreg.scripts.util.**imgResample**(*img*, *spacing*, *size=[]*, *useNearest=False*, *origin=None*, *outsideValue=0*)

Resample image to certain spacing and size.

Parameters

- **img** (*SimpleITK.Image*) – Input 3D image.
- **spacing** (*list*) – List of length 3 indicating the voxel spacing as [x, y, z]
- **size** (*list, optional*) – List of length 3 indicating the number of voxels per dim [x, y, z] (the default is [], which will use compute the appropriate size based on the spacing.)
- **useNearest** (*bool, optional*) – If True use nearest neighbor interpolation. (the default is False, which will use linear interpolation.)
- **origin** (*list, optional*) – The location in physical space representing the [0,0,0] voxel in the input image. (the default is [0,0,0])
- **outsideValue** (*int, optional*) – value used to pad are outside image (the default is 0)

Returns Resampled input image.

Return type SimpleITK.Image

cloudreg.scripts.util.**run_command_on_server**(*command*, *ssh_key_path*, *ip_address*, *username='ubuntu'*)

Run command on remote server

Parameters

- **command** (*str*) – Command to run
- **ssh_key_path** (*str*) – Local path to ssh key neeed for this server
- **ip_address** (*str*) – IP Address of server to connect to
- **username** (*str, optional*) – Username on remote server. Defaults to “ubuntu”.

Returns Errors encountered on remote server if any

Return type str

cloudreg.scripts.util.**start_ec2_instance**(*instance_id*, *instance_type*)

Start an EC2 instance

Parameters

- **instance_id** (*str*) – ID of EC2 instance to start
- **instance_type** (*str*) – Type of EC2 instance to start

Returns Public IP address of EC2 instance

Return type str

cloudreg.scripts.util.**tqdm_joblib**(*tqdm_object*)

Context manager to patch joblib to report into tqdm progress bar given as argument

cloudreg.scripts.util.**upload_file_to_s3**(*local_path*, *s3_bucket*, *s3_key*)

Upload file to S3 from local storage

Parameters

- **local_path** (*str*) – Local path to file

- **s3_bucket** (*str*) – S3 bucket name
- **s3_key** (*str*) – S3 key to store file at

class cloudreg.scripts.visualization.**S3Url**(*url*)

```
>>> s = S3Url("s3://bucket/hello/world")
>>> s.bucket
'bucket'
>>> s.key
'hello/world'
>>> s.url
's3://bucket/hello/world'
```

```
>>> s = S3Url("s3://bucket/hello/world?qwe1=3#ddd")
>>> s.bucket
'bucket'
>>> s.key
'hello/world?qwe1=3#ddd'
>>> s.url
's3://bucket/hello/world?qwe1=3#ddd'
```

```
>>> s = S3Url("s3://bucket/hello/world#foo?bar=2")
>>> s.key
'hello/world#foo?bar=2'
>>> s.url
's3://bucket/hello/world#foo?bar=2'
```

Attributes

bucket

key

url

cloudreg.scripts.visualization.**create_viz_link**(*s3_layer_paths*, *affine_matrices=None*, *shader_controls=None*, *url='https://json.neurodata.io/v1'*, *neuroglancer_link='https://ara.viz.neurodata.io/?json_url='*, *output_resolution=array([0.0001, 0.0001, 0.0001])*)

Create a viz link from S3 layer paths using Neurodata's deployment of Neuroglancer and Neurodata's json state server.

Parameters

- **s3_layer_paths** (*list*) – List of S3 paths to precomputed volumes to include in the viz link.
- **affine_matrices** (*list of np.ndarray, optional*) – List of affine matrices associated with each layer. Affine matrices should be 3x3 for 2D data and 4x4 for 3D data. Defaults to None.
- **shader_controls** (*str, optional*) – String of shader controls compliant with Neuroglancer shader controls. Defaults to None.
- **url** (*str, optional*) – URL to JSON state server to store Neueroglancer JSON state. Defaults to "https://json.neurodata.io/v1".

- **neuroglancer_link** (*str, optional*) – URL for Neuroglancer deployment, default is to use Neurodata deployment of Neuroglancer.. Defaults to "https://ara.viz.neurodata.io/?json_url=".
- **output_resolution** (*np.ndarray, optional*) – Desired output resolution for all layers in nanometers. Defaults to np.array([1e-4] * 3) nanometers.

Returns viz link to data

Return type str

cloudreg.scripts.visualization.**get_layer_json**(*s3_layer_path*, *affine_matrix*, *output_resolution*)

Generate Neuroglancer JSON for single layer.

Parameters

- **s3_layer_path** (*str*) – S3 path to precomputed layer.
- **affine_matrix** (*np.ndarray*) – Affine matrix to apply to current layer. Translation in this matrix is in microns.
- **output_resolution** (*np.ndarray*) – desired output resolution to visualize layer at.

Returns Neuroglancer JSON for single layer.

Return type dict

cloudreg.scripts.visualization.**get_neuroglancer_json**(*s3_layer_paths*, *affine_matrices*, *output_resolution*)

Generate Neuroglancer state json.

Parameters

- **s3_layer_paths** (*list of str*) – List of S3 paths to precomputed layers.
- **affine_matrices** (*list of np.ndarray*) – List of affine matrices for each layer.
- **output_resolution** (*np.ndarray*) – Resolution we want to visualize at for all layers.

Returns Neuroglancer state JSON

Return type dict

cloudreg.scripts.visualization.**get_output_dimensions_json**(*output_resolution*)

Convert output dimensions to Neuroglancer JSON

Parameters **output_resolution** (*np.ndarray*) – desired output resolution for precomputed data.

Returns Neuroglancer JSON for output dimensions

Return type dict

3.4 License

CloudReg is distributed with Apache 2.0 license.

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
```

(continues on next page)

(continued from previous page)

```
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
```

(continues on next page)

(continued from previous page)

```
      of the following places: within a NOTICE text file distributed
      as part of the Derivative Works; within the Source form or
      documentation, if provided along with the Derivative Works; or,
      within a display generated by the Derivative Works, if and
      wherever such third-party notices normally appear. The contents
      of the NOTICE file are for informational purposes only and
      do not modify the License. You may add Your own attribution
      notices within Derivative Works that You distribute, alongside
      or as an addendum to the NOTICE text from the Work, provided
      that such additional attribution notices cannot be construed
      as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
```

(continues on next page)

(continued from previous page)

```
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

E

F

G

I

M

O

P

R

S

T

U

W