VISOR User Guide

VICAR SOFTWARE FOR MARS AND BEYOND

DEEN, ROBERT G (US 398E)

Robert Deen
Heather Lethcoe
Deborah Padgett
Kevin Gill
Oleg Pariser
Hallie Abarca
Jacqueline Ryan
Shari Mayer

Jet Propulsion Laboratory, California Institute of Technology

Version 1.0
June 26, 2023

1. INTRODUCTION 5
1.1 Background 5
1.2 Audience 5
1.3 How To Use This Document 5
1.4 History 6
1.5 Exclusions and Limitations 7
1.6 Relationship to VICAR 8
1.7 Camera SISs 8
1.8 Acknowledgments 9
2. INSTALLATION 11
2.1 Platform Requirements 11
2.2 Source vs. Binary installation 11
2.3 Install VICAR/VISOR Code 11
2.4 Calibration Data 12
2.5 Sample Data 12
3. OBTAINING DATA 13
3.1 PDS 13
3.2 Searching for Data via Web Browser 14
4. PRODUCT OVERVIEW 16
4.1 EDRs and FDRs 16
4.2 Bayer processing and Color 16
4.3 Radiometric Correction 16
4.4 Photometric Correction 16
4.5 Stereo Correlation 17
4.6 XYZ/Range Data 17
4.7 Rover Masks 17
4.8 Terrain Meshes 17
4.9 Surface Normals 18
4.10 Slope maps 18
4.11 Roughness 18
4.12 Reachability 18
4.13 Goodness 18
4.14 Instrument/Helicopter Placement 19
4.15 Mosaics 19
4.15.1 Cylindrical Mosaic 19
4.15.2 Cylper (stereo) Mosaic 19
4.15.3 Polar Mosaic 19
4.15.4 Vertical Mosaic 20
4.15.5 Orthorectified Mosaic 20
4.15.6 Perspective Mosaic 20
4.15.7 Overlay mosaics (XYZ etc) 20
4.16 Terrain Meshes 20
5. VICAR BASICS 21
5.1 Setup 21
5.1 PDF Files 21
5.2 Running Programs 22
5.2 VICAR File Format 22

5.3 VICAR Labels 23
5.4 Viewing Images 24
5.4.1 xvd 24
5.4.2 Marsviewer / Jadeviewer 25
5.5 Transcoder (File Format Conversion) 27
6. VISOR BASICS 29
6.1 PDS and ODL Labels 29
6.2 Filename Conventions 30
6.3 Three-Letter Codes 31
6.4 List Files 32
6.5 Camera Models 32
6.6 Calibration Files 33
6.7 Pipeline processing summary 34
6.8 History Label and Reconstructing Processing 34
6.9 Coordinate Frames 42
6.10 Quaternions 43
6.11 POINT_METHOD parameter 44
6.12 Linearization 46
6.13 RSF files 47
6.14 Parallelized Programs 49
6.15 Generic Mission 50
7. MOSAICS 51
7.1 Parallax Effects 51
7.2 Surface Models 52
7.3 Seam Correction 52
7.4 Pointing Correction Principles 53
7.5 Pointing Correction Process 53
7.6 Tiepoints 54
7.7 Bundle Adjustment for Mosaics 54
7.8 Bundle Adjustment for Meshes 55
8. MESHES 56
9. PROGRAM SUMMARIES 57
10. USE CASE SCENARIOS 60
10.1 Linearization 61
10.2 Radiometric Correction 63
10.3 Bayer Processing 65
10.4 Color Processing Using Filter Wheels 67
10.5 Radiometric Correction, Part 2 69
10.6 Calibrated Color Processing 73
10.7 Stereo Correlation – Script 77
10.8 Stereo Correlation - Separate Programs 82
10.9 Surface Normals and Slope Maps 87
10.10 Using Marsviewer Locally 91
10.11 Roughness 93
10.12 Cylindrical Mosaic 94
10.13 Seam Corrected Mosaic 98
10.14 Mosaic Brightness Adjustment 103
10.15 Cylper (stereo) Mosaic 107
10.16 Orthorectified Mosaic 110

10.17 Overlay mosaics (XYZ etc) ... 113
10.18 Using IDX/ICM Outputs to Make XYZ Mosaics ... 114
10.19 Making Movies/Blinks ... 117
10.20 Multi-Instrument Mosaic ... 119
10.21 In-Situ Mesh Creation ... 122
10.22 Making an orbital mesh ... 124
11. SOFTWARE ARCHITECTURE ... 129
11.1 Metadata (labels) ... 129
11.2 PIG Library ... 129
11.3 Class Structure ... 130
11.3.1 PigModelBase ... 130
11.3.2 PigMission ... 131
11.3.3 PigCameraModel ... 131
11.3.4 PigPointingModel ... 131
11.3.5 PigCoordSystem ... 132
11.3.6 PigFileModel ... 133
11.3.7 PigLabelModel ... 133
11.3.8 PigSufaceModel ... 133
11.3.9 PigSolutionManager ... 133
11.3.10 PigRoverStateManager ... 133
11.3.11 PigPointingCorrections ... 133
11.3.12 PigBrtCorrModel ... 134
11.3.13 RadiometryModel ... 134
11.3.14 PigColorModel ... 134
11.4 Support Subroutines ... 134
11.5 Application Programs ... 134
12. EXTENDING VISOR ... 136
12.1 Code Conventions ... 136
12.2 Augmenting Existing Programs ... 136
12.3 Augmenting Mars Subroutines ... 136
12.4 Augmenting the PIG library ... 137
12.5 Writing New Programs ... 137
12.6 Supporting New Missions ... 137
12.7 Future Enhancements ... 138
13. ACRONYMS ... 139
14. REFERENCES ... 143
15. APPENDIX A: CODE CONVENTIONS AND GUIDELINES ... 146
15.1 General Coding Style ... 146
15.2 Bob Deen's Coding Style Suggestions ... 147
15.3 VICAR Coding Guidelines ... 147
15.4 Mars/PIG subsystem Rules ... 149

1. INTRODUCTION

Welcome to Mars!

1.1 Background

"VISOR" is a set of VICAR (Video Image Communication And Retrieval) programs that implement a world-class in-situ image processing capability for rovers and landers. Despite the name, it is not specific to Mars - it was developed for use with the NASA Mars surface program (from Mars Pathfinder on), but it is usable for any in-situ spacecraft. Note that VISOR has been historically called "the Mars suite". VISOR stands for VICAR In-Situ Operations for Robotics.

VISOR is designed to be multimission, easily adaptable to new missions. It is built around a set of C++ class libraries (collectively called PIG - Planetary Image Geometry) that hide almost all mission-specific code behind generic interfaces. The bulk of the code is thus agnostic to mission, and reusable across all missions. The suite has been used successfully by the operations team for MER (Mars Exploration Rover), Phoenix, MSL (Mars Science Laboratory), InSight (Interior Exploration using Seismic Investigations, Geodesy and Heat Transport), Mars 2020 (both the Perseverance rover and the Ingenuity helicopter), and several testbeds, where it generated all the terrain models (meshes) used to command the spacecraft during their lifetimes.

In a nutshell, VISOR does radiometric, color, and (limited) photometric correction, stereo correlation to produce XYZ data and 3D terrain meshes, and makes mosaics in several projections, creates products ready for PDS (Planetary Data System) archive, and much more.

This document describes how to use VISOR with example scenarios and tips for image processing with your own mission using the software. VISOR was developed by the Multimission Image Processing Lab (MIPL) at the NASA Jet Propulsion Laboratory (JPL). We at MIPL are eager to help implement new missions, so please feel free to contact us. We can either implement your mission for you, or help you do it. We also conduct operations for many missions and would be happy to talk about applying our expertise to your mission.

1.2 Audience

The primary audience for this document is scientists, researchers, and enthusiasts who wish to apply VISOR to data taken by already-supported missions. This generally means using data that is available to the public via PDS, although science team members can also use VISOR on operations data. The intent is to be able to re-create the derived products that are in PDS (for scientific reproducibility reasons), as well as to create new products that are not archived in PDS (e.g., custom mosaics).

The secondary audience is people who wish to apply VISOR to new in-situ missions, or add capabilities to it for existing missions. The document provides an overview of capability, while discussing in broad terms how to design and implement a new mission or add functionality. People wishing to plan new missions or other contributions are encouraged to contact MIPL to discuss it in more detail.

1.3 How To Use This Document

Obviously, the Installation section is a good place to start, since without the software it doesn't do much good.

After that, if you have not used VICAR before, the VICAR Basics section may come in handy. Note that core VICAR has been out for some years, and there are additional getting started resources there.

You should download the Camera SIS for the mission you are interested in. It is not a document to read through, but skim and use it for reference.

VISOR Basics talks about how to use VISOR programs generally, introducing quite a few concepts that are common to many of the programs.

Product Overview discusses briefly the types of products that can be made with this software. It may be a review to those who are familiar with the mission data already.

Then, look through the Use Case Scenarios. The data you need is provided as part of the distribution package. Find a scenario that is of interest, and walk your way through it. This section is intended as a step-by-step guide.

Once you're comfortable with the Use Cases, go to Obtaining Data to fetch your own data to work on, and off you go!

Finally, the most hard-core users will want to read Software Architecture and Extending VISOR.

Please, contact MIPL with any questions you might have. We would be happy to help to the extent possible.

1.4 History

VISOR started as a small set of VICAR programs written primarily by Jean Lorre at MIPL to implement stereo correlation and mosaics for the Mars Pathfinder mission. It was used successfully for that mission, but the programs were specific to Mars Pathfinder.

In 1997, Robert Deen at MIPL was tasked with porting this software for use with the upcoming Mars Polar Lander mission. Realizing that this was only the first in a long line of in-situ missions, Deen took the time to completely overhaul the software, creating the PIG library. The PIG library hid all mission-specific code behind a C++ class structure, allowing the application programs to work cross-mission.

Although the Mars Polar Lander mission failed (it landed in Dec 1999 but failed to communicate), VISOR was ready to process data from it. It was also back-ported to support Mars Pathfinder as a proof of concept.

The suite was next ported to the Mars Athena 2001 testbed, and the Field Integrated Design and Operations (FIDO) rover at JPL. This effort proved the value of PIG, as a minimal investment in mission-specific code brought the entire MPL processing suite to bear on these small, limited-budget testbeds.

This success led to its use with the Mars Exploration Rovers (MER), Spirit and Opportunity. Due once again to the short time it took to implement the basic mission with PIG, funds and time were available to significantly enhance the capability of the suite.

A similar pattern then ensued for the Phoenix, Mars Science Laboratory (MSL), InSight, and Mars 2020 (M20) landers/rovers, as well as the Ingenuity helicopter. Only brief efforts were required to port to each new mission, freeing up resources to further expand the suite's capabilities.

It is important to realize that the suite is not just designed for Mars; it is for all in-situ missions. It was used for the MoonRise testbed (a proposal effort [Trebi-Ollennu2012]), and recently for the ColdArm experiment, a CLPS task intended for Lunar use, and CADRE, a CLPS mission.

