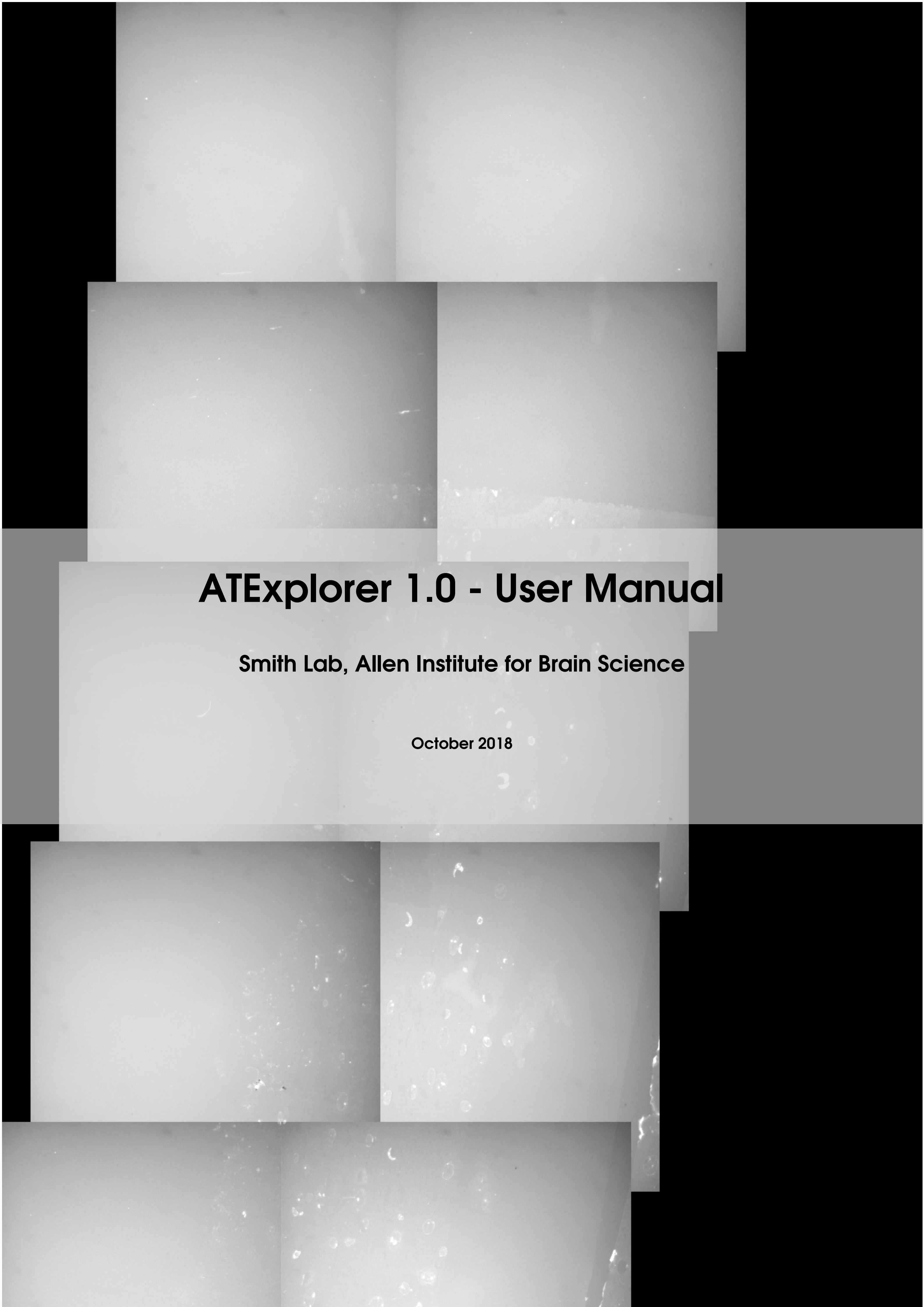


ATExplorer 1.0 - User Manual

Smith Lab, Allen Institute for Brain Science

October 2018

Preface

AT introduction here..

The ATExplorer application integrates a workflow, as well as a number of software components that are useful in the context of Array Tomography .

The following software components are the main building blocks that ATExplorer is built on top of:

- RENDER PYTHON by F Collman et al. RenderPython is a thin Python wrapper for *Render*.
- RENDER by ??? et.al
- FIJI by et. al....
- DOCKER et. al.

In addition to the above, semi specialized software packages, a number of open source, C++, libraries are employed by the ATExplorer application:

- VTK by
- POCO by ??? et.al
- LIBCURL by et. al....
- TINYXML2 et. al.
- DUNE SCIENTIFIC LIBRARY (DSL) et. al.

The ATExplorer application was designed and implemented in the lab. of Stephen J Smith and

Forrest Collman, at the Allen Institute of Brain Science by Totte Karlsson.

The following people has been contributing to the effort; ......

Part One

1. Overview of the ATExplorer UI and API

1.1 Introduction

This chapter gives an overview of the software application that is named *ATExplorer*.

The ATExplorer application was designed and implemented due to an emerging need to allow *non-programmers* to process, manage and explore Array Tomography data on a routine basis.

Depending on the actual protocols, an Array Tomography data set can range in size from a few hundred megabytes, to very large, like several Terra bytes. Depending on the number of stains and sessions, the complexity of the data-sets range from trivial to complex.

One of the main challenges in Array Tomography is the precise digital reconstruction of an original volume, i.e. from individually cut and imaged slices of tissue.

However, before volume reconstruction can begin, various pre data processing algorithms need to be applied, such as median filtering, flat-field correction and de-convolution. These processing algorithms are all, to some extent, complex. ATExplorer provides the non-programmer user with intuitive and easy to use UI components to guide through this process, in order to quickly get to data that is useful for scientific discovery and exploration.

1.2 A manual Array Tomography workflow

In order to get from raw acquired data to fine aligned 3D volumes, a number of processing steps, i.e. a *pipeline*, is required. (Create a pipeline figure)

- FLATFIELD CORRECTION
- DECONVOLUTION
- STITCHING
- REGISTRATION
- ROUGH ALIGNING
- FINE ALIGNING
- OTHER

The pipeline is integrated into the ATExplorer UI. However, in certain situations it may be necessary to push data through the pipeline manually. ATExplorer provides a set of Python scripts that can be used for this purpose. The scripts are to be found in the folder `/atPipeline/source` and `/atPipeline`. These pipeline scripts can be executed on their own, or from within the UI.

Each pipeline script is discussed in terms of their purpose, input and output data, in the sections below.

1.2.1 Creation of Renderstacks from raw image data

The first step in order to get easy and useful access to raw data is to create *stacks* in Render. A stack, in this context, can be thought of as an object in the Render backend that that keep positional and orientational information about a set of imaged section tiles.

However, before this data can be populated, a set of files, *statetables*, need to be created using the python script *createStateTables.py*. A statetable file contain meta data for individual section, by session.

