A Practical Guide to Drone Mapping
Using Free and Open Source Software

OpenDroneMap
The Missing Guide

Piero Toffanin

PIERO TOFFANIN

OpenDroneMap: The Missing Guide

A Practical Guide To Drone Mapping Using Free and Open Source Software

First published by MasseranoLabs LLC 2019

First edition

Cover art by Piero Toffanin
Proofreading by Danielle Y. Toffanin

This book was professionally typeset on Reedsy.
Find out more at reedsy.com

"Empowerment of individuals is a key part of what makes open source work, since in the end, innovations tend to come from small groups, not from large, structured efforts."

- Tim O'Reilly

Contents

Preface

"If you're wondering who's in charge of writing documentation, you are." - Piero Toffanin

I never thought I'd eventually end up writing a book about OpenDroneMap. I made my first code contribution to the project in 2016, after buying a drone and discovering that software can automatically turn 2D images into 3D models and maps. I was intrigued by the process and OpenDroneMap was one of the few open source programs that I could manage to get up and running. At the time the program was difficult to use and worked only from the command line. So over a few days I contributed a rough user interface. That interface later evolved into the NodeODM project. People noticed, loved it and asked for more. So that was the start of the WebODM project. My involvement stepped up once I started diving into the processing engine's internals and making some major contributions there along the way. At the time the program was changing so rapidly that even writing some simple documentation seemed like an impossible task. It would be obsolete in a few months, so why bother?

Today the software is still rapidly changing, but the general structure of the program is much more defined, making an attempt at documenting it feasible. People in the meanwhile kept asking for a comprehensive guide. So, one day I decided

to take up the effort and write it. Thus, OpenDroneMap: The Missing Guide was born.

I decided to offer it as a book and not as an online resource for several reasons:

- A book has a more discursive format and allows the information to be presented in a more linear fashion
- The project already has an online reference documentation and I didn't want to rewrite the work others have already made. This book does not replace the online documentation, it expands it
- It gives people an opportunity to financially support the project

I'm aware that for some people buying a book might not be an option. Reasons can range from financial hardship to the inability of making purchases with a credit card. To mitigate this problem, I have setup a page on the book's website at https://odmbook.com where people can apply for a free or discounted copy. Furthermore, if you purchased this book and you know somebody who is unable to get it, please feel free to forward them your copy (just please don't share it on a public site).

As soon as a second edition of the book is written, I pledge to release this book for everyone to download freely.

I have tried my best to write in a style that a complete novice could understand, while keeping it technical enough for advanced users to gain valuable insights. I have favored simplicity over correctness when discussing concepts, knowing that scholarly readers will know how to recognize the shortcomings of my descriptions and where to lookup the more formal

definitions.

I'm not a professional writer and English is not my first language, so I hope the reader will forgive me for the occasional awkwardness in sentence structures or a misspelling that might have been missed during review.

I believe that constructive criticism is a key component to learning and improving. How was the book? What could be improved for the next edition? What was not clear?

If you have feedback or any other comment in general, please feel free to drop me a note: pt@masseranolabs.com.

Enjoy the book!

-Piero

Acknowledgement

This book would not have existed without the amazing support of early enthusiasts who offered to purchase the book even before it was finished! There are no words to express my gratitude for the motivation they have provided me during many long days and nights of writing. In no particular order, I'd like to thank them individually.

Gold Supporters

Mike Finlayson, Timothy Viola (Viola Engineering, PC), Edward Wyrwas (Prepared for Flight, LLC), Dr. Bertram Bilek (Instituto Bilek), Owen Torgerson, Matt Harding (Osprey Mapping Solutions, LLC), Issouf Ouattara (FasoDrone), Reality Scout, Nathan Ryan, Bill Fredricks (Penn's Wall, LLC), Christoph Trockel (SMQ), Justin Cole, Tryhard (Tryhard Prospecting), Pascal Vincent (AOMS), Luca Delucchi, Matthew Cua (OneSquirrelMade), Keith Conley (Beyond Aerial, LLC), Greg Rouse (Ross County Ohio SWCD), Silva Diego Hemkemeier, Robert Nall (Bulldog Imagery, LLC), Maciej Cybulski (MC2Systems), Adam Steer (Spatialised), Pratyush Kumar Das (Asian Institute of Technology), Will Welker (Pi Farm), Shunichiro Nishino, Mapping Services Australia, Marcos Gomez-Redondo (Facultad de Ingeniería de la Universidad Nacional de Asunción), AeroSurvey New Zealand,

Bryan Jackson (kaaspad), Robert Hall (Pro Aerial Digital), Jon Lee (Data Aero, LLC), Chris Mather (Bendigo Aerial Australia), Qopter360 Ltd, David Bradburn (DMB Consulting), Mauricio Gaviria (Tecnidrones S.A.S. - Colombia), Arne Wulff, Andy Lyons (University of California), Francisco Flores (FPFI Consulting Ltda., Chile), University of Copenhagen Forest and Landscape College, Masakazu Oshio (Land and House Investigator, Japan), NC State Center for Geospatial Analytics, Sasanai Chanate

Silver Supporters

David E. Gorla (CONICET, Argentina), Jarrett Totton (GoMapping), Japar Sidik Bujang (Universiti Putra Malaysia), Jeong Hyung-sik (Garam Forest Technology), Geco Enterprises Ltda Centro de I+D Chile, Neill Glover (Land IQ Insights), Luigi Pirelli (Freelance QGIS core developer), Jorge Lama, Michael Nielsen (Skyfair), Loren Abdulezer (Evolving Technologies Corporation), Jonathan Quiroz Valdivia, Mark W. Fink, Tomasz Nycz (GIS w Górach), Daniel Kendall (Daktech Pty Ltd), Joe Martin, Jerome Maruéjouls (Geoek), Álvaro Perdigão (TAW-S2i-Software), Kim Junseong (KMAP), Richard Marshall (Yacht Pogeyan), Randy Niedz, The New Zealand Institute for Plant & Food Research Limited, Gary Sieling (Element 84), Ryan Howell, Leon Schulpen (Kapla), Carlos Sousa, Marco Rizzetto, Sandy Thomson, Tero Keso (Häme University of Applied Sciences), Evan Watterson (Bluecoast Consulting Engineers), Alexander R. Groos (University of Bern)

Thank you!

I

Introduction

The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked. "Begin at the beginning", the King said gravely, "and go on till you come to the end: then stop."

- Lewis Carroll, Alice's Adventures in Wonderland

1

Why OpenDroneMap?

> "What makes OpenDroneMap different than software XYZ?" - *Most software XYZ users*

I would be tempted to list the many features of the software, how it's based on open standards, why thousands of organizations have chosen it as their preferred solution for processing aerial data, why it scales horizontally, and *blah blah blah*.

However, OpenDroneMap is not unique in functionality. At the time of writing (mid 2019) there are dozens of photogrammetry solutions that can deliver results comparable and sometimes better than OpenDroneMap.

What makes the software unique, is that it offers people a choice. Prior to its creation people were mostly confined to the lands of proprietary, black-box systems. If you wished to unlock the insights of aerial data, brought to us by the recent availability of inexpensive UAVs (Unmanned Aerial Vehicles), you had no choice. Pay up, get locked-in and trust that the results are correct.

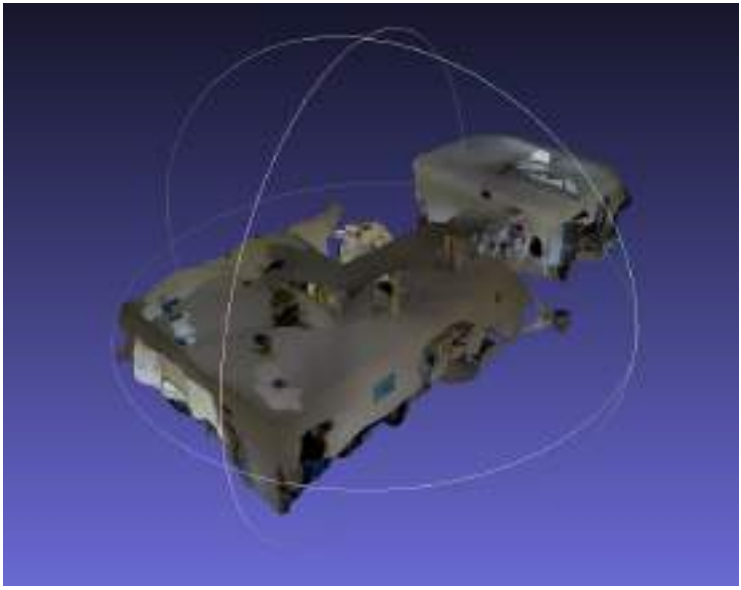
By distributing OpenDroneMap under a free and open source license, we have given people the ability to use, modify, examine, distribute and sell the software under very permissive terms. We have given people a choice.

Philosophical reasons aside, most users do not care about software freedom as much as they care about free pizza, so I'm excited to tell you that OpenDroneMap can be installed at no cost on all the computers you like and scale to thousands of clusters without licensing costs, with an active community of thousands of users and organizations, all while the project receives love from different organizations that fund and benefit from its development since 2013. We offer more features than any other open source software in the field, are growing at steady pace and have plans to take over the world when the autonomous robot apocalypse arrives!

2

What You Can Do with OpenDroneMap

The name OpenDroneMap immediately seems to narrow the scope of the program exclusively to the domain of drone image processing. While the history and focus of the program is centered around processing aerial imagery, it's not limited to that. The application can be used to perform photogrammetry tasks in many other fields such as archaeology and architecture. Indoor scanning and modeling with the use of a classic hand-held camera (or even a phone) is also supported.



Apartment building 3D model processed with OpenDroneMap

However, OpenDroneMap really shines when it comes to aerial imagery. By aerial imagery I mean pictures taken from UAVs, planes, kites or balloons. The software can generate georeferenced point clouds, 3D models, digital elevation models and maps, using Ground Control Points (GCPs) for better accuracy or without any GPS information at all. The program can be installed on a local machine or on cloud servers such as those provided by Amazon Web Services (AWS) or Google Compute Engine. It can be used from the command line or with a user-friendly interface. It can be used from the Python programming language via a dedicated Software Development Kit (SDK) or with other programming languages using an Application Programming Interface (API). The software can be integrated into new and existing platforms and allows organizations to

scale their processing capabilities as needed.

Today companies, government entities, professionals and hobbyists alike use some or all parts of OpenDroneMap to perform a varieties of tasks, including:

- Monitoring crops in agriculture.
- Mapping land areas.
- Reporting construction progress.
- Classifying and counting trees.
- Analyzing stockpile volumes.
- Documenting car crashes.
- Inspecting roofs and cell towers.
- Documenting proof of work completion.
- Improving OpenStreetMap.
- Stitching historical aerial images.

This list is by no means exhaustive. We often encounter new ingenuous ways in how people are using the software. OpenDroneMap is a collection of solutions for collecting, processing, analyzing and displaying aerial data, with a photogrammetry toolkit at its core. Aside from the classical uses, let creativity and imagination be the limit.

The project has a website hosted on https://www.opendronemap.org. If you haven't looked around the website yet, I encourage you to take a peek and read some of the (often humorous) blog posts. You might find inspiration for novel things to do with the software.

3

The Key To Becoming a Successful User

As an open source project, users can choose to download the software, read the documentation, start using it and be done. Maybe file a bug report when problems occur. We're happy when people just want to use the software and be done.

However, we recommend people join the vibrant OpenDroneMap community if they are using the software. This unlocks many benefits, such as:

- The ability to propose the addition of a missing feature.
- Becoming an expert in one or more areas of the program by helping others.
- Getting bugs that affect them personally resolved more quickly.
- Helping to steer the direction of the project.
- Participating in community events around the globe, meeting people and making new friends.

People are often hesitant to join and participate in a community, mostly because they are new and think they have nothing

worthy to say or contribute. But literally anyone can contribute.

Before going to the next chapter, I invite you to join the friendly OpenDroneMap community at https://community.opendronemap.org. Introduce yourself, tell people how you are hoping to use the software or anything else about you. Later on, after reading this book, try to answer a question from a fellow user. Even if you don't know for sure the answer, try to help anyway. Some help is better than no help. Propose a new feature or report bugs as you find them. The cycle of reciprocal help will come back to you.

Plus, you might even have some fun along the way and have a chance to meet other fellow drone mappers in the process!

II

Getting Started

"The way to get started is to quit talking and begin doing."

- Walt Disney

4

Installing The Software

Until recently OpenDroneMap was the term used to refer to a single command line application (what is now known as the ODM project). Not anymore. With the recent development of a web interface, an API and other tools, OpenDroneMap has become an ecosystem of various applications to process, analyze and display aerial data. This ecosystem is made of several components:

- **ODM** is the processing engine, which can be used from the command line. It takes images as input and produces a variety of outputs, including point clouds, 3D models and orthophotos.
- **NodeODM** is a light-weight API built on top of ODM. It allows users and applications to access the functions of ODM over a computer network.
- **WebODM** is a friendly user interface that includes a map viewer, a 3D viewer, user logins, a plugin system and many other features that are expected of modern drone mapping platforms.
- **CloudODM** is a small command line client to communi-

cate with ODM via the NodeODM API.

- **PyODM** is a Python SDK for creating tasks via the NodeODM API. We cover it in more detail in the *Automated Processing With Python* chapter.
- **ClusterODM** is a load balancer for connecting together multiple NodeODM instances and is covered in the *Processing Large Datasets* chapter.

I believe in a practical and incremental approach to learning. To keep things as simple as possible, we will begin with the installation and usage of WebODM, which under the hood also installs ODM and NodeODM. For the purpose of installing WebODM, we will briefly need to use the command line, but not much. We'll leave the command line aside until part III, where we will cover more advanced uses that require it. By then readers will be comfortable with the core concepts of the program and the learning curve will be more gradual. Advanced users that are familiar with the installation process may wish to skip this chapter.

ODM, NodeODM and WebODM are available on all major platforms (Windows, macOS and Linux) via a program called docker, which is required to run the software. Docker is a tool to run *containers*, which are packaged copies of an entire system, its software and its dependencies. These containers run within a virtual environment. On Linux this virtual environment is available from the operating system and is very efficient. On macOS and Windows the containers run within a Virtual Machine (VM), so there's a bit of overhead, but not a lot. Once installed users do not have to worry much about docker, as it can be used as a simple tool to launch WebODM and nothing else. We dedicate an entire chapter to more advanced uses of

docker in Part III of the book.

We often get asked why we use docker, since it tends to be a pain to install and configure on platforms other than Linux. ODM is a complex software. It relies on many dependencies, some of which simply do not run on Windows. Without docker it would not be possible to run ODM on Windows or macOS. On these platforms ODM cannot run natively. Future development efforts are being focused on leveraging the new Windows Subsystem for Linux (WSL) and the possibility to make a native port of all dependencies to macOS, which is going to make the installation much easier. But for the time being, we are stuck in a love/hate relationship with docker.

On Ubuntu Linux 16.04 it's feasible to run all OpenDroneMap software natively. However, because there's very little performance penalty for running docker on Linux[1] and docker is straightforward to setup on this platform, I don't recommend it. On Linux the advantages of containerization far outweigh a tiny performance penalty. With docker users also get easy one-step updates of the software, which is nice.

Hardware Requirements

The bare minimum requirements for running the software are:

- 64bit CPU manufactured on or after 2010
- 20 GB of disk space
- 4 GB RAM

[1] IBM Research Report: An Updated Performance Comparison of Virtual Machines and Linux Containers: https://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf

No more than 100-200 images can be processed with these specifications (the software will run out of memory). Recommended requirements are:

- Latest generation CPU
- 100 GB of disk space
- 16 GB RAM

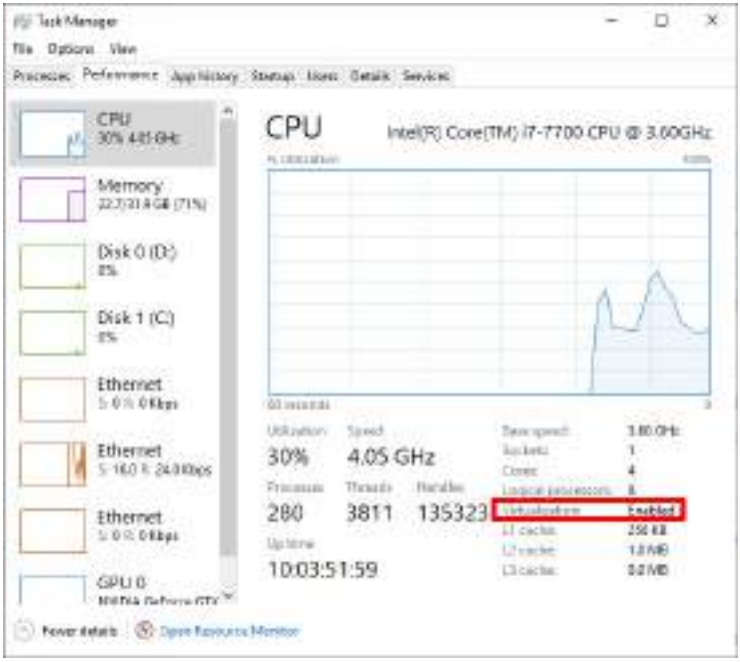
Which will allow for a few hundred images to be processed without too many issues. A CPU with more cores will allow for faster processing, while a graphics card (GPU) currently has no impact on performance. For processing more images, add more disk space and RAM linearly to the number of images you need to process.

Installing on Windows

To run OpenDroneMap you need at least Windows 7. Previous versions of Windows are not supported.

Step 1. Check Virtualization Support

Docker requires a feature from your CPU called virtualization, which allows your computer to run VMs. Make sure it's enabled! Sometimes this is disabled. To check, on Windows 8 or higher you can open the **Task Manager** (press CTRL+SHIFT+ESC) and switch to the **Performance** tab.



Virtualization should be enabled

On Windows 7 to see if you have virtualization enabled you can download the Microsoft® Hardware-Assisted Virtualization Detection Tool[2] instead.

If virtualization is disabled, you'll need to enable it. The procedure unfortunately is a bit different for each computer model, so the best way to do this is to look up on a search engine "how to enable vtx for <type your computer model here>". Often times it's a matter of restarting the computer, immediately pressing F2 or F12 during startup, navigating the boot menu and changing the settings to enable virtualization

[2] Microsoft® Hardware-Assisted Virtualization Detection Tool: http://www.microsoft.com/en-us/download/details.aspx?id=592

(often called *VT-X*).

Vendor	Key
Acer	Esc, F9, F12
ASUS	Esc, F8
Compaq	Esc, F9
Dell	F12
EMachines	F12
HP	F9
Intel	F10
Lenovo	F8, F10, F12
NEC	F5
Packard Bell	F8
Samsung	Esc, F12
Sony	F11, F12
Toshiba	F12

Common keys to press at computer startup to access the boot menu for various PC vendors

After you've enabled virtualization, proceed to step 2.

Step 2. Install Requirements

First, you'll need to install:

- Git: https://git-scm.com/downloads
- Python (the latest 3.x version): https://www.python.org/downloads/windows/

For Python 3, make sure you check **Add Python 3.x to PATH** during the installation.

Don't forget to add the Python executable to your PATH

Then, only if you are on Windows 10 Home, Windows 8 (any version) or Windows 7 (any version), install:

- Docker Toolbox: https://github.com/docker/toolbox/releases/download/v18.09.3/DockerToolbox-18.09.3.exe

If you are on Windows 10 Professional or a newer version, you should install instead:

- Docker Desktop: https://download.docker.com/win/stable/Docker%20for%20Windows%20Installer.exe

Please do **NOT** install both docker programs. They are different and will create a mess if they are both installed.

After installing docker, launch it from the Desktop icon that is created from the installation (**Docker Quickstart** in the

case of Docker Toolbox, **Docker Desktop** otherwise). This is important, do not skip this step. If there are errors, follow the prompts on screen to fix them.

Step 3. Check Memory and CPU Allocation

Docker on Windows works by running a VM in the background (think of a VM as a sort of computer emulator). This VM has a certain amount of memory allocated and WebODM can only use as much memory as it's allocated.

If you installed Docker Toolbox (see below if you installed Docker Desktop instead):

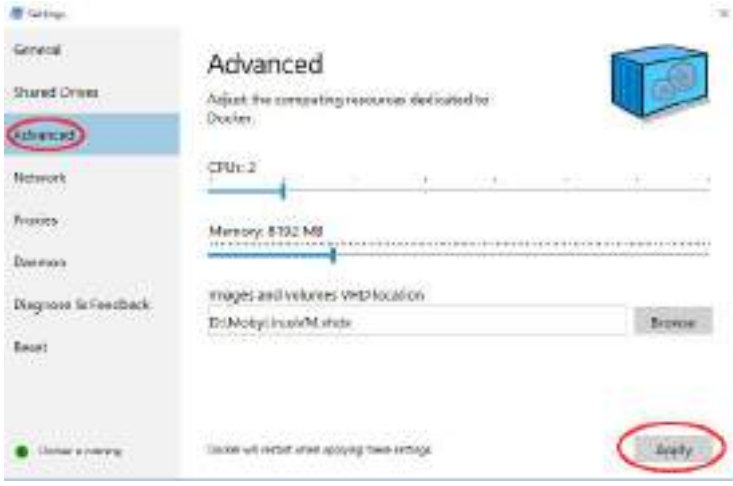
1. Open the **VirtualBox Manager** application.
2. Right click the **default** VM and press **Close (ACPI Shutdown)** to stop the machine.
3. Right click the **default** VM and press **Settings...**
4. Move the **Base Memory** slider from the **System** panel and allocate 60-70% of all available memory, optionally adding 50% of the available processors from the **Processor** tab also.

VirtualBox settings

Then press **OK**, right click the **default** VM and press **Start**.

If you installed Docker Desktop instead:

1. Look in the system tray and right click the *white whale* icon.
2. From the menu, press **Settings...**
3. From the panel, click **Advanced** and use the sliders to allocate 60-70% of available memory and use half of all available CPUs.
4. Press **Apply**.



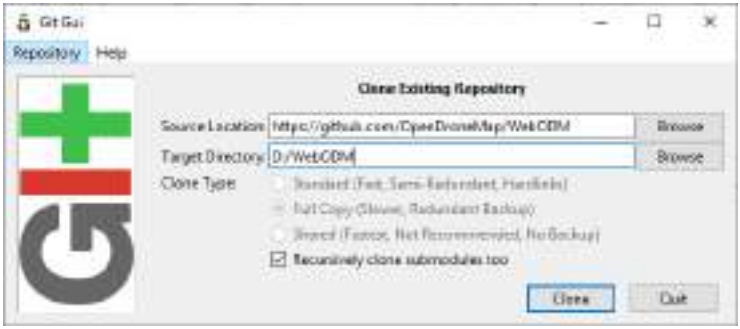
Step 4. Download WebODM

Open the **Git Gui** program that comes installed with Git. From there:

- For the **Source Location** field type: https://github.com/Open-

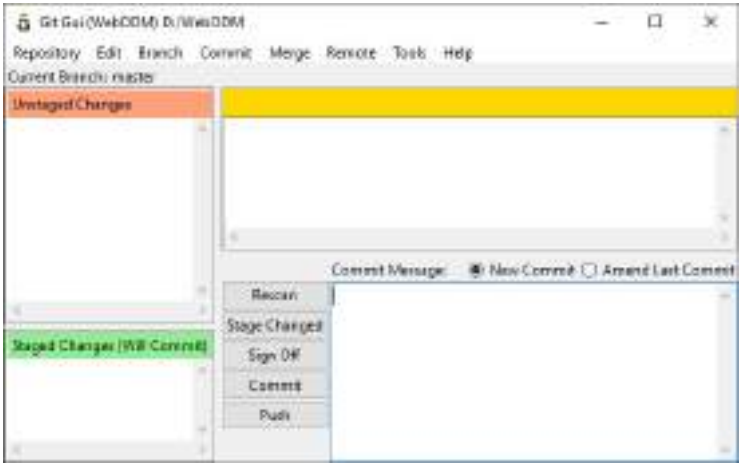
DroneMap/WebODM

- For the **Target Directory** field, click browse and navigate to a folder of your choosing (create one if necessary).
- Press **Clone**.



Git Gui

If the download succeeded, you should now see this window:



Git Gui after successful download (clone)

Go to the **Repository** menu, then click **Create Desktop Icon**. Clicking the newly created desktop icon will allow you to re-open this window in the future.

Step 4. Launch WebODM

From Git Gui, go to the **Repository** menu, then click **Git Bash**. From the command line terminal type:

```
$ ./webodm.sh start
```

Several components will download to your machine at this point, including WebODM, NodeODM and ODM. After the download you should be greeted by the following screen:

Console output after starting WebODM for the first time

- If you are using **Docker Desktop**, open a web browser to http://localhost:8000
- If you are using **Docker Toolbox**, find the IP address to connect to by typing:

```
$ docker-machine ip
192.168.1.100
```

Then connect to http://192.168.1.100:8000 (replacing the IP address with the proper one).

Installing on macOS

Most modern (post 2010) Mac computers running macOS Sierra 10.12 or higher can run OpenDroneMap using docker, as long as hardware virtualization is supported (see below).

Step 1. Check Virtualization Support

Open a **Terminal** app and type:

```
$ sysctl kern.hv_support
kern.hv_support: 1
```

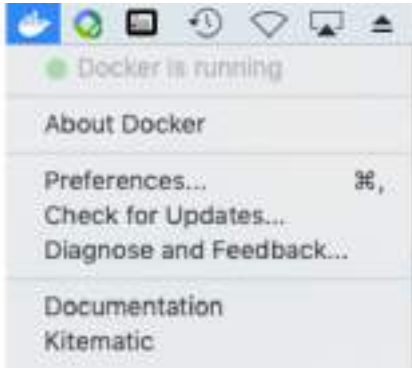
If the result is **1**, then your Mac is supported! Continue with Step 2.
If the result is **0**, unfortunately it means your Mac is too old to run OpenDroneMap. :(

Step 2. Install Requirements

There are only two programs to install:

1. Docker: https://download.docker.com/mac/stable/Docker.dmg
2. Git: https://sourceforge.net/projects/git-osx-installer/files/

After installing docker you should find an icon that looks like a whale in the task bar.



You can verify that docker is running properly by opening a **Terminal** app and typing:

```
$ docker run hello-world
[...]
Hello from Docker!
```

To verify that git is installed, simply type:

```
$ git --version
git version 2.20.1 (Apple Git-117)
```

If you get a *bash: git: command not found*, try to restart your **Terminal** app and double-check for any errors during the install process.

Step 3. Check Memory and CPU Allocation

Docker on macOS works by running a VM in the background (think of it as a sort of computer emulator). This VM has a certain amount of memory allocated and WebODM can only use as much memory as it's allocated.

1. Right click the whale icon from the task bar and click **Preferences...**
2. Select the **Advanced** tab.
3. Adjust the CPUs slider to use half of all available CPUs and the memory to use 60-70% of all available memory.
4. Press **Apply & Restart**.

Docker advanced settings

Step 4. Download and Launch WebODM

From a **Terminal** type:

```
$ git clone https://github.com/OpenDroneMap/WebODM
$ cd WebODM
$ ./webodm.sh start
```

Then open a web browser to http://localhost:8000.

Installing on Linux

OpenDroneMap can run on any Linux distribution that supports docker. According to docker's documentation website[3] the officially supported distributions are CentOS, Debian, Ubuntu and Fedora, with static binaries available for others (I use Arch Linux quite successfully). If you have to pick a distribution solely for running OpenDroneMap, Ubuntu is the recommended way to go.

Step 1. Install Requirements

There are four programs that need to be installed:

1. Docker
2. Git
3. Python (2 or 3)
4. Pip

[3] Docker Documentation: https://docs.docker.com/install/

We cannot possibly cover the installation process for every Linux distribution out there, so we'll limit the instructions to those that are distributions officially supported by docker. In all cases it's just a matter of opening a terminal prompt and typing a few commands.

Install on Ubuntu / Debian

Commands to type:

```
$ sudo apt update
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sh get-docker.sh
$ sudo apt install -y git python python-pip
```

Install on CentOS / RHEL

Commands to type:

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sh get-docker.sh
$ sudo yum -y install git python python-pip
```

Install on Fedora

Commands to type:

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sh get-docker.sh
$ sudo dnf install git python python-pip
```

Install on Arch

Commands to type:

```
$ sudo pacman -Sy docker git python python-pip
```

Step 2. Check Additional Requirements

In addition to the three programs above, the **docker-compose** program is also needed. Sometimes it's already installed with docker, but sometimes it isn't. To verify if it's installed type:

```
$ docker-compose --version
docker-compose version 1.22.0, build f46880f
```

If you get a:

```
docker-compose: command not found
```

you can install it by using **pip:**

```
$ sudo pip install docker-compose
```

Step 3. Download and Launch WebODM

From a terminal type:

```
$ git clone https://github.com/OpenDroneMap/WebODM
$ cd WebODM
$ ./webodm.sh start
```

Then open a web browser to http://localhost:8000.

Basic Commands and Troubleshooting

The cool thing about using docker is that 99% of the tasks you'll ever need to perform while using WebODM can be done via the ./webodm.sh script. You have already encountered one of them:

```
$ ./webodm.sh start
```

which takes care of starting WebODM and setting up a default processing node (node-odm-1). If you want to stop WebODM, you can press **CTRL+C** or use the following command:

```
$ ./webodm.sh stop
```

There are several other commands you can use, along with different parameters. Parameters passed to the ./webodm.sh command and are typically prefixed with two dashes. The **–port** parameter for example instructs WebODM to use a different network port:

```
Run WebODM on port 80 instead of 8000
$ ./webodm.sh restart --port 80
```

Other useful commands are listed below:

```
Restart WebODM (useful if things get stuck)
$ ./webodm.sh restart

Reset the admin user's password if you forget it
$ ./webodm.sh resetadminpassword newpass

Update everything to the latest version
$ ./webodm.sh update

Store processing results in the specified folder
    instead of the default location (inside docker)
$ ./webodm.sh restart --media-dir /path/to/
    webodm_results

See all options
$ ./webodm.sh --help
```

For general maintenance tasks, including backups and troubleshooting, the README page of WebODM has the most up-to-date instructions[4] and it's well worth a read. The community forum[5] is also a great place to ask for help if you get stuck during any of the installation steps and for general questions on using the ./webodm.sh script.