The table below summarizes the lines of code (LOC) in VISOR for each mission and the multimission core. It also includes a *very* approximate order of magnitude development time estimate just for pure porting, without adding capabilities.

Component	Approx LOC	Approx Dev Time
Applications (multimission)	178k	10+ years
PIG multimission base	21k	(Incl above)
Multimission total	199k	10+ years
Component	Approx LOC	Approx Dev Time
PIG MPF	3.0k	3 weeks
PIG MPL	2.8k	6 weeks
PIG M01	0.6k	2 days
PIG Generic	1.0k	1 week
PIG FIDO	1.2k	3 weeks
PIG MER	2.5k	2 months
PIG PHX	2.8k	1 month
PIG MSL	3.0k	2 months
PIG InSight	1.5k	1 month
PIG M20	4.1k	3 months
PIG Psyche (partial)	1.4k	1 day
PIG Coldarm	1.4k	2 days
Total for 12 missions	25.3k	50 weeks

1.5 Exclusions and Limitations

Every attempt was made to include as much of VISOR software as possible in the open source

release. However, a small subset of programs use bits of flight software (FSW) from the various missions, which cannot at this point be Open Sourced. The arm reachability programs fall into this category, as they use the FSW kinematics code to determine where in the workspace the arm can reach. Also excluded is some of the rover masking capability, which uses a simplified volumetric model of the rover to indicate which pixels are potentially on the rover rather than terrain or sky. This uses a FSW capability in MSL and Mars 2020 to prevent the ChemCam/SuperCam LIBS laser from shooting the rover itself, so the mask is somewhat conservative (covering some bits of ground).

These two data sets thus cannot be re-created by users. Unfortunately, users will have to rely on what is available from PDS for masking and reachability.

Another major exclusion are the telemetry processors, commonly called “edrgen,” although the module names are specific to the mission and, often, per instrument. The telemetry processors need access to the spacecraft telemetry dictionary, which is problematic to release due to security concerns. However, the raw data products that come down from the spacecraft are not generally made available to PDS, so there is no need to have the telemetry processor to process them. We thus assume that the EDR = “raw” image is the starting point for this open source release.

Although the software is generally backwards-compatible all the way to MER and before, it has evolved considerably over the years. Thus, it may not be possible to *exactly* reproduce a given product in PDS, especially for older missions. However, it should be possible to *substantively* reproduce any product in all situations we are aware of.

1.6 Relationship to VICAR

VISOR is built on Video Image Communication And Retrieval (VICAR) software, and most of the programs are VICAR programs. The VICAR core is required in order to fully utilize VISOR. They are included together in the installation packages.

VICAR is a set of computer programs and procedures designed to facilitate the acquisition and processing of multi-dimensional imaging data. VICAR was developed and is maintained by the Multimission Image Processing Lab (MIPL) at NASA’s Jet Propulsion Laboratory (JPL) for use with JPL’s planetary missions. Most of JPL’s planetary missions have cameras that utilize VICAR, beginning with Surveyor in 1966 and continuing to present day missions such as Mars 2020, Mars Science Laboratory (MSL), InSight, and more.

VICAR is a command-line-oriented system consisting of approximately 350 application programs (not counting VISOR) that perform a wide variety of image processing functions. These functions are suited to compute trivial to highly complex image processing techniques. VICAR combines these applications to scripts to systematically accomplish complicated image processing.

The VICAR core has been open source since 2015. We are happy that we are finally able to add the Mars application suite to Open Source VICAR.

1.7 Camera SISs

Each supported mission has a document called a Camera SIS (Software Interface Specification). The SIS describes all the data products and most of the processing for images

from that mission. It is the primary reference for image data stored in the Planetary Data System (PDS) archive for that mission. As such, the SIS's are the definitive sources of information; if something in a SIS contradicts this guide, the SIS is more likely to be correct.

The intent of this guide is not to reproduce all of the information in the SIS. If you are working with data from a specific mission, it is highly recommended that you download and examine the relevant SIS. Rather, this guide helps provide pointers to get you started in using the Mars software suite.

It should be noted that MIPL also processes non-camera instrument data for most of the missions, and each of those instruments has a SIS as well. For the most part, this processing for non-imaging instruments consists of converting telemetry data products to "raw" science data files (EDRs), rather than calibrating frames and producing reduced or derived data products (RDRs). Since the Mars software suite is specifically focused on images, this guide is mostly irrelevant to non-camera instruments and instrument modes.

Note that some of the SISs below (notably M2020 and InSight) have appendices that are in separate files. These can be accessed by trimming the filename from the end of the URL. The URLs refer to the main document.

Mission	Camera SIS Link
MER	https://pds-imaging.jpl.nasa.gov/data/mer/opportunity/mer1ho_0xxx/document/CAMSIS_latest.PDF
PHX	https://pds-imaging.jpl.nasa.gov/data/phoenix/phxmos_0xxx/document/cam_edr_rdr_sis.pdf
MSL	https://pds-imaging.jpl.nasa.gov/data/msl/MSLHAZ_0XXX/DOCUMENT/MSL_CAMERA_SIS_latest.PDF
InSIGHT	https://pds-imaging.jpl.nasa.gov/data/nsyt/insight_cameras/document/insight_cameras_sis.pdf
M2020	https://pds-geosciences.wustl.edu/m2020/urn-nasa-pds-mars2020_mission/document_camera/Mars2020_Camera_SIS.pdf
MSAM	https://pds-imaging.jpl.nasa.gov/data/individual_investigation/deen_pdart16_msl_msam/document/MSAM_sis.pdf

1.8 Acknowledgments

The following people are known to have contributed to VISOR development through the years. Sincere apologies to anyone who has been inadvertently missed.

- Robert Deen
- Jean Lorre
- Francois Ayoub
- Oleg Pariser
- Walt Bunch

- Igor Yanovsky
- Edwin Sarkissian
- Matthew Yeates
- Robert Crocco
- Jacqueline Ryan
- Mauricio Hess-Flores
- Hyun Lee
- Justin Maki
- Alice Stanboli
- Steven Myint
- Todd Litwin
- Marjorie Lucs
- Nicholas Ruoff
- Stirling Algermissen
- Marsette Vona
- Shari Mayer
- You Lu
- Gary Yagi
- Casey Handmer
- Chris Leger
- John Wright
- Gerhard Klimeck

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004).

2. INSTALLATION

VICAR can be installed using either pre-built binaries (highly recommended) or built from source. In either case, is distributed with a collection of third-party libraries, called “externals”, which are also pre-built binaries but could be built from source.

The open source page, from which VICAR can be downloaded, is here:
https://www-mipl.jpl.nasa.gov/vicar_open.html

The github page, which can also be used for downloads, is:

https://github.com/NASA-AMMOS/VICAR

The VICAR Installation Guide can be found here:

https://github.com/NASA-AMMOS/VICAR/blob/master/vos/docsource/vicar/VICAR_build_5.0.pdf

The VICAR Quick Start Guide can be found here:

https://github.com/NASA-AMMOS/VICAR/blob/master/vos/docsource/vicar/VICAR_guide_5.0.pdf

2.1 Platform Requirements

VICAR supports the following platforms with prebuilt binaries:

- Linux (64-bit)
- Mac OS X (64-bit)

The code itself supports other platforms, mostly historical (for example, 32-bit Linux), which could be built from source. However, not all externals are necessarily available for other platforms.

2.2 Source vs. Binary installation

The whole point of Open Source is to make source code available, so users can build it from scratch and make modifications. However, unless you have a specific need (like an incompatible OS, or you want to modify the code), we strongly recommend using a prebuilt binary installation rather than building from source. It is difficult to get all the dependencies right, and we have not had a lot of resources to devote to making builds easier. You can build applications or even parts of the PIG library without doing a full VICAR build.

If you must do a full build, consult the VICAR Quick-Start Guide.

2.3 Install VICAR/VISOR Code

The first step is to install the VICAR code. Consult the VICAR installation guide. Although it doesn’t mention VISOR by name, the VISOR code gets installed along with VICAR (in `$MARSLIB` instead of `$R2LIB`).

It is recommended that you review the VICAR Quick Start Guide next to familiarize yourself with VICAR generally.

2.4 Calibration Data

VISOR needs calibration data specific to each mission in order to function, see Section 6.6 for details. Although the calibration data is generally in PDS for each mission, it is gathered here in the proper format for VISOR for user convenience.

The download page above contains a set of links for the calibration files. Download the mission(s) you want to work with. If it's most of them, there's a big tarball that has all of the missions together. Install them somewhere, and set the `$MARS_CONFIG_PATH` environment variable to point to them. Note that it is a colon-separated list that points to the top-level directory of each mission (e.g. the level that contains the `camera_model`, `param_files`, etc directories). For example:

```
setenv MARS_CONFIG_PATH /home/caldata/m20:/home/caldata/msl:/home/caldata/mer
```

2.5 Sample Data

Section 10 consists of use-case scenarios demonstrating the use of VISOR on real Mars data. All of the data sets used as inputs for these scenarios come from PDS and can be downloaded from PDS directly (see Section 4). However, that is an extreme inconvenience. For that reason, the download page above also contains a tarball of all the sample input data used in the use cases.

3. OBTAINING DATA

3.1 PDS

The Cartography and Imaging Sciences Discipline Node (aka "Imaging Node") of the Planetary Data System is the curator of NASA's primary digital image collections from past, present and future planetary missions. Imaging provides to the NASA planetary science community the digital image archives, ancillary data, sophisticated data search and retrieval tools, and cartographic and technical expertise necessary to develop and fully utilize the vast collection of digital planetary images of many terrestrial planetary bodies, including icy satellites. Imaging science expertise includes orbital and landed camera instrument development and data processing, data engineering and informatics, planetary remote sensing at UV to RADAR wavelengths, and cartographic and geospatial data analysis and product development.

Most (but not all) of the data relevant to VISOR is available from the Imaging Node.

https://pds-imaging.jpl.nasa.gov

Below is a table showing the primary (top-level) web pages for the data for each mission.

Mission	Link
MER	https://pds-imaging.jpl.nasa.gov/volumes/mer.html
PHX	https://pds-imaging.jpl.nasa.gov/volumes/phx.html
MSL	https://pds-imaging.jpl.nasa.gov/volumes/msl.html
InSIGHT	https://pds-imaging.jpl.nasa.gov/volumes/insight.html
M2020	https://pds-imaging.jpl.nasa.gov/volumes/mars2020.html
M2020 MEDA	https://pds-atmospheres.nmsu.edu/PDS/data/PDS4/Mars2020/mars2020_meda/
MSAM	https://pds-imaging.jpl.nasa.gov/data/individual_investigation/deen_pdart16_msl_msam/

Within each mission are a variety of “Data Volumes” (PDS3: MER, PHX, MSL) or “Bundles” (PDS4: InSight, Mars 2020, MSAM) which are generally split out by instrument and sometimes type of data (e.g. EDR vs RDR). In general, each bundle/volume contains the following directories:

Directory	Purpose
browse	Images in commonly used formats such as TIFF, JPEG, and/or PNG. The contents will be laid out in a similar manner to `data`. The image may have some level of compression applied to their encoding and has limited metadata.
calibration or calib	Camera models, flat fields, ILT tables, etc, as used for calibration of the images.

data	Image data files, generally organized by Sol.
document	Supporting instrument SIS, volume/bundle documentation, and reports
extras or miscellanous	Additional browse images and thumbnails with higher compression, Velocity templates, etc.
index	Data contents database (PDS3 only)

See the PDS documentation for more details.