The script: *create_state_tables.py*

```
def run(sessionFolder, firstsection, lastsection, dockerContainer):

    [projectroot, ribbon, session] = atutils.parse_session_folder(sessionFolder)
    print ("Processing session folder: " + sessionFolder)

    for sectnum in range(firstsection, lastsection+1):
        print("Processing section: " + str(sectnum))

        #create state table file name
        statetablefile = "statetable_ribbon_%d_session_%d_section_%d"%(ribbon, session,
                                                                       sectnum -1)
```

```python
        statetablefile = projectroot + os.path.join("scripts", statetablefile)

        if os.path.exists(statetablefile):
           print("The statetable: " + statetablefile + " already exists. Continuing..")
        else:
            #Example data
            #docker exec renderapps python /pipeline/make_state_table_ext_multi_pseudoz.
                                                                py
            #--projectDirectory /mnt/data/M33
            #--outputFile  /mnt/data/M33/statetables/test
            #--oneribbononly True
            #--ribbon 4
            #--session 1
            #--section 0

            #make state table
            #Need to pass posix paths to docker
            cmd = "docker exec " + dockerContainer
            cmd = cmd + " python /pipeline/make_state_table_ext_multi_pseudoz.py"
            cmd = cmd + " --projectDirectory %s"%(atutils.toPosixPath(projectroot,  "/
                                                                mnt"))
            cmd = cmd + " --outputFile %s"%(atutils.toPosixPath(statetablefile , "/mnt"))
            cmd = cmd + " --ribbon %d"%ribbon
            cmd = cmd + " --session %d"%session
            cmd = cmd + " --section %d"%(sectnum - 1) #Start at 0
            cmd = cmd + " --oneribbononly True"
            print ("Running: " + cmd)

            p = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr=
                                                                subprocess.STDOUT)
            for line in p.stdout.readlines():
                print (line)
```

Input: Dataroot folder and ribbons, sessions
Output: state table files in folder: *scripts*

The script: *create_rawdata_render_multi_stacks.py*

Creates down-sampled images and tile specs...

```python
def run(sessionFolder, firstsection, lastsection, dockerContainer, renderProject):

    channels = atutils.getChannelNamesInSessionFolder(sessionFolder)
    [projectroot, ribbon,session] = atutils.parse_session_folder(sessionFolder)
    print ("Project root folder: " + projectroot)

    for sectnum in range(firstsection,lastsection+1):
        print("Processing section: " + str(sectnum))

        #create state table
        statetablefile = projectroot + "scripts\\statetable_ribbon_%d_session_%
                                                    d_section_%d"%(ribbon, session,
                                                    sectnum -1)

        #Example
        #docker exec renderapps python -m renderapps.dataimport.create_fast_stacks_multi
        #--render.host localhost
        #--render.client_scripts /shared/render/render-ws-java-client/src/main/scripts
        #--render.port 8080
        #--render.memGB 5G
        #--log_level INFO
        #--statetableFile /data/test
        #--render.project test_project
        #--projectDirectory /data/M33
        #--outputStackPrefix ACQ_Session01
        # --render.owner test

        #upload acquisition stacks
```

```
        cmd = "docker exec " + dockerContainer + " python -m renderapps.dataimport.
                                                  create_fast_stacks_multi"
        cmd = cmd + " --render.host %s"        %renderProject.host
        cmd = cmd + " --render.owner %s "      %renderProject.owner
        cmd = cmd + " --render.project %s"     %renderProject.name
        cmd = cmd + " --render.client_scripts /shared/render/render-ws-java-client/src/
                                                  main/scripts"
        cmd = cmd + " --render.port 8080"
        cmd = cmd + " --render.memGB 5G"
        cmd = cmd + " --log_level INFO"
        cmd = cmd + " --statetableFile %s"    %(atutils.toPosixPath(statetablefile, "/
                                                  mnt"))
        cmd = cmd + " --projectDirectory %s"  %(atutils.toPosixPath(projectroot, "/mnt"
                                                  ))
        cmd = cmd + " --outputStackPrefix ACQ_"
        print ("Running: " + cmd)

        p = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.
                                                  STDOUT)
        for line in p.stdout.readlines():
            print (line)
```

Input: Dataroot folder and ribbons, sessions
Output: Renderstacks in render: ACQ_

1.2.2 Creation of median files

The script: ***create_median_files.py***

```
def run(firstsection, lastsection, sessionFolder, dockerContainer, renderProject):

    [dataRootFolder, ribbon, session] = atutils.parse_session_folder(sessionFolder)

    #Output directories
    median_dir        = os.path.join("%s"%dataRootFolder, "processed", "medians")
    median_json        = os.path.join(median_dir, "median_%s_%s_%d_%d.json"%(ribbon,
                                                  session, firstsection-1, lastsection-1))

    #stacks
    acq_stack         = "ACQ_Session%d"%(session)
    median_stack      = "MED_Session%d"%(session)

    #Make sure output folder exist
    if os.path.isdir(median_dir) == False:
       os.mkdir(median_dir)

    with open(atutils.mediantemplate) as json_data:
         med = json.load(json_data)

    atutils.savemedianjson(med, median_json, renderProject.host, renderProject.owner,
                                                  renderProject.name, acq_stack,
                                                  median_stack, atutils.toPosixPath(
                                                  median_dir, "/mnt"), ribbon*100 +
                                                  firstsection -1, ribbon*100 +
                                                  lastsection -1, True)

    #Run ============
    cmd = "docker exec " + dockerContainer + " python -m rendermodules.
                                                  intensity_correction.
                                                  calculate_multiplicative_correction"
    cmd = cmd + " --render.port 80"
    cmd = cmd + " --input_json %s"%(atutils.toPosixPath(median_json, "/mnt"))
    print ("Running: " + cmd)

    p = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.
                                                  STDOUT)
    for line in p.stdout.readlines():
        print (line)
```

Input:
Output:

The script: *create_flatfield_corrected_data.py*

```
def run(firstsection, lastsection, sessionFolder, dockerContainer, renderProject):

    [dataRootFolder, ribbon, session] = atutils.parse_session_folder(sessionFolder)
    #Output directories
    flatfield_dir     = os.path.join("%s"%dataRootFolder, "processed", "flatfield")

    #stacks
    acq_stack         = "ACQ_Session%d"%(session)
    median_stack      = "MED_Session%d"%(session)
    flatfield_stack   = "FF_Session%d"%(session)

    #Make sure output folder exists
    if os.path.isdir(flatfield_dir) == False:
       os.mkdir(flatfield_dir)

  #Create json files and apply median.
    for sectnum in range(firstsection -1, lastsection):
        z = ribbon*100 + sectnum
        with open(atutils.flatfieldtemplate) as json_data:
             ff = json.load(json_data)

        flatfield_json     = os.path.join(flatfield_dir, "flatfield""_%s_%s_%s_%d.json"%(
                                                       renderProject.name, ribbon, session,
                                                        sectnum))
        atutils.saveflatfieldjson(ff, flatfield_json, renderProject.host, renderProject.
                                                       owner, renderProject.name, acq_stack
                                                       , median_stack, flatfield_stack,
                                                       atutils.toPosixPath(flatfield_dir, "
                                                       /mnt"), z, True)
        cmd = "docker exec " + dockerContainer + " python -m rendermodules.
                                                       intensity_correction.
                                                       apply_multiplicative_correction"
        cmd = cmd + " --render.port 80"
        cmd = cmd + " --input_json %s"%(atutils.toPosixPath(flatfield_json, "/mnt"))

        #Run =============
        print ("Running: " + cmd)

        p = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.
                                                       STDOUT)
        for line in p.stdout.readlines():
            print (line)
```