Hello, WebODM!

After running **./webodm.sh start** and opening WebODM in the browser, you will be greeted with a welcome message and will be asked to create the first user. Take some time to familiarize yourself with the web interface and explore its various menus.

WebODM dashboard

Notice that under the **Processing Nodes** menu there's a *node-odm-1* node already configured for you to use. This is

[4] WebODM README: https://github.com/OpenDroneMap/WebODM

[5] OpenDroneMap Community Forum: https://community.opendronemap.org

a NodeODM node and has been created automatically by WebODM. This node is running on the same machine as WebODM. In *The Command Line* chapter we will explore how to add new nodes, even from different machines.

Now it's time to process some data.

5

Processing Datasets

If you own a drone, I recommend trying to process images you've collected: it's more gratifying than using somebody else's. OpenDroneMap does not (yet) have a flight planner or a flight controller application, but there are many freely available such as DroneDeploy[6], DJI GS Pro[7] or QGroundControl[8] that can help you capture a dataset. For advice on data collection, also check the *Flying Tips* chapter. If you need some sample data, there are many freely available datasets on the community forum[9] and you are welcome to use them.

Dataset Size

Depending on the amount of RAM available on your computer, you might want to choose a small dataset (less than 100 images)

---

[6] DroneDeploy: https://dronedeploy.com

[7] DJI GS Pro: https://www.dji.com/ground-station-pro

[8] QGroundControl: http://qgroundcontrol.com

[9] OpenDroneMap Community Forum - Datasets: https://community.opendronemap.org/c/datasets

to start with. Memory requirements for processing are mostly proportional to the number of images and there are no reliable benchmarks at the time of writing that can tell you exactly how much memory you will need ahead of time. So the best way is to start small and gradually increase. Some task options can also affect memory requirements. Task options are covered in detail in the *Task Options in Depth* chapter.

File Requirements

Currently the software only supports JPEG files. Multiband files such as .TIFF are currently not supported at the time of writing, but efforts are underway[10].

The images can come from different cameras and can be taken at different angles. Most drone and phone cameras will also add geolocation information in the JPEG files in the form of Exchangeable Image File Format (EXIF) tags. EXIF tags are small pieces of information embedded within an image, which often times include the geographical location of where a picture was taken. Geolocation information is required to produce georeferenced orthophotos and elevation models, but is not required for creating point clouds and 3D models. You can use pictures that have no geolocation information, but you will receive a warning that an orthophoto cannot be created.

If you have pictures with no geolocation information, you can either use a Ground Control Point (GCP) file or a tool such as exiftool[11] to add geolocation information to the images individually. GCPs are covered in more detail in the *Ground*

[10] Add support for GeoTIFF images: https://github.com/OpenDroneMap/ODM/issues/865

[11] Exiftool: https://www.sno.phy.queensu.ca/~phil/exiftool/

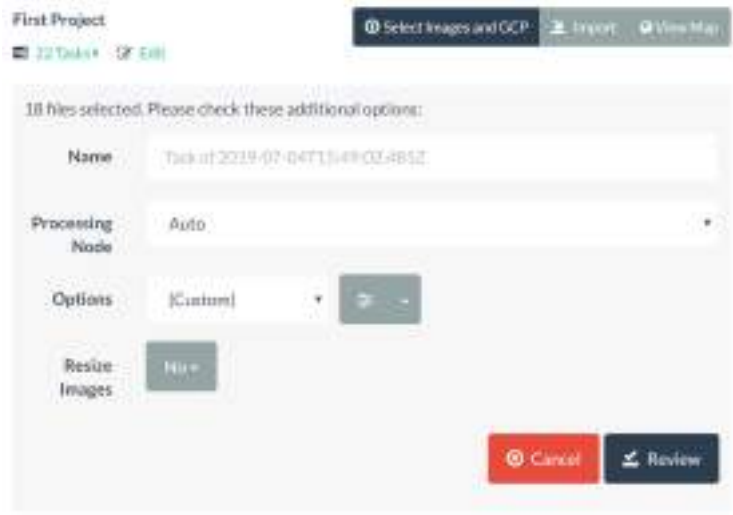
Control Points chapter.

You can use GIMP[12] to check whether your images have geolocation information. If you open an image with GIMP, press the **Image** menu, then go to **Metadata — View Metadata**. From the **EXIF** tab you should be able to find the GPSAltitude, GPSLatitude and GPSLongitude tags. If you don't see them, the image does not have geolocation information. You can also modify or add GPS information for a single image from GIMP by using the **Image — Metadata — Edit Metadata** panel. It can be lengthy to add geolocation for many images with GIMP, so if you have many images exiftool is a better tool. We cover batch geotagging of images with exiftool in *The Command Line* chapter.

Process Tasks

Simply press the **Select Images and GCP** button or drag and drop your images in a project. You can also press the button multiple times to add files from multiple folders.

[12] GIMP: https://www.gimp.org



WebODM's new task panel

There are a few settings you can choose:

- **Name:** a label for the task.
- **Processing Node: Auto** will simply pick a processing node for you (the node with the least number of running tasks will be picked first). Otherwise you can manually select a processing node.
- **Options:** you can choose one from a predefined list of presets. You can hover your mouse cursor over the currently selected preset to see what options are being affected. Several default presets are available, but you are encouraged to experiment and create your own presets. Pressing the button next to the preset list brings up the edit task options panel, while pressing the arrow button next to it allows you to save, edit or remove existing presets. Task options are covered in detail in the *Task Options in Depth*

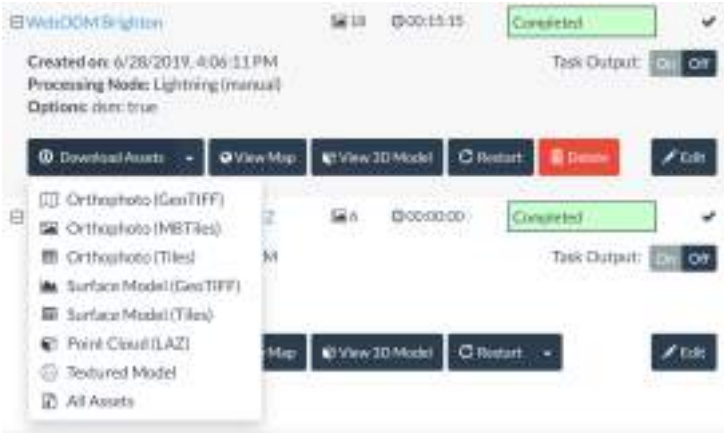
chapter. For now simply choose the **Default** preset.

- **Resize Images:** for reducing storage requirements, lowering memory usage and increasing processing speed, at the expense of potentially lower quality results, you can choose to resize your images prior to processing.

When you are ready, press **Review** and then **Start Processing**. After processing starts:

1. Images are uploaded to the app/media folder within WebODM. Note that this folder is not accessible from your computer unless you passed the **–media-dir** parameter when starting WebODM as explained at the end of the "Installing The Software" chapter.
2. A processing node is selected and the images are sent to the processing node. If this seems redundant (why not upload directly to the processing node?) remember that processing nodes can be located on remote computers.
3. A task is started and information such as time elapsed and console output are refreshed every few seconds.

Once the task is completed the results are transferred from the processing node to WebODM. Often a copy of the task results is kept on the processing node for a period of time (typically 2 days). This allows WebODM to restart tasks from the middle of processing.



Results can be downloaded or viewed directly from one of two interfaces:

- **View Map**: displays a 2D map where orthophotos and elevation models can be explored. Tools are available for generating contours, making volume measurements and more.
- **View 3D Model:** shows an interactive point cloud visualization. If a textured model is available, it can be toggled for inspection. Various tools can be used to make measurements, create elevation profiles, clip areas of the point cloud and more.

If a project contains multiple tasks, clicking the **View Map** button from the top right of each project displays orthophotos and elevation models of all tasks in the project simultaneously.

Map View

3D View

Output Results

If you download the results by pressing the **Download Assets — All Assets** button you will find an archive containing several directories:

- **dsm_tiles** contains the tiles for the color shaded digital surface model (if one was generated). Tiles can be used to display results on the web by using a viewer such as Cesium or Leaflet.
- **dtm_tiles** same as above, but for the digital terrain model.
- **orthophoto_tiles** same as above, but for the orthophoto.
- **odm_dem** stores the digital elevation model files.
- **odm_georeferencing** contains the dense point clouds.
- **odm_texturing** stores the textured 3D model (both a georeferenced version and a non-georeferenced version).
- **entwine_pointcloud** is a representation of the point cloud that can be streamed efficiently over the web using a viewer such as plas.io or potree[13].

Share With Others

Pressing the **Share** button from either **Map View** or **3D Model** will generate links that can be shared publicly with others. Note that the links will only work if WebODM has been installed on a server with a public IP address. If you are running WebODM on your local computer, you will not be able to share those links with people (unless you have configured the proper forwarding rules on your router/firewall). Typically if you want to share tasks with others you should install WebODM on a server.

[13] Viewing Entwine Data: https://entwine.io/quickstart.html#viewing-the-data

Export To Another WebODM

Tasks can also be downloaded and imported between WebODM installations. To do that, simply download a task assets by pressing **Download Assets — All Assets** and subsequently pressing the **Import** button from the **Dashboard**.

Manage Plugins

Some functionality in WebODM is implemented via plugins. By default many plugins are enabled, but they can be toggled off by visiting **Administration — Plugins**.

Change The Look & Feel

You can customize the colors, logos and names of WebODM by visiting the **Administration — Theme** and **Administration — Brand** panels.

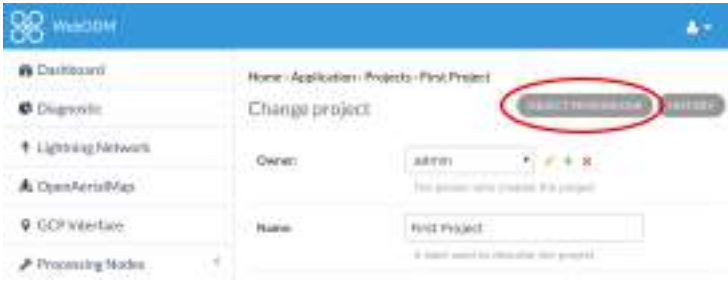
Create New Users

You can create user accounts (and groups) for people in your organization by visiting the **Administration — Accounts** panel.

Manage Permissions

You can manage who has access to which projects, who can create new projects and many other permissions. The permission settings can be set within the **Administration — Application** panel. Click **Projects**, then select the project you want to

modify. Then click the **Object Permissions** button from the top right section of the screen.



Object Permissions button

By default a project is owned by the user that created it. All tasks part of a project inherit the permissions from that project. Admin users (also called *superusers*) in the system have access to all resources, while normal users have only access to the tasks they have created. To share a project between users in the WebODM system you can type the name of a specific user or a group in the appropriate box and press the **Manage User** or **Manage Group** button respectively.

How Does WebODM Process Images?

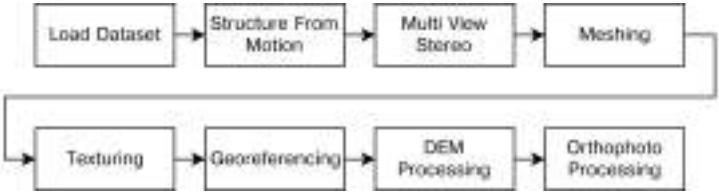
If this was the first task you've ever processed, or if you can remember the first time you processed one, it's likely that at some point you had a ***woah!*** moment: how could a computer take simple 2D images and turn them into georeferenced mosaics, 3D models and point clouds?

That's what we'll find out in the next chapter.

6

The Processing Pipeline

Going from images to 3D models and orthophotos is a process best visualized as a series of incremental steps. Each step relies on the work of previous steps.



ODM's processing pipeline

In this chapter we will explore an overview of the pipeline. We will not cover too many details, as each step's behavior can be tweaked by changing the task options. We will discuss in detail of how task options affect the inner workings of each step in the next chapter.

Load Dataset

Images can be corrupted. They may contain full or partial GPS coordinates (embedded within the EXIF tags). This step counts the images, extracts dimensions and parses GPS information from all available images.

Input: images + GCP (optional)
Output: image database

Structure From Motion

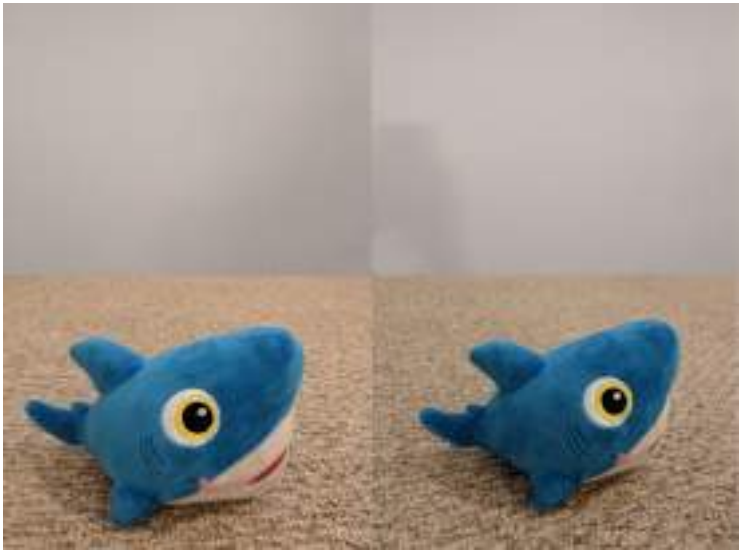
Structure From Motion (SFM) is a photogrammetry technique for estimating 3D objects (structures) from overlapping image sequences (from the motion of a camera taking pictures). At a very high level, the idea of SFM begins from the intuition that we can perceive a whole lot of information about a scene by just observing it from multiple view points. Using perspective geometry and optics (fascinating fields which would each require a book to cover in depth), the position and angle of the camera can be recovered for every picture. Astute readers might wonder why this is needed. After all, doesn't every picture already embed GPS and gimbal information? One of the problems is that GPS information (without Real Time Kinematics, a technique used to increase GPS accuracy) is not all that precise and gimbal information is not always present. SFM is much more accurate in calculating the position of cameras. As a bonus, it doesn't require any GPS information at all.

SFM performs several sequential steps:

- Extracts the camera information from the images' EXIF tags (if they are available). The optics equations require

a good estimate of the camera parameters (focal length, sensor size and others) for the process to work. The information from EXIF tags is used as a best guess initial estimate which is refined in later steps.

- Each image is scanned for easily identifiable features such as edges, points of interest and other unique objects. This is a crucial step and it's critically important to understand why. Take a look at the image below:

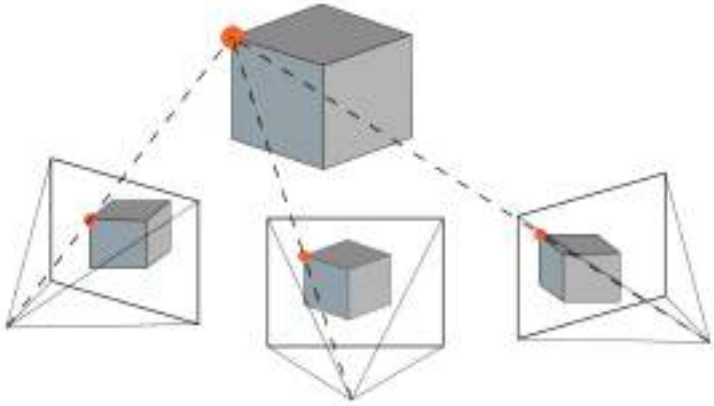


Two pictures of a white wall (top) vs. two pictures of a shark (bottom). Can you tell the camera moved left?

If we take two pictures of a white wall, we cannot tell how they relate to each other (did we move the camera left or right?). Compare them with two pictures that share an identifiable object.

In the second set of pictures we can tell that our camera has moved left (and tilted a bit). A computer cannot recognize any movement in the white wall pictures, which makes it impossible to solve the SFM problem. Next time you wonder why that grass field shot at low altitude does not want to be reconstructed, you'll know why (not enough identifiable features)!

- Image features from the previous step are now compared with each other. When many features are shared between two images (the same objects appear in two pictures) the images are matched. Some optimizations are employed to remove likely impossible candidates, such as images that are far away from each other using GPS information.
- Starting from a single pair of images and progressively adding more images, the program begins to recover the positions and angles of the cameras as well as recording a *sparse* collection of triangulated points (a sparse point cloud). This is accomplished using knowledge of optics (the physics laws that govern light and in this case its interaction with camera lenses), perspective geometry (the studies about representing 3D objects on 2D surfaces) and approximation methods for generating a consistent estimate of the camera information, orientation and position of the resulting triangulated points.



The SFM problem. What kind of camera took these pictures and where was the camera when the pictures were taken?

Photogrammetry is not a new field and its history dates back hundreds of years[14]. It's just that we've recently discovered that computers can be really good (and fast) at it. For those interested in learning more about SFM, coursera.org has some really good lectures[15]. ODM uses a software package called OpenSfM[16] (Open Structure From Motion) for efficiently solving the SFM problem.

Input: images + GCP (optional)
Output: camera poses + sparse point cloud + transform

---

[14] History of Photogrammetry: http://wayback.archive-it.org/all/20090227061949/http://www.ferris.edu/faculty/burtchr/sure340/notes/History.pdf

[15] Robotics: Perception: https://www.coursera.org/learn/robotics-perception

[16] OpenSfM: https://github.com/mapillary/OpenSfM/

Multi View Stereo

While SFM focuses mostly on the estimation of camera poses, Multi-View Stereo (MVS) focuses on the reconstruction of 3D models from multiple overlapping image pairs. MVS programs expect that information about cameras has already been computed and this allows them to focus on one thing: create a highly detailed set of 3D points (a *dense point cloud*). ODM currently offers two options for MVS:

- Multi-View Environment (MVE), a software suite developed at TU Darmstadt[17].
- OpenSfM, which we already discussed, has a dense reconstruction feature also.

Lots of 3D points make a dense point cloud

Input: images + camera poses + (sometimes) sparse point cloud
Output: dense point cloud

[17] Multi-View Environment: https://github.com/simonfuhrmann/mve

Meshing

When you think of a *3D model* you most likely imagine the type of models you see in videogames or movies. These models are more precisely called *polygonal meshes* or *meshes* for short.

3D mesh

Whenever a 3D model is scanned or derived from a photogrammetry process, the result is typically represented with 3D points. To go from 3D points to polygonal meshes we have to perform two steps:

1. "Connect the dots" using many triangles to obtain a mesh. Points may be moved or eliminated to create a better looking mesh.
2. Add color to the mesh (a process referred to as *texturing*).

Meshing is the process of "connecting the dots". ODM supports two different algorithms for meshing and uses one or the other depending on the situation and the user settings:

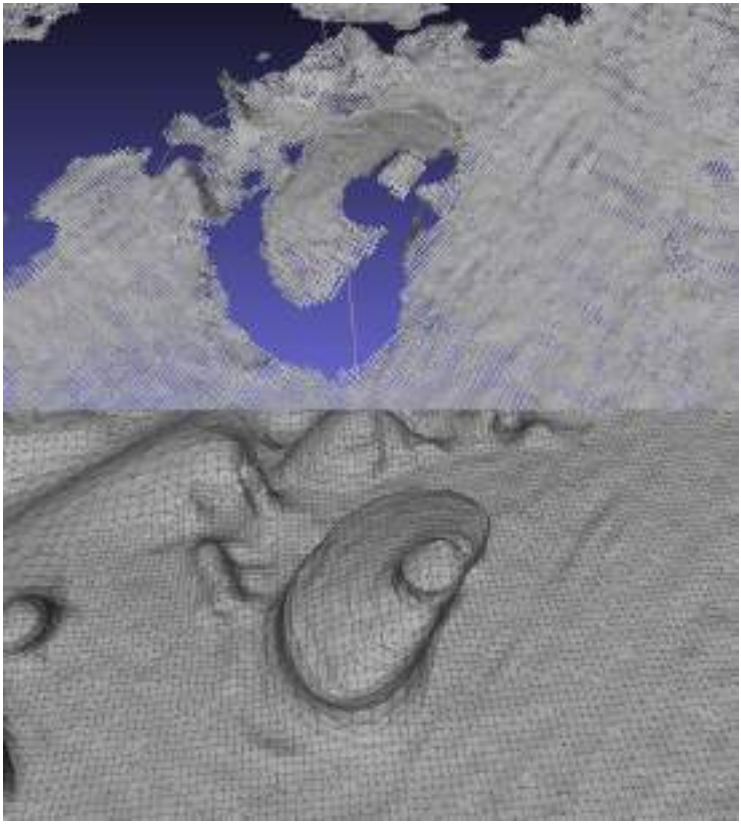
1. Screened Poisson Surface Reconstruction is a robust, memory efficient and battle tested algorithm for creating

3D surfaces by Michael Kazhdan[18]. It's used for generating full 3D models with a high degree of accuracy.

2. dem2mesh[19] is a program I developed for generating 2.5D meshes. 2.5D meshes are meshes that look 3D, but are simple *extrusions* of a 2D surface. These are used for generating orthophotos, as orthophotos do not require full 3D models for rendering and results often tend to look better.

[18] Screened Poisson Surface Reconstruction: http://www.cs.jhu.edu/~misha/-Code/PoissonRecon/Version8.0/

[19] dem2mesh: https://github.com/OpenDroneMap/dem2mesh



From 3D points to mesh

Input: dense or sparse point cloud
Output: 3D and 2.5D meshes

Texturing

At this point the mesh does not have any colors associated with it. It's just a polygon soup. Texturing is the process of adding colors to meshes. It does so by using specially computed images

(texture images) and by assigning each polygon to a section of the texture images. The process of creating the texture images and creating the associated mappings is performed by MvsTexturing[20], a software also developed at TU Darmstadt. At a very high level the program works as follows:

- Loads camera poses and images from the SFM process.
- Loads the mesh.
- For each polygon in the mesh, it finds the best image to fill it.
- It creates one or more texture images based on the information from the step above, also checking and attempting to remove moving objects (cats, cars, etc.).
- Sections of the texture images are color adjusted to compensate for differences in illumination.
- The borders (*seams*) between neighboring sections are blended to reduce color differences.

Textured mesh

[20] MvsTexturing: https://github.com/nmoehrle/mvs-texturing

Due to some randomness in the texturing algorithm, you are not guaranteed to get the same results if you run the process twice on the same mesh. That's why you might notice that the same dataset processed twice yields slightly different looking models (and orthophotos).

Input: images + camera poses + meshes
Output: textured meshes

Georeferencing

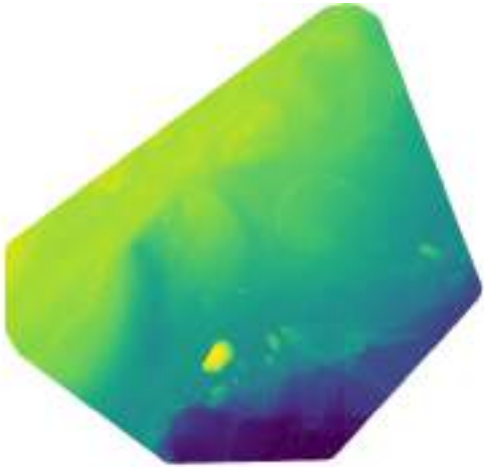
Up to this point all outputs have been represented using a *local* coordinate system (a made up coordinate system). A local coordinate system has no correlation to real world positions. Georeferencing is the process of converting (*transforming*) a local coordinate system into a world coordinate system. ODM can do this only if location information about the world is available, either via GPS coordinates embedded in the input images or a GCP file. When GPS coordinates are available, they are incorporated during the SFM step to align the reconstruction as to minimize the error between all the GPS locations and the computed camera positions. When GCPs are available, the GPS information is ignored and GCPs are used for the alignment instead. One of the outputs of the SFM step is a *transform* file, which allows the georeferencing step to convert point clouds and 3D models from local to world coordinates.

Once georeferenced, the point cloud is used to generate an estimate of the geographical boundaries of the dataset. These boundaries are used in subsequent steps for cropping orthophotos and digital elevation models (DEMs).

Input: transform + point cloud + textured meshes
Output: georeferenced point cloud + textured meshes + crop boundaries

Digital Elevation Model Processing

Point clouds are cool to look at, but much analysis is usually done using simpler 2D DEMs, which represent XY coordinates as pixel locations on the screen and pixel intensities (or colors) as elevation values. During this step ODM takes the georeferenced point cloud and extracts a surface model by using an inverse distance weighting interpolation method. If there are any holes in the model (perhaps an area is missing), they are filled using interpolation. Finally, the model is smoothed using a median filter to remove noise (bad values). With certain settings it can also attempts to classify the point cloud into ground vs. non-ground points and generate a terrain model by first removing all non-ground points. Finally, the results are cropped.

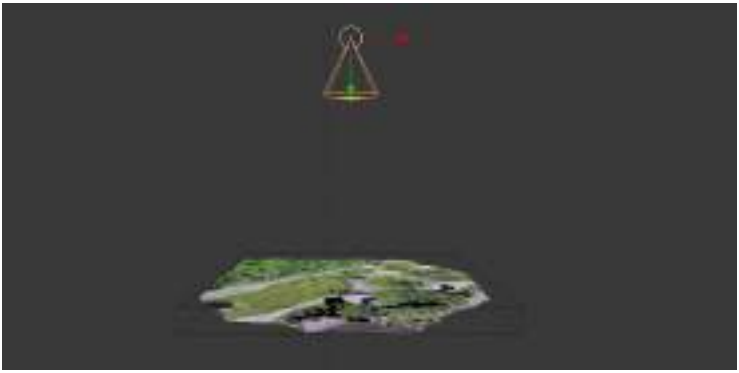


Digital surface model

Input: georeferenced point cloud + crop boundaries
Output: digital surface models + digital terrain models + classified georeferenced point cloud

Orthophoto Processing

The orthophoto is generated by taking a picture of the textured 3D mesh from the top. A dedicated program loads the textured mesh into an orthographic scene and saves the result to an image using the appropriate resolution. The image is then georeferenced and converted to a GeoTIFF using the information computed in the georeferencing step. Finally, the result is cropped.



Orthographic camera taking a picture of the 3D model from the top

Orthophoto

Input: textured mesh + crop boundaries
Output: orthophoto

We covered a general overview of the processing pipeline to de-

scribe how ODM goes from images to end results. We avoided going into too much detail about the specific implementation of each step, since any effort to discuss implementation details would inevitably be inaccurate or even obsolete by the time this book is completed. The beauty of open source is that you don't need a manual to tell you the details of the implementation. Those interested can and are encouraged to go look for details directly from the source code[21].

Now we turn our eyes to one of the most practical and important topics of this book: mastering the long list of task options and understanding what in the world each one does.

[21] ODM Stages Source Code: https://github.com/OpenDroneMap/ODM/tree/master/stages

7

Task Options in Depth

There are several components involved in the data processing pipeline. Each component has several adjustable settings that influence the output. The software exposes a subset of these available knobs through various options. When creating a task, a user can choose to tweak one or more options to change the behavior of the pipeline.

Options as shown in WebODM when creating a task

If the list seems overwhelming, just remember that this is a subset of all possible options that could be available from the various components of the data pipeline! This should raise some curiosity. Hidden features and processing capabilities could be hiding in the source code of OpenDroneMap, in the form of an option not yet exposed! The software exposes only those options that seem to have the biggest impact on results, or

those necessary to handle different workflows. But many, many more options, under the hood, remain unexposed in order to keep their number somewhat manageable.

Tuning options is more art than science. There are no clear guidelines on how to tune options to achieve optimal results. That's mostly because the best options for a certain dataset do not automatically transfer over to another. Given the big variety of possible scenes, cameras and mission planning strategies, it would be immensely time consuming to write an exhaustive guide. Plus, the tools are evolving quickly, so by the time such guide would be written, it would already be obsolete. But fear not!

This chapter is about understanding in detail what each option does. By the end of the chapter you'll be able to quickly improve your results, explain why certain models turn out the way they do and know what to tweak if the results don't turn out the way you want.

A few of these options might be missing from the graphical interface and might be available only from the command line. This is because sometimes the option does not make sense in the context of the interface workflow, or it's simply not supported.