If you need to bulk-download a large amount of data, contact the Imaging Node directly.

Note that not all imaging data is at the Imaging Node. For example, the Mars 2020 MEDA images are at the Atmospheres node (see link in the table above). Additionally, some M20 instrument-specific data for PIXL, SHERLOC, and SuperCam are at the Geosciences node (but that data is not necessarily compatible with VISOR directly). See https://pds-geosciences.wustl.edu/missions/mars2020/index.htm .

The MER, MSL, and M2020 data consist of an image in VICAR/ODL format, with a separate PDS3 or PDS4 label file. PHX and InSIGHT consist of VICAR files (no ODL) and a separate PDS3 or PDS4 label file. For use with VISOR, the separate label file is not used - the VICAR file itself contains all the necessary information. You may wish to obtain the detached PDS label file for other reasons, but it is not used by VISOR.

Occasionally, data may be available in PDS3/ODL format only, with no VICAR label. For example, some Pancam, Mastcam, and Mastcam-Z data produced by the instrument teams are this way. For these, the image (with the attached ODL-format label) can be converted back to VICAR using the transcoder. In most cases, this preserves all the metadata and makes the file usable with VISOR. This is not guaranteed though; not all such images have sufficient metadata. Such data can be converted via:

```
java -Xmx1024m jpl.mipl.io.jConvertIIO inp=file.img out=file.vic format=vicar
ri=true xsl=file.xsl
```

Where file.xsl is a pointer to one of the converters here:

```
$V2TOP/java/jpl/mipl/io/xsl/PDStoVicar*.xsl
```

Pick one that is closest to your mission (for M20, if there is not a specific M20 one, use MSL). There's some chance you might need to add:

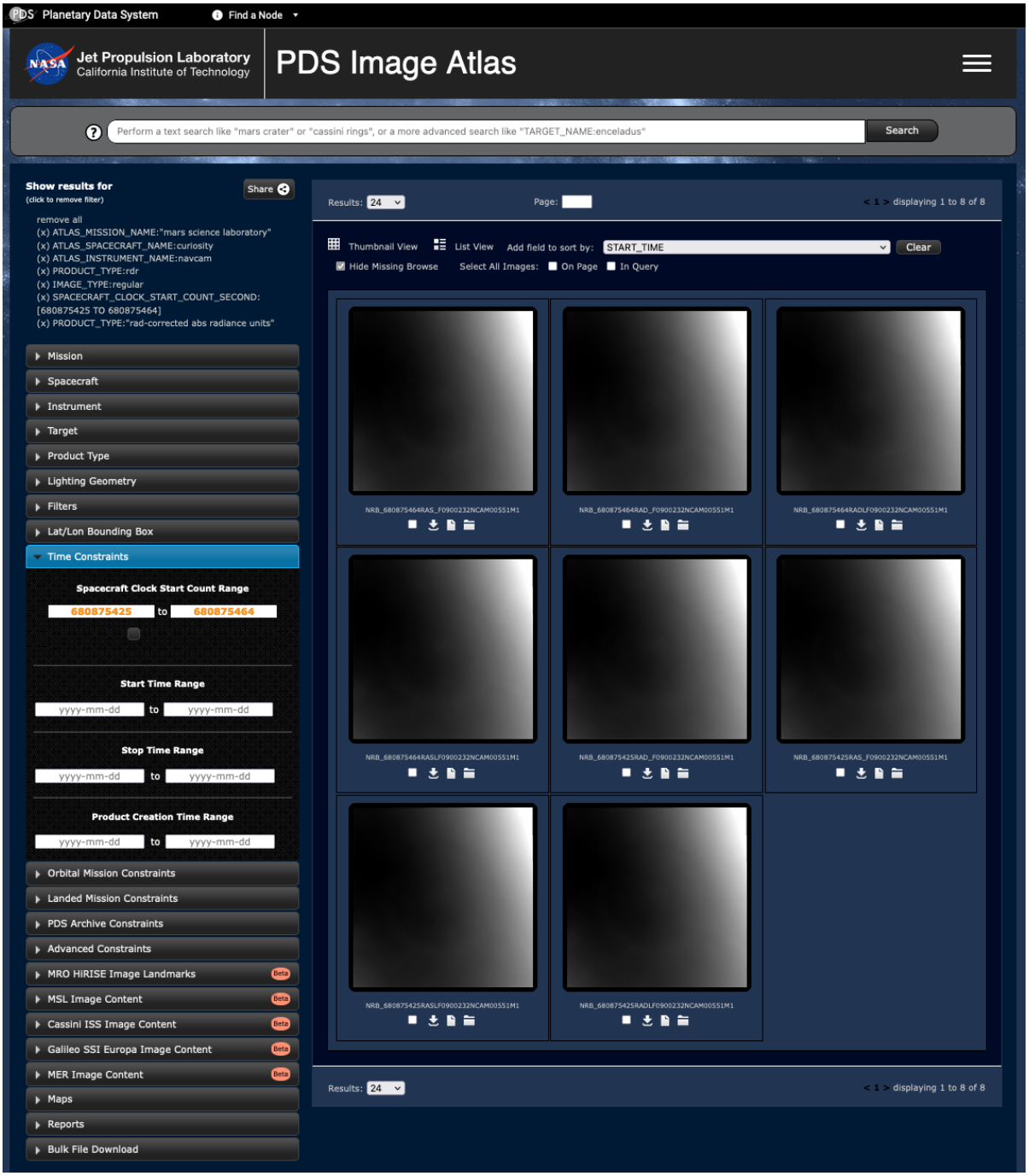
```
-Djavax.xml.transform.TransformerFactory=org.apache.xalan.processor.Transform
erFactoryImpl
```

to the command line, depending on your Java setup.

3.2 Searching for Data via Web Browser

Mars data, along with various other orbital and landed data, can be searched, browsed, and downloaded via the web tool 'PDS Image Atlas' at https://pds-imaging.jpl.nasa.gov/search/

The tool allows a search to be narrowed down by mission, spacecraft, instrument, target, time, and a number of mission and PDS-specific parameters and constraints. Downloading files is done either on a per-image basis or bulk using file selection and an autogenerated download script. The bulk download script supports Windows, MacOS, and Linux/Unix, and includes an instructions screen for first-time users. The `wget` command is used by the script on the command-line, so ensure that it is installed and working.



4. PRODUCT OVERVIEW

This section talks in broad terms about the more common products made for the Mars missions. See the individual mission SISs for more detailed information on each product. Programs listed are in `$MARSLIB` unless otherwise specified.

4.1 EDRs and FDRs

The EDRs (Experiment Data Records) are the original images delivered by the spacecraft in telemetry, with minimal processing. Some missions (notably M2020) have multiple individual products that are collectively called EDRs. These products are the starting point for VISOR processing.

Mars 2020 introduced the concept of FDR (Fundamental Data Record), which is a common starting point for downstream processing, after the basic EDR processing (which consists of decompanding, debayering, and tile reassembly). For M20 data, this is the product you should start with for most operations (such as mosaics).

4.2 Bayer processing and Color

Color images are produced in two primary ways for VISOR missions. The first is a filter wheel, used on MER Pancam and Phoenix SSI. For these, images of the same scene taken through different filters are combined into a color image. Simple approximate color can be obtained by e.g. just combining Left filters 2,5,7 with `$R2LIB/viccub`. More sophisticated processing is possible, see for example `$R2LIB/spec2xyy` and the oddly-named `$R2LIB/xyy2hdtv` .

The second way is with a Bayer pattern on the CCD. This is used on MSL Mastcam, and many of the cameras on InSight and Mars 2020. The Bayer pattern must be converted to a color image before use. Sometimes that is done onboard, sometimes the raw Bayer is sent down (see `marsdebayer`).

InSight and to a lesser extent M20 also have color correction, see `marscolor`. We do not claim “true” color; that is a topic for entire papers. Rather, we can get “approximate” true color in some cases. See the use cases for both methods.

4.3 Radiometric Correction

Radiometric correction is the process of correcting the data for radiometric effects, i.e. re-creating the scene as it was when it entered the camera lens (measured in physical units, typically W/m^2/sr/nm). This typically includes flat-field correction, exposure time compensation, and temperature responsivity correction, but it can also include corrections for dark current, shutter smear, bad pixels, and other effects. Radiometric correction is accomplished with `marsrad`, or can be done on-the-fly by the mosaic programs.

4.4 Photometric Correction

Photometric correction is the process of adjusting the scene for lighting effects, so it (ideally) looks the same regardless of lighting. VISOR currently has only limited photometric correction, consisting of Zenith correction. This adjusts the scene for the overall brightness based on solar elevation and (optionally) tau (a measure of atmospheric opacity). See the [M2020_Cam_SIS] for more on zenith correction. Images taken at different times of day typically work better in a mosaic when zenith corrected. Zenith correction is an option on `marsrad` and the mosaic programs.

4.5 Stereo Correlation

The stereo correlation process can be described simply: For each pixel in one image (the reference), find the location (to subpixel accuracy) of the pixel that best matches it in the other. Because the values of individual pixels are not unique enough to match, this process happens over a small area around each pixel (7 pixels high by 11 wide for most cases). By convention, the left image is the reference image and the matching location is found in the right image, so "left" and "right" will often be used thus. The actual left and right images may be swapped with no change to the process.

Finding the matching pixel is accomplished by searching the right image for an area that best matches the template (the area around the left image's pixel), using cross-correlation of the template with a similar window on the right side. The central pixel of this area is the desired result. The difference between the coordinates of the matching pixel in the left and right images (in line/sample terms) is called the stereo disparity, and directly relates to range from the camera. Simple enough in principle, but the devil is (as always) in the details [Deen2005].

Quite a number of programs are involved in stereo correlation (see the use cases), among them `marsjplstereo`, `marsecorr`, `marscor3`, `marsdispcompare`, and `marsmask`.

4.6 XYZ/Range Data

The fundamental product for all terrain modeling is the XYZ image. This is an image which contains the coordinate in Cartesian XYZ space of each pixel in the corresponding input image. From this XYZ image, all other terrain products are derived. The XYZ data is sometimes called a "point cloud", but, because it is stored in a raster format matching the image, additional information about pixel connectivity is implicitly present.
XYZ data is produced by taking the correlation results and projecting rays from each camera into space using the camera models. Where the rays come closest to crossing is the XYZ coordinate for that pixel. See `marsxyz`.

Closely related to XYZ images are Range maps, which provide the range to each pixel from a given point (generally either the camera C point, or the origin of a coordinate frame). They are computed by `marsrange` as the Cartesian distance between that point and the XYZ coordinate for each pixel.

4.7 Rover Masks

Included in the PDS data for most missions are a set of rover masks. These indicate where the rover is in the image, with some margin. They can be used to remove the rover from consideration for mosaics or meshes.