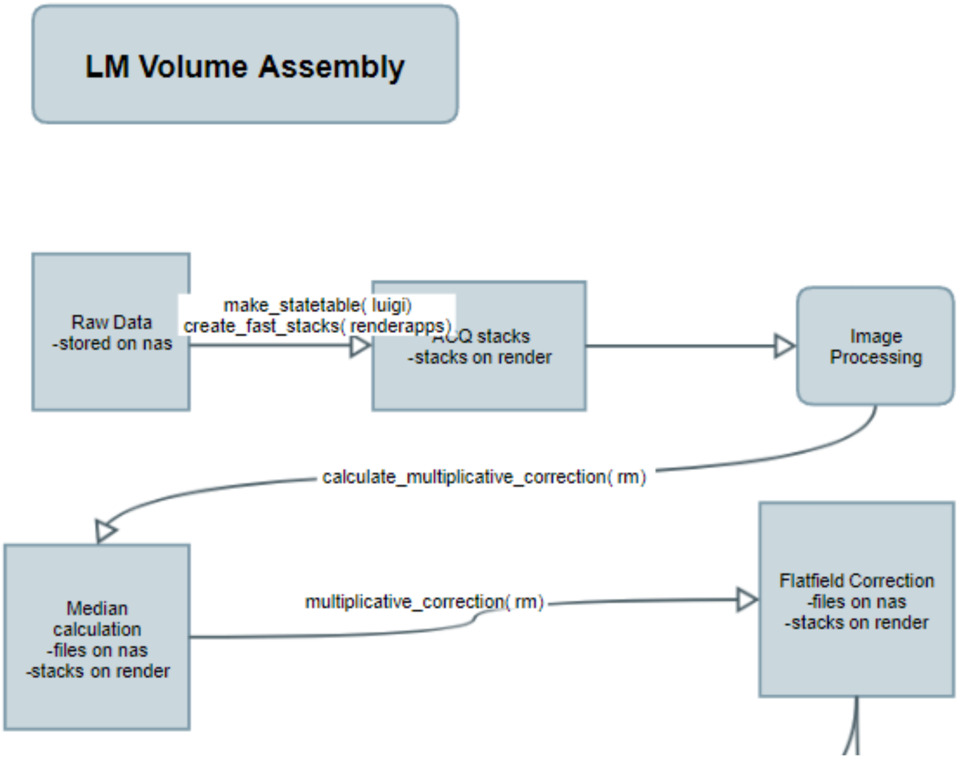


Figure 1.1: Processing ....

1.3 The Render Service

1.4 The ATExplorer UI

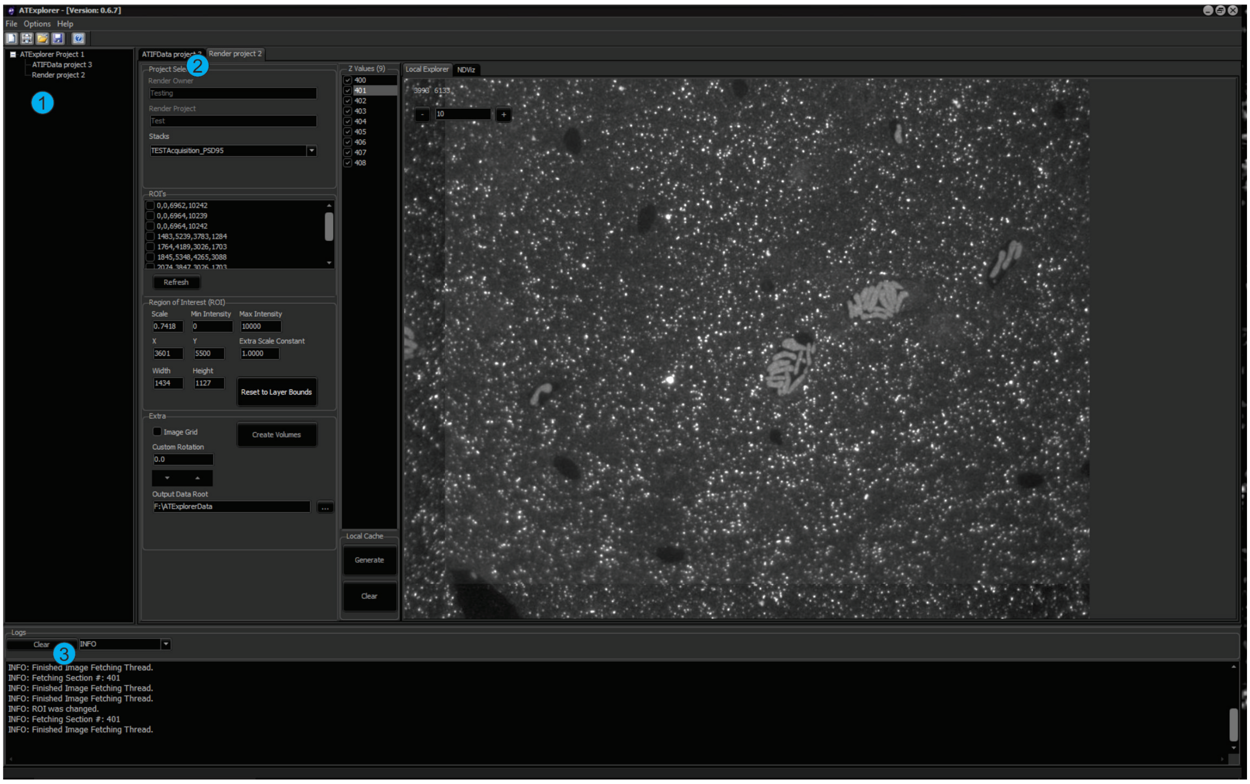


Figure 1.2: ATExplorer UI. The circled numbers in the figure indicate relevant elements of the UI; (1) Project(s) TreeView. (2) Tabbed Project Item View. (3) Information and Application Log Messages.

1.4.1 Importing Data

- **Importing process** Give an overview on what happens when data is being imported to ATExplorer .
- **Data Formats** Describe the Allen Institute format, and Kristinas format.