Feel free to jump around and use this chapter as a reference. As the software gets better, some of these options might disappear from future versions and new ones will be introduced. The list below is taken from the software as of June 26th 2019. I will try my best to keep up with updates to the software and I plan to start a second edition of the book after it's published. In alphabetical order:

build-overviews

When set, overviews are added to the orthophoto. This does not affect other 2D outputs such as DEMs. Overviews are an optimization available for GeoTIFFs that reduces the time it takes to open them, for the tradeoff of a larger filesize and some computational time. Think of overviews as downsized copies of the orthophoto, stored inside the orthophoto file itself. Overviews are useful when opening the orthophoto in GIS programs that support them, such as QGIS[22] (a popular free and open source GIS software). When overviews are available, the viewer program can load the overview most appropriate for the current zoom level instead of loading the entire file. If it takes forever to display a 400MB orthophoto in a GIS program, it's probably because overviews weren't built! ODM creates overviews at 1/2, 1/4, 1/8 and 1/16 of the original resolution using an average interpolation. If an orthophoto is 1000x1000 pixels, **build-overviews** will store copies of the same orthophoto at 500, 250, 125 and 62 pixels resolution for faster visualization.

cameras

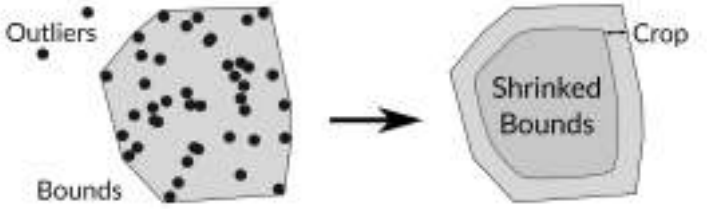
By default, during the SFM process, camera parameters are estimated from the input images. By using this option it's possible to choose a precomputed set of camera parameters instead, either as a path to a **cameras.json** file or by providing the contents of a **cameras.json** file. A **cameras.json** file is always computed after processing a dataset, so you can use

[22] QGIS: https://qgis.org

the camera parameters computed from one dataset to process another. The **cameras.json** file can also be generated using a special calibration target and a calibration software. Specifying a precomputed set of camera parameters can be useful to increase the accuracy of certain reconstructions, especially those that exhibit *doming* effects (point clouds that look concave or convex when they should be straight). We cover usage of this option in much more detail in the *Camera Calibration* chapter.

crop

In its raw form, the orthophoto contains irregular, jagged edges. These are the result of the texturing program attempting to fill areas that have little or no information. Cropping attempts to remove those edges to give us a nice, smooth looking orthophoto. First, the boundaries of the orthophoto are estimated by looking at the point cloud. The point cloud is first thinned and filtered to remove outliers. Then a polygon encompassing the result is saved in *odm_georeferencing/odm_georeferenced_model.bounds.gpkg*. This polygon is further smoothed and shrunk by the value specified in the **crop** option (as a value in meters). The final polygon is then used to crop the orthophoto.



Point cloud (left) and shrinked bounds that define the crop area (right)

Estimating where the bounds are is not a perfect science. Sometimes the area of an orthophoto that looks perfectly good will be removed in the cropping process. This option can be set to zero to skip cropping. For very large datasets, skipping crop can lower the run-time, since no computations have to be performed to estimate boundaries.

debug

Enable additional debug messages in the console output. Mostly useful for development purposes.

dem-decimation

All DEMs are computed from the point cloud output. The larger the point cloud, the longer it takes to compute a DEM. To speed things up, at the trade-off of possible accuracy loss, this option reduces (decimates) the number of points used to compute DEMs. The value specifies how many points to "skip" during the decimation. Let's look at an example by setting this option to 3. By taking all the points in the point cloud and placing them in a straight line, the program will reduce the

number of points by keeping one every three.

Only black points are included. Gray points are skipped

This way ~33% of all points are included, and ~66% are removed. If you use a value of 1 (the default), then all points are included. If you use a value of 50, then ~2% of the original points are kept and ~98% are removed. You can compute this percentage by doing:

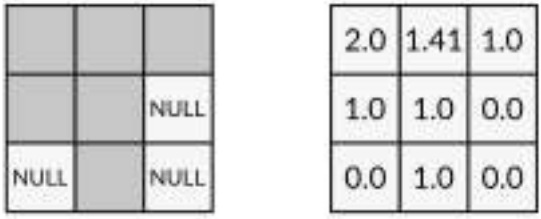
```
(1 / decimation) * 100
```

It should be noted that points are skipped sequentially and do not take into account spatial information, so this option should be used with care. Decimation also does not affect the original point cloud. The decimation is only used for the purpose of computing DEMs.

dem-euclidean-map

An euclidean map is a georeferenced image derived from DEMs (before any holes are filled) where each pixel represents the geometric distance of each pixel to the nearest *void*, *null* or *NODATA* pixel. It's an indicator (map) of how far a value in

the DEM is to an area where there are no values. This can be useful in cases when a person wishes to know which areas of a DEM were derived from actual point cloud values and which ones were filled with interpolation. Looking at the euclidean map, every pixel that has a value of zero indicates that the corresponding location on the DEM was filled with interpolation (because the distance of a *NODATA* pixel to itself is zero).



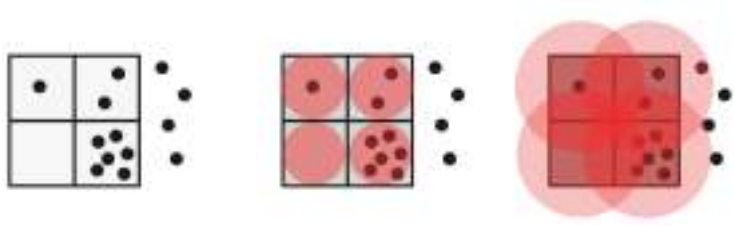
DEM before hole filling (left) and corresponding euclidean map (right)

Euclidean map results are stored in the *odm_dem* directory.

dem-gapfill-steps

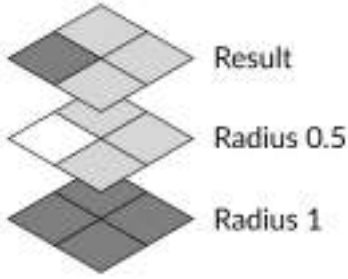
The process of going from point cloud to DEMs is not as straightforward as it may seem. Since DEMs are *rasters* (images), they have *cells* (pixels). Each cell, should have a value. Depending on the resolution of the raster, certain cells may have zero, one or more points that fall within it. All cells need a value, even if no points fall directly into it, otherwise there will be empty areas (gaps) in the DEM! One way to overcome this is to use a radius around each cell. Every point that falls

within the radius is considered part of the cell.



Pixels and points (left), radius of 0.5 (middle) and radius of 1 (right)

But how big should the radius be? If too small, as in the 0.5 radius example above, some cells might remain empty. If too big, there will be too much smoothing and accuracy will suffer. Since different point clouds have varying degrees of density, one solution is to compute multiple DEMs with different radiuses and stack them.



Gap fill interpolation with 2 DEM layers

Results with smaller radiuses (more accuracy, more gaps) are placed at the top, while results with bigger radiuses (less accuracy, less gaps) are placed at the bottom. If there are still

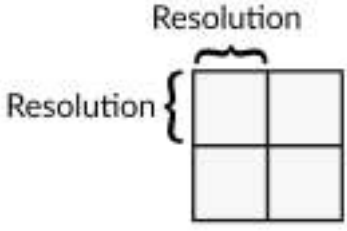
gaps at the end of this process, any remaining gaps are filled using a less accurate smoothed nearest neighbor interpolation.

How many layers should there be? However many this option says there should be. The initial value for the radius for the first layer is set to half of the raster resolution. Subsequent layers have twice the radius of their predecessors.

The local gridding method used to compute individual grid cell values is the same as the one used in the open source Points2Grid[23] project.

dem-resolution

This option specifies the output resolution of DEMs in cm / pixel.



Each square represents a pixel in a raster DEM

As an example, if the area covered by the point cloud is 100x50 meters and dem-resolution is set to 10 cm / pixel, the final image size of the DEM in pixels can be calculated by:

[23] Points2Grid: A Local Gridding Method for DEM Generation from Lidar Point Cloud Data https://opentopography.org/otsoftware/points2grid

```
(100 meters * 100 cm/meter ) / 10 cm/pixel
= 1000 pixels
(50 meters * 100 cm/meter ) / 10 cm/pixel
= 500 pixels
```

So the output DEM will be an image of 1000x500 pixels. The multiplication by 100 in the parenthesis is necessary because there are 100 centimeters in a meter.

depthmap-resolution

A depthmap is an image containing information relative to the distance of objects in a scene.

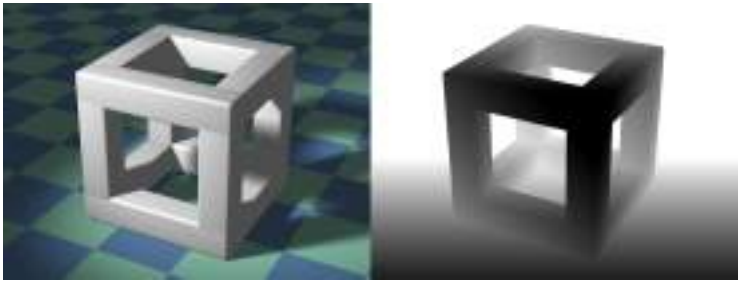


Image and corresponding depthmap. Darker areas are closer to the camera.
Image courtesy of Dominicus, Cubic Structure, CC BY-SA 3.0

Depthmap images are computed during the photogrammetry process to compute the dense point cloud. The higher the resolution of depthmaps, the longer the run-time to triangulate points. Higher resolution depthmaps increase the number of points, but also increase the amount of noise. Increasing

the depthmap resolution will increase run-time quadratically (twice the resolution will take 4x the time to compute). If you need to change the density of your point clouds, this is the option to tweak. Point clouds are the basis for 3D models, which in turn are used to generate orthophotos. When mapping urban areas, if the buildings come out with holes, look malformed or "wavy", a higher density point cloud could help obtain better looking buildings. Increasing this value too much can increase noise, so there's a subtle balance between point density and quality of results.

dsm

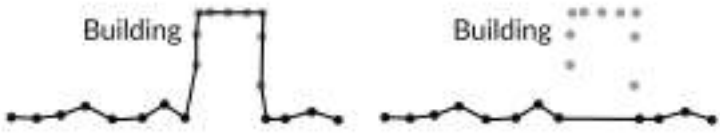
This option generates a digital surface model (DSM). DSMs are generated by taking the maximum elevation values in a point cloud, including both terrain and other structures such as buildings or trees. If two points fall on top of each other, only the tallest point is used. Gaps in the point cloud are filled using the process described in **dem-gapfill-steps**. The result is stored in *odm_dem/dsm.tif*.

dtm

This option generates a digital terrain model (DTM). DTMs are generated by classifying the point cloud using a simple morphological filter[24] (SMRF). Setting this option implicitly turns on the **pc-classify** option, which classifies point either as ground or non-ground. Non-ground points are discarded before computing the DTM. Gaps in the point cloud are filled

[24] smrf: A Simple Morphological Filter for Ground Identification of LIDAR Data. http://tpingel.org/code/smrf/smrf.html

using the process described in **dem-gapfill-steps**. For more information on tweaking the SMRF classification algorithm, see the **pc-classify** option. The result is stored in *odm_dem/dtm.tif*.



DSM (left) vs. DTM (right)

end-with

The processing pipeline is composed of several steps and processing is executed sequentially. Sometimes the results don't turn out as expected or a user might wish to compare the results of using different options. Since changing an option sometimes affects only a certain stage of the pipeline, there's no need to execute every single step all the way to the end. By using this option, the program will stop the execution at the chosen step. Possible values (in order of execution) are:

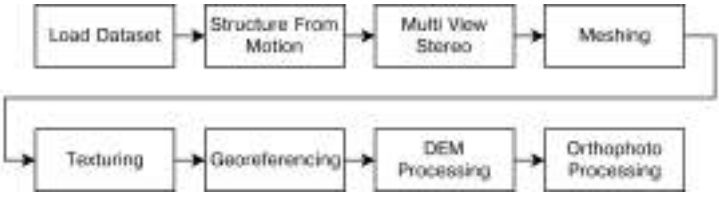
- **dataset**
- **split**
- **merge**
- **opensfm** (Structure From Motion)
- **mve** (Multi-View Stereo)
- **odm_filterpoints**
- **odm_meshing**
- **odm_25dmeshing**
- **mvs_texturing**

- **odm_georeferencing**
- **odm_dem**
- **odm_orthophoto**

This option is often used in conjunction with **rerun-from**.

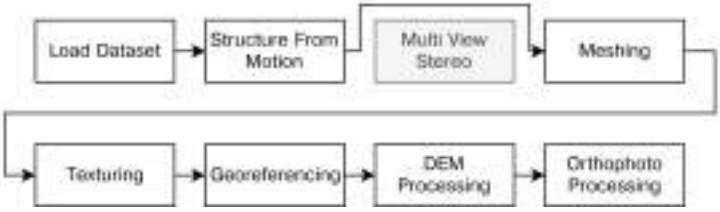
fast-orthophoto

Sometimes all that a user wants is an orthophoto, generated as fast as possible, using the least amount of resources. If we revisit the overview of the processing pipeline, this is how it's generated executed:



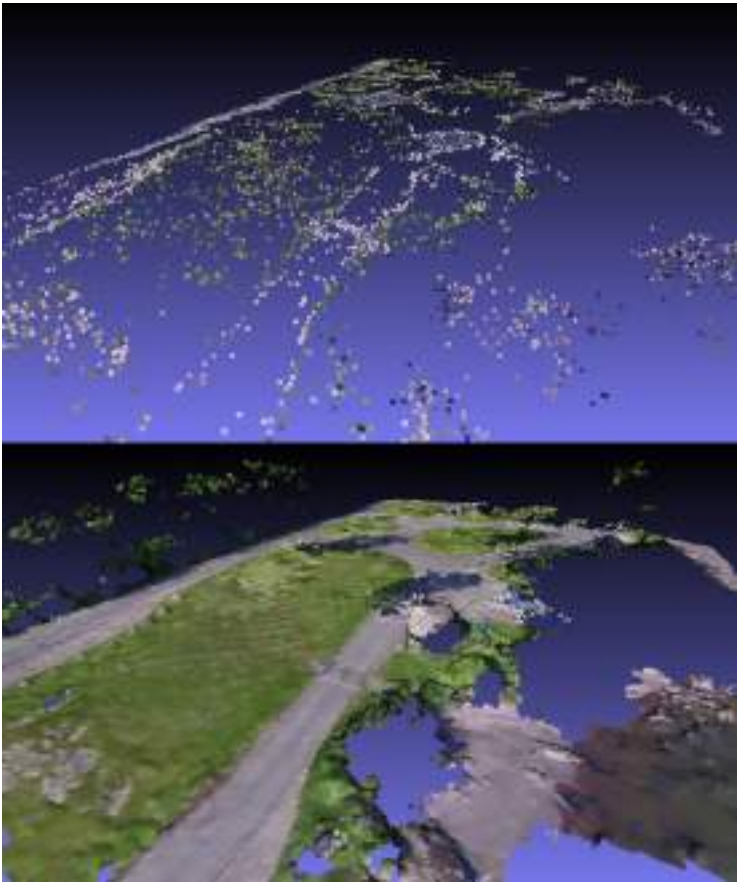
Processing Pipeline (normal)

When **fast-orthophoto** is used, the program is instructed to skip the expensive MVS step.



Processing Pipeline (fast-orthophoto)

Recall that SFM computes a sparse point cloud, while MVS generates a refined, denser point cloud.



Sparse (top) vs. dense (bottom) point cloud outputs

Both point clouds can be used to generate a mesh. However, it's better to have more points, as meshes can be created with more details. In the dense point cloud screenshot above, the

building in the middle of the scene is well defined, but it's almost missing in the sparse point cloud. Buildings are especially difficult to model without a dense point cloud, so this option tends to yield poor results in urban areas. For flat areas such as farmlands, however, the results are quite good, since there are fewer buildings or structures to model. This option also tends to work well with images captured from a really high altitude or with images that have less than 50% overlap (such as many historical aerial images).

Normal (top) vs. fast-orthophoto (bottom)

Comparing the two images above, the building on the top is much more defined than the bottom, but the terrain between the two is almost identical.

This option can make a big difference in run-time, especially for very large datasets.

gcp

By default the program looks for a file named *gcp_list.txt* in the project directory. If it exists, it's used as a ground control point file to increase the georeferencing accuracy of the results. With this option users can specify an alternate path for the GCP file. Ground control points and the GCP file format are explained in more detail in the *Ground Control Points* chapter.

This option is not shown in WebODM and is automatically set if a GCP file is uploaded with a dataset.

help

Shows all possible options and exits.

ignore-gsd

To achieve good processing speed, the program relies on optimizations. One of these optimizations uses the average Ground Sampling Distance (GSD) value from all images (plus some buffer) to achieve two goals:

1) Put a cap on the resolution of orthophotos and DEMs.
2) Compute a good target size for the images when texturing 3D models.

The reason for placing a cap on resolutions is simple. While the program allows output resolutions to be set via **orthophoto-resolution** and **dem-resolution**, it can be tedious to estimate what the maximum resolution can be. For example, if photos are captured at 400ft, it makes no sense to set the resolution

to 0.1 cm / pixel. The photos don't contain enough details to achieve that target resolution. So the optimization automatically lowers the resolution to a more reasonable value.

Similarly, when the program knows that the orthophoto is going to have a target resolution of 5 cm / pixel, it's wasteful to use full resolution images to create the textures for the 3D models. The orthophoto will be downsized anyway, so the program resizes the images prior to texturing, speeding things up and lowering memory usage.

There are a few caveats with this approach:

1) The GSD value is computed by averaging the GSD value of all images in the scene. Furthermore, the flight altitude necessary for the computation is estimated from an average plane height computed from the sparse point cloud.

Average plane height (dotted gray line) and terrain (black line)

This means areas that have large changes in elevation (hills, mountains) might turn into an inaccurate estimate for the GSD value. In such scenario, the resolution could be possibly capped at a value lower than ideal.

2) To color a mesh, the texturing program has to choose the best section of a photo, from a set of photos. To decide which photo and section is best, the algorithm uses several factors.

When **texturing-data-term** is set to gmi (the default), one of these factors is an indicator of *sharpness*. Resizing images prior to texturing has the side effect of reducing the *sharpness* of all images, thus decreasing the importance of this factor relative to others. This can result in more blurred areas.

When noticing excessive blur in an orthophoto, or when the resolution of the outputs is lower than expected, turning on **ignore-gsd** can improve results. The trade-off is longer runtime and higher memory usage.

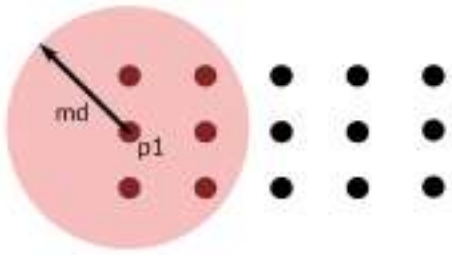
matcher-distance

The SFM process involves matching image pairs, that is, finding images that share features between them (images showing the same objects). The naive, brute force approach is to compare each image against each other image. This results in an exhaustive, but slow search. Finding all images pairs in a 100 images dataset requires a lot of comparisons. To be exact, it requires:

```
100 * (100 - 1)
= 9900 comparisons
```

This number increases rapidly with larger datasets. To speed things up, the program uses an optimization. The core idea is that, when processing datasets collected in a uniform pattern, most images will be paired with images that are within a short distance from each other. Since each image often contains GPS location information, the program can set a distance threshold for which image pairs should not be considered. This is called

preemptive matching.

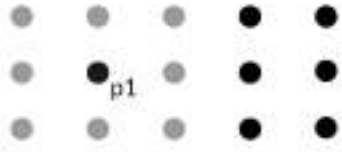


Dots represent approximate image locations, extracted from EXIF tags. The md value represents the maximum distance that the image p1 will search for other images. Images outside of the circle will not be considered for matching

The distance is expressed in meters. It can be set to zero to disable it. If no location information is embedded in the EXIF tags of the images, this option is disabled. This option works in conjunction with the **matcher-neighbors** option.

matcher-neighbors

Similarly to **matcher-distance**, this option performs preemptive matching by considering only the nearest neighbors of each image. The illustration below shows the result of setting this option to 8:



Dots represent approximate image locations, extracted from EXIF tags. When the matcher-neighbors is set to 8, only the 8 nearest neighbors (highlighted in gray) are considered for matching with image p1

For datasets with lots of overlap, it can be beneficial to increase this value since it's likely that valid matches will not be taken into consideration and decrease the accuracy of the reconstruction. It can be set to zero to disable it. If no location information is embedded in the EXIF tags of the images, this option is disabled. This option works in conjunction with the **matcher-distance** option.

max-concurrency

By default the program will attempt to use all available CPU resources. There are scenarios where this might not be desirable, for example on shared servers or when wanting to use the computer for other tasks while processing. This option attempts (but does not guarantee) to limit the maximum number of CPU cores that will be used at the same time.

merge

This option controls what assets should be merged during the merge step of the split-merge pipeline. By default all available assets are merged, but users can choose to merge only specific ones. We cover this option in more detail in the *Processing Large Datasets* chapter.

mesh-octree-depth

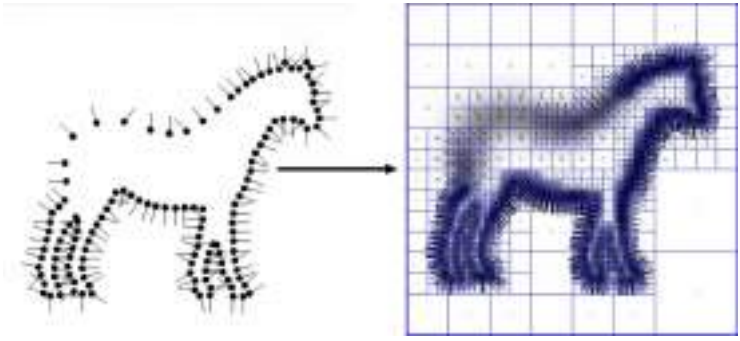
When it comes to generating 3D models, this is probably the most important option. It specifies a key variable for the Screened Poisson Reconstruction[25] algorithm, which is responsible for generating a mesh from the point cloud. The details of the algorithm are fascinating, but probably outside the scope of this chapter. For the curious ones, the best description I could find is available on Wikipedia under the *Surface Reconstruction* section at https://en.wikipedia.org/wiki/Poisson%27s_equation

To understand how this option affects the output, it helps to visually understand the concept of an octree. First, octree means *eight-tree* (okta is *eight* in Greek). Why eight? Because at each level (or *depth*) of the tree, each box (or *node or branch*) of the tree is divided in eight parts. At the first level there's only one branch. At the second level there's 8. At the third there's 64 and so forth.

[25] Screened Poisson Reconstruction: watertight surfaces from oriented point sets. http://www.cs.jhu.edu/~misha/MyPapers/ToG13.pdf

An octree with depth 1, 2 and 3

Lower depths in an octree allow finer details to be captured.



Points and resulting octree.
Image from http://www.cs.jhu.edu/~misha/Code/

The practical aspect of this option is that the higher the value, the finer the resulting mesh will be. The trade-off is exponentially longer run-time and memory usage. The default value of 9 works well for a lot of different cases. Flat areas can benefit from lower values (6-8) and urban areas can improve by setting this value higher (10-12). When increasing this option, **mesh-size** should also be increased as finer meshes require more triangles.

mesh-octree-depth 6 and mesh-size 10000 (top) vs.
mesh-octree-depth 11 and mesh-size 1000000 (bottom)

mesh-point-weight

Similarly to **mesh-octree-depth**, this option specifies a key variable for the Screened Poisson Reconstruction algorithm, which is responsible for generating a mesh from the point cloud. It affects the mesh by giving more importance to the location of the points. In practical terms, higher values can help create higher fidelity models, but can also lead to the generation of artifacts (undesired alterations). In general the default value works fairly well.

mesh-point-weight set to 0 (top) and 20 (bottom). Notice the lack of edges in the top image and the excessive bumps in the bottom image

mesh-samples

Similarly to **mesh-octree-depth**, this option specifies a key variable for the Screened Poisson Reconstruction algorithm, which is responsible for generating a mesh from the point cloud. It specifies how many points should fall within a node of the octree during its construction. In practical terms, this value should be tweaked between 1 and 20 to improve the smoothness of a model. If there's noise in the point cloud, this value should be increased. If there's little or no noise in the point cloud, this value should be set to 1 (the default).

mesh-samples set to 1 (top) 20 (bottom). Ideal values for this option are between 1 and 20

mesh-size

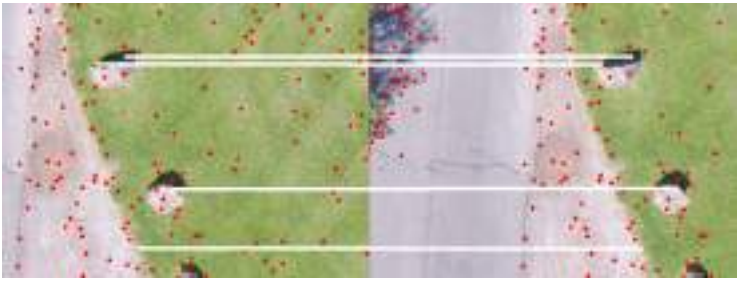
To keep memory usage and run-time under control, after a mesh is generated the program simplifies it by setting an upper limit on the number of triangles the mesh can contain. The more triangles a mesh contains, the longer it takes to process it in subsequent steps of the pipeline. A low triangle count can sometimes degrade the quality of the mesh. If details seem to be missing from the 3D model or sharp triangles are present, increasing this option could improve results. This is especially beneficial in urban areas where buildings require fine details for more appealing results.

Effects of reducing the number of triangles in a mesh. Image courtesy of Trevorgoodchild, Quadric error metric simplification applied to the Stanford bunny, CC BY-SA 3.0

min-num-features

During the SFM process, images have to be matched into pairs. The way the pairing happens is by means of finding matches between features in the images. An example of a feature is the corner of a building or the edge of a car. If the same feature is

found to be present between two images, it's used as evidence of a possible match. But features can be ambiguous. An identical-looking corner of a building between two images could be from two corners of two different building that just happen to share the same architecture style. So the program finds lots and lots of features in each image and uses all of them to find possible pairs. This option controls the minimum number of features the program tries to finds in each image, thus increasing the likelihood of finding matches between images. It does so by progressively lowering certain threshold values, which lead to more, but less ideal features.



Features (red points) and matches between overlapping images (white lines). min-num-features controls the target number of red points in each image

This option should be increased when trying to map areas that have few distinguishable features, such as forest areas:

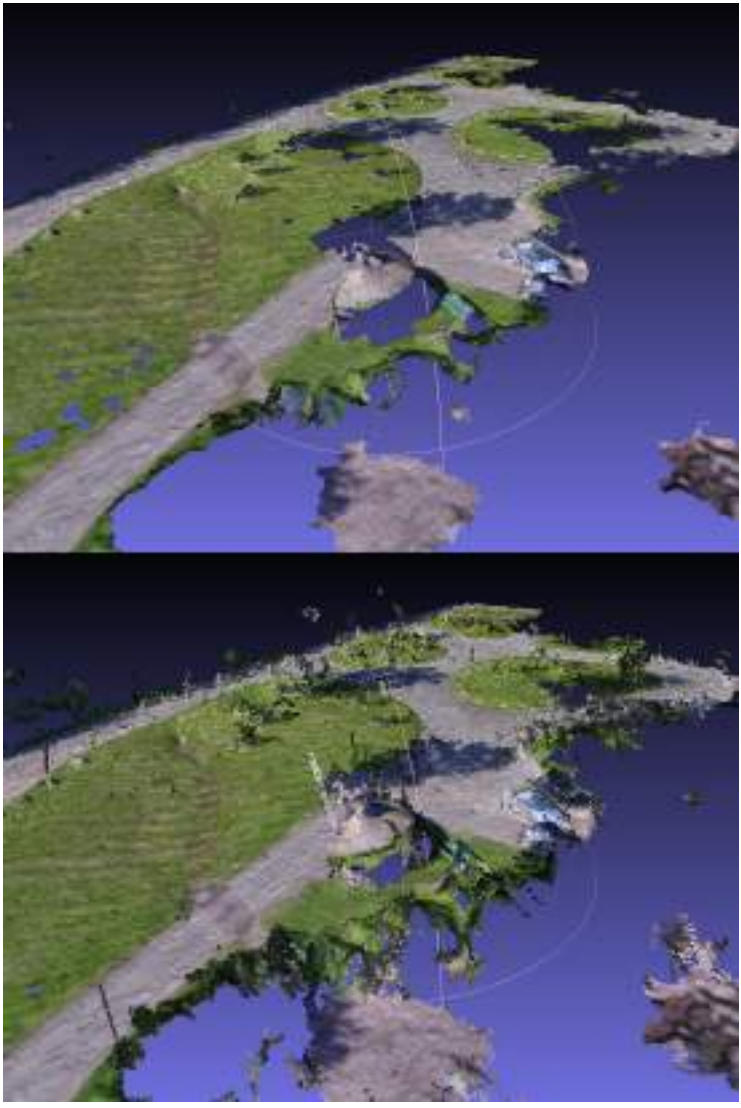
Can you easily find many good features / reference points in the image above? Neither can a computer. But increasing the number of features can increase the chances of a match between images

Increasing this option results in longer run-time, but increases the chances that a reconstruction will be properly generated. In certain instances, the program might also be able to generate only a partial reconstruction. In such cases users will notice that some areas for which images exists do not appear in the final results (a chunk of the orthophoto will appear to be missing). Increasing this option could help find matches that generate a more complete reconstruction.

An interesting side effect is that increasing this option also increases the number of points in the sparse point cloud, so it can be used in conjunction with **fast-orthophoto** to produce slightly better results.

mve-confidence

When the dense point cloud is computed using MVE (the default), each point is assigned a confidence value between zero and one. A value of zero indicates that the point is most likely noise, while a value of one indicates that the point is most likely a good point. Points below a certain confidence threshold are discarded. Users can increase this option to decrease noise (but potentially eliminate valid points) or decrease it to get more complete point clouds (but potentially increase noise).