Unfortunately, the programs that create the mask use parts of the flight software (FSW) and are thus not available in the Open Source release. However, the xml files that these programs create are delivered for some missions; the `marsfilter` program can create the mask from these object descriptions.

4.8 Terrain Meshes

Terrain meshes are created out of XYZ data and the corresponding image. They decompose the scene into a set of triangles, which can be used by 3D rendering programs to represent the terrain. The connectivity between pixels in the XYZ image format is an important part of this

step. See `marsmesh`.

4.9 Surface Normals

Surface normals provide the orientation of the surface as a unit vector. They are computed by fitting a plane to a patch of pixels (see `marsuvw`). The size of the patch is important; typical missions make surface normals at small scale (rocks and terrain features) and at rover scale (used for slope analysis when the rover drives).

4.10 Slope maps

Measuring the slope of the surrounding terrain is important for rover safety. It is easy to misinterpret the slope on which the rover sits, because the camera tends to be tilted on the same slope and thus the images often look flat. Most rovers cannot safely handle slopes more than about 30 degrees, and controllers prefer less slope than that.

There are actually several different slope products, created by `marsslope`. The first and most important is simply the overall slope, in degrees from horizontal. This same value is also presented as slope magnitude (cosine of the slope, 0-1). Slope heading provides the azimuthal direction toward which the slope tilts.

Some missions also compute other slope-related products. Slope in Rover Direction is the component of slope in the direction of the rover (radially). It was created by MER when Spirit was having difficulty with its wheel, to compute the amount of climb from the current location. Solar Energy indicates how much energy is available at local noon based on the season. Slope Northerly Tilt contains the component of slope that points North, which is also used for solar energy estimation.

4.11 Roughness

Surface roughness is a measure of the peak-to-peak variation of the surface from an ideal plane within an area. This is used to check safety for the drill and dust removal tool (DRT). Roughness is computed by two programs, `mslrough` (which is not entirely MSL-specific, despite the name), and `marsrough`. Mars 2020 also has a surface curvature product (computed, oddly enough, by `mslrough`).

4.12 Reachability

Robotic arms on rovers and landers typically have multiple instruments on them. Reachability maps analyze, for each pixel, the possible kinematic states of the arm and terrain interactions in order to determine whether it can “reach” that pixel for each instrument in each of the several kinematic configurations (e.g. elbow up or down). Unfortunately, the kinematic analysis routines are embedded in the FSW, making these programs not available in the Open Source release. An exception is `nsytwksp`, which provides reachability for the InSight arm without the FSW (and is therefore available).

4.13 Goodness

A set of goodness products for some missions combines roughness and reachability data together into a single metric indicating whether the terrain is “good” for something, such as abrading or coring. These are computed by `marsgood`.

4.14 Instrument/Helicopter Placement

A series of products indicates whether a device (seismometer, cover, or HP3 instrument on InSight, helicopter for Mars 2020) can be deployed to a given pixel in the image. These products typically look at the specific footprint of the instrument at various "clock" (rotation) angles and how it intersects with the terrain. On InSight the programs are `nsytfoot`, `nsytgood`, `nsytrough`, and `nsyttilt`. These have been replaced by multimission versions for M20 (which still work with InSight data and are thus recommended): `marsifoot`, `marsigood`, `marsirough`, and `marsitilt`.

4.15 Mosaics

Mosaics provide a unified view of the terrain by compositing multiple images. There are six mosaic projections, described below. Mosaics are described in much more detail in Section 7.

4.15.1 Cylindrical Mosaic

A cylindrical projected mosaic exhibits rows that are constant lines of elevation and columns that are constant azimuth. This is the standard mosaic projection for non-stereo in-situ views. Cylindrical mosaics are produced by `marsmap` with the `-cyl` option.

In order to view a mosaic in stereo, separation must be maintained between the left and right eye views. Mosaics must be computed from two different points of view as viewed from a single camera. If the camera is in two suitable places, then the result can be stereo. However, this only works for limited fields of view (not panoramas).

Cylindrical projection *cannot be used* for stereo panoramas! This is because cylindrical projections stem from a single point of view. If it is moved over for stereo, it works ahead and behind but with a loss of stereo separation to the sides. Simply projecting left and right eye views to the same stereo projection does not give proper depth. This results in a visual "wall" with bumps on the wall due to deviations from the surface model, which looks very unnatural.

4.15.2 Cylper (stereo) Mosaic

In a Cylindrical-Perspective Hybrid (Cylper) mosaic, each column has its own camera model from its own point of view (POV).

Cylindrical-perspective hybrid projection is suitable for stereo panoramas. This is because each column of the output mosaic is a perspective projection from a different POV. These POVs describe a circle in space as azimuth changes that match how the cameras move. This maintains stereo separation between the eyes. This results in a stereo output that looks natural – a flat plane extending to the horizon, with height variations on it. Perfect viewing requires tuning the disparity at the horizon to the viewer's interocular distance (the distance between their eyes). Doing this causes the horizon to appear to be at infinity.

Cylper mosaics are created by `marsmcauley`.

4.15.3 Polar Mosaic

Polar mosaics show lines of equal elevation as rings around the center (nadir). They thus provide a continuous view of the horizon. They are created by `marsmap` with the `-polar` option.

4.15.4 Vertical Mosaic

A vertical mosaic projects the image onto a flat plane. This gives an impression of an overhead view, but can suffer from severe distortion if the scene does not match the plane (surface model). In particular, rocks can have extreme "layover" effects and appear elongated away from the rover. Nevertheless, it is a useful projection for a quick overhead view. They are created with `marsmap` using the `-vert` option.

4.15.5 Orthorectified Mosaic

Orthorectified mosaics use XYZ data to create a "true" overhead view of terrain as depicted in visible imagery data. These mosaics take into account the XYZ locations of each pixel to create a true overhead view, with gaps in the invisible areas behind rocks and the areas occluded by the rover. The flat-plane surface model is no longer needed, and the distortion evident in a vertical projection vanishes. However, ortho mosaics only work where there is stereo data. Ortho mosaics are created using the `marsortho` program.

4.15.6 Perspective Mosaic

Perspective mosaics use a pinhole camera model for the output image. They thus provide the most natural view for small viewing angles. However, the size of the image goes to infinity as the field of view approaches 180 degrees. Perspective mosaics are created using the `marsmos` program.

4.15.7 Overlay mosaics (XYZ etc)

Mosaics need not be made from image data. They can in fact be made from any of the raster-form products above, including XYZ, slope, goodness, etc. There are two primary ways to make them. One is to simply rerun the mosaic program, giving it a list of e.g. XYZ files instead of image files. The other is to use the IDX/ICM files created in more recent missions (InSight/M20) and use the `marsremos` program to "remosaic" the overlay images into the same geometry.

4.16 Terrain Meshes

Terrain meshes provide the 3D geometry of a scene by use of triangles with XYZ coordinates for each vertex. An image is draped over this geometry (the "skin") to provide visual texture. Meshes can be used in 3D rendering programs to view the scene independently of where the camera was originally located. Meshes can be made from in-situ images, or from orbital images covering the area of interest. The program `marsmesh` makes meshes. A later section describes meshes in much more detail.

5. VICAR BASICS

Using VICAR effectively requires an understanding of some fundamental concepts. A summary of these is provided below. See the VICAR Quick-Start Guide [VICAR_QS] for more complete information.

5.1 Setup

VICAR requires a number of environment variables, which provide location independence. The setup scripts are designed to run with `csh` (C-shell), or better yet `tcsh`. Once the env vars are set up, you may start other subshells of different types (`bash`, `ksh`, etc) that inherit these env vars. Users may also create their own setup for whatever shell they prefer, which set the needed env vars. However, working with `csh`/`tcsh` is recommended.

Here is the basic user setup, where V2TOP is the top of the VICAR tree:

```
setenv V2TOP /usr/local/vicar/v1.0/vos
source $V2TOP/vicset1.csh
source $V2TOP/vicset2.csh
```

Note that there is a VICAR "shell" (called TAE) that can be used, but this is not generally used with Mars programs. See [VICAR_QS] if you want to use TAE.

5.1 PDF Files

Each VICAR application program has associated with it a .pdf file of the same base name (thus the program "label" has "label.pdf"). These Parameter Definition Files ***are NOT Adobe Portable Document File PDF's!*** (Unfortunately, Adobe chose the same name we had been using for years.) The VICAR PDF files are plain ASCII text files.

VICAR PDF files contain program-readable descriptions of each program parameter – data type, valid values, default, etc. They also – more importantly – contain the help for the program. This help is the primary documentation for individual programs.

The first part of the PDF contains the parameter definitions. These show all the parameters available to the program, with valid values, counts, and data ranges. It is the first place to go if you are familiar with the program to remember what a specific parameter is. These are read at runtime by the executable, so the PDF has to always be in the same directory as the executable.

The second part of the PDF is the help. It is in three sections. The first is overall program documentation. The second, starting with a ".level1" line, contains a short description of each parameter. The third, starting with ".level2", contains a more complete description of each parameter.

The overall PDF help describes the program, its operation, algorithms, caveats, etc. in detail. This PDF help should be the primary source of information for any given program. Note that many R2LIB PDF's were written in the VAX/VMS days, so examples often use VMS file paths, TAE syntax, etc. These should be easily translatable to Unix equivalents. This is not a problem for the MARSLIB programs.

You may occasionally find "procedure" PDFs (distinguished by the first line of the file). These

are TAE scripts and are not generally needed for Mars programs (see [VICAR_QS]).

5.2 Running Programs

Mars programs reside in `$MARSLIB`, while general programs are in `$R2LIB`. Programs can be invoked via:

```
$MARSLIB/programname [parameters]
```

Parameters take one of several forms:

`value` : just the keyword value. These are positional parameters and must be in the order parameters are specified in the PDF file. Usually positional parameters are just used for `inp` and `out` (the first two parameters for most programs).

`name=value` : Can be used for any parameters. Parameter names are listed in the PDF.

`-value` : Sets a keyword. Can also be `name=value` but the `-value` form is more convenient for most parameters.

Complex parameter values may need to be quoted in one of several ways. Strings containing spaces or special characters (e.g. `$R2LIB/f2` function, the `point_method` parameter, `$R2LIB/label` strings) are often best handled by two sets of quotes: single quote around a double quote. The outer quote is for the shell, the inner is passed on to the VICAR parser. For example:

```
$R2LIB/f2 in out func='"in1*10"'
$MARSLIB/marscahv in out point='"cm=label,cahv_fov=min"'
$R2LIB/label -add file -prop property=identification
    item='"product_id='${var}'"'
```

Note in the last example how we pop out of the single quotes to insert a variable value.

Multivalued parameters need to be enclosed in parentheses. These are usually just quoted with backslashes:

```
$R2LIB/f2 inp=\(a b\) out=c func='"(in1+in2)/2"'
```

There are many ways to provide the necessary quoting, but the above have been found to be the most convenient for us.

5.2 VICAR File Format

The full VICAR file format is documented in [Deen1992]. This is a summary of the relevant parts for VISOR.