1.4.2 Connect to a a Remote (or local) RenderHost

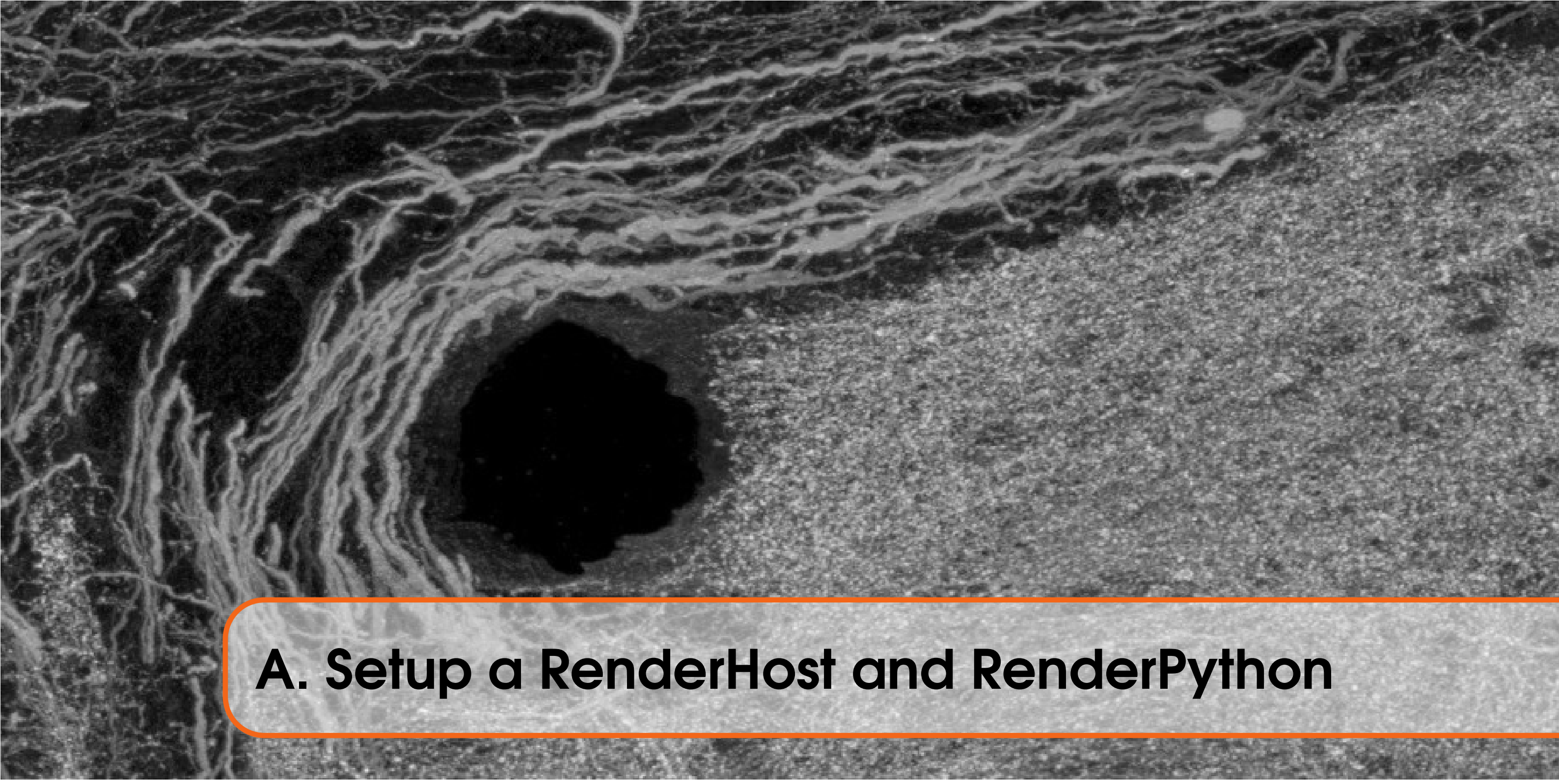
1.4.3 Create of RenderStacks

1.4.4 Manage Stacks in Render

1.4.5 Explore Data

1.5 Python Bindings