Confidence set to 0.6 (top) and 0 (bottom). Notices some areas are missing in the top image, but much more noise is present in the bottom image

opensfm-depthmap-method

When **use-opensfm-dense** is set, this option affects the point cloud by specifying the method used to compute depthmaps (see **depthmap-resolution** for a brief discussion on depthmaps). The three possible options are:

- **PATCH_MATCH** (default)
- **PATCH_MATCH_SAMPLE**
- **BRUTE_FORCE**

PATCH_MATCH and **PATCH_MATCH_SAMPLE** are faster, but sometimes miss some valid points, which can result in a point cloud with some empty areas. **BRUTE_FORCE** is slower, but does a more exhaustive job and can produces more complete results. The default value tends to work well and users should switch to **BRUTE_FORCE** only if the point cloud is missing significant chunks. The **PATCH_MATCH** approaches are based on the paper Accurate Multiple View 3D Reconstruction Using Patch-Based Stereo for Large-Scale Scenes[26]. **PATCH_MATCH** is slightly slower than **PATCH_MATCH_SAMPLE** but tends to create slightly denser point clouds.

opensfm-depthmap-min-patch-sd

When **use-opensfm-dense** is turned on and **opensfm-depthmap-method** is set to **PATCH_MATCH** or **PATCH_MATCH_SAMPLE** this option controls a key variable that helps define areas in the depthmaps that should be skipped for improving run-

[26] Accurate Multiple View 3D Reconstruction Using Patch-Based Stereo for Large-Scale Scenes. http://www.nlpr.ia.ac.cn/2013papers/gjkw/gk11.pdf

time performance. Patches are simply small sections in an image:

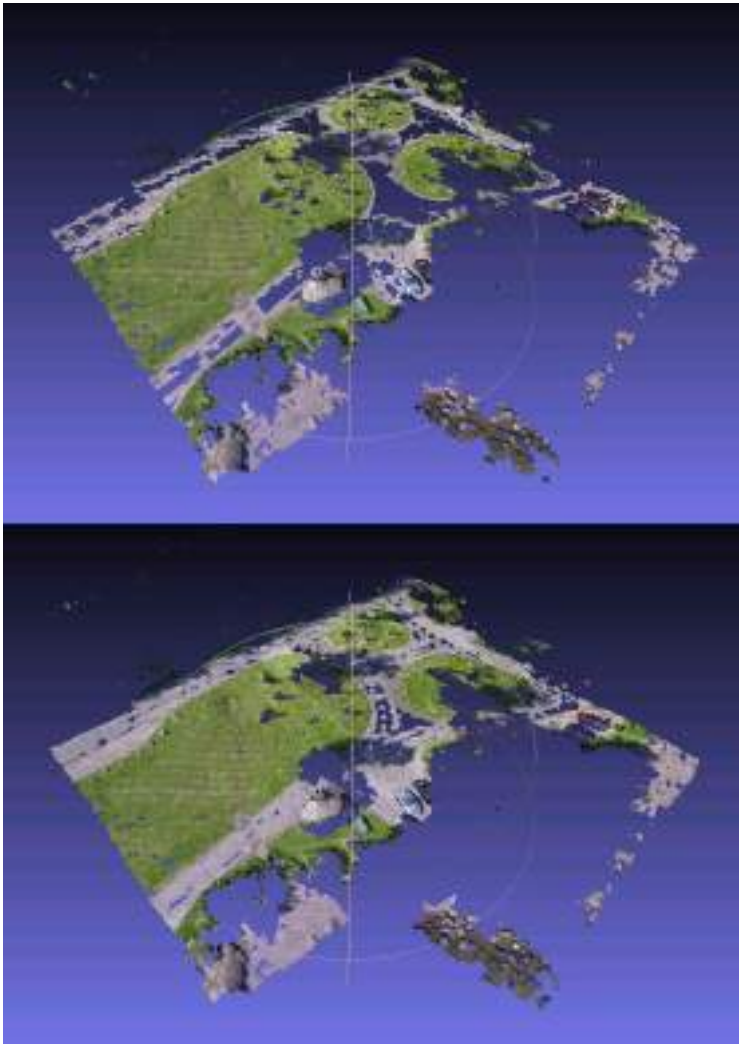
Patch in an image

During computation, input images are split into patches. Each patch is assigned a variance value computed from its pixels. The key idea is that areas in the image that are uniform (contain similar values), such as the sky, contain little to no information useful for triangulating points.

Patch with high variance (left) vs. patch with low variance (right)

Variance is simply standard deviation multiplied by itself. So this option defines the minimum standard deviation that an image patch must have during the OpenSfM depthmap calculation process for it to be considered in the computation. Setting this value too low will result in longer run-time, while

setting this value too high could potentially make areas with uniform values to be ignored. The screenshots below illustrates this behavior:



opensfm-depthmap-min-patch-sd set to 5 (top) and 2.5 (bottom). Notice the lack of roads in the image on the left. Roads have uniform colors (low variance). Setting a high value caused the roads to disappear!

orthophoto-bigtiff

BigTIFF is an extension of the TIFF format to support files larger than 4GB. The possible values for this option are:

- **YES**: force the output orthophoto to use BigTIFF
- **NO**: force the output orthophoto to use classic TIFF
- **IF_NEEDED**: will use BigTIFF if it is needed (image larger than 4GB and not using compression, see **orthophoto-compression**)
- **IF_SAFER** (default): will use BigTIFF if the resulting file might exceed 4GB. This is a heuristics that might not always work depending on compression

The BigTIFF format is not backward compatible with classic TIFF (programs compatible with TIFF do not necessarily support BigTIFF). A viewer needs to have explicit support for BigTIFF. Luckily most GIS programs support BigTIFF. Some legacy applications however might not. If you use one of these legacy applications, set this option to **NO**. If receiving a *TIFFAppendToStrip:Maximum TIFF file size exceeded* error, the heuristic used for **IF_SAFER** failed to guess the final size of the image. In this case changing this option to **YES** can fix the error.

orthophoto-compression

Compression is a method to save space in exchange for slightly longer run-time. The possible values for this option are:

- **JPEG:** Uses JPEG compression with quality value of 75. JPEG is a lossy compression method, meaning some image quality is lost during compression.

- **LZW:** Uses Lempel–Ziv–Welch compression. This is a lossless compression method, meaning image quality is not lost during compression.
- **PACKBITS:** This compression method, like LZW, is lossless. It's arguably more supported than LZW, but achieves less compression than LZW.
- **DEFLATE** (default): also referred as ZIP compression, deflate is a lossless compression method. It tends to yield slightly smaller file sizes when compared to LZW.
- **LZMA:** another lossless compression method.
- **NONE:** Skips compression. Speeds up the generation of the orthophoto, but creates larger files.

orthophoto-cutline

By turning on this option the program will generate a cutline. A cutline is a polygon within the orthophoto's crop area that attempts to follow the edges of features.

Cutline

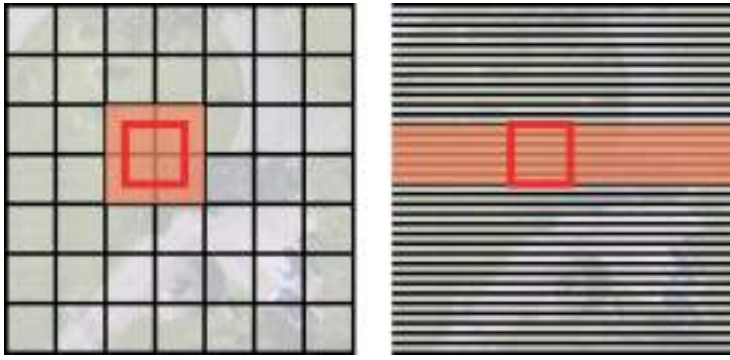
Cutline going around the edges of a car

A cutline can be used to merge overlapping orthophotos by minimizing the color differences between seams. It's used to merge orthophotos when processing large datasets using the split-merge pipeline (see the *Processing Large Datasets* chapter).

The cutline is saved in *odm_orthophoto/cutline.gpkg*.

orthophoto-no-tiled

By default the program will generate tiled TIFFs, with a tile size of 256x256 pixels. Tiling in this context is not the same as generating tiles for web viewers. Instead its related to the arrangement of data within the file. Data can be arranged either in tiles or stripes. When reading and displaying an entire file, tiles and stripes are equivalent in performance, since all data must be read regardless. When accessing a subsection of the image however, for example when zooming into an area, using tiles often results in needing to read less data.



Tiles (left) vs. stripes (right). The red rectangle is the area being accessed. The highlighted area shows the amount of data that needs to be read from the file. The smaller the highlighted area, the quicker it is to access the file

Tiles are the default. Tiled orthophotos take slightly longer to create compared to striped. To use stripes, users can turn on this option.

orthophoto-resolution

Same as **dem-resolution**, but applied to orthophotos instead of DEMs.

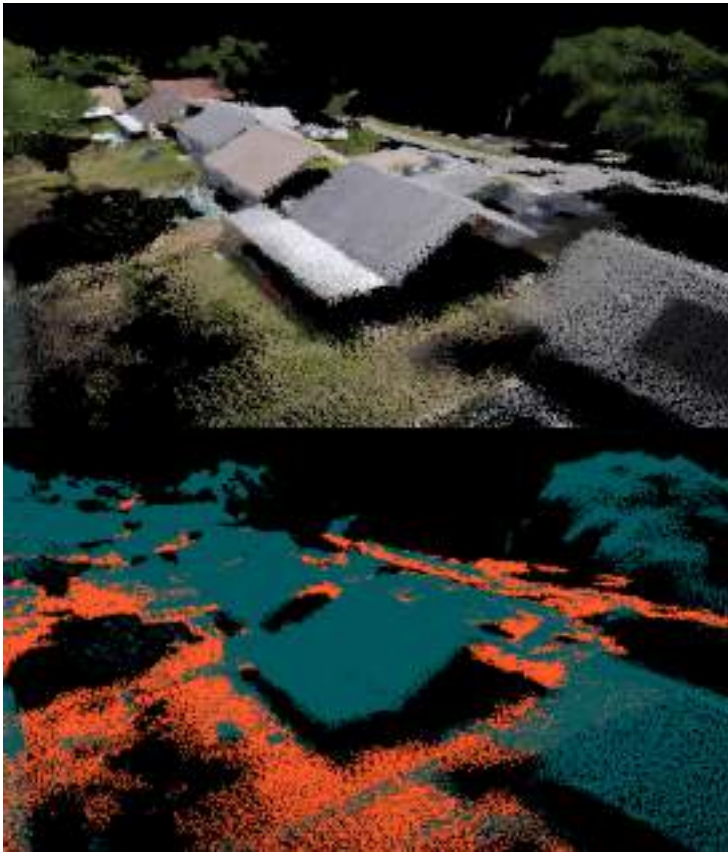
pc-classify

Points in a point cloud can be assigned one of several classification values[27] to indicate whether a point is part of the terrain (ground), of a building, of a tree (vegetation) and several other possible classifications. By default every point is simply labeled

[27] LAS 1.4 Specification: https://www.asprs.org/wp-content/uploads/2010/12/LAS_1_4_r13.pdf

as *unclassified* and the software does not attempt to label what each point represents. By turning on this option, a Simple Morphological Filter[28] (SMRF) is used to attempt to find the points that are part of the terrain (ground) and assign to them a ground classification value. The end result is a point cloud that is divided between ground and non-ground points. The ground point cloud can then be used for the purpose of computing a DTM.

[28] smrf: A Simple Morphological Filter for Ground Identification of LIDAR Data. http://tpingel.org/code/smrf/smrf.html



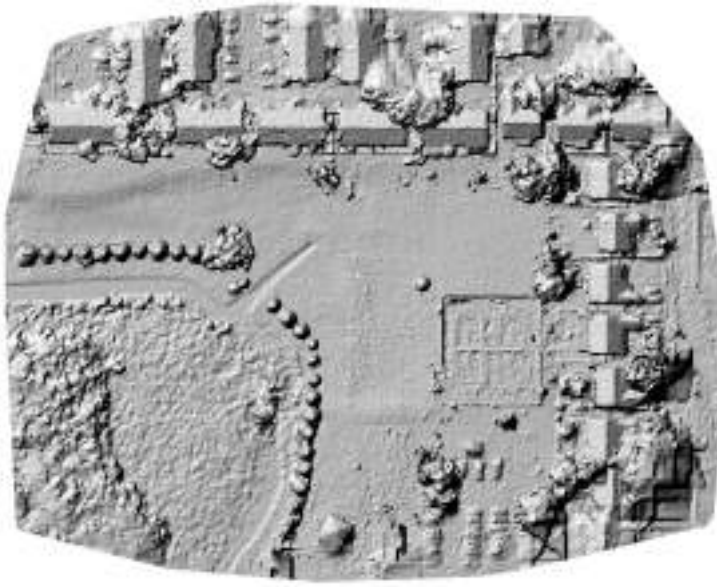
Point cloud (top) and classification results with SMRF (bottom)

The SMRF algorithm can be controlled via four options. It's usually recommended to try the default values, examine results and then make tweaks as needed.

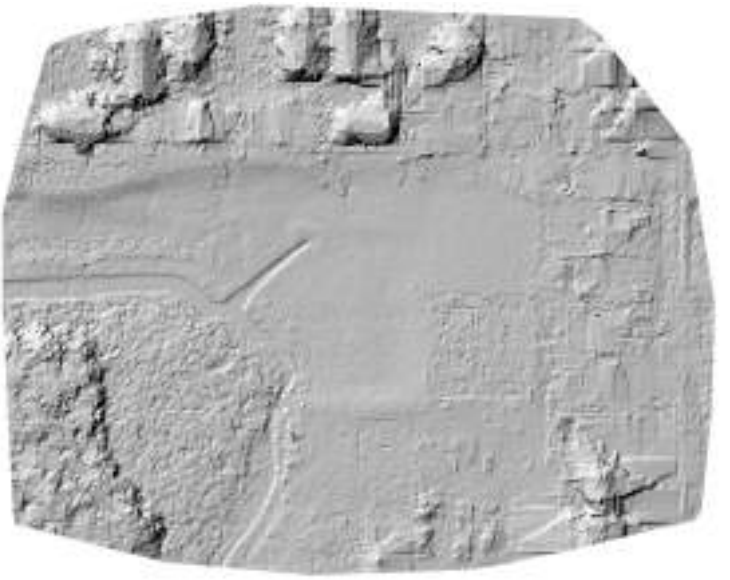
- **smrf-scalar:** is used to make the threshold parameter dependent on the slope. To improve results, this value can be decreased slightly if the **smrf-threshold** value is increased and vice-versa.

- **smrf-slope:** should be set to the *largest common terrain slope*, expressed as a ratio between change in elevation and change in horizontal distance (if elevation changes by 1.5 meters over a 10 meter distance, that's 1.5 / 10 = 0.15). It should be increased for terrains with large slope variation (hills, mountains) and decreased for flat areas. For best results it should be higher than 0.1, but not higher than 1.2.
- **smrf-threshold:** specifies the minimum height (in meters) of non-ground objects. For example, setting a value of 5 will likely be sufficient to identify buildings, but will not be sufficient to identify cars. To identify cars the value should be lowered to 2 or even 1.5 (the average car height). This parameter alone has the biggest impact on results.
- **smrf-window:** should be set to the size of the largest non-ground feature (in meters). For example, if a scene is full of small objects (trees), this value can be decreased. If the scene contains large objects (buildings), this value can be increased. It's recommended to keep this value above 10.

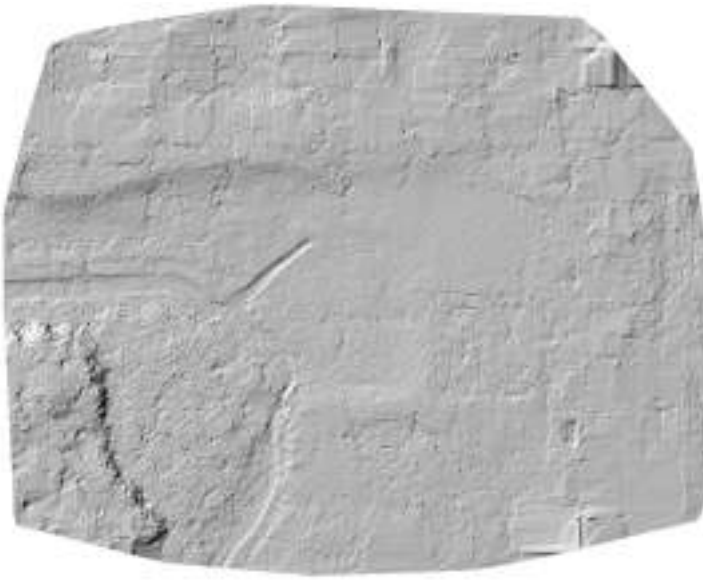
SMRF has limitations and it's important to understand them. In particular, the filter will sometimes mistakenly classify points that belong to buildings or trees as ground points (type II errors).



Input surface model



Terrain model obtained with default SMRF options. Note some houses were mistakenly included and artifacts are lingering around the edges of removed objects



A much improved terrain model obtained by setting smrf-threshold 0.3 (decreased), smrf-scalar 1.3 (increased), smrf-slope 0.05 (decreased) and smrf-window 24 (increased)

Automated methods for reliable classification of point clouds are an active area of research. SMRF performs remarkably well, but often requires some tweaking. Users wishing to generate high quality DTMs should always double check the results and adjust the SMRF options as needed. It's also sometimes worth comparing the classification results with those obtained from supervised or trained classification methods. CloudCompare[29] is a free and open source software that implements such methods[30].

[29] CloudCompare: http://www.cloudcompare.org

[30] CloudCompare CANUPO Plugin: http://www.cloudcompare.org/doc/wiki/index.php?title=CANUPO_(plugin)

pc-csv

By default the output point cloud is exported in a compressed LAZ format. The LAZ format is not human readable and cannot be opened with a simple text editor. Users can export a copy of the point cloud to CSV (Comma Separated Value) format by turning on this option. The CSV file format is not ideal for point cloud data, but can sometimes be useful for debugging the values of the point cloud or for import in programs that do not understand LAS/LAZ. The resulting point cloud is stored in *odm_georereferencing/odm_georeferenced_model.csv*.

pc-ept

By setting this option users can export the point cloud in Entwine Point Tile (EPT) format[31], which can be used to efficiently stream point clouds across networks. EPT datasets can also be efficiently analyzed, queried and transformed using tools such as PDAL[32]. The resulting EPT dataset is stored in the *entwine_pointcloud* directory.

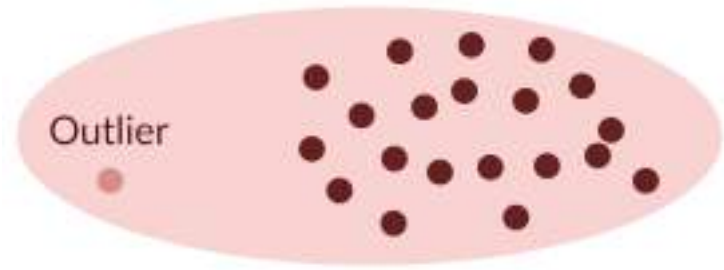
pc-filter

Noise from the point cloud can be partially removed using a statistical filter. This option sets the standard deviation threshold value for the filter. Standard deviation is a measure of how spread out points are relative to their neighbors. The filter looks at the closest 16 neighbors for each point and computes

[31] Entwine Point Tile: https://entwine.io/entwine-point-tile.html

[32] PDAL: https://pdal.io

their standard deviations (how far each point deviates from the average distance to each other point). If a point is found to be too far away relative to its neighbors, thus having a standard deviation higher than the threshold, the point is labeled as an outlier.



The gray point has a high standard deviation, so it's labeled as an outlier

Setting this value too high will keep some noisy points, while setting this value too low will possibly remove valid points. Filtering can be disabled by setting this option to zero.

pc-las

By default the output point cloud is exported in a compressed LAZ format. Since not all programs support LAZ, users can export a copy of the point cloud in uncompressed LAS format by turning on this option. The resulting point cloud is stored in *odm_georereferencing/odm_georeferenced_model.las*.

rerun

Shorthand for **rerun-from <step> end-with <step>**.

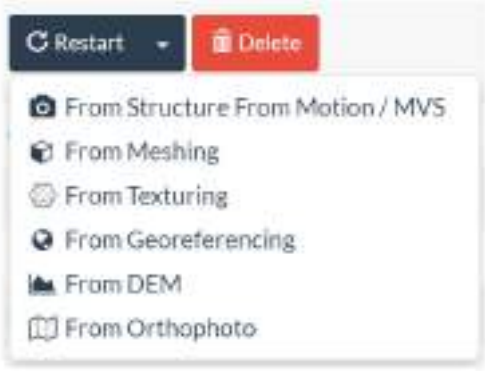
rerun-all

Shorthand for **rerun-from dataset**, while also removing all output folders before starting the process.

rerun-from

Same as **end-with**, except that it instructs the program to resume execution from a specific point in the pipeline, skipping previous steps.

When using WebODM the rerun-from option is automatically set when using the **Restart** button dropdown.



Restart drop-down in WebODM

It's not always possible to restart a task from a certain step in WebODM. The processing node has to support task restarts

and the task intermediate results need to have been kept on disk (by default they are kept only for 2 days). See *The NodeODM API* chapter for information on how to change the number of days results are kept.

resize-to

During the SFM process the program needs to extract features from each image. To speed things up, the program resizes all images prior to performing feature extraction . This option specifies the target size of the largest side of the images for the purposes of feature extraction. It's important to note that the input images are not affected by this option and neither are other stages of the pipeline. Changing this option will not degrade the quality of resulting orthophotos or 3D models. This option can be lowered with datasets that have lots of recognizable features (cars, buildings, etc.) and should be increased with datasets that lack them (forest areas, deserts, etc.).

skip-3dmodel

Sometimes all that a user wants is an orthophoto. In that case, it's not necessary to generate a full 3D model. This option saves some time by skipping the commands that produce a 3D model. A 2.5D model (a 3D model where elevation is simply *extruded* from the ground plane) is still generated. 2.5D models are not *true* 3D models as they cannot represent the true shape of objects such as overhangs, but work well for the purpose of rendering orthophotos.

3D model (top) vs. 2.5D model (bottom). Note the absence of overhangs on the bottom

By default both models are created. See also **use-3dmesh**.

sm-cluster

Specifies a URL to a ClusterODM instance. When combined with the **split** option, it enables the distributed split-merge pipeline for processing large datasets in parallel using multiple processing nodes. We cover this option in more detail in the *Processing Large Datasets* chapter.

smrf-scalar

Controls the scalar variable for SMRF. See **pc-classify**.

smrf-slope

Controls the slope variable for SMRF. See **pc-classify**.

smrf-threshold

Controls the threshold variable for SMRF. See **pc-classify**.

smrf-window

Controls the window variable for SMRF. See **pc-classify**.

split

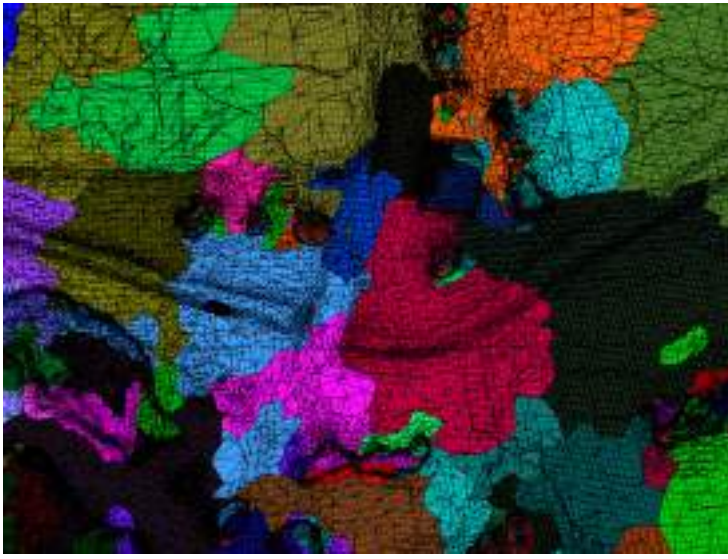
When set to a number lower than the number of input images, enables the split-merge pipeline. We cover this option in more detail in the *Processing Large Datasets* chapter.

split-overlap

Specifies the amount of overlap (in meters) that submodels should have during the split-merge pipeline. We cover this option in more detail in the *Processing Large Datasets* chapter.

texturing-data-term

A difficult part of texturing a mesh is answering the question of how to choose the best image for each part of the mesh, since due to overlap each part of the mesh has likely been photographed by multiple images. This process is known as *view selection* and is guided by the definition of a *data term* or a *cost function*.



The view selection process assigns different areas of a mesh with an image. In the screenshot above each color represents a different image assigned to an area

Two data terms are available:

1. gmi (default)
2. area

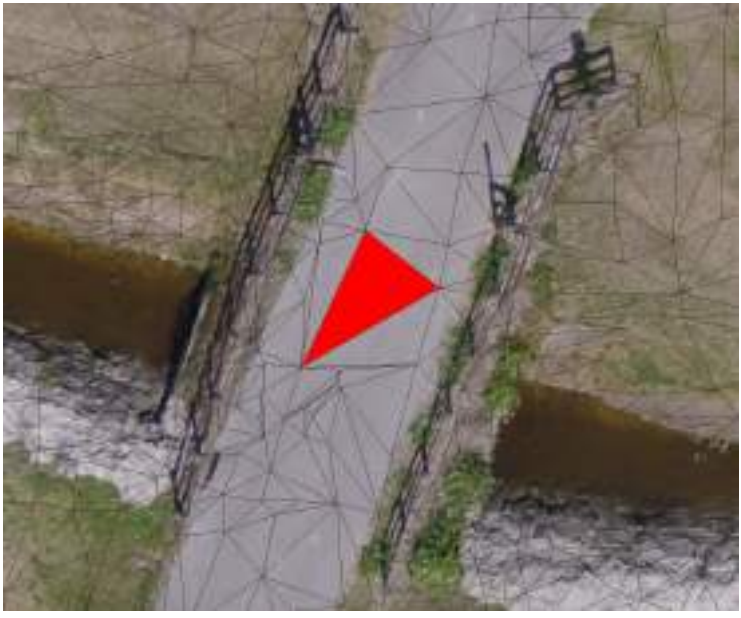
GMI stands for *Gradient Magnitude Image*. A gradient is a change in values and is often represented graphically for ease of understanding. Magnitude is a measure of how much the gradient *changes*. If a gradient changes gradually it will have lower magnitude than if the gradient changes abruptly.

A low magnitude gradient (black to white)

The purpose of the gradient is to prioritize areas in an image that are in focus (they will exhibit a higher gradient magnitude). The gradient image is computed by running a *Sobel* edge detector and computing a gradient over the result.

An image (top) vs. Sobel edge detector mapped to a black-white gradient (bottom). The water shows small changes in gradient, whereas the shoreline has a higher gradient magnitude

The *area* data term works differently. It simply prioritizes images that provide the largest area coverage for a particular section of the mesh.



A triangle in the textured mesh

The same triangle projected on two images. Because the triangle's area in the top image is bigger, the top image is chosen for coloring the triangle

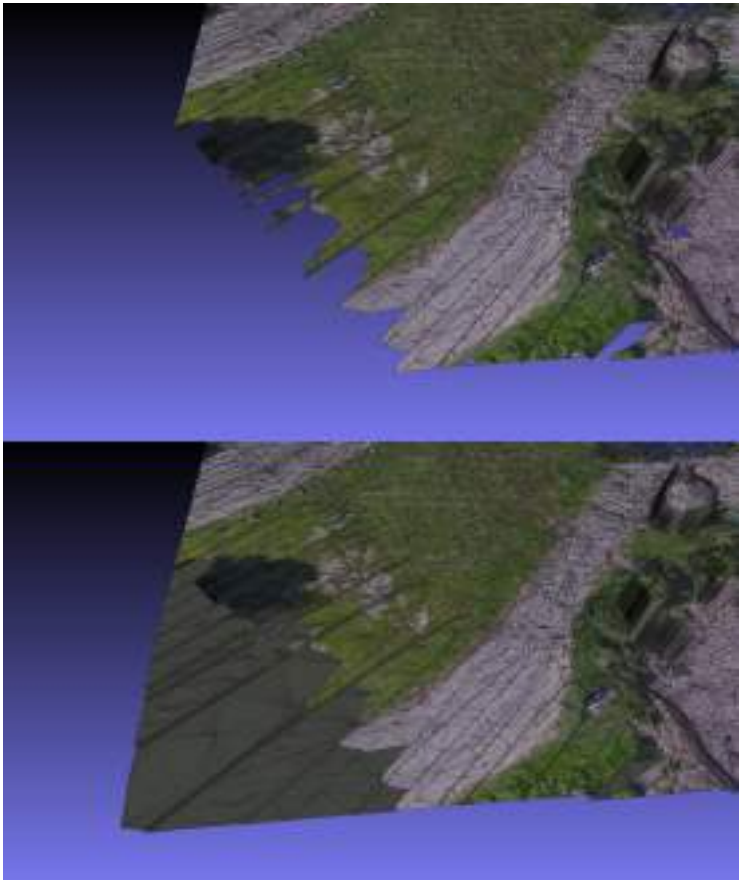
Data term gmi (top) vs. area (bottom). On the top, notice the sharpness of the ditch and the blue car, but the presence of a tree on the road

The tree was wrongly placed on the top image in this case. Trees have high gradient magnitudes, whereas roads do not.

Actual image captured from UAV for comparison

texturing-keep-unseen-faces

The input to the texturing part of the pipeline consists of a mesh, cameras and images. The input mesh is composed of triangles. The program at some point checks which triangles are visible by the cameras. By default if a triangle is not visible by any camera, it's discarded from the output.



Unseen faces are removed from the textured mesh (top) vs. faces are kept with no color (bottom)

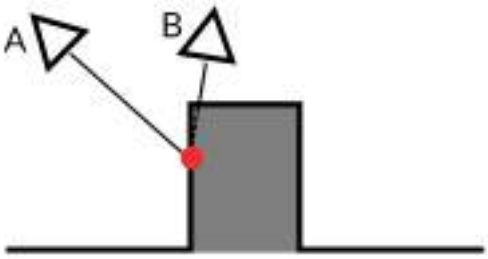
This option instructs the program to keep all triangles, regardless of whether they are seen by a camera or not.

texturing-nadir-weight

During texturing the program needs to choose the best image for each section of the mesh. In addition to using the data term (see **texturing-data-term**), if a 2.5D mesh is used for generating the orthophoto, the program enables a special mode called nadir. While in nadir mode, the program behaves differently with parts of the mesh that are vertical or almost vertical (think walls of buildings).