A VICAR image is a file of fixed-length records consisting of up to six parts:

- PDS3/ODL label (ignored by VICAR)
- VICAR label (primary label)
- binary label header - optional
- binary label prefix - optional

- pixel data
- end-of-dataset label (EOL) – optional

The binary labels are not used with Mars datasets. The EOL label is a continuation of the primary VICAR label and is used if the label expands too much after the image is written (this is transparent to users). Generally, data in PDS does not have EOL labels, but this is not 100% guaranteed. The VICAR file is treated as a series of fixed-length records, of size RECSIZE. The image area always starts at a record boundary, so there may be unused space at the end of the label, before the actual image data starts. See [Deen1992] for a full description.

The PDS3/ODL label allows an image to be simultaneously a VICAR file and a PDS3 image. This is quite common for the Mars missions. Even missions using PDS4 (such as Mars 2020) often include an ODL label, which is syntactically identical to a PDS3 label (although it need not follow PDS3 data dictionary rules). This PDS3/ODL label is completely ignored by VICAR programs and need not be present. Output from all VICAR programs are pure VICAR - without this extra label. The label is added by the transcoder if desired, see Section 5.5.

5.3 VICAR Labels

A VICAR label contains information that describes the size, origin, processing history and attributes of the associated image. All VICAR application programs are designed to read information from the VICAR labels of the input datasets and dynamically update them. VICAR labels are critical to Mars processing; the metadata contained in the label tells the programs how to work with this image.

A VICAR label is an ASCII string composed of label items which are keyword=value pairs separated by spaces.

Syntax:

```
keyword=value
```

where: **keyword** is a text keyword that identifies the label item
value is the information portion of the label item; may be of type string, integer, real, or double, and may be multi-valued

Examples:

```
NL=800
FORMAT='HALF'
SIZE=(1,1,800,800)
```

A VICAR label contains three different classes of keyword=value label items:

- System
- Property
- History

The system portion consists of those items that describe the actual image structure. These items include size of the image, its organization, its pixel format, host type and items indicating the existence of the optional sections of the label.

The property portion of the label contains items that describe properties of the image, such as

map projection, acquisition time, instrument temperatures, camera models, etc. This is the meat of the label, containing all the mission metadata. The property label is divided into property groups (each starting with the "PROPERTY=name" keyword) that help to organize the properties.

The history portion contains the processing history of the data. Each time a program processes a dataset, VICAR adds history items to the label. The history items include the name of the program (truncated to 8 characters), user identification, and processing date/time. The MARSLIB programs (and some R2LIB programs) also contain the program parameters, which allow the program call to be reconstructed. Each program invocation is marked by a "TASK=name" keyword in the label.

5.4 Viewing Images

There are two primary image display systems in VICAR: xvd and marsviewer/jadeviewer. The xvd program is recommended for general image display in most cases. Jadeviewer is useful for displaying RDR overlays and for displaying non-VICAR images (such as png or jpeg).

5.4.1 xvd

The "xvd" program is a high-performance display program for VICAR (and PDS3) images. It is written in C++ using X-windows and Motif. Using it requires an X-windows server (for Mac, look for XQuartz).

Running xvd is simple, as its location is put in $PATH for you by vicset1.

```
xvd &
```

This will bring up a file selection window, allowing you to select a file to view.

More commonly, a filename can be given on the command line. This can be a single-band or multiband (color) file. Alternatively, three files can be given, if the bands are separate:

```
xvd color.vic &
xvd x.red x.grn x.blu &
```

The trailing & puts the program in the background, freeing the shell window for other tasks. There are several options that can be provided to xvd (before the filename). The most useful are:

-min x : Sets the minimum data range for a non-byte image
-max y : Sets the maximum data range for a non-byte image
-fit : Does a zoom to fit, making the image fit the window size

Non-byte data is converted to byte for display using the data range. This is normally the minimum and maximum values in the image, but can be set with the File/Data Range menu or the -min/-max command line options. Stretches are applied *after* the conversion to byte. If an image looks all black, it's possible that the data range maximum is set too high; try reducing it.

Some other tricks with xvd. If you are given a stereo pair of images, you can view it as an anaglyph using the three-file form of the command line:

```
xvd left.vic right.vic right.vic &
```

Individual bands can be selected using (quoted) parentheses after the filename. So if you are given a stereo pair of *color* images, you can view it as a "colorglyph" like this:

```
xvd left.vic\(1\) right.vic\(2\) right.vic\(3\) &
```

which extracts the first (red) band from the left image, and the green and blue bands from the right image.

5.4.2 Marsviewer / Jadeviewer

Jadeviewer is a display system written in Java. It can thus run natively on most computers (including Windows and Mac) without X-windows. It can also be used over X-windows from a Linux server (the -Y option to ssh is often needed here - additionally, if you have trouble displaying on a Mac over X-windows, try adding `-Dsun.java2d.xrender=false` to the java call).

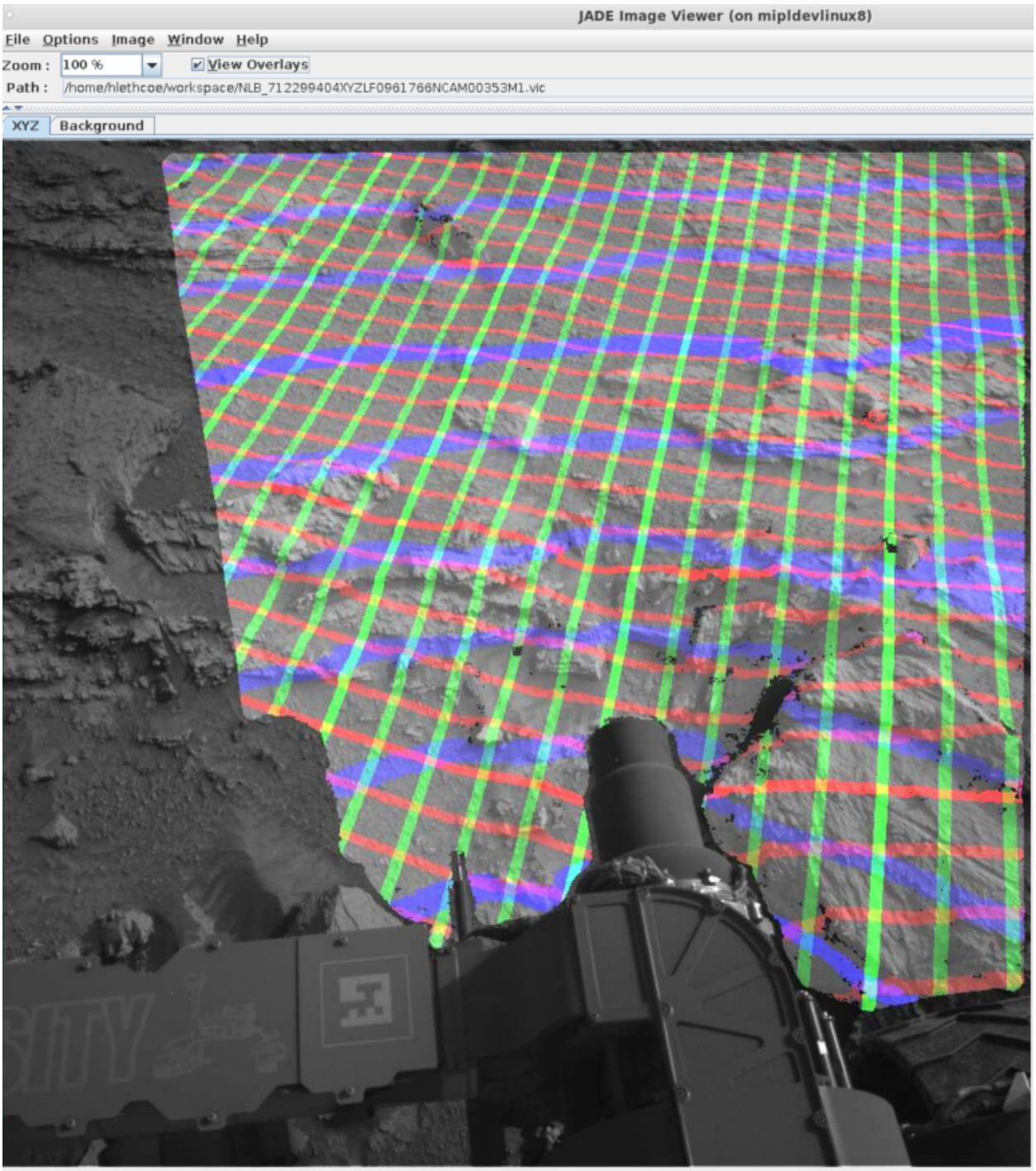
Jadeviewer can be used for generic image display. However, its primary purpose is to visualize derived products (called RDRs) generated by the Mars programs. For example, the following will display an XYZ image as a grid overlaid on a background image:

```
jadeviewer xyz.vic XYZ image.vic &
```

or as direct invocation without a wrapper script:

```
java jpl.mipl.jade.viewer.ImageViewer xyz.vic XYZ image.vic &
```

The second parameter is the 3-letter code (see Section 6.3) that tells jadeviewer how to interpret the image.



Marsviewer is a cousin of jadeviewer that understands filename conventions and directory structures for various missions. It is tremendously more convenient to use than jadeviewer if the data is properly named in a proper directory structure. The advantage of jadeviewer is that it doesn't care about filenames or directories. See Section 10.10 for information on how to set up data for viewing in Marsviewer.

5.5 Transcoder (File Format Conversion)

The Transcoder is a powerful Java program that does conversion amongst many common file formats, and can preserve metadata. It is based on the Java Image I/O package, with additional plugins courtesy of MIPL for VICAR, PDS 3, ISIS 5, and FITS images. It also has the beginnings of PDS 4 support.

To invoke the transcoder:

```
java jpl.mipl.io.jCovertIIO
```

Running it with no options will print a help list, describing each option in detail. However, the three most important are `inp=`, `out=`, and `format=`. Using these you can convert any known image format to any other, *without* preserving metadata. For example:

```
java jpl.mipl.io.jCovertIIO inp=file.vic out=file.png format=png
```

For a list of known formats, run it with "`plugins`" as the only argument.

The `2rgb=true` argument will convert a single-band input file to color for those formats that are naturally color (such as jpeg).

It's one thing to convert pixel data between formats - useful, to be sure, but not particularly unique. The power of this system is its ability to preserve the metadata (labels) while doing (certain kinds of) transforms.

Metadata-preserving transformations are implemented using an XSL stylesheet that says how to convert the metadata between formats. Several of these are provided in `$V2TOP/java/jpl/mipl/io/xsl/` . The most important of these are `VicarToPDS*.xsl`. These convert from VICAR to PDS3/ODL format and are how we create the dual-labeled products for Mars surface operations and archive. Use the highest numbered version available for the correct mission. Note that there are not a lot of mission dependencies here, so if there is not a version for the particular mission you are working with, try another (in general the later missions, such as m20, are more complete).