Appendices



A. Setup a RenderHost and RenderPython

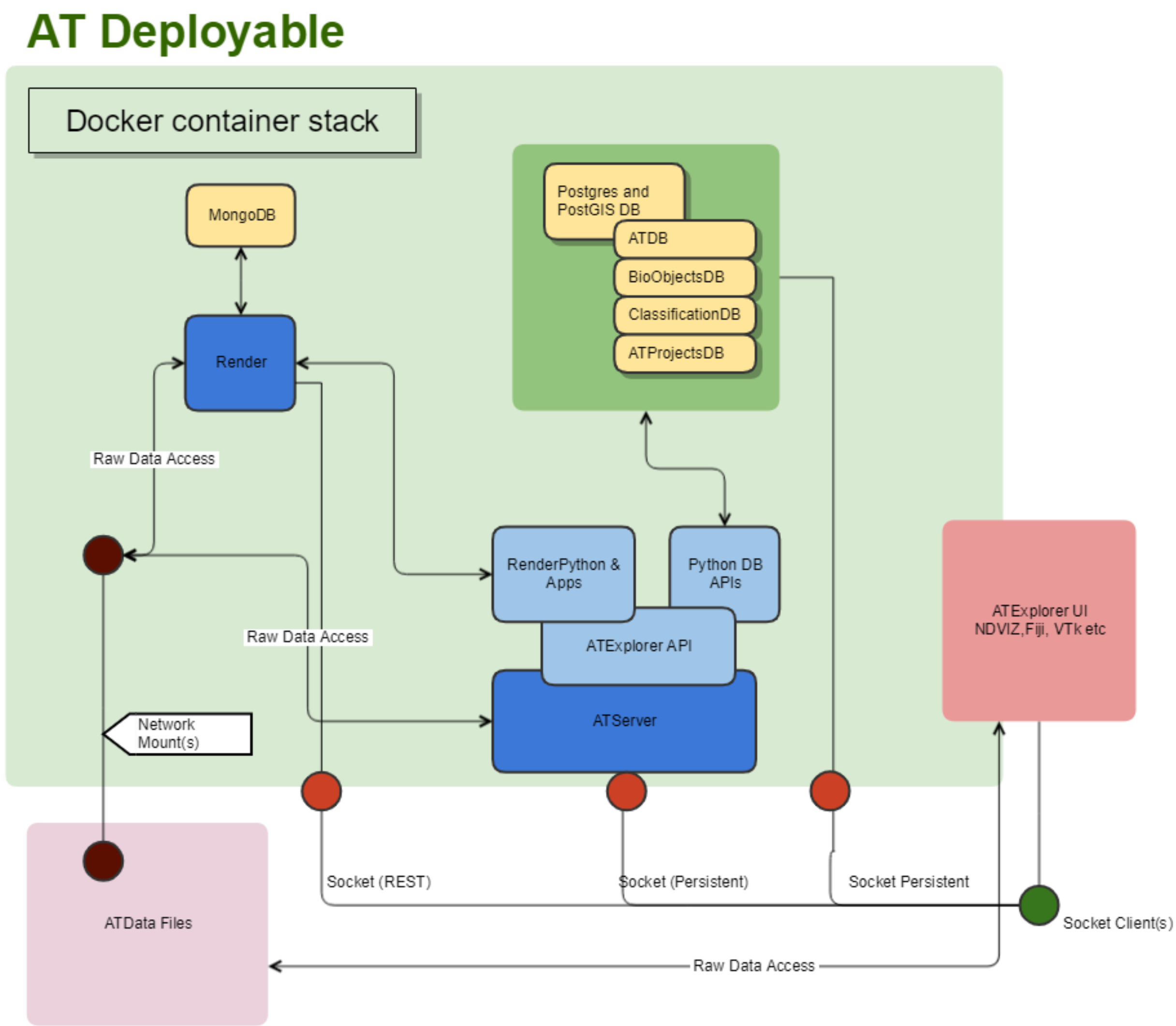
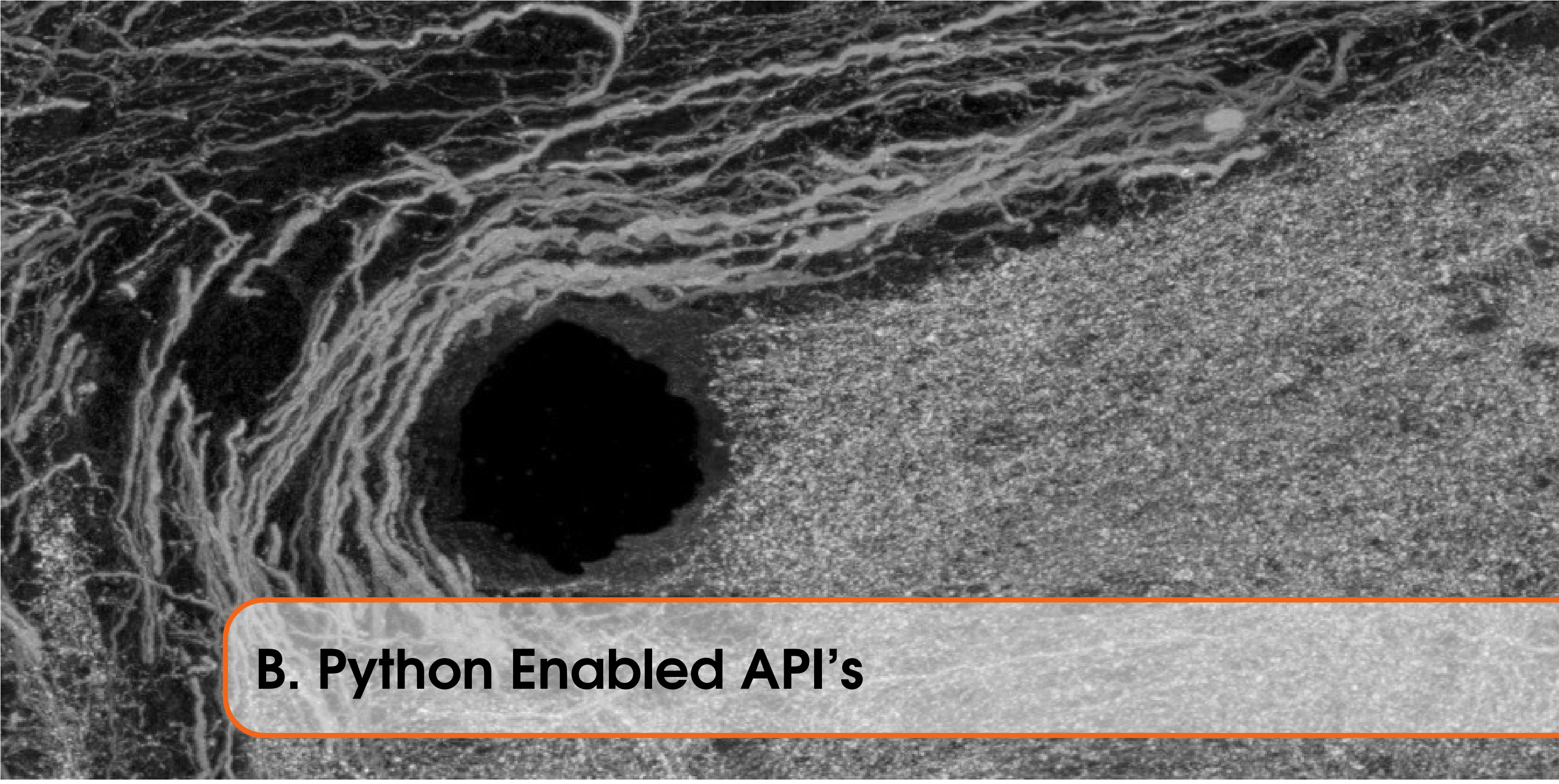
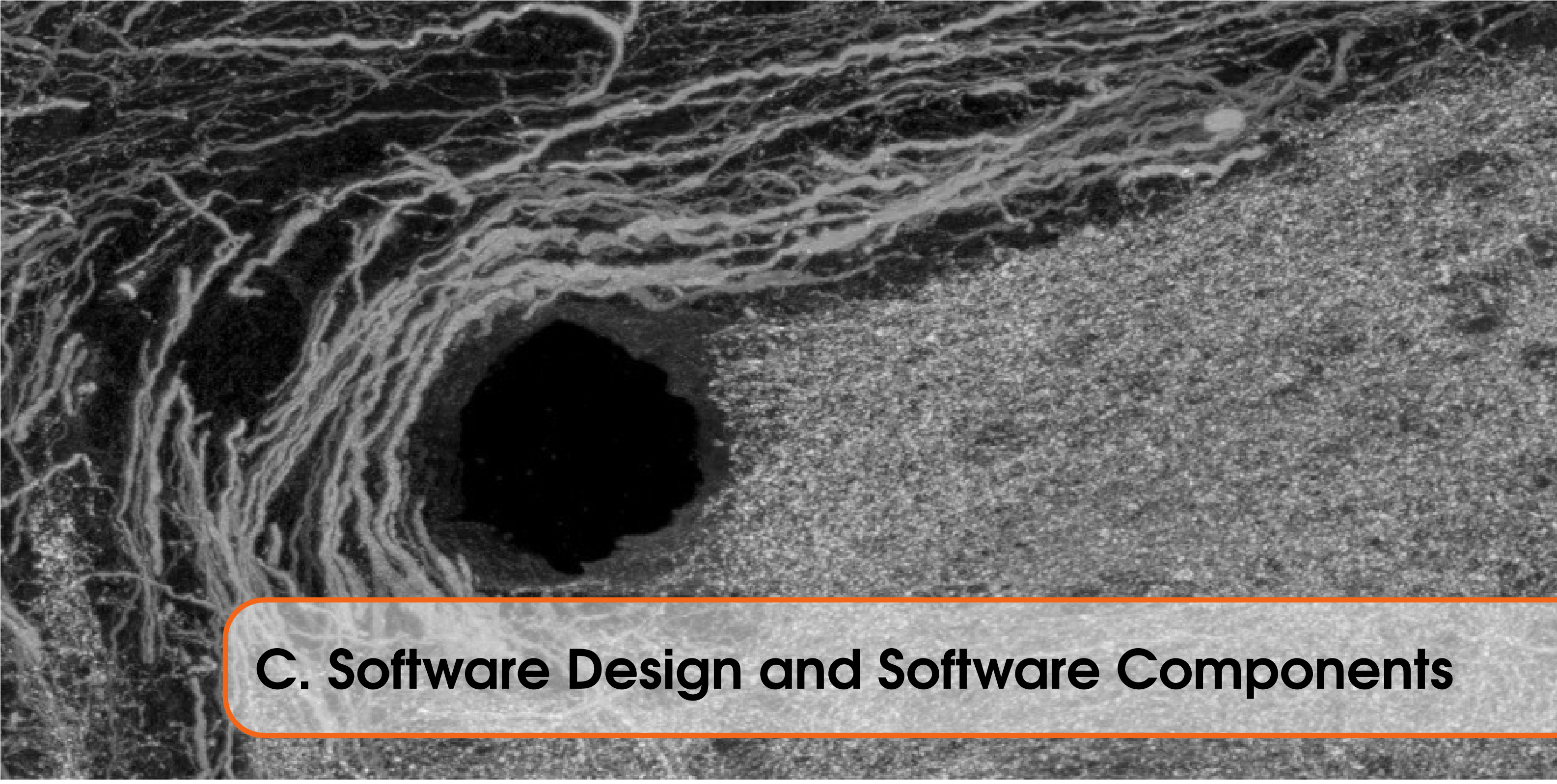