1. The visibility test for such vertical parts is disabled (see **texturing-skip-visibility-test** for a description of the visibility test).
2. The quality score obtained either via the *gmi* or *area* data term is replaced with a value proportional to the *nadir-ness* of the images (how straight down is an image?), multiplied by a weight. Images that are captured straight down are given more priority than images that are captured at an angle.

This option controls the weight of the nadir factor. The higher this option, the more nadir images are favored for texturing vertical areas of the mesh and vice versa.



Camera A has a better view of the side of the building and Camera B is occluded. But because visibility testing is skipped and Camera B is more nadir, in nadir mode camera B is selected.

Nadir mode can substantially the quality of orthophotos, especially around the corners of buildings.

texturing-nadir-weight set to 0 (top) and 32 (bottom). Note the quality improvements near the edges of the building

This option affects only the 2.5D textured mesh (not the 3D textured mesh). The default of 16 works well for most datasets. A higher value can increase the quality of buildings, but can lose details in other areas of the orthophoto.

texturing-outlier-removal-type

Aerial imagery is not always static. Sometimes moving objects (cars, bicycles, cats, etc.) are captured between multiple photos. Because the objects are moving however, they could end up appearing in multiple parts of the textured mesh during the texturing process.

Moving object, captured between two pictures

To prevent these artifacts, the program checks for consistency between two or more images. If inconsistencies are found, they are labeled as outliers and removed. There are two methods to

do that:

1. gauss_clamping (default)
2. gauss_damping

Both are based on evaluating each photo in which an outlier could be visible using a statistical method. Gauss Clamping is more aggressive. Upon finding a high likelihood of an outlier, it will reject the photo containing the outlier from use in the area being textured. Gauss Damping on the other hand will progressively lower the picking priority of the offending photo, so it's a less aggressive approach and can lead to smoother results.

Gauss Damping (top) vs. Gauss Clamping (bottom). Note a part of the cart was missed in the top image

texturing-skip-global-seam-leveling

During texturing the program needs to merge together images that have different characteristics, for example different light intensities.

Seams in the textured model due to different light intensities (top) and global seam leveling applied (bottom)

This kind of seam blending is referred to as *global* because it's evaluated on all texture patches. The goal is to minimize the difference between the patches as to achieve consistent

luminosity throughout.

texturing-skip-hole-filling

During the texturing process, some parts of the mesh cannot be assigned a texture. This could happen because not enough information is available to assign a particular face of the mesh to one of the input images. To mitigate this, small holes in the mesh are filled by interpolating the textures of nearby faces.

Small holes (left) are filled via interpolation (right). Interpolation comes out as a smooth blur, which is not very noticeable for small areas

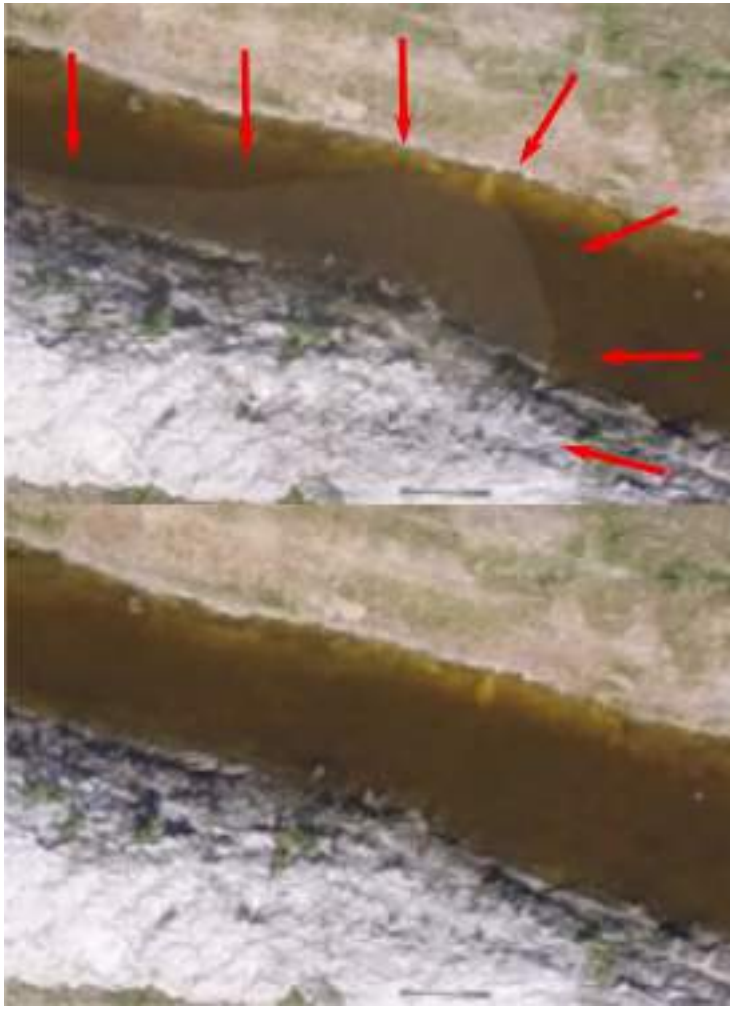
This option disables the hole filling feature (not recommended).

texturing-skip-local-seam-leveling

When texturing, the program needs to merge together images that have different characteristics, for example different light intensities. Applying global seam leveling (discussed in **texturing-skip-global-seam-leveling**) is a necessary but often insufficient step to remove all visible seams from the texture patches.

To overcome this problem, the program applies localized Poisson editing (a way to blend two images) on all texture patches. The method is *local* because it affects only a local buffer around the boundary of the texture patches to keep runtime under control. An in-depth discussion of the method is outside the scope of this chapter, but is clearly explained in the paper Let There Be Color! Large-Scale Texturing of 3D Reconstructions[33] under the *Poisson Editing* section.

[33] Let There Be Color! Large-Scale Texturing of 3D Reconstructions: https://www.gcc.tu-darmstadt.de/media/gcc/papers/Waechter-2014-LTB.pdf

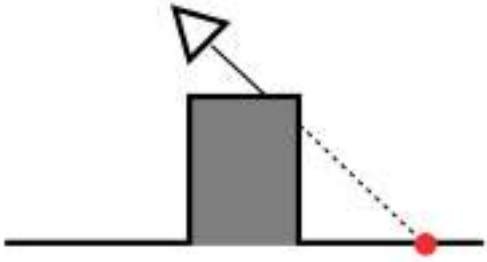


Result with global seam leveling, but not local seam leveling (top) and with local seam leveling (bottom)

This option disables local seam leveling (not recommended).

texturing-skip-visibility-test

During the texturing process, mesh faces are checked for visibility. If an obstacle exists between a face and a camera (such as a building), that face will ignore image information coming from that particular camera. This is a good way to assure consistency.



A camera ray projected onto a face (red dot). A building is in the way, so the camera is ignored.
Without visibility testing, images from building rooftops would show up on the ground!

texturing-tone-mapping

This option can sometimes help enhance the quality of textured meshes and thus the quality of the orthophotos. When setting this option to **gamma**, the program applies gamma correction to the texture maps, prior to applying local and global seam leveling. Gamma correction is a method for mapping luminance values in an image (for which an in-depth discussion is available at https://www.cambridgeincolour.com/tutorials/gamma-correction.htm). The use of gamma correction in the

context of texturing can generate more vivid results.

Raw input image (left), gamma corrected image (right)

time

Generates a *benchmark.txt* file stored in the project directory showing the time it took to process each step of the pipeline. Useful for measuring performance. By default no benchmark file is generated.

use-3dmesh

By default a 2.5D textured mesh is used to render the orthophoto. 2.5D meshes tend to work well for most aerial datasets, but can sometimes lead to sub-par results, especially if there are no nadir images in the datasets (images with the camera pointed straight or almost straight at the ground). The reason for it is that the texturing step is performed differently between 3D and 2.5D meshes. 2.5D meshes give priority to nadir images, and if these are missing, the texturing might be of lesser quality compared to a 3D mesh. For points of interests,

such as one obtained by orbiting a single building at close range with oblique images, a 2.5D mesh will also perform poorly. This option instructs the program to use the full 3D model for generating an orthophoto and to skip the generation of the 2.5D model. See also **skip-3dmodel**.

use-exif

A Ground Control Point file will always be used if either a *gcp_list.txt* file exists in the project directory (ODM) or if a GCP file has been uploaded with a dataset (WebODM). By turning on this option, the program ignores the GCP file and relies on the location information from the images' EXIF tags instead.

use-fixed-camera-params

During the SFM process, the camera's internal parameters need to be estimated and refined to achieve a good solution. Sometimes, due to poor image collection practices or excessive lens distortion, the camera parameters are wrongly estimated and can result in reconstructions that exhibit a *doming* effect. While better solutions to the doming effect are explained in the *Camera Calibration* chapter and through the use of the **cameras** option, by turning on this option it's possible to instruct the software not to optimize camera parameters at all (keeping the parameters fixed). This can sometimes improve results if there's little to no geometric distortion in the images and the focal length value embedded in the images' EXIF tags is accurate.

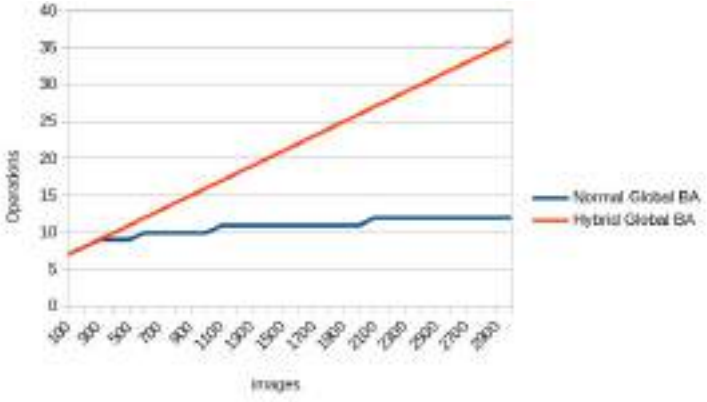
use-hybrid-bundle-adjustment

Bundle adjustment (BA) is a refinement step during the SFM process that improves the location of cameras, 3D points and camera parameters. This process needs to be done at regular intervals during the reconstruction to avoid the accumulation of errors, but is also computationally expensive. It comes in two varieties, local and global. Local BA only refines a subset of the reconstruction, global BA refines the entire reconstruction. Performing global bundle adjustment requires re-evaluating the entire scene, which is slow. By default, the program will perform global BA only after the number of new triangulated points in the scene has increased by 20%. It will also perform local BA every time a new camera is added to the scene by comparing the cameras that are within 3 levels of connectivity (capped to 30 cameras).

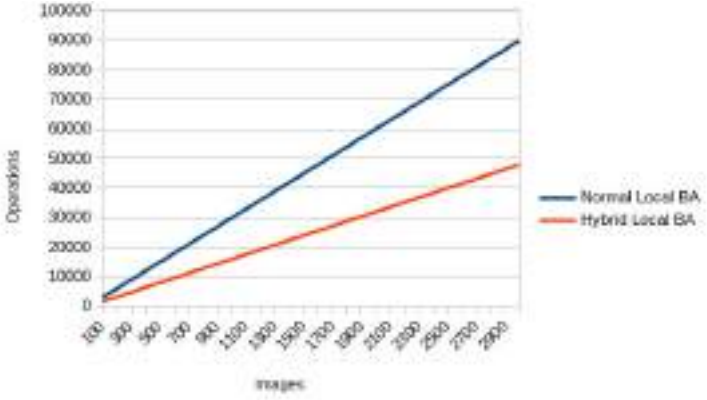
Local Bundle Adjustment with 3 (left) and 1 (right) levels of connectivity.
Gray cameras (up to 30) are used in the local BA calculation.
Camera icon by FontAwesome CC BY 4.0

By turning on this option, the program changes the logic that triggers the local and global bundle adjustment. First, a global bundle adjustment is forced every time 100 new cameras are added to the scene, regardless of the number of new points

added to the reconstruction. Second, when performing local bundle adjustment, the program only looks at the the cameras that are within 1 level of connectivity instead of 3. In short, this option increases the number of times that global BA is performed, but reduces the number of local BA operations.



Estimate of global BA operations



Estimate of local BA operations

For small datasets (< 1000 images) there's not much difference. As the number of images increases, the cost of running more local BA operations starts to outweigh the cost of running more global BA. For very large datasets, turning on this option can reduce the total run-time. It can also increase the accuracy of the reconstruction since a global bundle adjustment is performed more frequently.

use-opensfm-dense

The program has two options for generating the dense point cloud:

1. MVE (default)
2. OpenSfM

By default MVE (Multi-View Environment) is used. By turning on this option OpenSfM's depthmap reconstruction algorithm is used instead.

verbose

This option enables verbose messages. When this option is enabled, the processing output will contain more messages which can be used to diagnose possible issues.

version

This option causes the program to print the ODM version number and exit.

8

Ground Control Points

A ground control point (GCP) is a position measurement made on the ground, typically using a high precision GPS[34]. Measurements are made near identifiable structures such as street corners or by placing visible markers on the terrain.

[34] I use GPS as a synonym for GNSS, since most people recognize GPS as a word.

A ground control point marker.
Image courtesy of Michele M. Tobias & Alex Mandel Creative Commons Attribution-ShareAlike 4.0 International CC BY-SA 4.0

Images that contain the visible markers can then be *tagged* by creating a correspondence between the image location of the markers and their real world positions.

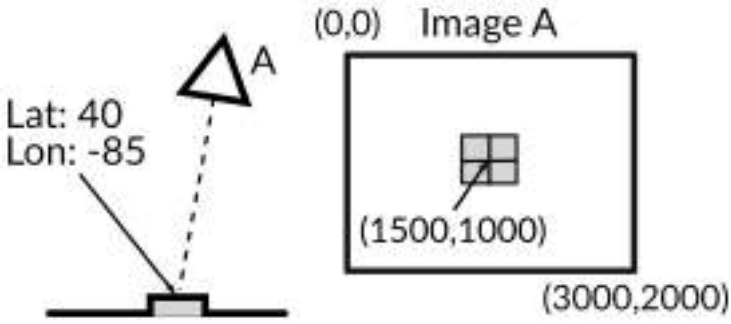


Figure 1: A GCP marker is photographed by camera A to produce Image A. In a second step, the pixel location of the marker (1500,1000) from Image A can be manually tagged with its real world coordinates (latitude 40, longitude -85).

Using ground control points can increase the georeferencing accuracy of a reconstruction, since measurements of static (non-moving) objects using a high precision GPS are often better than those obtained from the GPS of moving UAVs.

The ideal number of ground control points ranges between 5 to 8, placed evenly across the area to be flown. Adding more than 8 ground control points does not necessarily result in increased accuracy.

If the same marker is visible from multiple images, it should be tagged multiple times for each image. Ideally each marker should be tagged at least 3 times. Another way to think of it is to capture each marker on at least 3 images. This is so that the marker's location can be triangulated during computation.

Ground control points can be used by providing an additional text file along with the input images. The file follows a simple format:

- The first line indicates the spatial reference system (SRS) of

the world coordinates. There are no restrictions on the type of SRS you can use. Internally the program will convert the coordinates to the nearest WGS84 UTM[35] projection. The SRS can be specified using 3 different formats.

```
1) WGS84 UTM <zone number><hemisphere>
2) EPSG:<code>
3) <proj4>
```

Format #1 is just a human readable format specifying a UTM projection using the WGS84 reference ellipsoid and datum. Format #2 uses EPSG codes[36], which are a standard to reference many common spatial reference systems. Format #3 uses Proj4 strings[37] , which are used to explicitly to define spatial reference systems. Format #1 and #2 are preferred over #3.

These are all valid examples of SRS definitions for a GCP file:

```
WGS84 UTM 16N
WGS84 UTM 32S
EPSG:4326
EPSG:32616
+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs
+proj=utm +zone=16 +ellps=WGS84 +datum=WGS84 +units=m
    +no_defs
```

[35] Universal Transverse Mercator: https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system

[36] EPSG: http://www.epsg.org/

[37] Proj Quick Start: https://proj.org/usage/quickstart.html

A great resource to lookup spatial reference systems and their definitions is https://www.spatialreference.org.

- Subsequent lines are the world X, Y, Z coordinates, the associated pixel coordinates and the image filename (case sensitive). Optionally, extra fields (such as labels) can be specified after that. Tabs and spaces can be used interchangeably to separate fields.

```
<spatial reference system>
<geo_x> <geo_y> <geo_z> <im_x> <im_y> <image_name> [<
    extras>]
<geo_x> <geo_y> <geo_z> <im_x> <im_y> <image_name> [<
    extras>]
...
```

The GCP in Figure 1, for example, would be represented as:

```
EPSG:4326
-85 40 0 1500 1000 ImageA.jpg gcp1
```

The **geo_z** field in this case is set to zero because Figure 1 doesn't have an altitude value.

Creating a GCP file using POSM GCPi

The task of manually finding and measuring pixel coordinates from all images and tying marker locations to world coordinates can be tedious and error prone at best.

The Portable OpenStreetMap Ground Control Point Inter-

face (POSM GCPi) provides a way to make this process a bit easier. The application is already loaded as a default plugin in WebODM and can be accessed via the **GCP Interface** panel.

Alternatively it can also be downloaded and run as a standalone application by following the instructions on the project's home page[38]. It's also hosted online at https://webodm.net/gcpi.

Step 1. Create a GCP file stub

After measuring the location of your markers it's likely that you'll end up having a list of labeled point coordinates. Exporting them to CSV format and opening them in a spreadsheet application such as LibreOffice Calc[39] should yield something similar to:

```
X,Y,Z,Label
-91.9943320967465,46.8423713026218,0,gcp1
-91.9938849653384,46.8423668860772,0,gcp2
-91.9942463047423,46.8425277454029,0,gcp3
```

From here, let's add two new columns for the *px*, *py* fields (initialized to zero) and shift the *Label* column to the right, so that our file now looks like:

```
X,Y,Z,px,py,Label
-91.9943320967465,46.8423713026218,0,0,0,gcp1
-91.9938849653384,46.8423668860772,0,0,0,gcp2
-91.9942463047423,46.8425277454029,0,0,0,gcp3
```

[38] POSM GCPi: https://github.com/posm/posm-gcpi

[39] LibreOffice: https://www.libreoffice.org

Finally, we remove the first line, replacing it with a SRS definition and save the CSV file making sure to use *tabs* or *spaces* instead of *commas* as a separator character. In LibreOffice Calc you can achieve this by clicking **File - Save As...** and from the bottom left of the save window check **Edit Filter Settings**. The final result opened in a text editor should look like:

```
EPSG:4326
-91.9943320967465 46.8423713026218 0 0 0 gcp1
-91.9938849653384 46.8423668860772 0 0 0 gcp2
-91.9942463047423 46.8425277454029 0 0 0 gcp3
```

Step 2: Import the GCP file stub

From POSM GCPi, press the **Load existing Control Point File** button and select the file you just created in step 1. After pressing **Load** the map on the right should update to reflect the position of your points.

Loading a GCP stub into POSM GCPi

Step 3. Import the images and start tagging

Now import the images by pressing **Choose images**. Clicking an image will expand the panel on the left. Due to a usability quirk, the interface will add a new ground control point, which we need to remove. You can click the point on the right panel and delete it. Now from the left panel you can pan and zoom around the image to move the target icon at the location of your marker. Once it's in the proper position, simply click the target icon once from the left panel and click the corresponding target icon from the right panel. The target icon should turn green, indicating a correspondence has been set.

Tagging images with points using POSM GCPi

Now you can repeat the process for every marker and every image. If you get tired or want to save your work, simply press **Export File** and save the result in a location of your choice. You can resume your work by reloading the images and the exported file.

Notice the green dot on the top left corner. The dot shows the number of points that have been connected to at least one image (in this case, one). At the end of the process you'll want to have at least 5 or 8 green dots.

Step 4. Export the GCP file

Once you are done, you can press the **Export File** button to export your finished GCP file. This is the file that you will be using for the next steps.

Using GCP files

With WebODM and NodeODM using a GCP file is as simple as including it along with the images during file upload. With ODM, the GCP file should be placed in your project folder and be named **gcp_list.txt**. Alternatively, you can use the **gcp** option to specify a path to a GCP file. For example, if images are stored in **D:\odmbook\project\images** and your GCP file is stored in **D:\odmbook\project\gcp.txt**, you can simply invoke from the command line:

```
$ docker run -ti --rm -v //d/odmbook/project:/datasets
    /code opendronemap/odm --project-path /datasets --
    gcp /datasets/code/gcp.txt
```

Using ODM from the command line is covered in detail in *The Command Line* chapter.

How GCP files work

Sometimes the final results might not align perfectly with your ground control points. It's important to understand why. GCP obversations are used during the SFM step of the processing pipeline. During that step, camera positions are estimated based on an error minimization problem with many variables involved. GCP observations are used to to minimize the georeferencing alignment error, but they can't compensate for other factors such as an incorrect camera model estimate due to a sub-optimal flight path or excessive camera lens distortion (see the *Camera Calibration* chapter). If your results do not align perfectly, first check for mistakes in your GCP file. If there are

none, read and apply the concepts from the *Camera Calibration* chapter to improve results.

9

Flying Tips

Data collection is sometimes more art than science. There are however many useful guidelines you can follow to increase the quality and accuracy of your results. Follow these recommendations and you will achieve better results.

Fly Higher

This is probably the most useful tip. Once you know the target resolution you're aiming for, don't fly lower than you absolutely need. If you need an orthophoto at 5 cm / pixel resolution, don't fly at an elevation that gives you you a 1 cm / pixel resolution. Most flight planner apps will tell you what altitude you need to fly to achieve a target resolution. This is not only for saving space and processing time: the results will look better also! When you fly at a higher elevation more features can be matched across images during the structure from motion process (especially for areas with lots of trees or farmlands). Building rooftops will also look better due to the way texturing works. Finally, you can achieve greater image overlap and cover

a larger area in a single flight.

Fly on Overcast Days

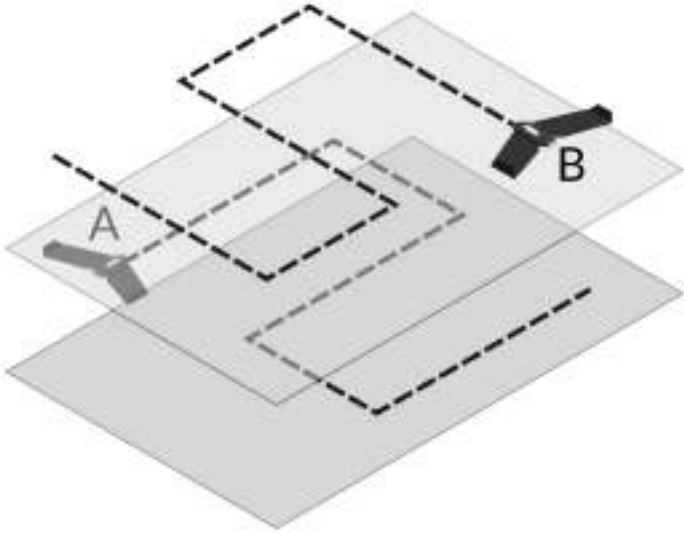
When possible, try to plan your flights when clouds are covering the sky. Clouds diffuse the rays from the sun, creating a soft light that reduces blur, shadows and produces more pleasing colors, increasing the overall quality of orthophotos and 3D models.

Fly Between 10am and 2pm

When the sun is directly above you, there will be less shadows and more uniform lighting.

Fly at Different Elevations and Capture Multiple Angles

OpenDroneMap can produce more accurate results when you capture images from different elevations, using both nadir (straight down) and non-nadir (at an angle) images. A cross pattern flown at two different altitudes with varying angles is much better than a single nadir-only pattern. When capturing angled images, be careful to set a value that avoids the horizon! Capturing the horizon can deteriorate results instead of improving them.



Flying cross-pattern at different elevations with B capturing nadir images and A capturing images at a slight angle (or vice-versa) is a better for camera calibration

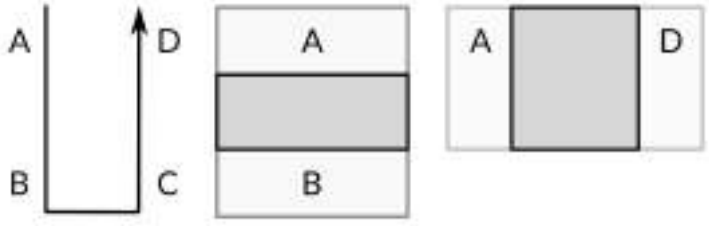
We discuss this in more details in the *Camera Calibration* chapter.

Fly on Calm Days

When it's windy your drone will have a harder time stabilizing the camera and can produce images that are blurrier. Fly when it's calm for better results.

Increase Overlap

More overlap increases the number of features that can be matched across images. Side overlap is more important than front overlap, so increase side overlap first, then increase front overlap.



Flight path with 4 images (left), front overlap (middle) and side overlap (right). Darker areas indicate the overlap area between images. More overlap is better

Set Drone to Hover While Taking Images

If your drone camera is equipped with a rolling shutter (most consumer grade drones), you should instruct the flight controller to bring the drone to a hover before taking a picture, as OpenDroneMap currently does not implement a rolling shutter correction model[40]. Doing this will increase the accuracy of the reconstruction[41]. This is not something to worry about if

[40] Rolling shutter correction: https://github.com/OpenDroneMap/ODM/issues/313

[41] Improved accuracy for rolling shutter cameras: https://www.pix4d.com/blog/rolling-shutter-correction

your drone camera is equipped with a global shutter.

Check Camera Settings

Make sure that image quality is set to high and auto focus is disabled. To do that, fly at your target altitude before mission start, set the focus, then disable auto focus.

This chapter concludes part II of the book. Most of what you need to know to start using OpenDroneMap efficiently has been covered. Congratulations for making it this far!

Part III delves into some more advanced topics, such as using OpenDroneMap from the command line, the mysteries of docker, processing humongous datasets at scale and an introduction to using Python for automating processing using the NodeODM API.

III

Advanced Usages

"Any sufficiently advanced technology is indistinguishable from magic."
- Arthur C. Clarke

10

The Command Line

In the first part of this book we explored how to use WebODM, the friendly graphical interface to ODM. WebODM hides some of the complexities of ODM, but this convenience comes at a cost. Here are a few things you cannot easily do in WebODM:

- Process tasks without performing web uploads.
- Inspect intermediate result files.
- Restart a task from an arbitrary point in the pipeline (WebODM supports task restarts, but only for a subset of them).
- Restart tasks without time expirations (WebODM can restart tasks only within 2 days of the task completing, unless the NodeODM settings are changed).

In this chapter we're going to leave the comforts of the user interface and dive into the realm of power users, using the command line to process the tasks directly with ODM.

This is not meant to be an exhaustive guide on using the command line for different operating systems. It's an overview of the basic commands you'll likely need to know for the

purpose of using ODM.

If you are already familiar with the command line, feel free to skip this chapter.

Command Line Basics

First, we should clarify that a *command line* is any application that allows a user to interact with it by means of typing commands. There are many varieties of command line applications and each operating system tends to have its own flavor(s). To provide a unified set of instructions for all three major operating systems (Windows, macOS, Linux), when we say *command line* we will always refer to *Bash* or one of its variants:

- Windows: use **Git Bash** (installed by following the instructions in the *Installing The Software* chapter). Do **not** use the Command Prompt or Powershell.
- macOS: use the **Terminal** app.
- Linux: Most distributions already use **Bash** by default, but in case your shell is different, just launch a bash shell by typing **bash** in your terminal.

Below are some commands you should be familiar with. Once you have opened a command line, try to type them:

- **ls -al:** list files and directories
- **cd <dir>**: change directory
- **pwd**: show me the current directory
- **cat <file>**: show the contents of file
- **head -n <lines> <file>:** show the first lines of file
- **tail -n <lines> <file>:** show the last lines of file
- **find . -name *.JPG:** find all JPG files in the current directory (and subdirectories)

- **whoami:** show the name of the current user
- **chown -R $(whoami):$(whoami) <directory>**: change ownership of directory to the current user and group
- **sudo <command>**: execute command with admin privileges (Linux and Mac only)

Adding the **–help** flag to any of the commands above will show a description of the command along with usage information.

```
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current
    directory by default).
Sort entries alphabetically if none of -cftuvSUX nor
    --sort is specified.
...
```

Paths in Bash are separated by forward slashes (as in **/c/Users/myuser**). This is sometimes a source of confusion for Windows users who are used to back slashes (as in **C:\Users\myuser**). Paths and filenames are also case-sensitive, so **/c/file** is different than **/c/FiLe**.

Pressing the **TAB** key while typing commands can autocomplete paths, for example if there's a directory in **/c/myVeryLongPathname** and we're currently in **/c**:

```
$ pwd
/c
$ cd myV<PRESS TAB>
$ cd myVeryLongPathname/ <-- auto completed
```

To navigate up one level from a directory you can reference the special ". ." (two dots) directory:

```
$ pwd
/c/dir
$ cd ..
$ pwd
/c
```

Note there's a space between **cd** and the two dots.

Using ODM

Now that we know how to navigate around directories using the **cd** command, place some images in a directory of your choice (e.g. **C:\odmbook\projects\test\images**) and navigate to:

```
$ cd /c/odmbook/
```

On Windows, if while running any of the commands below you get a *the input device is not a TTY. If you are using mintty, try prefixing the command with 'winpty'* error message you will need to type the following:

```
echo "alias docker='winpty docker'" >> ~/.bash_profile
```

then restart Git Bash before proceeding.

We can start processing the images with ODM by typing:

```
$ docker run -ti --rm -v /$(pwd)/projects/test:/
    datasets/code opendronemap/odm --project-path /
    datasets [options]
```

In place of **[options]** you can add any of the task options we've covered in the *Task Options in Depth* chapter. For example, to change the orthophoto resolution, generate a DSM and restart a task from the Multi-View Stereo step, we can type:

```
$ docker run -ti --rm -v /$(pwd)/projects/test:/
    datasets/code opendronemap/odm --project-path /
    datasets --orthophoto-resolution 2 --dsm --rerun-
    from mve
```

If you forget what options are available, you can simply run:

```
$ docker run -ti --rm opendronemap/odm --help
```

If the docker commands above looks ominous, don't worry. The next chapter contains a more in-depth discussion of docker.