For example, here is a script that will create the MSL dual-labeled files, along with a PDS 3 detached label:

```
#! /bin/csh
# Simple script to transcode (vicar -> pds/odl) and image.
set base = ${1:r}

java -Xmx1024m jpl.mipl.io.jCovertIIO inp=$1 out=${base}.IMG xml=false
format=pds embed_vicar_label=true ri=true
xsl=$V2TOP/java/jpl/mipl/io/xsl/VicarToPDSmsl_Blob_ODL12.xsl
pds_label_type=ODL3

java -Xmx1024m jpl.mipl.io.jCovertIIO inp=${base}.IMG out=${base}.LBL
format=pds pds_detached_only=true ri=true
xsl=$V2TOP/java/jpl/mipl/io/xsl/VicarToPDSmsl_Blob_ODL2PDS_10.xsl
pds_label_type=PDS3
```

You can also use the transcoder "backwards", to convert PDS3 labeled images to VICAR. Use

one of the `PDSToVicar*` stylesheets for this (for M20, use either the MSL or MSAM ones). For example:

```
java jpl.mipl.io.jConvertIIO inp=file.pds out=file.vic format=vicar ri=true
xsl=$V2TOP/java/jpl/mipl/io/xsl/PDSToVicarMSAM.xml
```

The transcoder is also used for the now-critical PDS4 label creation using the Velocity templates. The script $V2JBIN/LabelImage will create a detached PDS4 label for VICAR/ODL input images:

```
$V2JBIN/LabelImage image.vic
```

There are other scripts in $V2JBIN to do things like create labels for meshes, browse products, etc. A full treatment of Velocity templates is beyond the scope of this document; consult the PDS documentation or [Deen2019]. It is however an exceptionally powerful way to create PDS4 labels for most any data set.

The Labelocity tool is especially helpful when dealing with PDS4 labels, see [Deen2023] and [Labelocity].

6. VISOR BASICS

This section talks about aspects of using VISOR that are common across most of the programs.

6.1 PDS and ODL Labels

The MER project required that all science instrument products, including image products, be generated using the PDS3 file format. However, the legacy software was written to use the VICAR file format, which was developed at MIPL and had been used for decades by MIPL software. To preserve the considerable software heritage, a ‘‘dual-labeled’’ file format was designed so that both a PDS3 and a VICAR label were attached to the images (PDS3 first, then VICAR).

The VICAR I/O system was modified to recognize and skip over the PDS3 label on read, while the application programs would write pure VICAR format with a separate ‘‘transcoder’’ program to add an equivalent PDS3 label afterward. The PDS3 format already supported a mechanism to skip over the embedded VICAR label. So, the dual label design accommodated MIPL software written to use only the VICAR label as well as software developed outside of MIPL written to use only the PDS3 format. This had the additional advantage of making available the large base of legacy image processing applications written for VICAR.

As a note, the only significant part of the VICAR label that was not duplicated in PDS3 (and vice versa) was the history label. This label is occasionally useful, since it lists the parameter values used by programs when the data was processed, and will be especially useful with the release of the Mars software (see the section History Label and Reconstructing Processing below). Phoenix used the MER model.

On MSL, things got a little messier. PDS3 is implemented using a format known as ODL (Object Description Language). PDS got more strict on MSL than it was on MER, meaning the embedded PDS3 label was not actually PDS3 compliant (mostly due to a local data dictionary with a MSL: prefix on keywords), yet a large amount of software was written for MER using the attached label and users wanted to use that software for MSL. So a triple label resulted. The attached label looks the same as on MER, but is called an ODL label rather than a PDS3 label. A separate, detached PDS3 label was created. Syntactically ODL and PDS3 are identical; semantically the difference is that the PDS3 label conforms to the PDS3 data dictionaries while the ODL label does not.

Mars 2020 moved to PDS4, which is an XML label format. The dual ODL/VICAR attached label was retained. The PDS3 detached label was removed, and a PDS4 detached label took its place.

InSight dispensed with the ODL label, containing just the VICAR label and a detached PDS4 label.

In all cases, great pains are taken to ensure that the semantic content of all labels are identical for a given file. In general the VICAR label is the source, and the ODL, PDS3, and PDS4 labels are generated from the VICAR label. As a result, you can look at any label to find the information you need.

Some products generated by teams other than MIPL may not have the embedded VICAR

label. If a PDS3/ODL label exists, the transcoder can be used to create a VICAR label for use with the Mars programs (Section 5.5). As of this writing, no PDS4->VICAR converter exists, but no known Mars data is written without either a VICAR or PDS3/ODL label.

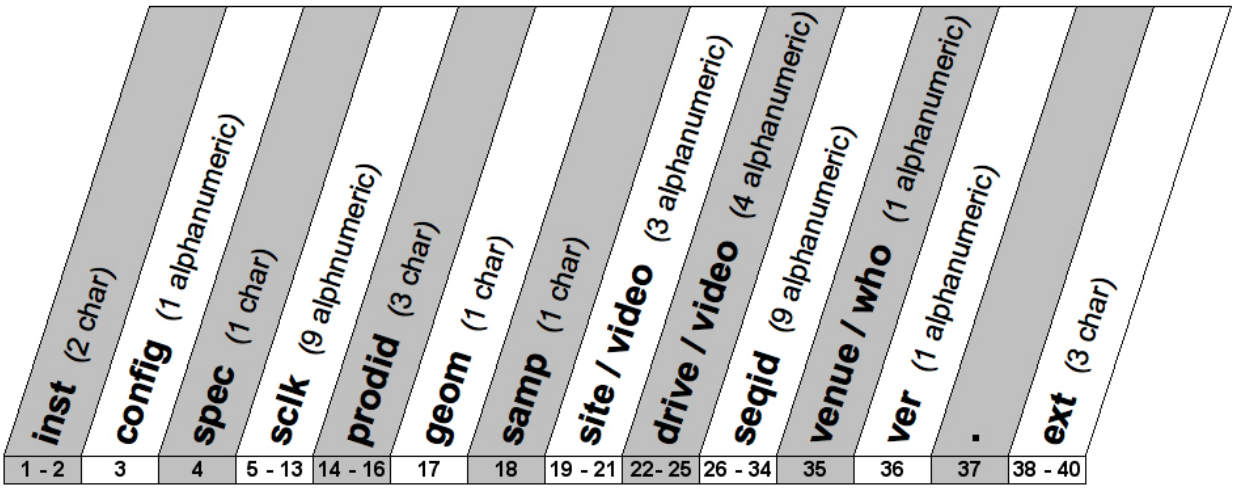
It should be noted that Mars Pathfinder was processed using VICAR labels but these were stripped to become only PDS3 labels when the data was sent to PDS. This caused any number of issues, because the PDS data was no longer compatible with the processing programs, and that lesson learned became the driving force behind the multiple label scheme. There is limited support for Mars Pathfinder in PIG, but you have to find the VICAR files. It should be possible to write transcoder XSL templates to convert MPF data back to VICAR, but this has not yet been done.

6.2 Filename Conventions

The filenames used by the Mars products in PDS are rather long and complex. However, there is a lot of very meaningful information in there, once you know how to read them. It may look intimidating, but once you learn what's there, it makes working with the data much, much easier.

Each mission describes its specific conventions in the mission's camera SIS. Users should find the convention for their mission and study it. Note that there are typically different conventions for single-file images, meshes, and mosaics.

Here we break down a MSL single-frame image filename, as an example. Refer to section 6.1.1 of the MSL Camera SIS. Here is the summary chart, reproduced as an example.



Once you learn the patterns, you won't need to count characters, but can instead visually go right to the field you want. In general, the filenames alternate between numeric and alpha fields as much as possible to help with this visual recognition.

Here's a sample MSL filename:

```
NLB_712299404XYZLF0961766NCAM00353M1.IMG
```

Using the above, we can break this down into the individual fields:

inst = "**NL**" = Navcam Left

config = "**B**" = "B-side" configuration
spec = "**_**" = No special processing method
sclk = "**712299404**" = Spacecraft Clock Start (time since epoch)
prodid = "**XYZ**" = XYZ RDR
geom = "**L**" = Linearized
samp = "**F**" = Full Frame
site = "**096**" = Site 96
drive = "**1766**" = Drive (Position-within-Site) 1766
seqid = "**NCAM00353**" = Command Sequence ID
venue/who = "**M**" = Producer is MIPL at JPL
ver = "**1**" = Version 1
ext = "**IMG**" = Image product with ODL label

Referring back to the full filename, the "XYZLF" in the middle stands out as alphas amongst a sea of numbers. This is the 3-letter code (see below) with some related flags. The sequence ID also stands out. Once you learn that the site/drive are between those, and the SCLK is in front, you've learned to successfully decode the filename.

Almost all missions have instrument, eye, sclk, 3-letter code, site/drive, sequence ID, and version, in some order. Various other flags are also common across missions. Observe the patterns and you'll be well on your way to being a filename expert.

6.3 Three-Letter Codes

There are many different product types that can be made with VISOR. These are identified by a three-letter code. These codes are used throughout this document, for example XYZ, UVW, DSP, EDR.

The three-letter code indicates the format of the data (e.g. is it XYZ, surface normal, disparity, image, or what), as well as other salient characteristics, such as the frame in which the data is measured, the type of radiometric or color correction, the correlation stage, etc. The DERIVED_IMAGE_TYPE label describes just the format of the data without the other characteristics; this label would be useful for example in a visualization program to display the data in the correct manner.

We have attempted to standardize these three-letter codes across missions as much as possible. The primary products are largely the same. However, there are some differences and name collisions amongst the less commonly used products between missions.

Each mission lists the three-letter-codes it uses in the SIS, generally in the section on filename convention. On M20, for example, there are 99 of them and they are listed in Tables 17-2 and 17-3.

MER and PHX are slightly different. There is still a three-letter code, and for non-linearized products the three-letter codes generally match the later missions. However, the 3rd letter is overloaded as a linearization/thumbnail flag. So for example, linearized XYZ data is called XYL. For MSL and later missions, this proved insufficient, so we went to a full 3 letter code and moved the linearization flag to a separate character (sometimes you will see references to e.g. XYZL which is the 3-letter code plus the linearization flag).

6.4 List Files

A list file contains a list of images to be used for processing (typically, this is for mosaics). A list file is simply a list of filenames, one name per line. The names can be full pathnames or relative to the current directory. Lines starting with "`#`" are comments and are ignored.

List files are a convenience. It is possible in most cases to actually put each of the files on the command line, for example:

```
$MARSLIB/marsmap inp=\(a.img b.img c.img d.img e.img\) …
```

but this quickly gets unwieldy. Instead you can simply do:

```
$MARSLIB/marsmap inp=mosaic.lis
```

where `mosaic.lis` looks like:

```
# optional comment
a.img
b.img
c.img
d.img
e.img
```

Most programs accept list files for the INP parameter and occasionally another (e.g. the RINP parameter for marsmcauley). This is true even for programs that take just one or two input files. It is possible to supply the left and right images to the correlator as a list file rather than listing them separately for example, but this is rarely done.

The order of the images in the list is important. In the context of a mosaic, the first image generally goes "on top" of the mosaic, so it hides subsequent images where they overlap. Thus, ordering changes where seams occur in a mosaic.