Figure A.1: A deployed system



B. Python Enabled API's



C. Software Design and Software Components

C.1 ATExplorer UI

The ATExplorer UI is a Microsoft Windows desktop application implemented using Embarcadero's C++ Builder tools. The C++ Builder environment provide a programmer with hundreds of components for efficient and rapid development of Windows applications. In addition, thousands of third party components are available as well.

This appendix discusses some of the software designs and software components employed when implementing the ATExplorer

C.1.1 Observers and subjects

The Tree view and PageControl .

C.1.2 The TreeView

The items in the Treeview stores data objects as (void*) pointers. Any object registered with the tree (as a node) need to be a descendant of the class ExplorerObject. Typical scenario:

```
ExplorerObject* eo = (ExplorerObject*) node->Data


if(dynamic_cast<...>(eo))
{
```

```
...
}
```

C.1.3 Populating an ATData object

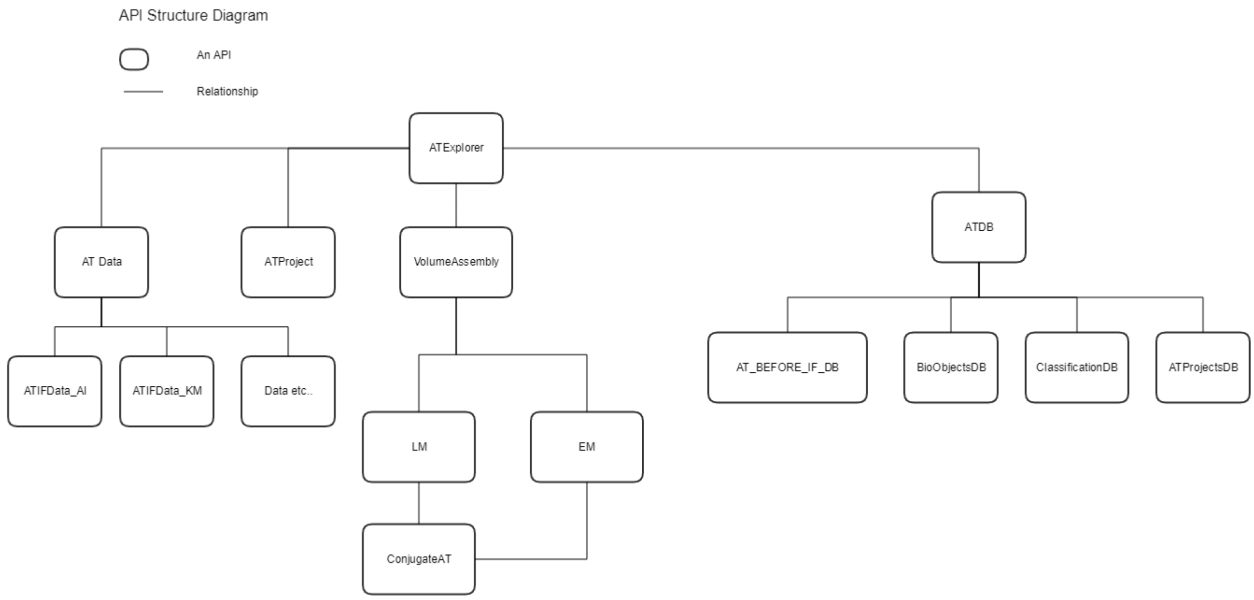


Figure C.1: An overview of some of ATExplorer API's

C.2 ATExplorer Software API's

C.2.1 atCore

C.2.2 atData

C.2.3 atVCLCore

C.3 ThirdParty libraries

C.3.1 Poco

C.3.2 libCurl

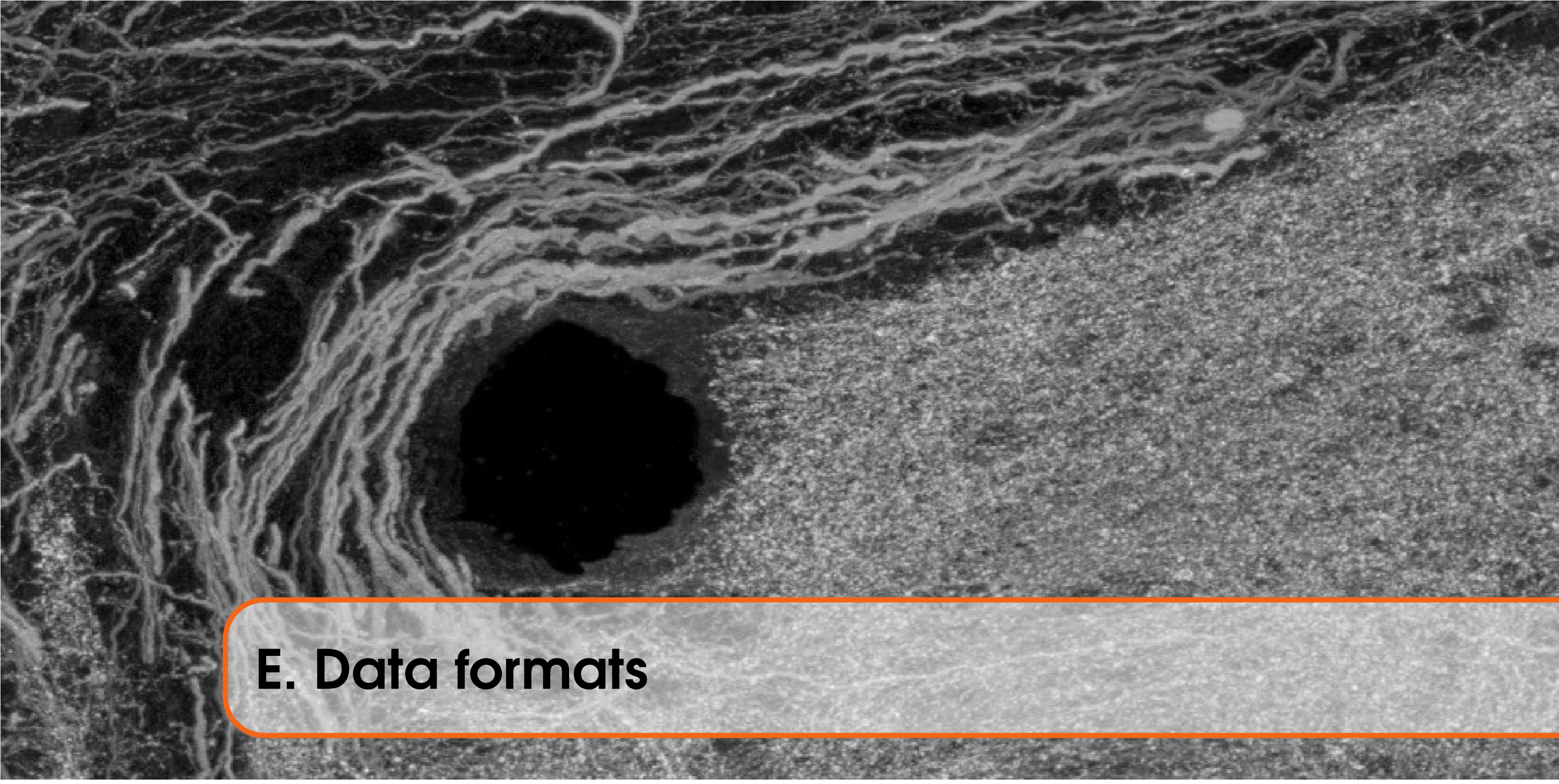
C.3.3 SQLite 3

C.3.4 TinyXML2

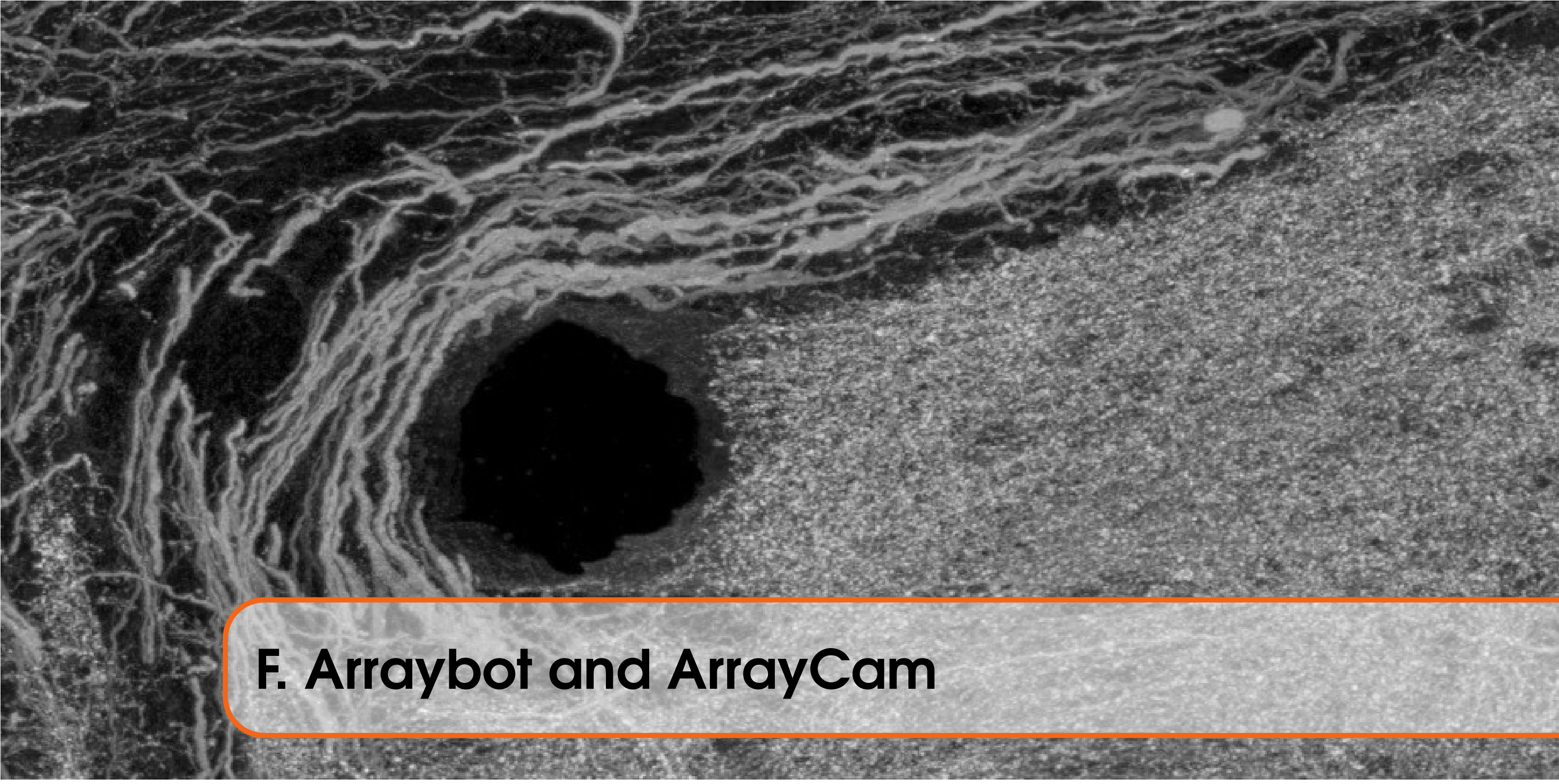
C.3.5 Dune Scientific libraries: dslFoundation