Processed Files Owned By Root

On Linux and Mac you might notice that once your images are done processing the resulting files cannot be changed or deleted! This is a peculiarity of docker in which the output files are created within the docker container, and since the container runs with a root (admin) user, all files are also owned by root. To get back control to the files, simply run:

```
$ sudo chown -R $(whoami):$(whoami) /path/to/project
```

You can check who owns a directory by typing:

```
$ ls -al
drwxrwxrwx  2 foo bar 4.0K Jun 10 18:02 images
```

In the output above the **images** directory is owned by the user *foo* and group *bar*.

Add New Processing Nodes to WebODM

If you have a second computer, you can launch a new NodeODM node on that computer by typing:

```
docker run --rm -it -p 3000:3000 opendronemap/nodeodm
    -q 1 --token secret
```

The command asks docker to launch a new container using the **opendronemap/nodeodm** image (the latest version of NodeODM), using port 3000, setting a maximum number of concurrent tasks to 1 and protecting the node from unauthorized access using the password *secret*.

From WebODM you can then press the **Add New** button under the **Processing Nodes** menu. For the *hostname/IP* field type the IP of the computer. For the *port* field type *3000.* For the *token* field type *secret*. You can also add an optional *label* for

your node. Then press **Save**.

You should now be able to process multiple tasks in parallel using multiple machines.

Batch Geotagging of Images Using Exiftool

You can use exiftool[42] to add geolocation information to many images at once. The first step is to use a software such as LibreOffice Calc[43] to create a spreadsheet with the following columns.

```
SourceFile | GPSLatitude | GPSLongitude | GPSAltitude
    | GPSLatitudeRef | GPSLongitudeRef |
    GPSAltitudeRef
```

Then for each image you want to tag, add new rows as follows:

```
image1.JPG | 46.8425212 | -91.9942096 | 198.609 | N |
    W | 0
image2.JPG | 46.8424584 | -91.9938293 | 198.609 | N |
    W | 0
[...]
```

When you are done, export the spreadsheet to CSV format. Finally, type:

```
$ exiftool -GPSLatitude -GPSLongitude -GPSAltitude -
    GPSLatitudeRef -GPSLongitudeRef -GPSAltitudeRef -
```

[42] Exiftool: https://www.sno.phy.queensu.ca/~phil/exiftool/

[43] LibreOffice: https://www.libreoffice.org/

```
    csv="myfile.csv" -o geotagged_images/ input_images
    /
```

Images in the **input_images** directory will be geotagged and saved in the **geotagged_images** directory.

Further Readings

While not required for the purpose of using OpenDroneMap, users interested in expanding their skills with the command line should read the *Learn the Bash Command Line* tutorial available at https://ryanstutorials.net/linuxtutorial/. It contains a much more comprehensive introduction, including file manipulation, editing, pipes and process management.

11

Docker Essentials

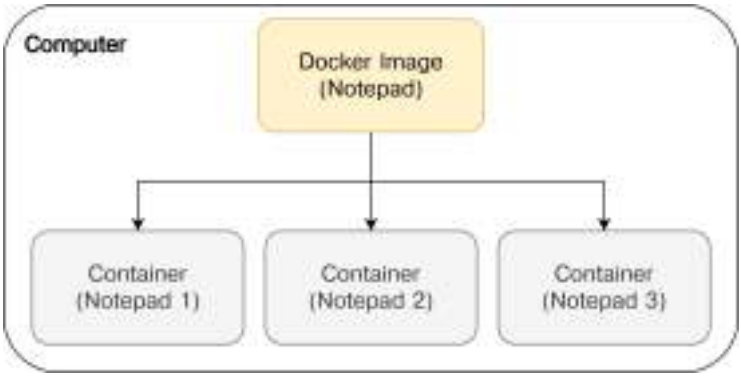
Most OpenDroneMap projects make extensive use of Docker as an installation and management tool. If you have ever had to deal with it, you probably know that it's confusing, seems to eat your disk space without reason and will unexpectedly error out with some cryptic message once in a while.

For the good and the bad however, Docker is here to stay. This chapter is here to help you understand its concepts and covers some basic commands as they are applicable to using OpenDroneMap more efficiently. I encourage readers to follow along and type the commands in a terminal while going through this chapter.

Docker Basics

Without getting too technical, docker is a tool that lets people wrap software and all their dependencies into *docker images*. Think of these *images* as programs. Each image can be used to start one or more *containers*. Think of containers as running instances of the program. To make an analogy, if Notepad is

a docker image, opening the Notepad program 3 times is the equivalent of running 3 containers.



Unlike normal programs however, docker images carry with them the entire operating system from which they are built! Among many other advantages, this allows us to run a program that was built for Linux under a different operating system. Of course there are disadvantages too (extra space, some overhead, etc.), but life is about trade-offs isn't it?

Docker images have names and they follow this convention:

```
username/imagename[:tag]
```

The **:tag** part of the name is optional and when omitted it defaults to **latest**. As an example, let's look at the full command we typically use to start an ODM process in Windows:

```
$ docker run -ti --rm -v //d/odmbook/project:/datasets
    /code opendronemap/odm --project-path /datasets
```

We are asking docker to start a new container using the **opendronemap/odm:latest** image.

- **-ti** (short for **-t -i**) asks docker to create a terminal and to keep it open (even if we close the window). Just remember to pass this, or no console output will be displayed.
- **–rm** asks to remove the container once it's done (by default containers are not destroyed after they are done, they are left in a *stopped* state).
- **-v** maps a volume (we'll discuss volumes below). Finally, we pass the **–project-path** option to the ODM process inside the container.

The **docker run** command follows this syntax:

```
docker run [docker options] [image name] [program
    options]
```

Managing Containers

We mentioned earlier that by default containers are not removed. A container begins in a running state and when it's done it switches to a stopped state. Let's see what happens when we forget to pass the **–rm** flag to a run command.

```
$ docker run -ti -v //d/odmbook/project:/datasets/code
     opendronemap/odm --project-path /datasets
```

When the command stops, we list all containers by issuing:

```
$ docker ps -a

CONTAINER ID        IMAGE
    COMMAND                  CREATED
    STATUS                       PORTS
    NAMES
ab89e4b71b65        opendronemap/odm
    "python /code/run.py..."   9 minutes ago
    Exited (1) 9 minutes
```

We can see that the **opendronemap/odm** container was not removed after exiting (because we didn't specify **–rm**). The **ps -a** command shows all containers, whether they are running or are stopped.

Each container has a unique identifier (or *hash*), which can be used to reference the container in other commands. In these examples, the container's hash is **ab89e4b71b65**.

To remove the container we just launched we can type:

```
$ docker rm ab89e4b71b65
```

If there are no conflicting hashes, you can also shorthand the hash by typing just one or more of the hash's beginning characters:

```
$ docker rm ab8
```

If you get the error message *Error response from daemon: You cannot remove a running container*, it's because only containers that are in stopped state can be removed. To stop a container issue:

```
$ docker stop ab8
```

We can verify that the container has been removed by issuing:

```
$ docker ps -a

CONTAINER ID        IMAGE
    COMMAND                  CREATED
    STATUS                        PORTS
    NAMES
```

You should be comfortable creating, listing and removing containers.

Two other important flags for the **run** command are **-d** and **-p:**

```
$ docker run -d -p 3000:3000 opendronemap/nodeodm

ab89e4b71b65
```

The **-d** flag can be used to start a container in the background. When a container is launched in the background the console does not *attach* to the container but returns immediately with the hash of the created container. This way you can launch

multiple containers without having to open new terminal windows. The **-p** flag exposes a network port from the container so that you can access it from the outside:

```
-p <port of computer>:<port inside container>
```

In the example above, the **opendronemap/nodeodm** image contains a web server that is configured to run on port 3000. By passing **-p 3000:3000** we ask docker to make the web server (running on port 3000 inside the container) available on port 3000 on our computer. This might seem confusing, but allows you to do cool things such as:

```
$ docker run -d -p 3000:3000 opendronemap/nodeodm
$ docker run -d -p 3001:3000 opendronemap/nodeodm
```

Which launches two separate NodeODM instances on ports 3000 and 3001, respectively.

Managing Images

Docker images can be created from a Dockerfile[44], which is a text file that tells docker how to create a particular image.

You don't necessarily need to build your own images. The OpenDroneMap developers have already built and published docker images for everyone to use. You can see what images are available by visiting https://hub.docker.com/

[44] Dockerfile Reference: https://docs.docker.com/engine/reference/builder/

r/opendronemap/.

There are some advantages to building your own images. You can make modifications to the software and in some cases (mostly just for ODM), you can gain a modest speed-up! For example, the public **opendronemap/odm** image has been built to support a large variety of computers (old and new), so certain optimizations that are available only on newer computers have been disabled. If you have a shiny new computer, building your own image can take advantage of those optimizations. To build your own image of ODM, first download ODM's source code, then navigate to the folder that contains the Dockerfile and type:

```
$ docker build -t myusername/odm .
```

After building it (which will take a while), use **myusername/odm** to run your image:

```
$ docker run --rm -ti [...] myusername/odm [...]
```

The first time you use **docker run**, Docker checks if the image you are requesting already exists on the computer. If it does, docker runs it. If it doesn't, docker attempts to download it from hub.docker.com. You can list the images that exist on your computer by using:

```
$ docker images

REPOSITORY                    TAG                 IMAGE
```

```
      ID            CREATED             SIZE
opendronemap/odm             latest
     f2275dac6ee1        22 hours ago        3.14GB
```

Notice that every image has a unique identifier (or *hash*) associated with it. With time you might decide that you do not need some images anymore and you can reclaim some disk space by removing older images. For example, let's remove the **opendronemap/odm** image:

```
$ docker rmi f22
```

When a new image becomes available on hub.docker.com (for example, when a new version of ODM becomes available) you need to manually specify that you'd like to download the new version using the **pull** command:

```
$ docker pull opendronemap/odm
```

Managing Volumes

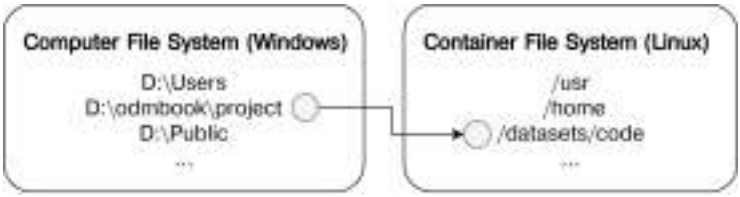
Up to this point, we've been using this **run** command quite a bit:

```
docker run -ti -v //d/odmbook/project:/datasets/code
    opendronemap/odm --project-path /datasets
```

But what does the **-v** //d/odmbook/project**:/datasets/code** part do? It creates a mapped volume:

```
-v <path on computer>:<path in container>
```

Containers are isolated environments. A container has its own internal directory structure, separate from that of the computer that is running the container. If we want to allow the container to access some of the files on our computer, we need to specifically allow it and we need to specify where we want to make our files accessible inside the container.



*Docker volume mapping. Files in **D:\odmbook\project** will be available in the container's **/datasets/code** path*

If you have ever setup a network drive, you map a network location to a drive so that you can access some path such as **G:\folder**, even though **folder** might be a folder residing on a remote computer. Docker volumes are just like network drives, but from your computer to containers.

If this still doesn't make sense, just accept as an universal fact that in order to get files in and out of a container you need to setup a mapped volume.

In our example we took the **D:\odmbook\project** folder

from our computer and made it available in the container's **/datasets/code** directory:

```
Computer --> Container
D:\odmbook\project\images\1.JPG --> /datasets/code/
    images/1.JPG
D:\odmbook\project\images\2.JPG --> /datasets/code/
    images/2.JPG
...
```

A quick note on the choice of **/datasets/code** as a path for the ODM container. **/datasets** is an arbitrary path, which we specify via **–project-path /datasets** when running ODM:

```
docker run -ti -v //d/odmbook/project:/datasets/code
    opendronemap/odm --project-path /datasets
```

code is the default *project name*. We can also explicitly specify a different project name and rewrite the command above as:

```
docker run -ti -v //d/odmbook/project:/datasets/
    example1 opendronemap/odm --project-path /datasets
     example1
```

But because the developers are lazy (the good kind of lazy) if a project name is omitted it defaults to **code**, so it's shorter to write.

A word of caution for Windows users: mapping volumes with

Docker Toolbox (Windows 7/8 and Windows 10 Home users) requires the path you want to map to be shared, or if you don't want to deal with extra configurations, make sure you only attempt to map folders that are within your user's home folder (C:\Users\youruser\)[45].

Some readers might have noticed that I prefixed the volume paths with two forward slashes instead of one. This is a quirk of Git Bash on Windows. You can omit the two forward slashes on macOS and Linux.

Docker-Compose Basics

While docker is used for running individual containers, docker-compose is used for running multiple containers. It's helpful to look at WebODM as an example of an application that uses docker-compose. WebODM, for example, is made of several components:

- A web application (opendronemap/webodm_webapp)
- A database (opendronemap/webodm_db)
- A message broker (library/redis)
- A processing engine (opendronemap/nodeodm or dronemapper/node-micmac)

You can see the parts that make WebODM by looking at the contents of the various **docker-compose*.yml** files from the WebODM source code. These .yml (YAML) files control the

[45] How to use a directory outside C:\Users with Docker Toolbox/Docker for Windows
: http://support.divio.com/local-development/docker/how-to-use-a-directory-outside-cusers-with-docker-toolboxdocker-for-windows

behavior of docker-compose. In Part II of this book when we launched WebODM via:

```
./webodm.sh start
```

all we did was to start docker-compose. In fact *webodm.sh* is mostly an interface to docker-compose. It provides a way to coordinate the launch of multiple containers, decide which containers should be launched before others, how storage should be configured and many other features. It's outside the scope of this book to have an exhaustive overview of docker-compose, but interested readers can find an exhaustive guide from the docker documentation[46]. I invite you to open *webodm.sh* as well as the various docker-compose*.yml files with a text editor to see how they are defined.

Docker-compose can combine multiple .yml files together. For example:

```
$ docker-compose -f docker-compose.yml -f docker-
    compose.nodeodm.yml up
```

reads the configuration from **docker-compose.yml**, then applies the configuration from **docker-compose.nodeodm.yml**, overriding or extending previous configurations. In this case **docker-compose.nodeodm.yml** (start WebODM using a NodeODM processing node) extends **docker-compose.yml** (just start WebODM, no processing nodes). The **up** command

[46] Docker-compose: https://docs.docker.com/compose/

asks to create and start all the containers defined in the configurations. Other useful commands include:

```
Stop the containers (but don't remove them)
$ docker-compose -f docker-compose.yml stop

Stop the containers and remove them
$ docker-compose -f docker-compose.yml down

Update all images from hub.docker.com
$ docker-compose -f docker-compose.yml pull
```

Managing Disk Space

Without the occasional cleaning, over time docker will happily eat up all of your disk space! This can happen when containers have not been removed, when many images have been downloaded, or when stray volumes have been abandoned. Docker is supposedly doing you a favor by not removing things automatically (what if you needed that image you built 3 years ago?). Luckily there's a useful command for cleaning things up:

```
$ docker system prune
```

Changing Entrypoint

Starting and stopping containers can take time. Also when processing fails, it might be difficult to inspect what went wrong. The author suggests a two step approach to running ODM datasets with docker:

```
$ docker run -ti -v //d/odmbook/project:/datasets/
    example1 --entrypoint bash opendronemap/odm
root@898747d1f3a8:/code# ./run.py --project-path /
    datasets example1
root@898747d1f3a8:/code# exit
$
```

By passing the **–entrypoint** bash flag we ask docker to ignore the default startup command of the container (*run.py*) and to run bash (a Linux shell) instead. The # confirms that we are inside the container, running a bash shell. Now we can run the ODM process by invoking *run.py* directly. The difference with this approach is that if something fails, we can more easily check for problems and restart the process. To exit the container we simply type **exit**.

Assigning Names To Containers

When you have lots of containers, nothing is more frustrating than remembering hash IDs. You can assign names when running a container and then reference that name in future commands.

```
$ docker run -d -p 3000:3000 --name mynode
    opendronemap/nodeodm
$ docker stop mynode
$ docker rm mynode
```

Jumping Into Existing Containers

Sometimes you'd really like to know what the heck is going on inside a container, without restarting it (you would lose its state). For example, I sometimes wonder if WebODM is really done downloading that long running task that seems to take forever.

```
$ docker ps

CONTAINER ID        IMAGE
    COMMAND                  CREATED
    STATUS              PORTS
    NAMES
aff69c390477        opendronemap/webodm_webapp   "/bin
    /bash -c 'chmod..."   46 hours ago        Up 3
    hours          0.0.0.0:8000->8000/tcp    webapp

$ docker exec -ti aff69c390477 bash

root@aff69c390477:/webodm# ls app/media/project/<id>/
    task/<id>/assets
all.zip    images.json    odm_orthophoto/    odm_dem/
    ...
```

The **exec** command allows you to execute a command from

a running container. In this instance we choose to execute a bash shell, which gives us a command prompt to inspect the contents of the container while it's running!

Making Changes Without Rebuilding Images

This is probably most useful to people that want to tinker with ODM's source code. You can modify ODM's source code using your IDE of choice, then create a container running the latest version of ODM and volume map the source code directory to the container's /code directory as follows (assuming the source code is in D:\ODM):

```
$ docker run -ti -v //d/ODM:/code -v //d/odmbook/
    project:/datasets/example1 --entrypoint bash
    opendronemap/odm
root@# bash configure.sh reinstall
# ./run.py --project-path /datasets example1
```

You can now edit the files in D:\ODM and the changes will be reflected inside the container!

Docker is a powerful tool, but can also be a bit intimidating. I hope this chapter removed some of the mysteries that surround it, hopefully while sparking some curiosity. While it's not necessary to become a master of docker to use any of the OpenDroneMap software, familiarity with it will certainly help. The docker documentation website[47] is a much more comprehensive resource for those wishing to expand their

[47] Docker Documentation: https://docs.docker.com/

knowledge.

12

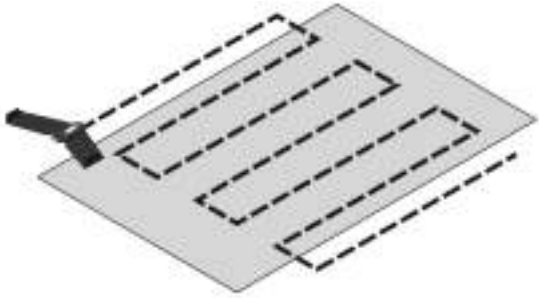
Camera Calibration

In order to create accurate reconstructions, the software needs to know the internal details of the cameras used for taking the photos. These details are referred to as *intrinsic camera parameters* and include values such as focal lengths, camera centers and distortion parameters. Reading the specification values from the manufacturer or looking this information from a database is not sufficient to get accurate values. Defects in the manufacturing process, vibrations and other factors can cause these values to vary, even between identical camera models.

You might have noticed that you can process datasets and obtain good results without performing any kind of camera calibration. This is because modern photogrammetry software (including OpenDroneMap) performs a kind of self-calibration directly from the input images. Self-calibration tends to work very well in OpenDroneMap, as long as:

1. Images are captured at different elevations.
2. Images are captured at varying angles (including both nadir and angled shots).

High overlap, near nadir images are not recommended[48]. This is important to remember, as most mission planning software will create exactly this type of pattern.



Typical flight path from mission planning software. Not great for self-calibration

This doesn't mean a person should never fly this pattern. It just means that when flying this pattern, people need to be aware that the reconstruction will not be as accurate. Over large areas, this error accumulates and typically results in a *doming* effect.

Point cloud exhibiting doming. The terrain appears arched instead of straight

[48] Camera Calibration Considerations for UAV Photogrammetry: https://www.isprs.org/tc2-symposium2018/images/ISPRS-Invited-Fraser.pdf

Doming is best cured by following best practices while collecting aerial imagery: flying at different elevations (maximize scale variation) and varying angles.

Unfortunately the luxury of capturing perfect images is not always available. Perhaps a dataset has already been captured and there's no opportunity for a retake, or the area is too large to cover in multiple passes.

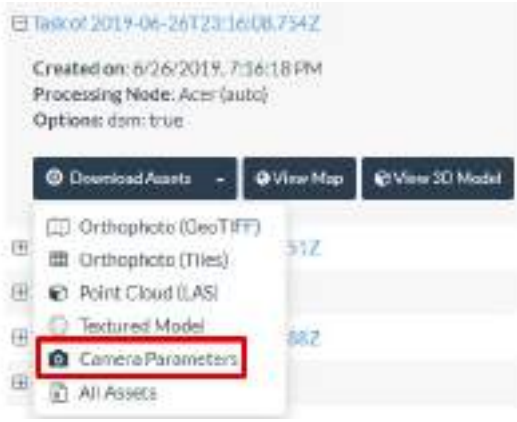
A few options are available to attempt to correct this problem.

Option 1: Use an Existing Camera Model

This is the preferred method and the most likely to yield improvements. With OpenDroneMap it's possible to process a good dataset (lots of overlap, varying elevations, different angles, etc.) captured with the same camera and reuse the camera parameters obtained from self-calibration on a different dataset.

WebODM Instructions

After processing a good dataset, simply click **Download Assets - Camera Parameters**.



This will save a **cameras.json** file. When creating a new task, from the task options set the **cameras** option by selecting the **cameras.json** file. The dataset will be processed using the camera parameters previously computed.

ODM Instructions

After processing is complete, a **cameras.json** file is always saved in the root directory of the project. To reuse camera parameters from an existing project, simply pass a path to a **cameras.json** file via **–cameras.**

For example, if **cameras.json** is stored in **D:\odm-book\projects\test**:

```
$ docker run -ti --rm -v //d/odmbook/projects/test:/
    datasets/code opendronemap/odm --project-path /
    datasets --cameras /datasets/code/cameras.json
```

Option 2: Generate a Camera Model From a Calibration Target

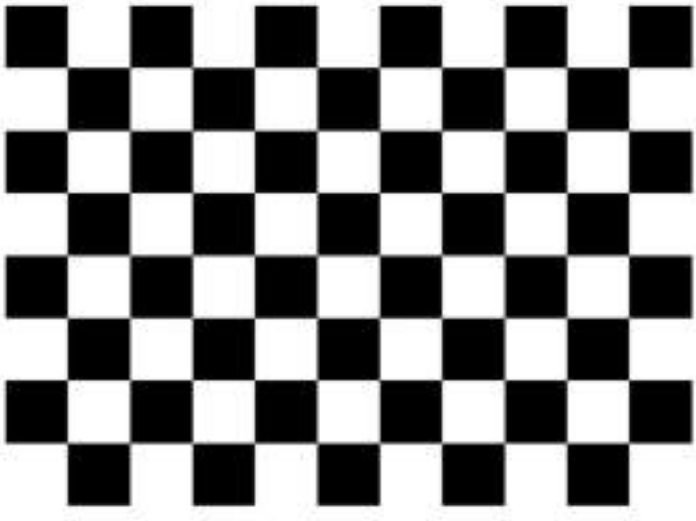
An alternative method of obtaining good camera parameters is from a calibration target. This is less likely to yield good results, since self-calibration methods tend to work better. However, it can be a useful method to know if it's not possible to capture a good dataset or if the self-calibration approach fails to generate good results.

There are three steps to this process:

1. Taking pictures of a calibration target (usually a chessboard pattern shot from different angles).
2. Extracting a camera profile from the the pictures.
3. Manually writing a **cameras.json** file.

Taking Pictures of a Calibration Target

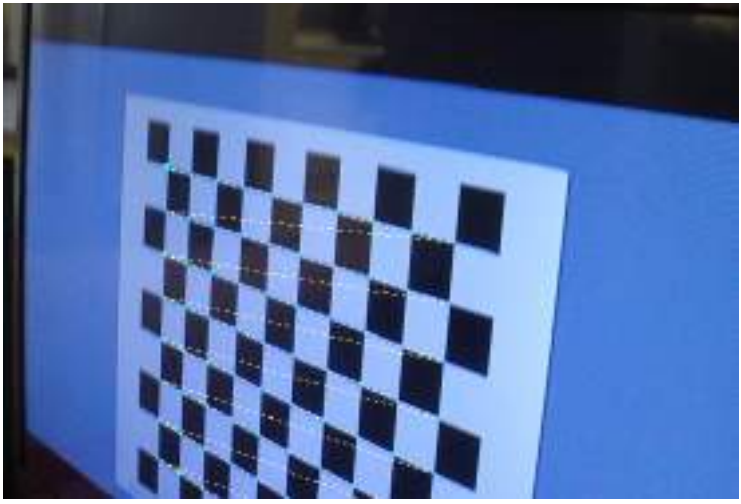
A calibration target is simply a known object that has recognizable features. A chessboard pattern is frequently used since the black/white contrast makes it easy to track via software.



Chessboard pattern

You can buy physical calibration targets from many retailers. Projecting the pattern onto a large surface, such as a TV or a large monitor also tends to work well. These patterns can be easily generated from several websites such as calib.io[49] .

[49] Calib.io calibration pattern generator: https://calib.io/pages/camera-calibration-pattern-generator



Chessboard calibration target displayed using a Chromecast on a TV

Go ahead and generate a 11x9 chessboard pattern, then load it onto your monitor. Once the target is in place, using the same camera you used for capturing a dataset, take between 5-9 pictures of the target at different angles. Try to retain the same camera focus as you do that (disable auto focus).

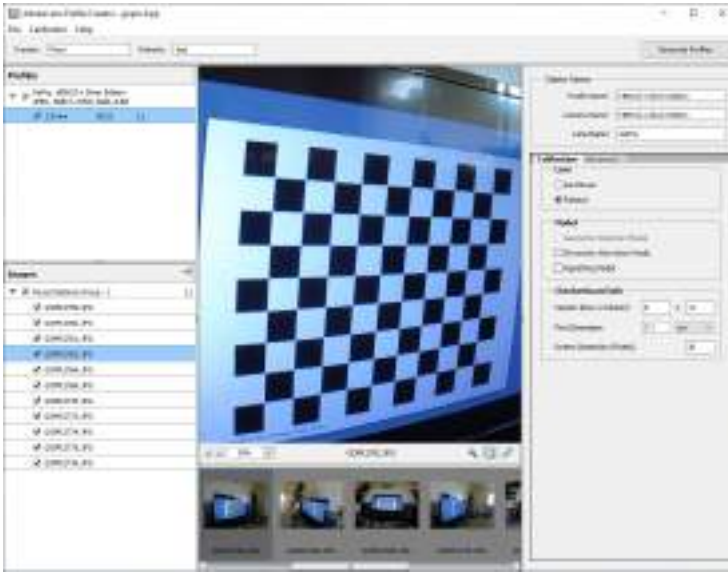
Extracting a Camera Profile

There are many programs for extracting a camera profile from the pictures of a calibration target. Because of it's simplicity, we'll use Adobe™ Lens Profile Creator. It's freely available for download for Windows and Mac from Adobe's site[50].

Once downloaded and installed, open the program and press

[50] Adobe™ Lens Profile Creator download page: https://helpx.adobe.com/photoshop/digital-negative.html#resources

File — Add Images to Project (CTRL+N). Select the images previously shot of the calibration target and press **Open**.



Lens Profile Creator

The images should have loaded. Now, turning the attention to the right side of the screen:

1. Type a **Lens Name** of your choice.
2. Select the appropriate type of lens of your camera. **Rectilinear** lenses are lenses where straight objects such as walls appear straight. **Fisheye** lenses make straight objects appear curved (see below for an example).
3. From the **Model** section check only **Geometric Distortion Model**. We are not going to need chromatic aberration and vignetting.
4. For **Checkerboard Info** type the number of rows and

columns of your checkerboard target.

5. For **Print dimension** and **screen dimension** type to the approximate size of a single square on the pattern in physical units and pixels respectively (or leave the defaults).
6. From the **Advanced** tab, for the **Rectilinear Lens Model** and **Fisheye Lens Model** choose **Two-Parameter Radial Distortion** (if creating a simplified *perspective* or *fisheye* camera model) or **Five-Parameter Radial Distortion** (if creating a more complex *brown* camera model).

Fisheye (top) vs. Rectilinear lenses (bottom).
Image courtesy of Ashley Pomeroy, An example of Panorama Tools, CC BY-SA 3.0

From the top right corner, press the **Generate Profiles** button. If the calibration is successful, you will be prompted to **Save a Profile (.lcp)** file. Select a folder and remember its location for the next step.

If you get an error, follow the recommendations on screen. Errors are typically solved by taking better pictures of the calibration target or adjusting the values in the **Checkerboard Info** section. For troubleshooting issues we recommend reading the Lens Profile Creator User Guide[51].