6.5 Camera Models

Each camera has associated with it a camera model, which describes the geometry of the camera. Specifically, the relationship between pixels and XYZ coordinates in the real world. It allows you to take an XYZ point in space and project it into the camera to the specific line,sample coordinate where that point appears in the image. Conversely, it allows you to take a line,sample coordinate in the image and project it out into space, specifying the origin and direction of the ray along which the light for that pixel came (you cannot get an XYZ coordinate directly because you do not know the distance; that's where stereo processing comes in).

The camera model is foundational for most of what VISOR does. Without it, we really can't do much with the image. It's worth noting that some cameras do not have a camera model (such as the M20 SkyCam and CacheCam), so you will find little in terms of RDRs from these instruments other than some radiometric or color correction.

It should be noted that as used in VISOR, the term "camera model" refers to the actual numbers that specify a particular model, while "camera model type" refers to the math that uses those numbers to do the projection (for example, CAHV, CAHVOR, CAHVORE, PSPH). The type used is implicit in the camera model by the number and names of parameters.

There are two types of camera model parameters: internal and external. Internal parameters describe properties of the camera itself: focal length, pixel pitch, lens distortion, etc. External parameters describe how the camera relates to the world, i.e. where it is located and how it is pointed. The CAHV-family of models conflates those two; for example the H and V vectors encode both focal length and pixel scale, and the rotation of the camera in 3D space. As a result, VISOR generally does not distinguish between external and internal parameters.

Calibration camera models describe the state of the camera when the calibration data was acquired. They thus point (external parameters) at the cal target when calibration was run. These models are transformed into the actual camera model for an image by first "unpointing" the calibration model so it points down an axis and then "pointing" it based on the pointing parameters (azimuth, elevation, arm position, etc). This transforms both the orientation and position of the camera model. The actual camera model, which is what is stored in the image label, thus describes the external (and internal) parameters applicable to that specific image.

See [M2020_Cam_SIS], Section 8.3, for a description of how mast kinematics is used to point the camera model for Mars 2020 (the general concept is applicable to most missions).

6.6 Calibration Files

The PDS delivery for each mission contains the set of calibration files used for that mission. The specifics vary per mission but below is a summary of the types of files that generally exist.

`camera_models` : Contains the calibration camera models
`flat_fields` : Contains the flat fields for each camera. Also dark current and other related files for some cameras.
`ilut` : Contains the inverse lookup tables used to decompand data
`param_files` : Contains various parameter files, including the camera mapping file (basic characteristics and serial number of each camera), point files (files containing kinematics parameters for pointing cameras), radiometric correction parameters, color coefficients, rover filters, workspace definitions, etc.

Some missions have other calibration directories, see the mission documentation.

VISOR accesses calibration files via the `MARS_CONFIG_PATH` environment variable. This is a colon-separated list of directories (similar to the standard Unix `$PATH`) in which to search for calibration files. It should contain the top-level cal dir, i.e. `camera_models`, `param_files`, etc should be children of the directory listed in `$MARS_CONFIG_PATH`.

The calibration files are all named using the mission name (or testbed venue) as a prefix, so calibration files for all missions can coexist in one `$MARS_CONFIG_PATH` (this is how we do it at MIPL).

For convenience, this release contains the cal dirs for each of the 6 primary missions supported by PIG at the time of this writing. It is important to realize, these are just a snapshot; the cal dirs for active missions may continue to be updated throughout the mission (or even after). So, while these calibration files are useful for processing most relevant data in PDS, they may not be the most up-to-date.

If you'd like to use your own calibration data, we strongly recommend not changing the existing cal dirs, but rather augmenting them. Create a directory with the appropriate subdirs, install just your updated files, and put that dir at the front of $MARS_CONFIG_PATH. Since the path is searched in order, it will find and use your files in preference to the standard ones.

For example, if you wanted to have your own flat field for the M20 Left Navcam, you'd do the following. First, look in `param_files/M20_camera_mapping.xml` in the M20 cal dir to determine the serial number of the camera, in this case 0103. You can find the existing flat field in `flat_fields/M20_FLAT_SN_0103.IMG` (or `M20_FLAT_SN_0103_M_RAWBAYER.IMG`). Create a new flat field file, and then do the following to install it (assumes `$MARS_CONFIG_PATH` is already set up normally):

```
mkdir /home/myuser/mycal/
mkdir /home/myuser/mycal/param_files
cp /home/myuser/awesome_ncam_flat.IMG
/home/myuser/mycal/param_files/M20_FLAT_SN_0103.IMG
setenv MARS_CONFIG_PATH /home/myuser/mycal:${MARS_CONFIG_PATH}
```

Then if you run $MARSLIB/marsrad on a navcam left image, it should use your new flat field (study the stdout output, the cal files used are often reported, to make sure you got the right one).

6.7 Pipeline processing summary

In operations, data is processed using a pipeline. There are at least 3 major pipeline technologies used by the 5 primary flight missions. The pipeline itself is not being open sourced (some of the pipelines, such as CWS used by M20, are independently open sourced, although not the M20 adaptations as of this writing). For that reason, only the highest level overview of pipelines is presented here.

Basically a pipeline routes data through processing programs. At some points (like to do stereo analysis or mosaics), files must be matched up according to various criteria in order to be processed together.

The M20 SIS in particular has an extensive pipeline data flow diagram, see Appendix D (which is in a separate physical file from the rest of the document).

Since all VISOR programs can be called from the command line, just about any pipeline technology can be used - even simple shell scripts. The Coldarm mission for example is implemented only using shell scripts. See the use case on correlation, below, for an example script of this type.

If you wish to re-create a partial (or even complete) pipeline, analysis of history labels (below) can provide the sequence of calls and parameters used to build the products.

6.8 History Label and Reconstructing Processing

VICAR history labels are used to see what parameters were used during processing for a particular product. They can be used to reconstruct the program executions and parameters that went into building a product.

There are some caveats, however. First, all of the Mars programs provide their complete parameter set in the history label. However, not all of the non-Mars VICAR programs do so. Generally though, the non-Mars VICAR programs used in the processing pipeline are fairly straightforward and the parameters can be guessed at.

Second, the program name is limited to 8 characters. There are some sets of programs that match within the first 8 characters, for example marsautoloco/marsautotie/marsautotie2, marsdispcompare/marsdispinvert/marsdispwarp, marsproj/marsprojfid. Generally context can be used to disambiguate those, or the sets of parameters allowed by each program.

Finally, the filenames in the history label are the files and pathnames presented to the program. The pathnames almost certainly no longer exist (often they are pipeline-temporary directories). The filename part is usually preserved, as in the PDS product will have the same filename (except perhaps for a IMG vs VIC extension). But there is no guarantee that the file will exist in PDS; some are temporaries that are thrown out. For the most part though, the processing chain can be reconstructed via the history label.

As an example, let's walk through the processing for the provided Navcam DSP RDR (`NRB_712299404DSPLF0961766NCAM00353M1.IMG`). First, let's look at the complete history label:

```
$R2LIB/label -list NLB_712299404DSPLF0961766NCAM00353M1.IMG -history
```

The results are as follows:

```
Beginning VICAR task LABEL
LABEL version 2017-03-30
************************************************************

        ************  File NLB_712299404DSPLF0961766NCAM00353M1.IMG ************
---- Task: TASK -- User: urmslopg -- Thu Jul 28 23:43:18 2022 ----
---- Task: LABEL -- User: urmslopg -- Thu Jul 28 23:43:19 2022 ----
---- Task: MARSINVE -- User: urmslopg -- Thu Jul 28 23:44:44 2022 ----
INP=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/inverterpipe/processque/NLB_712299404EDR_F0961766NCAM00353M1.VI
C'
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/inverterpipe/output/NLB_712299404ILT_F0961766NCAM00353M1.tmp'
DATA_SET_NAME='MSL MARS NAVIGATION CAMERA 5 RDR V2.0'
DATA_SET_ID='MSL-M-NAVCAM-5-RDR-V2.0'
---- Task: MARSRELA -- User: urmslopg -- Thu Jul 28 23:44:44 2022 ----
INP=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/inverterpipe/output/NLB_712299404ILT_F0961766NCAM00353M1.tmp'
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/inverterpipe/output/NLB_712299404ILT_F0961766NCAM00353M1.VIC'
CM='CM'
---- Task: MARSRAD -- User: mslopgs -- Thu Jul 28 23:45:03 2022 ----
INP=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/radpipe/processque/NLB_712299404ILT_F0961766NCAM00353M1.VIC'
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/radpipe/output/NLB_712299404RAD_F0961766NCAM00353M1.VIC'
DNSCALE=100.0
BITS=15
---- Task: MARSCAHV -- User: urmslopg -- Thu Jul 28 23:50:24 2022 ----
INP=
```

```
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/linradpipe/processque/NLB_712299404RAD_F0961766NCAM00353M1.VIC'
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/linradpipe/output/NLB_712299404RADLF0961766NCAM00353M1.VIC'
POINT_METHOD='cahv_fov=min'
OMP_ON='OMP_ON'
---- Task: MARSJPLS -- User: urmslopg -- Thu Jul 28 23:50:43 2022 ----
INP=(
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/lstereopipe/localcache/NLB_712299404RADLF0961766NCAM00353M1.VIC
',
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/lstereopipe/processque/NRB_712299404RADLF0961766NCAM00353M1.VIC
')
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/lstereopipe/output/NLB_712299404DFFLF0961766NCAM00353M1.VIC'
PYRLEVEL=3
WINDOWSIZE=13
MAXDISP=2032
BLOBSIZE=50
POINT_METHOD='cm=label'
---- Task: MARSCOR3 -- User: urmslopg -- Thu Jul 28 23:51:00 2022 ----
INP=(
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/tmp_cache/NLB_712299404RADLF0961766NCAM00353M1.VIC',
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/tmp_cache/NRB_712299404RADLF0961766NCAM00353M1.VIC')
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marscorpipe/output/NLB_712299404DSRLF0961766NCAM00353M1.VIC'
IN_DISP=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marscorpipe/processque/NLB_712299404DFFLF0961766NCAM00353M1.VIC
'
TEMPLATE=(7, 11)
SEARCH=25
QUALITY=0.5
GORES='GORES'
GORE_QUALITY=0.6
GORE_PASSES=3
GORE_REVERSE='GORE_REVERSE'
DISP_PYRAMID=3
MODE='amoeba4'
FTOL=0.004
MULTIPASS='multipass'
FILTER='filter'
OMP_ON='OMP_ON'
---- Task: MARSDISP -- User: urmslopg -- Thu Jul 28 23:52:34 2022 ----
INP=(
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marsdispcompipe/localcache/NLB_712299404DSRLF0961766NCAM00353M1
.VIC',
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marsdispcompipe/processque/NRB_712299404DSRLF0961766NCAM00353M1
.VIC')
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marsdispcompipe/output/NLB_712299404MDSLF0961766NCAM00353M1.VIC
'
---- Task: MARSMASK -- User: urmslopg -- Thu Jul 28 23:52:35 2022 ----
INP=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marsdispcompipe/localcache/NLB_712299404DSRLF0961766NCAM00353M1
.VIC'
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marsdispcompipe/output/NLB_712299404DSPLF0961766NCAM00353M1.VIC
'
MASK=
```

```
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marsdispcompipe/output/NLB_712299404MDSLF0961766NCAM00353M1.VIC
'

************************************************************
```

This example shows 10 different programs being run. Let's walk through each and build command lines for them.

```
---- Task: TASK -- User: urmslopg -- Thu Jul 28 23:43:18 2022 ----
```

The task name "TASK" is used when the program name is unavailable. This happens for some Java-based code, but in this case it represents the telemetry processor, m20edrgen. Since the telemetry processors are not included in this release, and the data products needed as input for them are generally not available from PDS, we ignore this step.

```
---- Task: LABEL -- User: urmslopg -- Thu Jul 28 23:43:19 2022 ----
```

This is an example of a non-Mars VICAR program (`$R2LIB/label`) that does not save its parameters into the history label. Thus we cannot reconstruct specifically what this program was doing. However, it is part of the processing needed to create the initial EDR, so we can ignore it.