D. How to Build ATExplorer and its API's

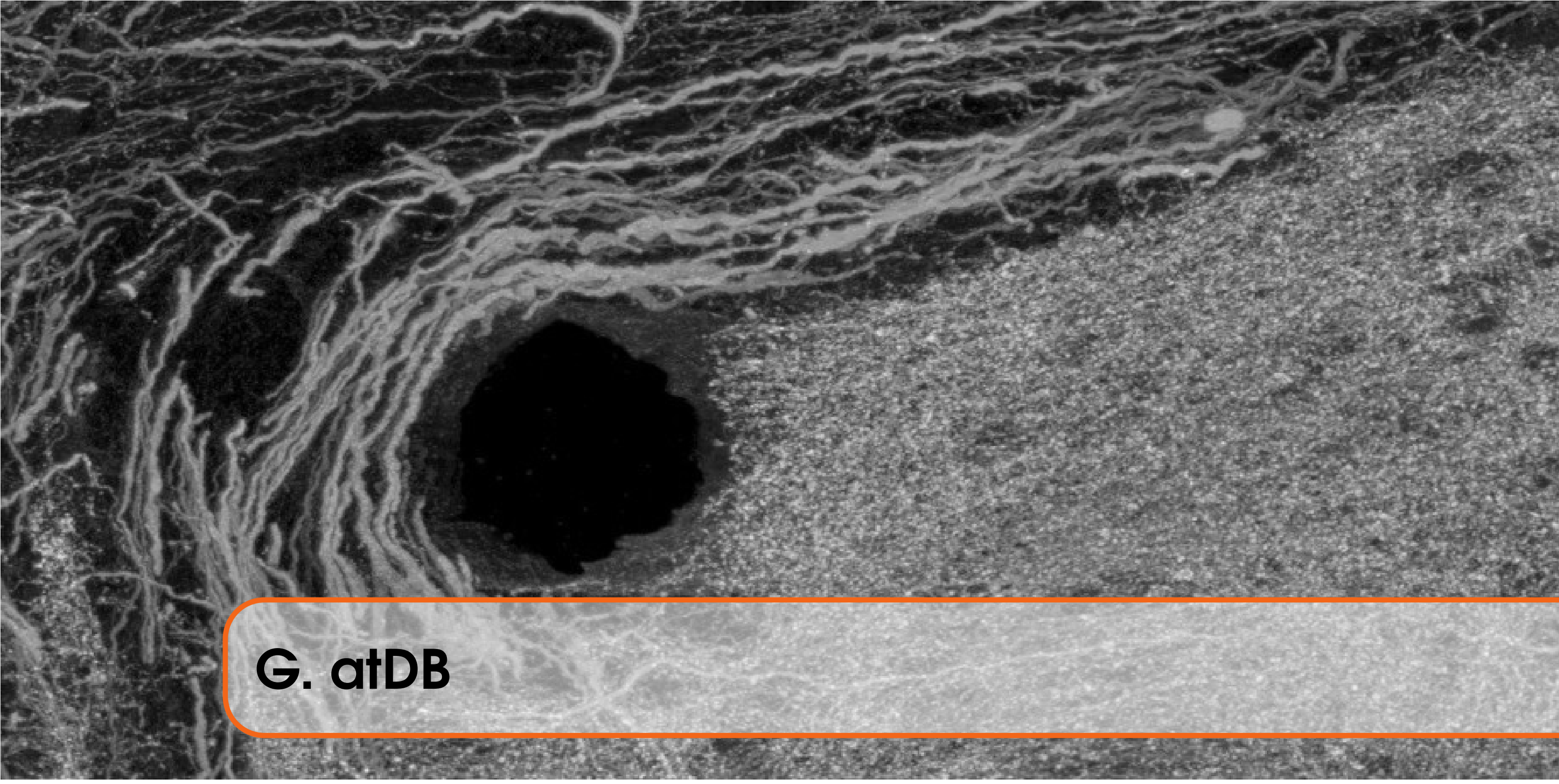
Public Software Repository: **git@github.com : TotteKarlsson/ATExplorer.git**



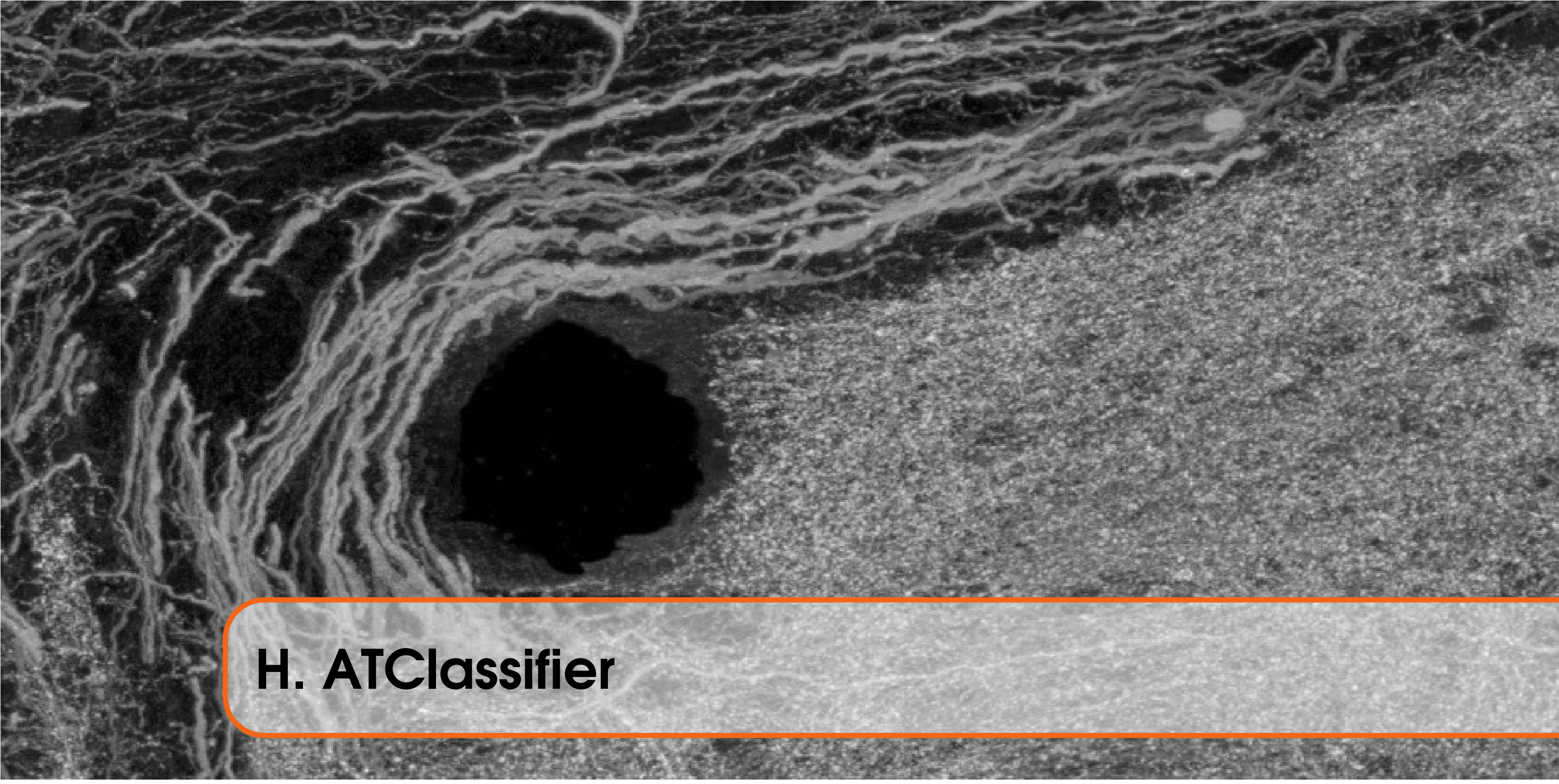
E. Data formats



F. Arraybot and ArrayCam



G. atDB



H. ATClassifier