Manually Writing a cameras.json File

With a text editor of your choice, go ahead and open both an existing **cameras.json** file and the **.lcp** file you just generated. The goal is to grab the calibration values from the **.lcp** file and apply them to the **cameras.json** file. From the **.lcp** file look for a **rdf:parseType="resource"** entry:

```
<stCamera:PerspectiveModel rdf:parseType="Resource">
    <stCamera:Version>2</stCamera:Version>
    <stCamera:FocalLengthX>0.581968</stCamera:
        FocalLengthX>
    <stCamera:FocalLengthY>0.581968</stCamera:
        FocalLengthY>
    <stCamera:ImageXCenter>0.500000</stCamera:
        ImageXCenter>
    <stCamera:ImageYCenter>0.500000</stCamera:
        ImageYCenter>
    <stCamera:RadialDistortParam1>0.017524</stCamera:
        RadialDistortParam1>
```

[51] Adobe™ Lens Profile Creator User Guide: https://wwwimages2.adobe.com/content/dam/acom/en/products/photoshop/pdfs/lensprofile_creator_userguide.pdf

```
    <stCamera:RadialDistortParam2>-0.074705</stCamera:
        RadialDistortParam2>
    <stCamera:ResidualMeanError>0.000134</stCamera:
        ResidualMeanError>
    <stCamera:ResidualStandardDeviation>0.000227</
        stCamera:ResidualStandardDeviation>
</stCamera:PerspectiveModel>
```

Currently your **cameras.json** file might look something like:

```
{
    "v2 dji fc300s 4000 2250 perspective 0.5555": {
        "focal_prior": 0.5555555,
        "width": 4000,
        "k1": 0.0,
        "k2": 0.0,
        "k1_prior": 0.0,
        "k2_prior": 0.0,
        "projection_type": "perspective",
        "focal": 0.555555,
        "height": 2250
    }
}

(Your file will be different, this is an example)
```

Taking the focal and radial distortion values from the **.lcp** file, rewrite the **cameras.json** file as follows:

```
{
    "v2 dji fc300s 4000 2250 perspective 0.5555": {
        "focal_prior": 0.581968,
```

```
        "width": 4000,
        "k1": 0.017524,
        "k2": -0.074705,
        "k1_prior": 0.017524,
        "k2_prior": -0.074705,
        "projection_type": "perspective",
        "focal": 0.581968,
        "height": 2250
    }
}
```

The mapping of values between the two files is as follows:

```
.lcp -> cameras.json

FocalLengthX,FocalLengthY -> focal,focal_x,focal_y
RadialDistortParam1 -> k1
RadialDistortParam2 -> k2
RadialDistortParam3 -> k3
TangentialDistortParam1 -> p1
TangentialDistortParam1 -> p2
ImageXCenter,ImageYCenter -> c_x,c_y
```

Depending on the type of camera and the computed lens model some values might not be present (for example, there are no tangential parameters in a simple **perspective** camera model, but there are in a **brown** model).

The **projection_type** field is always one of **perspective**, **brown** (a more complex perspective model named after the work of Duane C. Brown and Alexander E. Conrady) or **fisheye**. Any **prior** field should be filled with the same value as its non-prior counterpart.

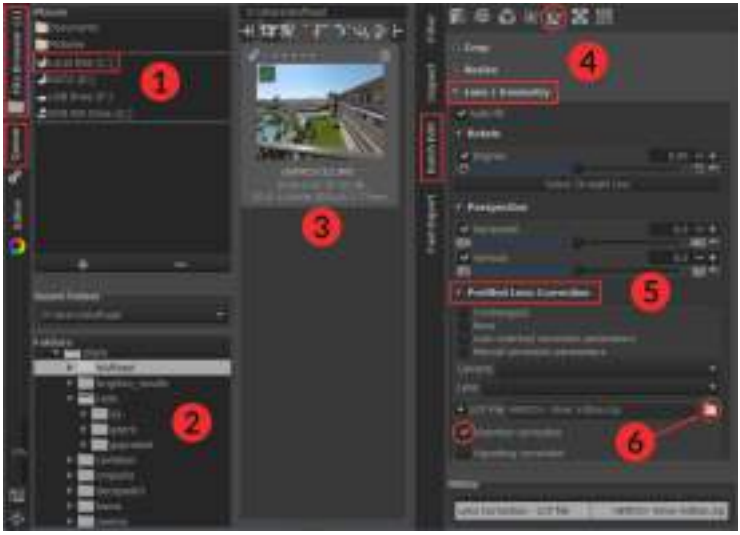
Once completed, simply save the new **cameras.json** file. It's ready to be used.

Bonus: Checking Your LCP File by Manually Removing Geometric Distortion

How do you know if your LCP file was computed correctly before processing a dataset? One way to check for obvious problems is to try to use it for manually removing lens distortion from images and see if the distortion goes away.

There are several programs that can read **.lcp** camera profiles. We'll use Rawtherapee, because it's a stable, free and open source program available on all major platforms. Installers can be downloaded directly from Rawtherapee's website[52]. Once installed, open it. Rawtherapee has lots of features and the user interface might look a bit intimidating at first. Use the reference image below to help navigate the next steps.

[52] Rawtherapee's download page: https://rawtherapee.com/downloads



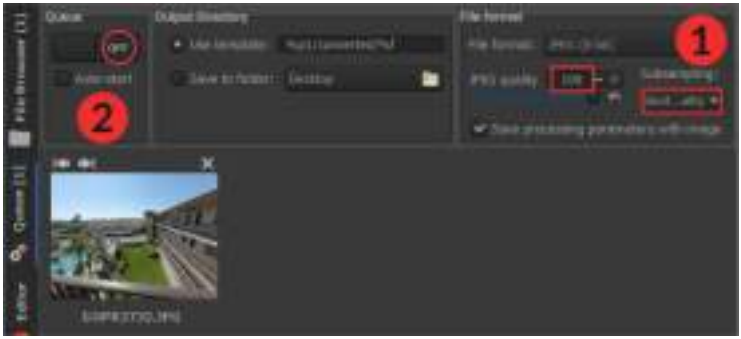
Rawtherapee's reference image

1. Open the **File Browser** tab and click the drive where the folder with your images is stored.
2. Navigate to the folder where your images are stored and **double-click** it.
3. Select all images by pressing **CTRL+A** (or right-click and **Select all**).
4. Open the **Batch Edit** tab, press the 3rd icon from the top right and expand **Lens / Geometry**.
5. Expand **Profiled Lens Correction**.
6. Press the folder icon to select the previously exported .lcp camera profile. After selecting it, make sure only **Distortion Correction** is checked.

You should see the images in the center panel (3) changing when you toggle the **Distortion Correction** checkbox. If you

don't see the images changing, triple-check that you performed the steps above in order. The changes might be subtle if your images do not have a lot of distortion. Once you are ready to export the images, **right-click** the center panel (3) and press **Put to queue** and open the **Queue** tab (1).

From the Queue tab:



1. Set the JPEG quality to **100** and subsamling to **Best Quality** (or values that are reasonable if you don't want to lose image quality).
2. Press the **Off** button to start processing the images.

Geometrically undistorted images will be saved in the **converted** folder within the directory of the input images.

Original (top) and undistorted (bottom) image

Congratulations! If you've followed the steps, you should now be comfortable with the process of using existing camera parameters, using a calibration target to generate camera parameters and the process of manually undistorting images. Keep the camera files you generate in a folder for future use. Just remember that vibrations and the environment could cause your camera parameters to change. So once in a while it's a good idea to capture new ones.

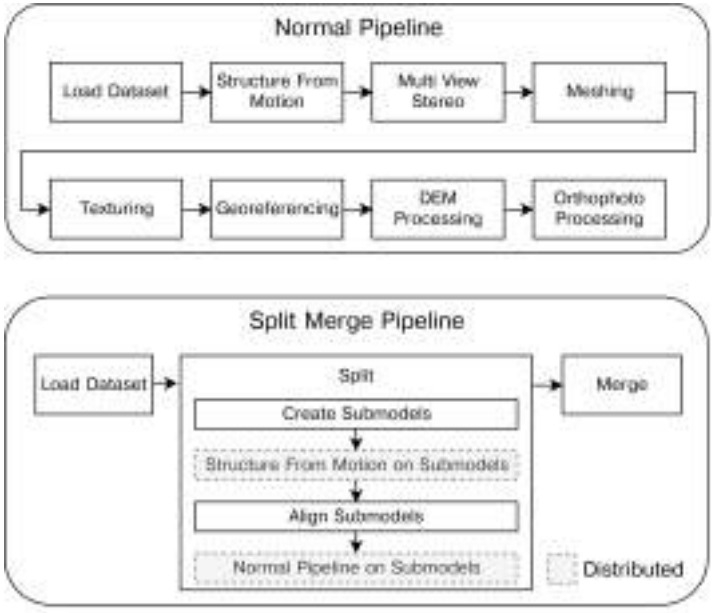
Camera calibration is not a bullet-proof method for obtaining better reconstructions. It is not a substitute for performing good data capture. But it's a step that can certainly help in many scenarios, especially when witnessing the *doming* effect.

13

Processing Large Datasets

ODM can use a lot of memory while processing. These memory requirements increase almost proportionally with the number of images and their resolution. Twice the images? Need twice the RAM! Of course at some point you might encounter a very large dataset made of dozens of thousands of images, but don't have 800GB of RAM just lying around.

This is were a nice feature called **split-merge** comes into play. This feature splits a large dataset into smaller, manageable parts called submodels that have some overlap between them. Each submodel is processed independently, either one at a time on a single machine (local split-merge) or even in parallel on multiple machines (distributed split-merge)! Once all submodels have been processed, the results are merged back together into a single consistent model. With this approach people can process much larger datasets using much less powerful computers. Enabling split-merge changes the ODM execution pipeline:



Split-merge pipeline. Step 2 and 4 of the split section can be performed in parallel on separate machines when using distributed split-merge

Split-Merge Options

split

Split-merge can be used either from the command line (ODM) or from WebODM and is turned on/off by setting the **split** option. Split-merge will be turned on anytime the following condition is true:

```
Split Option < Number of Images
```

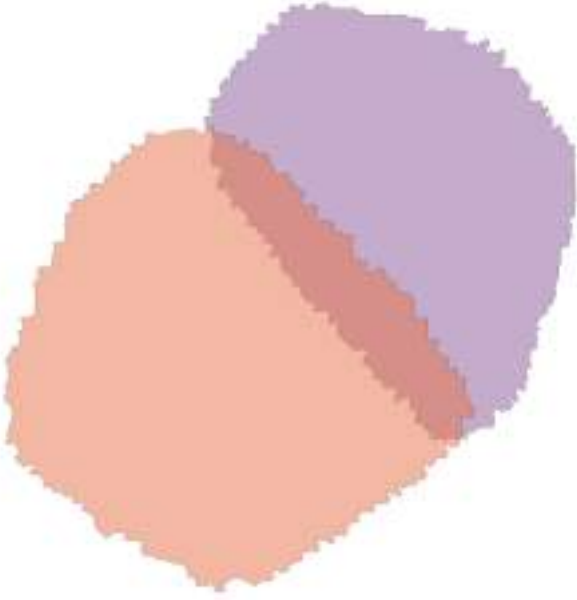
This option specifies the average number of images that should be included in each submodel. Note that this does not guarantee that your submodels will have this exact amount of images. In fact, some submodels might end up having twice as many images as others. It just means that the following expression holds true:

```
(Sum of submodel images) / (number of submodels) =~
    Split Value

In plain words: The sum of all submodel images divided
     by the number of submodels is approximately equal
     to the split value.
```

split-overlap

In order to align and merge results, each submodel must be reconstructed with a certain amount of overlap and redundancy with other submodels.



Overlap area between two submodels

The amount of overlap in meters is specified with this option. Datasets captured at higher altitudes should use a larger value, while those captured at lower altitudes can use lower values. More overlap will significantly slow down computation because of the redundancy of processed data, but can help achieve better model alignment during the merge step. If the resulting DEMs and point clouds from different submodels show big gaps in elevation, try increasing the overlap.

sm-cluster

This option enables distributed split-merge by specifying a URL to a ClusterODM instance. The process of setting up ClusterODM is described later in the *Distributed Split-Merge* section of this chapter.

merge

After splitting and computing each individual submodel, results need to be merged back together. By default all outputs (point clouds, DEMs and orthophotos) are merged. A user can use this option to specify that only a particular type of output should be merged. Valid options are:

- all
- pointcloud
- orthophoto
- dem

Local Split-Merge

It's really easy to use local split-merge. Simply pass the **split** option and **split-overlap** options:

```
$ docker run -ti --rm -v //d/odmbook/project:/datasets
    /code opendronemap/odm --project-path /datasets --
    split 100 --split-overlap 75
```

In the example above, submodels will be stored in **D:\odm-**

book\project\submodels while the merged results will be stored in the usual folders.

Each submodel folder (**submodels\submodel_xxxx**) is itself a valid OpenDroneMap project. So if a submodel fails to process, you can re-run the individual submodel to isolate potential issues by running:

```
$ docker run -ti --rm -v //d/odmbook/project:/datasets
    /code opendronemap/odm --project-path /datasets/
    code/submodels --orthophoto-cutline --dem-
    euclidean-map submodel_xxxx
```

The **orthophoto-cutline** and **dem-euclidean-map** options are always required for the purpose of merging DEMs and orthophotos. If an error occurs while processing one of the submodels, the process will stop. After fixing the problem, you can resume by re-running the initial command:

```
$ docker run -ti --rm -v //d/odmbook/project:/datasets
    /code opendronemap/odm --project-path /datasets --
    split 100 --split-overlap 75
```

Submodels that correctly processed the first time will not be re-processed, unless the **rerun-from split** option is passed. Execution will resume from the failed submodel.

If a task fails at the merge step, after all submodels have finished processing, you can check for issues and resume the merge step by typing:

```
$ docker run -ti --rm -v //d/odmbook/project:/datasets
    /code opendronemap/odm --project-path /datasets --
    split 100 --split-overlap 75 --rerun merge --merge
     all
```

Distributed Split-Merge

Distributed split-merge works just like local split-merge, but can be much faster, as submodels are processed independently in parallel by many machines. All that is required is to setup an instance of ClusterODM and link some NodeODM nodes to it.

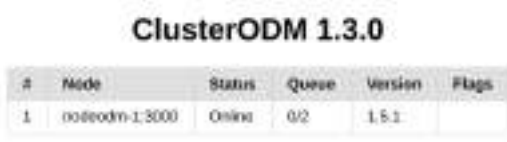
ClusterODM

ClusterODM is a program to connect together NodeODM API compatible nodes. It allows for tasks to be distributed across multiple nodes while taking into consideration factors such as maximum number of images, queue size and slot availability. It can also automatically spin up/down nodes based on demand using cloud computing providers (at the time of writing only Digital Ocean and Amazon Web Services are supported, but more are on the roadmap).

A ClusterODM looks like a normal NodeODM node from the outside and can operate with any tool that also works with NodeODM. To start ClusterODM, simply type:

```
$ git clone https://github.com/OpenDroneMap/ClusterODM
$ cd ClusterODM
$ docker-compose up
```

If you open your browser to http://localhost:10000 you should be greeted with:



ClusterODM web admin page

ClusterODM has setup a default NodeODM node on the same machine. Let's add some more nodes. If you have another computer with docker installed, you can run:

```
$ docker run -d -p 3000:3000 opendronemap/nodeodm
```

which will launch a new instance of NodeODM on port 3000. Now it's time to connect our new NodeODM node to ClusterODM. For that we'll need to use an archaic but functional tool called **telnet.** On Linux and macOS this tool is usually installed by default (if it isn't, Google how to install it). On Windows you usually need to enable it. From an elevated shell (right-click **Git Bash** and select **Run As Administrator**) type:

```
$ pkgmgr /iu:"TelnetClient"
```

Then restart the shell. Once **telnet** is available, type:

```
$ telnet localhost 8080
Connected to ...
Escape character is '^]'.
...
#
```

Typing **HELP** will show you all available commands. To add a NodeODM node use the **NODE ADD** command:

```
# NODE ADD ipofmachine 3000
# NODE LIST
1) nodeodm-1:3000 [online] [0/2] <version 1.5.1>
2) ipofmachine:3000 [online] [0/2] <version 1.5.1>
```

You'll need to change **ipofmachine** with the IP address or hostname of the computer running NodeODM.

To verify that things are working you can now open http://localhost:4000. If things are working you should see the NodeODM web interface. This means ClusterODM is properly forwarding requests to the NodeODM nodes.

You can connect as many nodes as you want to ClusterODM. If you have a variety of computers, some more powerful than others, you can start NodeODM instances with the **max_images** option:

```
$ docker run -d -p 3000:3000 opendronemap/nodeodm --
    max_images 300
```

This way NodeODM will be instructed to process datasets only up to 300 images. ClusterODM will take this factor into consideration when processing new tasks and will forward the task to the first machine capable of handling it.

A useful command is **NODE LOCK** which will prevent a certain node from being used by ClusterODM:

```
# NODE LOCK 1
# NODE LIST
1) nodeodm-1:3000 [online] [0/2] <version 1.5.1> [L]
2) ipofmachine:3000 [online] [0/2] <version 1.5.1>
```

In the setup above, all tasks will be forwarded to machine 2. Nodes that are locked can be unlocked with **NODE UNLOCK**.

ClusterODM can also use cloud computing providers such as Digital Ocean to spin up/down computing instances on demand. Because this feature is relatively new, check the ClusterODM README[53] for the latest instructions on how to configure it.

Distributed Run

Now that ClusterODM is setup, running distributed split-merge is the same as running local split-merge, except that we pass an additional **sm-cluster** option that specifies the URL to a ClusterODM instance.

```
$ docker run -ti --rm -v //d/odmbook/project:/datasets
    /code opendronemap/odm --project-path /datasets --
    split 100 --split-overlap 75 --sm-cluster http://
```

[53] ClusterODM: https://github.com/OpenDroneMap/ClusterODM

```
    ipofclusterodm:4000
```

There's a small gotcha, if you are running this command on the same machine as ClusterODM: **ipofclusterodm** cannot be **localhost**, because **localhost** refers to the **opendronemap/odm** container and not your machine. To find out the correct IP address, run:

```
$ docker ps

CONTAINER ID        IMAGE
    COMMAND                  CREATED
    STATUS              PORTS
    NAMES
5a86a35fe643        opendronemap/clusterodm
                "/usr/bin/nodejs /va..."   4 days ago
              Up 2 hours          0.0.0.0:4000->3000/
    tcp   node-odm-1

$ docker inspect -f "{{range .NetworkSettings.Networks
    }}{{.IPAddress}}{{end}}" 5a86

172.23.0.5
```

and use that IP instead. In the example above, we can then use **–sm-cluster http://172.23.0.5:4000**.

Using Image Groups and GCPs

You can control how a dataset should be split by placing a **image_groups.txt** file in your project folder. For example, if your images are in **D:\odmbook\project\images**, you can create a **D:\odmbook\project\image_groups.txt** file with the following content:

```
1.JPG A
2.JPG A
3.JPG B
4.JPG B
5.JPG C
[...]
```

where the items on the left are image names and items on the right represent submodel groups. If you run out of letters simply use *AA*, *BB*, etc. The file is case-sensitive so uppercase and lowercase letters are treated differently.

If this file is detected during split-merge, the **split** value will be ignored and the dataset will be split according to the rules specified in the **image_groups.txt** file. You will still need to pass **split** to enable the split-merge workflow.

At the time of writing, image groups are currently not supported in WebODM.

Image groups are important when using GCPs. GCPs can be used with split-merge, but care should be exercised to make sure there are at least 3 GCPs that fall within each submodel. If less than 3 GCPs are present in a submodel, the submodel will be aligned with the GPS information from EXIF data and

by looking at the position of other submodels, but won't be as accurate.

You can use GCPs with split-merge just like you would use GCPs for a normal run.

Limitations

With split-merge you get point clouds, DEMs and orthophotos, but not 3D textured meshes. You can still access the individual 3D textured meshes from each submodel, but no *global* 3D textured mesh is generated.

All of the problems listed in the *Camera Calibration* chapter are magnified when using split-merge. It's imperative to follow the best practices for data acquisition and if possible, to use a well calibrated camera model.

While the longest parts of processing can be parallelized in distributed split-merge, a dataset still needs to be split, aligned and merged on a single machine. The merge step in particular, for really large datasets, could still make the machine run out of memory. Choose **orthophoto-resolution**, **dem-resolution** and **depthmap-resolution** parameters conservatively. Use the most powerful machine you have available to start a distributed split-merge task.

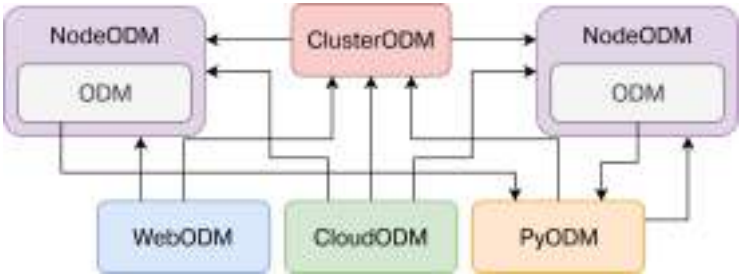
14

The NodeODM API

ODM is a processing engine and WebODM is a friendly user interface. NodeODM[54] was historically built to allow WebODM to communicate with ODM over a network. Today NodeODM has expanded its role and is the glue that binds together the entire suite of OpenDroneMap projects. Each project understands the API that NodeODM defines. When we say *NodeODM* we are referring to the reference implementation of the NodeODM API available at https://github.com/OpenDroneMap/NodeODM.

At its core, the API defines ways to easily create new tasks, manage such tasks (cancel, delete, restart), download results and query status information.

[54] Node is a reference to Node.js, the language NodeODM is written in.



OpenDroneMap projects use the NodeODM API to communicate with each other

It's interesting to note the NodeODM API has also been adopted by projects outside the OpenDroneMap ecosystem. For example, the NodeMICMAC project[55] has successfully implemented an aerial image processing pipeline using the open source MicMac photogrammetry engine as an alternative to ODM. This move has allowed the project to re-use and work in sync with all of the other tools within the OpenDroneMap ecosystem.

In this chapter we'll learn to manually launch a NodeODM instance, explore its web interface and do some manual interaction with the API using cURL, a program that allows us to make web requests. Familiarity with the NodeODM API can give users a better understanding of the network interactions that happen between the various projects.

The API has different versions and tries to be backward compatible with previous implementations whenever a new version is released. The most up-to-date specification is available online[56], while a copy of the latest specification at

[55] NodeMICMAC: https://github.com/dronemapper-io/NodeMICMAC/

[56] NodeODM specification: https://github.com/OpenDroneMap/NodeODM/blob/master/docs/index.adoc

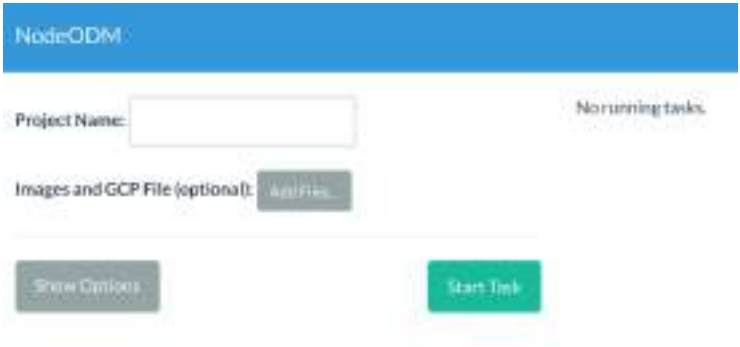
the time of writing is reported for reference at the end of this chapter.

Launching a NodeODM Instance

Starting NodeODM is as simple as running:

```
$ docker run -d -p 3000:3000 opendronemap/nodeodm
```

Opening a web browser to http://localhost:3000 loads the NodeODM interface. The interface is minimal on purpose as it doesn't try to compete with the more advanced functionalities of WebODM. It's mostly a test bed for testing the API functions.



NodeODM's web interface

From the web interface you can upload images, set task options, monitor and manage tasks, retrieve console outputs and download results. Not all functions of the API are exposed through the web interface, but most are.

NodeODM Configuration

You can pass several options while launching NodeODM. The full list is available by running:

```
$ docker run --rm -ti opendronemap/nodeodm --help

Usage: node index.js [options]

Options:
        --config <path> Path to the configuration file
             (default: config-default.json)
        -p, --port <number>     Port to bind the
            server to (default: 3000)
        --odm_path <path>       Path to OpenDroneMap's
             code     (default: /code)
        --log_level <logLevel>  Set log level
            verbosity (default: info)
        -d, --deamonize         Set process to run as
            a deamon
        -q, --parallel_queue_processing <number>
            Number of simultaneous processing tasks (
            default: 2)
        --cleanup_tasks_after <number> Number of
            minutes that elapse before deleting
            finished and canceled tasks (default:
            2880)
        --cleanup_uploads_after <number> Number of
            minutes that elapse before deleting
            unfinished uploads. Set this value to the
            maximum time you expect a dataset to be
            uploaded. (default: 2880)
        --test Enable test mode. In test mode, no
            commands are sent to OpenDroneMap. This
            can be useful during development or
            testing (default: false)
```

```
--test_skip_orthophotos If test mode is
    enabled, skip orthophoto results when
    generating assets. (default: false)
--test_skip_dems        If test mode is
    enabled, skip dems results when generating
     assets. (default: false)
--test_drop_uploads     If test mode is
    enabled, drop /task/new/upload requests
    with 50% probability. (default: false)
--test_fail_tasks       If test mode is
    enabled, mark tasks as failed. (default:
    false)
--test_seconds  If test mode is enabled, sleep
     these many seconds before finishing
    processing a test task. (default: 0)
--powercycle    When set, the application
    exits immediately after powering up.
    Useful for testing launch and compilation
    issues.
--token <token> Sets a token that needs to be
    passed for every request. This can be used
     to limit access to the node only to token
     holders. (default: none)
--max_images <number>   Specify the maximum
    number of images that this processing node
     supports. (default: unlimited)
--webhook <url> Specify a POST URL endpoint to
     be invoked when a task completes
    processing (default: none)
--s3_endpoint <url>     Specify a S3 endpoint
    (for example, nyc3.digitaloceanspaces.com)
     to upload completed task results to. (
    default: do not upload to S3)
--s3_bucket <bucket>    Specify a S3 bucket
    name where to upload completed task
    results to. (default: none)
```

```
        --s3_access_key <key>   S3 access key,
            required if --s3_endpoint is set. (default
            : none)
        --s3_secret_key <secret>        S3 secret key,
             required if --s3_endpoint is set. (
            default: none)
        --s3_signature_version <version>        S3
            signature version. (default: 4)
        --s3_upload_everything  Upload all task
            results to S3. (default: upload only .zip
            archive and orthophoto)
        --max_concurrency   <number>    Place a cap on
             the max-concurrency option to use for
            each task. (default: no limit)
Log Levels:
error | debug | info | verbose | debug | silly
```

The most important ones are described below.

- **-q** controls how many tasks can be processed in parallel.
- **–max_images** puts a limit on the maximum number of images that the node can accept.
- **–token** sets an access key to authenticate users. This is useful if you are setting up NodeODM on a public server and want to restrict access.
- **–cleanup_tasks_after** allows you to shorten/lengthen the time that it takes for completed tasks to be automatically removed from disk (expressed in number of seconds). If you are running out of space, this option can help free disk space more quickly.

Using the API with cURL

In the following examples we will use the curl program to interact directly with the API to upload some images, query task status and download results. Curl can be downloaded for any platform from https://curl.haxx.se/download.html.

There's almost no practical reason for using cURL to interact with NodeODM, aside from the learning experience and for the occasional troubleshooting. If you need to interact with NodeODM from the command line, CloudODM[57] is probably a much better tool for the job.

But since we are learning, cURL will be used here.

Create a New Task

If you have some images in **D:\odmbook\project\images** you can issue:

```
$ pwd
/d/odmbook/project

$ curl -F images=@images/DJI_0018.JPG -F images=
    @images/DJI_0019.JPG -F name=Test -X POST http://
    localhost:3000/task/new

{"uuid":"c99f32a8-d3b2-48d2-adfb-6c8e14e405e3"}
```

You can use more images by adding more **-F images** parameters.

With this command we created a new task and named it

[57] CloudODM: https://github.com/OpenDroneMap/CloudODM/

Test. The *name* field was passed via a FormData parameter. Some API functions accept parameters via Query and Body parameters also. We'll look at examples of both later. The API communicates via a format called JSON (**J**ava**S**cript **O**bject **N**otation). JSON is a simple human readable format. For example, if an error occurs, the program will output:

```
{"error": "description of the error"}
```

In the case of a successful call to **/task/new** we received back the task ID (identifier) that was created.

Query Task Information

After a task is created, we can query its status by referencing its ID using **/task/<uuid>/info**:

```
$ curl http://localhost:3000/task/c99f32a8-d3b2-48d2-
    adfb-6c8e14e405e3/info

{"uuid":"131ee2a5-9757-46e9-8491-81d2d4558680","name
    ":"Test","dateCreated":1560702697082,"
    processingTime":17655,"status":{"code":30,"
    errorMessage":"Process exited with code 1"},"
    options":[],"imagesCount":2,"progress":100}
```

We can also display the entire task output by invoking:

```
$ curl http://localhost:3000/task/c99f32a8-d3b2-48d2-
    adfb-6c8e14e405e3/output

["[INFO]    Initializing OpenDroneMap app - Sun Jun 16
     16:28:16  2019","[DEBUG]   ==============","[
    DEBUG]   build_overviews: False","[DEBUG]   crop:
    3","[DEBUG]   dem_decimation: 1","[DEBUG]
    dem_euclidean_map: False","[DEBUG]
    dem_gapfill_steps: 3","[DEBUG]   dem_resolution:
    5","[DEBUG]   depthmap_resolution: 640", ...
```

Or use the optional **?line=** Query parameter to retrieve the last 2 lines of output:

```
$ curl http://localhost:3000/task/c99f32a8-d3b2-48d2-
    adfb-6c8e14e405e3/output?line=-2

["if bbox is None: raise Exception(\"Cannot compute
    bounds for %s (bbox key missing)\" %
    input_point_cloud)","Exception: Cannot compute
    bounds for /var/www/data/c99f32a8-d3b2-48d2-adfb-6
    c8e14e405e3/odm_filterpoints/point_cloud.ply (bbox
     key missing)"]
```

Query parameters are passed directly via URLs.

So far we saw examples of passing Query and FormData parameters. Next we'll look at passing Body parameters.

Remove a Task

We can invoke **/task/delete** to delete a task:

```
$ curl -d uuid=c99f32a8-d3b2-48d2-adfb-6c8e14e405e3
    http://localhost:3000/task/remove

{"success":true}
```

Note the task ID is passed via a *uuid* Body parameter.

API Specification

Version: 1.5.3

GET /auth/info

Description: Retrieves login information for this node.
Response:

HTTP Code	Description	Schema
200	LoginInformation	Response 200

Response 200

Name	Description	Schema
loginUrl *required*	URL (absolute or relative) where to make a POST request to obtain a token, or null if login is disabled.	string
message *required*	Message to be displayed to the user prior to login/registration. This might include instructions on how to register or login, or to communicate that authentication is not available.	string
registerUrl *required*	URL (absolute or relative) where to make a POST request to register a user, or null if registration is disabled.	string

POST /auth/login

Description: Retrieve a token from a username/password pair.
Parameters:

Type	Name	Description	Schema	Default
Body	**password** *required*	Password	string	
Body	**username** *required*	Username	string	

Responses:

HTTP Code	Description	Schema
200	Login Succeeded	Response 200
default	Error	Error

Response 200

Name	Description	Schema
token *required*	Token to be passed as a query parameter to other API calls.	string

POST /auth/register

Description: Register a new username/password.
Parameters:

Type	Name	Description	Schema	Default
Body	**password** *required*	Password	string	
Body	**username** *required*	Username	string	

Responses:

HTTP Code	Description	Schema
200	Response	Response

GET /info

Description: Retrieves information about this node.
Parameters:

Type	Name	Description	Schema	Default
Query	**token** *optional*	Token required for authentication (when authentication is required)	string	

Responses:

HTTP Code	Description	Schema
200	Response	Response

Response 200

Name	Description	Schema
availableMemory optional	Amount of RAM available in bytes	integer
cpuCores optional	Number of CPU cores (virtual)	integer
engine required	Lowercase identifier of processing engine	string
enginesVersion required	Current version of processing engine	string
maxImages required	Maximum number of images allowed for new tasks or null if there's no limit.	integer
maxParallelTasks optional	Maximum number of tasks that can be processed simultaneously	integer
taskQueueCount required	Number of tasks currently being processed or waiting to be processed	integer
totalMemory optional	Amount of total RAM in the system in bytes	integer
version required	Current API version	string

GET /options

Description: Retrieves the command line options that can be passed to process a task.
Parameters:

Type	Name	Description	Schema	Default
Query	**token** *optional*	Token required for authentication (when authentication is required).	string	

Responses:

HTTP Code	Description	Schema
200	Options	< Option > array

Option

Name	Description	Schema
domain *required*	Valid range of values (for example, "positive integer" or "float > 0.0")	string
help *required*	Description of what this option does	string
name *required*	Command line option (exactly as it is passed to the OpenDroneMap process, minus the leading '--')	string
type *required*	Datatype of the value of this option	enum (int, float, string, bool)
value *required*	Default value of this option	string

POST /task/cancel

Description: Cancels a task (stops its execution, or prevents it from being executed).
Parameters:

Type	Name	Description	Schema	Default
Query	**token** *optional*	Token required for authentication (when authentication is required).	string	
Body	**uuid** *required*	UUID of the task.	string	

Responses:

HTTP Code	Description	Schema
200	Command Received	Response

POST /task/new

Description: Creates a new task and places it at the end of the processing queue. For uploading really large tasks, see **/task/new/init** instead.
Parameters:

Type	Name	Description	Schema	Default
Header	**set-uuid** *optional*	An optional UUID string that will be used as UUID for this task instead of generating a random one.	string	
Query	**token** *optional*	Token required for authentication (when authentication is required)	string	
FormData	**dateCreated** *optional*	An optional timestamp overriding the default creation date of the task.	integer	
FormData	**images** *optional*	Images to process, plus an optional GCP file (*.txt) and/or an optional seed file (seed.zip). If included, the GCP file should have .txt extension. If included, the seed archive pre-populates the task directory with its contents.	file	
FormData	**name** *optional*	An optional name to be associated with the task	string	

FormData	**options** *optional*	Serialized JSON string of the options to use for processing, as an array of the format: [{name: option1, value: value1}, {name: option2, value: value2}, ...]. For example, [{"name":"cmvs-maxImages","value":"500"}, {"name":"time","value":true}]. For a list of all options, call /options	string
FormData	**outputs** *optional*	An optional serialized JSON string of paths relative to the project directory that should be included in the all.zip result file, overriding the default behavior.	string
FormData	**skipPostProcessing** *optional*	When set, skips generation of map tiles, derivate assets, point cloud tiles.	boolean
FormData	**webhook** *optional*	Optional URL to call when processing has ended (either successfully or unsuccessfully).	string
FormData	**zipurl** *optional*	URL of the zip file containing the images to process, plus an optional GCP file. If included, the GCP file should have .txt extension	string

Responses:

HTTP Code	Description	Schema
200	Success	Response 200
default	Error	Error

Response 200

Name	Description	Schema
uuid *required*	UUID of the newly created task	string

POST /task/new/commit/{uuid}

Description: Creates a new task for which images have been uploaded via **/task/new/upload**.
Parameters:

Type	Name	Description	Schema	Default
Path	**uuid** *required*	UUID of the task	string	
Query	**token** *optional*	Token required for authentication (when authentication is required)	string	

Responses:

HTTP Code	Description	Schema
200	Success	Response 200
default	Error	Error

Response 200

Name	Description	Schema
uuid *required*	UUID of the newly created task	string

POST /task/new/init

Description: Initialize the upload of a new task. If successful, a user can start uploading files via **/task/new/upload**. The task will not start until **/task/new/commit** is called.
Parameters:

Type	Name	Description	Schema	Default
Header	**set-uuid** *optional*	An optional UUID string that will be used as UUID for this task instead of generating a random one.	string	
Query	**token** *optional*	Token required for authentication (when authentication is required)	string	
FormData	**dateCreated** *optional*	An optional timestamp overriding the default creation date of the task.	integer	
FormData	**name** *optional*	An optional name to be associated with the task	string	
FormData	**options** *optional*	Serialized JSON string of the options to use for processing, as an array of the format: [{name: option1, value: value1}, {name: option2, value: value2}, ...]. For example, [{"name":"cmvs-maxImages","value":"500"}, {"name":"time","value":true}]. For a list of all options, call /options	string	
FormData	**outputs** *optional*	An optional serialized JSON string of paths relative to the project directory that should be included in the all.zip result file, overriding the default behavior.	string	
FormData	**skipPostProcessing** *optional*	When set, skips generation of map tiles, derivate assets, point cloud tiles.	boolean	
FormData	**webhook** *optional*	Optional URL to call when processing has ended (either successfully or unsuccessfully).	string	

Responses:

HTTP Code	Description	Schema
200	Success	Response 200
default	Error	Error

Response 200

Name	Description	Schema
uuid *required*	UUID of the newly created task	string

POST /task/new/upload/{uuid}

Description: Adds one or more files to the task created via **/task/new/init**. It does not start the task. To start the task, call **/task/new/commit**.
Parameters:

Type	Name	Description	Schema	Default
Path	**uuid** required	UUID of the task	string	
Query	**token** optional	Token required for authentication (when authentication is required)	string	
FormData	**images** required	Images to process, plus an optional GCP file (*.txt) and/or an optional seed file (seed.zip). If included, the GCP file should have .txt extension. If included, the seed archive pre-populates the task directory with its contents.	file	

Responses:

HTTP Code	Description	Schema
200	File Received	Response
default	Error	Error

POST /task/remove

Description: Removes a task and deletes all of its assets.
Parameters:

Type	Name	Description	Schema	Default
Query	**token** *optional*	Token required for authentication (when authentication is required).	string	
Body	**uuid** *required*	UUID of the task.	string	

Responses:

HTTP Code	Description	Schema
200	Command Received	Response

POST /task/restart

Description: Restarts a task that was previously canceled, that had failed to process or that successfully completed.
Parameters:

Type	Name	Description	Schema	Default
Query	**token** optional	Token required for authentication (when authentication is required).	string	
Body	**options** optional	Serialized JSON string of the options to use for processing, as an array of the format: [{name: option1, value: value1}, {name: option2, value: value2}, ...]. For example, [{"name":"cmvs-maxImages","value":"500"}, {"name":"time","value":true}]. For a list of all options, call /options. Overrides the previous options set for this task.	string	
Body	**uuid** required	UUID of the task	string	

Responses:

HTTP Code	Description	Schema
200	Command Received	Response

GET /task/{uuid}/download/{asset}

Description: Retrieves an asset (the output of OpenDroneMap's processing) associated with a task.
Parameters:

Type	Name	Description	Schema	Default
Path	**asset** *required*	Type of asset to download. Use "all.zip" for zip file containing all assets.	enum (all.zip, orthophoto.tif)	
Path	**uuid** *required*	UUID of the task	string	
Query	**token** optional	Token required for authentication (when authentication is required)	string	

Responses:

HTTP Code	Description	Schema
200	Asset File	file
default	Error message	Error

GET /task/{uuid}/info

Description: Gets information about this task, such as name, creation date, processing time, status, command line options and number of images being processed.
Parameters:

Type	Name	Description	Schema	Default
Path	**uuid** required	UUID of the task.	string	
Query	**token** optional	Token required for authentication (when authentication is required).	string	
Query	**with_output** optional	Optionally retrieve the console output for this task. The parameter specifies the line number that the console output should be truncated from. For example, passing a value of 100 will retrieve the console output starting from line 100. By default no console output is added to the response.	integer	"0"

Responses:

HTTP Code	Description	Schema
200	Task Information	TaskInfo
default	Error	Error

TaskInfo

Name	Description	Schema
dateCreated *required*	Timestamp	integer
imagesCount *required*	Number of images	integer
name *required*	Name	string
options *required*	List of options used to process this task	< options > array
output *optional*	Console output for the task (only if requested via ?output=<linenum>)	< string > array
processingTime *required*	Milliseconds that have elapsed since the task started being processed.	integer
status *required*		status
uuid *required*	UUID	string

options

Name	Description	Schema
name *required*	Option name (example: "odm_meshing-octreeDepth")	string
value *required*	Value (example: 9)	string

status

Name	Description	Schema
code *required*	Status code (10 = QUEUED, 20 = RUNNING, 30 = FAILED, 40 = COMPLETED, 50 = CANCELED)	integer

GET /task/{uuid}/output

Description: Retrieves the console output of the OpenDroneMap's process. Useful for monitoring execution and to provide updates to the user.

Parameters:

Type	Name	Description	Schema	Default
Path	**uuid** *required*	UUID of the task	string	
Query	**line** optional	Optional line number that the console output should be truncated from. For example, passing a value of 100 will retrieve the console output starting from line 100. Defaults to 0 (retrieve all console output).	integer	"0"
Query	**token** optional	Token required for authentication (when authentication is required)	string	

Responses:

HTTP Code	Description	Schema
200	Console Output	string
default	Error	Error

Definitions

Error

Name	Description	Schema
error *required*	Description of the error	string

Response

Name	Description	Schema
error *optional*	Error message if an error occured	string
success *required*	true if the command succeeded, false otherwise	boolean

Exercises

Armed with your new knowledge, read the NodeODM specification document and try to perform the following tasks:

- The API defines two ways to new tasks. We've already used **/task/new**, which is a simplified method that uploads all images at once. NodeODM also exposes a chunked API for uploading images in parallel. Using **/task/new/init**, **/task/new/upload** and **/task/new/commit**, can you create a task using them?

- JSON can be used to specify task options. For example, to set the processing option **orthophoto-resolution** we first encode it to JSON [{"name":"orthophoto-resolution","value":"2"}] then, we can pass it to the *options* FormData parameter of **/task/new**. Search Google for information on the JSON format and how to use arrays. Afterwards, can you find a way to pass multiple options to **/task/new**?
- Can you restart NodeODM with the **token** parameter to add an authentication token and then invoke any of the API functions using the **?token=** Query parameter in the URLs? What happens if you forget to pass the token to your URLs?

If you get stuck, ask for help on the OpenDroneMap forum[58].

[58] OpenDroneMap Forum: https://community.opendronemap.org

15

Automated Processing With Python

What's better than aerial data processing? Automated aerial data processing of course!

In *The NodeODM API* chapter we've learned how to use the NodeODM API by using cURL. Now we will learn about PyODM, a Python library for communicating with the NodeODM API. Python is a programming language that is used by many OpenDroneMap projects and is popular language in the GIS community. This chapter will not try to teach the fundamentals of Python, as there are already plenty of free online resources for that[59]. Previous knowledge of Python is preferred, but not required. Python is a very descriptive language and readers should be able to follow the examples even without formal training in Python.

Why would you want to use Python (instead of cURL or CloudODM)? With Python you can leverage the image processing capabilities of ODM to create new, custom applications that have an aerial image processing component. For example, you

[59] Python for Beginners: https://www.python.org/about/gettingstarted/

could build:

- A platform for counting trees using modern computer vision techniques after a user uploads drone images.
- An application that detects when SD cards are inserted in a computer and automatically processes images without user interaction.
- An application for extracting video frame segments from YouTube and automatically generating 3D reconstructions from scenes of your favorite movies.

Obviously each of these applications can be complex and requires coding skills, but PyODM would help you with the image processing part of the implementation.

It should be noted that PyODM does a bit more than a simply communicating with NodeODM, as it deals with things such as managing parallel downloads (a sort of download accelerator which makes network transfers faster), parallel uploads, automatic retries for fault tolerance, as well as dealing with backward-compatibility issues between API versions.

Getting Started

If you have already installed Python (see *Installing The Software* chapter), all you need to do from a terminal is type:

```
$ pip install -U pyodm
```

which will install the *pyodm* package. Afterwards, make sure to start a NodeODM instance via:

```
$ docker run -d -p 3000:3000 opendronemap/nodeodm
```

You can use any text editor you want to write Python code. I prefer to use the free and open source Visual Studio Code[60] but any text editor will do. All examples below are also available for download from https://github.com/MasseranoLabs/odmbook-assets.

Example 1: Hello NodeODM

Type the following program into a new file using your text editor, then save it as **hello.py**.

```
from pyodm import Node, exceptions

node = Node('localhost', 3000)
try:
    print(node.info())
except exceptions.NodeConnectionError as e:
    print("Cannot connect: " + str(e))
```

Then run it:

```
$ python hello.py

{'version': '1.5.2', 'task_queue_count': 0, '
    total_memory': 1021136896, 'available_memory':
    436518912, 'cpu_cores': 2, 'max_images': None, '
```

[60] Visual Studio Code: https://code.visualstudio.com/

```
    max_parallel_tasks': 2, 'engine': 'odm', '
    engine_version': '0.6.0', 'odm_version': '?'}
```

We have successfully retrieved the NodeODM instance information. We also check for any connection errors just in case we have forgotten to launch the NodeODM instance and print an error message if we can't connect to the node.

There are a few different types of errors (*exceptions* in Python jargon) that we can handle:

- **OdmError**: A generic catch-all exception related to anything PyODM
- **NodeServerError**: The server replied in a manner which we did not expect. Usually this indicates a temporary malfunction of the node
- **NodeConnectionError**: A connection problem (such as a timeout or a network error) has occurred
- **NodeResponseError**: The node responded with an error message indicating that the requested operation failed
- **TaskFailedError**: A task did not complete successfully

Example 2: Process Datasets

For this example, save it as **process.py** and place the file in the same folder where some aerial images are stored (e.g. **D:\odmbook\project\images**):

```
import glob
from pyodm import Node, exceptions

node = Node("localhost", 3000)

try:
    # Get all JPG files in directory
    images = glob.glob("*.JPG") + glob.glob("*.jpg") +
         glob.glob("*.JPEG") + glob.glob("*.jpeg")

    print("Uploading images...")
    task = node.create_task(images, {'dsm': True, '
        orthophoto-resolution': 2})
    print(task.info())

    try:
        def print_status(task_info):
            msecs = task_info.processing_time
            seconds = int((msecs / 1000) % 60)
            minutes = int((msecs / (1000 * 60)) % 60)
            hours = int((msecs / (1000 * 60 * 60)) %
                24)
            print("Task is running: %02d:%02d:%02d" %
                (hours, minutes, seconds), end="\r")
        task.wait_for_completion(status_callback=
            print_status)

        print("Task completed, downloading results
            ...")

        # Retrieve results
        def print_download(progress):
            print("Download: %s%%" % progress, end="\r
                ")
        task.download_assets("./results",
            progress_callback=print_download)
```

```
        print("Assets saved in ./results")
    except exceptions.TaskFailedError as e:
        print("\n".join(task.output()))

except exceptions.NodeConnectionError as e:
    print("Cannot connect: %s" % e)
except exceptions.OdmError as e:
    print("Error: %s" % e)
```

Then run it via:

```
$ pwd
/d/odmbook/project/images

$ python process.py

Uploading images...
{'uuid': 'cc751818-36af-41e9-92d2-bc146cdee10c', 'name
    ': 'Task of 2019-06-16T21:00:02.502Z', '
    date_created': datetime.datetime(2019, 6, 16, 21,
    0, 2), 'processing_time': 1, 'status': <TaskStatus
    .RUNNING: 20>, 'last_error': '', 'options': [{'
    name': 'orthophoto-resolution', 'value': 2}, {'
    name': 'dsm', 'value': True}], 'images_count': 18,
     'progress': 0, 'output': []}
Task completed, downloading results...
Assets saved in ./results
```

This is a more comprehensive example. First, we create a **Node** instance. From that instance we can create new tasks via **create_task()** which take as input a list of image paths (which we generate via the **glob** function) and returns a **Task** instance. We then wait for the results to be ready via

wait_for_completion() and we ask to be notified of status updates by displaying how long the task has been running for. If a task fails at any point in time, **wait_for_completion()** raises a **TaskFailerError.** We "catch" that error and display the task output to the user if that happens. When the task completes, we download the results and display download progress information.

Concluding Remarks

PyODM is a simple module that makes it straightforward to leverage the capabilities of OpenDroneMap with Python. It's also a module used within ODM and WebODM and will continue to be well supported in the future. If you need to use NodeODM with a different programming language, you can use PyODM as a reference to write a client for your language of choice.

API Reference

The following reference is from PyODM version 1.5.2b (the latest version as of the writing of this book). The reference to the latest version can be found online at https://pyodm.readthedocs.io

class pyodm.api.Node(host, port, token='', timeout=30)

A client to interact with NodeODM API.

Parameters:

- **host** (*str*) – Hostname or IP address of processing node
- **port** (*int*) – Port of processing node

- **token** (*str*) – token to use for authentication
- **timeout** (*int*) – timeout value in seconds for network requests

create_task(*files*, *options{}*, *name=None*, *progress_callback=None*, *skip_post_processing=False*, *webhook=None*, *outputs=[]*, *parallel_uploads=10*, *max_retries=5*, *retry_timeout=5*)

Start processing a new task. At a minimum you need to pass a list of image paths. All other parameters are optional.
Parameters:

- **files** (*list*) – list of image paths + optional GCP file path.
- **options** (*dict*) – options to use, for example {'orthophoto-resolution': 3, ...}
- **name** (*str*) – name for the task
- **progress_callback** (*function*) – callback reporting upload progress percentage
- **skip_post_processing** (*bool*) – When true, skips generation of map tiles, derivate assets, point cloud tiles.
- **webhook** (*str*) – Optional URL to call when processing has ended (either successfully or unsuccessfully).
- **outputs** (*list*) – Optional paths relative to the project directory that should be included in the all.zip result file, overriding the default behavior.
- **parallel_uploads** (*int*) – Number of parallel uploads.
- **max_retries** (*int*) – Number of attempts to make before giving up on a file upload.
- **retry_timeout** (*int*) – Wait at least these many seconds before attempting to upload a file a second time, multiplied by the retry number.

Returns:

Task()

static **from_url**(*url*, *timeout=30*)

Create a Node instance from a URL.

```
>>> n = Node.from_url("http://localhost:3000?token=abc
    ")
```

Parameters:

- **url** (*str*) – URL in the format proto://hostname:port/?token=value
- **timeout** (*int*) – timeout value in seconds for network requests

Returns:

Node()

get_task(*uuid*)

Helper method to initialize a task from an existing UUID

Parameters:

uuid – Unique identifier of the task

info()

Retrieve information about this node

Returns:

NodeInfo()

options()

Retrieve the options available for creating new tasks on this node.

Returns:
[**NodeOption()**]

url(*url*, *query={}*)
Get a URL relative to this node.
Parameters:
- **url** (*str*) – relative URL
- **query** (*dict*) – query values to append to the URL

Returns:
Absolute URL (str)

version_greater_or_equal_than(*version*)
Checks whether this node version is greater than or equal than a certain version number
Parameters:
version (*str*) – version number to compare
Returns:
bool

class pyodm.api.Task(node, uuid)

A task is created to process images. To create a task, use **create_task()**.
Parameters:
- **node** (**Node()**) – node this task belongs to
- **uuid** (*str*) – Unique identifier assigned to this task.

cancel()
Cancel this task.
Returns:

task was canceled or not (bool)

download_assets(*destination, progress_callback=None, parallel_downloads=16, parallel_chunks_size=10*)

Download this task's assets to a directory.

Parameters:

- **destination** (*str*) – directory where to download assets. If the directory does not exist, it will be created.
- **progress_callback** (*function*) – an optional callback with one parameter, the download progress percentage
- **parallel_downloads** (*int*) – maximum number of parallel downloads if the node supports http range.
- **parallel_chunks_size** (*int*) – size in MB of chunks for parallel downloads

Returns:

path to saved assets (str)

download_zip(*destination, progress_callback=None, parallel_downloads=16, parallel_chunks_size=10*)

Download this task's assets archive to a directory.

Parameters:

- **destination** (*str*) – directory where to download assets archive. If the directory does not exist, it will be created.
- **progress_callback** (*function*) – an optional callback with one parameter, the download progress percentage.
- **parallel_downloads** (*int*) – maximum number of parallel downloads if the node supports http range.
- **parallel_chunks_size** (*int*) – size in MB of chunks for parallel downloads

Returns:
path to .zip archive file (str)

info(*with_output=None*)
Retrieves information about this task.
Returns:
TaskInfo()

output(*line=0*)
Retrieve console task output.
Parameters:
line (*int*) – Optional line number that the console output should be truncated from. For example, passing a value of 100 will retrieve the console output starting from line 100. Negative numbers are also allowed. For example -50 will retrieve the last 50 lines of console output. Defaults to 0 (retrieve all console output).
Returns:
console output (one list item per row) ([str])

remove()
Remove this task.
Returns:
task was removed or not (bool)

restart(*options=None*)
Restart this task.
Parameters:
options (*dict*) – options to use, for example {'orthophoto-resolution': 3, ...}
Returns:

task was restarted or not (bool)

wait_for_completion(*status_callback=None, interval=3, max_retries=5, retry_timeout=5*)

Wait for the task to complete. The call will block until the task status has become **COMPLETED()**. If the status is set to **CANCELED()** or **FAILED()** it raises a TaskFailedError exception.

Parameters:

- **status_callback** (*function*) – optional callback that will be called with task info updates every interval seconds.
- **interval** (*int*) – seconds between status checks.
- **max_retries** (*int*) – number of repeated attempts that should be made to receive a status update before giving up.
- **retry_timeout** (*int*) – wait N*retry_timeout between attempts, where N is the attempt number.

class pyodm.types.NodeInfo(json)

Information about a node

Parameters:

- **version** (*str*) – Current API version
- **task_queue_count** (*int*) – Number of tasks currently being processed or waiting to be processed
- **total_memory** (*int*) – Amount of total RAM in the system in bytes
- **available_memory** (*int*) – Amount of RAM available in bytes
- **cpu_cores** (*int*) – Number of virtual CPU cores

- **max_images** (*int*) – Maximum number of images allowed for new tasks or None if there's no limit.
- **max_parallel_tasks** (*int*) – Maximum number of tasks that can be processed simultaneously
- **odm_version** (*str*) – Current version of ODM (deprecated, use engine_version instead)
- **engine** (*str*) – Lowercase identifier of the engine (odm, micmac, ...)
- **engine_version** (*str*) – Current engine version

class pyodm.types.NodeOption(domain, help, name, value, type)

A node option available to be passed to a node.
Parameters:

- **domain** (*str*) – Valid range of values
- **help** (*str*) – Description of what this option does
- **name** (*str*) – Option name
- **value** (*str*) – Default value for this option
- **type** (*str*) – One of: ['int', 'float', 'string', 'bool', 'enum']

class pyodm.types.TaskInfo(json)

Task information
Parameters:

- **uuid** (*str*) – Unique identifier
- **name** (*str*) – Human friendly name
- **date_created** (*datetime*) – Creation date and time
- **processing_time** (*int*) – Milliseconds that have elapsed

since the start of processing, or -1 if no information is available.
- **status** (**pyodm.types.TaskStatus()**) – status (running, queued, etc.)
- **last_error** (*str*) – if the task fails, this will be set to a string representing the last error that occured, otherwise it's an empty string.
- **options** (*dict*) – options used for this task
- **images_count** (*int*) – Number of images (+ GCP file)
- **progress** (*float*) – Percentage progress (estimated) of the task
- **output** (*[str]*) – Optional console output (one list item per row). This is populated only if the with_output parameter is passed to info().

class pyodm.types.TaskStatus

Task status
Parameters:
- **QUEUED** – Task's files have been uploaded and are waiting to be processed.
- **RUNNING** – Task is currently being processed.
- **FAILED** – Task has failed for some reason (not enough images, out of memory, etc.
- **COMPLETED** – Task has completed. Assets are be ready to be downloaded.
- **CANCELED** – Task was manually canceled by the user.

Glossary

2.5D Model: A model where elevation is simply *extruded* from the ground plane and thus is not a true 3D model.
Artifacts: undesired alterations generated as the result of a digital process.
API: Application Programming Interface. A set of functions allowing the creation of applications that access the features or data of another application.
AWS: Amazon Web Services is a cloud service provider.
Bundle Adjustment: a refinement step during the Structure From Motion process that improves the location of cameras, the 3D points of the scene and the camera parameters.
CloudODM: A command line tool to process aerial imagery in the cloud.
ClusterODM: A NodeODM API compatible autoscalable load balancer and task tracker for connecting multiple NodeODM nodes under a single network address.
CRS: Coordinate Reference System. A CRS is a coordinate-based system used to locate geographical entities.
CSV: Comma Separated Value is a textual file format where fields are typically separated by commas or some other character such as a space or a tab.
cURL: a software providing a library and command-line tool for transferring data using many protocols.
DEM: Digital Elevation Model (either a DSM or a DTM).

Depthmap: An image containing distance information for objects in a scene relative to the camera plane.
Docker: a tool used to launch *containers*, lightweight standalone packages of software. OpenDroneMap uses docker to run many of its software. Docker is also the name of the company that develops the tool.
DSM: Digital Surface Model. A 2D representation of elevation that includes terrain, buildings, trees and other structures.
DTM: Digital Terrain Model. A 2D representation of elevation that includes terrain only.
EXIF: Exchangeable file format for images and their auxiliary tags. EXIF tags are pieces of information embedded within an image. They can include information such as camera model, GPS location of where the image was shot, focal length information and many more.
GCP: Ground Control Point. A GCP is a position measurement made on the ground, often taken at the location of a clearly identifiable marker, to increase the georeferencing accuracy of a reconstruction.
GSD: Ground Sampling Distance. In an aerial photo, it's the distance between pixels measured on the ground.
Mesh: A collection of vertices, edges and faces that define the shape of a 3D model. A mesh does not include color information such as textures.
MVE: Multi-View Environment is a suite of software packages developed to ease the work with multi-view datasets and to support the development of algorithms based on multiple views. It features Structure from Motion, Multi-View Stereo and Surface Reconstruction algorithms.
MVS: Multi-View Stereo is a branch of study in computer vision that focuses on the reconstruction of 3D models from

multiple overlapping image pairs. MVS programs expect that information about cameras has already been computed.

NodeODM: A lightweight REST API to access aerial image processing engines such as ODM or MicMac.

Noise: An unwanted interference. When applied to point clouds, it indicates points that should not be present or that were missed during outlier filtering.

Nadir: The direction pointing directly below a particular location.

ODM: A command line toolkit to generate maps, point clouds, 3D models and DEMs from drone, balloon or kite images.

OpenSfM: Open source structure from Motion library written in Python. The library serves as a processing pipeline for reconstructing camera poses and 3D scenes from multiple images.

Orthophoto: An image that has been *orthorectified*, warped in such a way that distances and scales are uniform.

Photogrammetry: the process of obtaining reliable information about physical objects and the environment through the process of using photographic images.

Preemptive Matching: During the Structure From Motion process, the act of reducing the possible number of pair candidates by using the location information stored in the EXIF tags of the images.

PyODM: A Python SDK for adding aerial image processing capabilities to applications.

RTK: Real Time Kinematics. A satellite navigation technique used to enhance the precision of position data derived from satellite-based positioning systems.

SDK: Software Development Kit. A set of libraries, tools and examples that help software developers to build software with

a particular technology.
SFM: Structure From Motion is a photogrammetry technique for estimating 3D objects (structures) from overlapping image sequences (from motion).
Texturing: The act of creating 2D images suitable for use on 3D models, also known as *texture maps* or *texture skins*. Textures give 3D models a realistic appearance.
UAV: Unmanned Aerial Vehicle. An aircraft piloted by remote control or on-board computers.
UI: User Interface.
UTM: Universal Transverse Mercator is a coordinate reference system. It ignores altitude and treats the earth as a perfect ellipsoid.
Virtualization: the act of creating a virtual version of computer hardware platforms, storage devices, and computer network resources.
VM: Virtual Machine. A software program or operating system that works like a separate computer.
WebODM: User-friendly, extendable application and API for processing aerial imagery.
WSL: Windows Subsystem for Linux. A feature of Windows 10 that allows Linux programs to run natively on Windows.

About the Author

Piero Toffanin is a software developer currently focused on geospatial and drone software development. He has been working on open source software for over 21 years. Since 2016 he is an OpenDroneMap core developer and frequently speaks at conferences about OpenDroneMap, geospatial and open source.

You can connect with me on:

https://piero.dev

https://masseranolabs.com