Note that many images have a call to `marsrelabel` at this point; that is what replaces the telemetered camera model with the one from the ground calibration files.

```
---- Task: MARSINVE -- User: urmslopg -- Thu Jul 28 23:44:44 2022 ----
INP=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/inverterpipe/processque/NLB_712299404EDR_F0961766NCA
M00353M1.VIC'
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/inverterpipe/output/NLB_712299404ILT_F0961766NCAM003
53M1.tmp'
DATA_SET_NAME='MSL MARS NAVIGATION CAMERA 5 RDR V2.0'
DATA_SET_ID='MSL-M-NAVCAM-5-RDR-V2.0'
```

This is a call to marsinverter. It takes an EDR and produces an ILT. A strict translation results in the command line:

```
$MARSLIB/marsinverter NLB_712299404EDR_F0961766NCAM00353M1.VIC
NLB_712299404ILT_F0961766NCAM00353M1.tmp data_set_name=’”MSL MARS NAVIGATION
CAMERA 5 RDR V2.0”’ data_set_id=’”MSL-M-NAVCAM-5-RDR-V2.0”’
```

There are several things to note here. First, VICAR parameters may be expressed positionally, as long as they're strictly in order in the PDF. Once you skip a parameter, positional parameters are no longer allowed. This facility is often used for INP and OUT (almost always the first two parameters), but rarely beyond that. You could have also said inp=xxx and out=yyy on the command line.

Second, parameters are not case-sensitive. You can say INP= or inp= or even InP= as you wish.

Third, complex strings (containing spaces or odd characters) need to be passed in to the VICAR

parser so the parser sees double quotes as part of the parameter value. That means quoting the double quotes somehow so they pass through the shell. The easiest way we have found to do that is to enclose the double-quoted string inside single-quotes, as shown in this example. You can pop out of the double quotes to use variable substitution in a script, e.g.

```
data_set_name=’”MSL MARS NAVIGATION CAMERA 5 RDR V’${version}’.0”’
```

There are many other ways to accomplish the same thing (shell quotes are a tremendously rich topic) but we have found the above to work the best for us.

Finally, the DATA_SET_NAME and DATA_SET_ID parameters are ***NOT*** required. They set those particular labels in the output, which are needed for PDS3. However, it is unlikely that you will be generating data for PDS3 - PDS is not accepting new data sets in anything other than PDS4 - so these parameters can be completely ignored. They did serve a purpose in explaining the quoting rules here though.

So a simpler form, omitting those parameters and (for variety) including inp= and out=, is:

```
$MARSLIB/marsinverter inp=NLB_712299404EDR_F0961766NCAM00353M1.VIC
out=NLB_712299404ILT_F0961766NCAM00353M1.tmp
```

Next:

```
---- Task: MARSRELA -- User: urmslopg -- Thu Jul 28 23:44:44 2022 ----
INP=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/inverterpipe/output/NLB_712299404ILT_F09617
66NCAM00353M1.tmp'
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/inverterpipe/output/NLB_712299404ILT_F09617
66NCAM00353M1.VIC'
CM='CM'
```

This replaces the camera model with a ground-derived model. MSL does that here; other missions do it before inverting.

Parameters that look like `CM=’CM’` are usually keyword parameters. They could be entered literally as `CM=CM` (quotes are not needed), but keywords are generally specified in the form `-keyword`, which is easier to type and read. That’s the form we’ll use here, but if in doubt you can say `CM=CM`. (The program PDF lists all the parameters; those of type KEYWORD can use the `-keyword` form).

```
$MARSLIB/marsrelabel NLB_712299404ILT_F0961766NCAM00353M1.tmp
NLB_712299404ILT_F0961766NCAM00353M1.VIC -cm
```

Next:

```
---- Task: MARSRAD -- User: mslopgs -- Thu Jul 28 23:45:03 2022 ----
INP=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/radpipe/processque/NLB_712299404ILT_F096176
6NCAM00353M1.VIC'
```

```
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/radpipe/output/NLB_712299404RAD_F0961766NCA
M00353M1.VIC'
DNSCALE=100.0
BITS=15
```

Straightforward, with numeric values for the parameters.

```
$MARSLIB/marsrad NLB_712299404ILT_F0961766NCAM00353M1.VIC
NLB_712299404RAD_F0961766NCAM00353M1.VIC dnscale=100 bits=15
```

Next:

```
---- Task: MARSCAHV -- User: urmslopg -- Thu Jul 28 23:50:24 2022 ----
INP=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/linradpipe/processque/NLB_712299404RAD_F096
1766NCAM00353M1.VIC'
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/linradpipe/output/NLB_712299404RADLF0961766
NCAM00353M1.VIC'
POINT_METHOD='cahv_fov=min'
OMP_ON='OMP_ON'
```

The `POINT_METHOD` parameter is special, see section on it below. In this case special quoting is not needed, but if you were to also need, say, `point=cm=label` they would be combined with the quote-in-quote trick, i.e.

```
      point='"cahv_fov=min,cm=label"'
```

The `OMP_ON` parameter is another keyword. It is redundant since ON is the default.

```
$MARSLIB/marscahv NLB_712299404RAD_F0961766NCAM00353M1.VIC
NLB_712299404RADLF0961766NCAM00353M1.VIC point=cahv_fov=min -omp_on
```

Next:

```
---- Task: MARSJPLS -- User: urmslopg -- Thu Jul 28 23:50:43 2022 ----
INP=(
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/lstereopipe/localcache/NLB_712299404RADLF09
61766NCAM00353M1.VIC',
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/lstereopipe/processque/NRB_712299404RADLF09
61766NCAM00353M1.VIC')
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/lstereopipe/output/NLB_712299404DFFLF096176
6NCAM00353M1.VIC'
PYRLEVEL=3
WINDOWSIZE=13
MAXDISP=2032
```

```
BLOBSIZE=50
POINT_METHOD='cm=label'
```

The program here is marsjplstereo. If in doubt you can do `ls $MARSLIB/marsjpls*` to see what the program is (or could be).

In this case we have two inputs. The first comes from the prior run, the second is included in the sample data but comes from a similar sequence of calls on the Right eye. When you have a multivalued parameter, the values have to be enclosed in parentheses. Since parentheses are special to the shell, they must be quoted. Again, there are many ways to do this, but we almost always use backslashes for the parentheses. Note that you can have either comma or spaces (or both) separating the values. It is also handy to put a space after the opening parenthesis so that shell tab-completion of filenames can happen - such spaces are allowed but not required.

```
$MARSLIB/marsjplstereo \( NLB_712299404RADLF0961766NCAM00353M1.VIC
NRB_712299404RADLF0961766NCAM00353M1.VIC\)
NLB_712299404DFFLF0961766NCAM00353M1.VIC pyr=3 window=13 maxdisp=2032 blob=50
point=cm=label
```

Note that parameter names can be abbreviated, as long as they are unique within the PDF parameter list. Same for keywords. It is best not to carry this to extreme, especially when writing scripts (because future versions may add keywords that break the uniqueness of the abbreviation), but the abbreviations are often handy.

```
---- Task: MARSCOR3 -- User: urmslopg -- Thu Jul 28 23:51:00 2022 ----
INP=(
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/tmp_cache/NLB_712299404RADLF0961766NCAM00353M1.VIC',

'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/tmp_cache/NRB_712299404RADLF0961766NCAM00353M1.VIC')
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marscorpipe/output/NLB_712299404DSRLF0961766NCAM0035
3M1.VIC'
IN_DISP=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marscorpipe/processque/NLB_712299404DFFLF0961766NCAM
00353M1.VIC'
TEMPLATE=(7, 11)
SEARCH=25
QUALITY=0.5
GORES='GORES'
GORE_QUALITY=0.6
GORE_PASSES=3
GORE_REVERSE='GORE_REVERSE'
DISP_PYRAMID=3
MODE='amoeba4'
FTOL=0.004
MULTIPASS='multipass'
FILTER='filter'
OMP_ON='OMP_ON'
```

Here is an example of a keyword (`MODE`) whose value is not the same as the keyword name. It can still be specified as either `MODE=amoeba4` or `-amoeba4`. Also shown is abbreviation of keyword values, in this case `-gore_rev` and `-multi`. Abbreviations are completely optional.

```
$MARSLIB/marscor3 inp=\( NLB_712299404RADLF0961766NCAM00353M1.VIC
NRB_712299404RADLF0961766NCAM00353M1.VIC\)
out=NLB_712299404DSRLF0961766NCAM00353M1.VIC
in_disp=NLB_712299404DFFLF0961766NCAM00353M1.VIC templ=\(7,11\) search=25
qual=0.5 -gores gore_q=0.6 gore_pass=3 -gore_rev disp_pyr=3 -amoeba4
ftol=.004 -multi -filter -omp_on
```

One other thing. There is no requirement on any particular order for the parameters, once you get past any positional parameters. It is usually convenient to put them in the same order as in the history label, or the same order as in the PDF, but it does not matter.

```
---- Task: MARSDISP -- User: urmslopg -- Thu Jul 28 23:52:34 2022 ----
INP=(
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marsdispcompipe/localcache/NLB_712299404DSR
LF0961766NCAM00353M1.VIC',
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marsdispcompipe/processque/NRB_712299404DSR
LF0961766NCAM00353M1.VIC')
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marsdispcompipe/output/NLB_712299404MDSLF09
61766NCAM00353M1.VIC'
```

In this case, the program name is ambiguous, as noted above. The options are marsdispcompare, marsdispinvert, and marsdispwarp. Looking at the program PDF files, we see that mardispwarp has a required (`COUNT=1`) `DISP` parameter, which is not present, so it is not marsdispwarp. Similarly, marsdispinvert has a required `IN_DISP` parameter, which is also not present. So by process of elimination, it must be marsdispcompare.

Also note that the second input again comes from the result of doing all this again on the right side. To save the trouble, the image is in the sample input directory.

```
$MARSLIB/marsdispcompare \(NLB_712299404DSRLF0961766NCAM00353M1.VIC
NRB_712299404DSRLF0961766NCAM00353M1.VIC\)
NLB_712299404MDSLF0961766NCAM00353M1.VIC
```

Next:

```
---- Task: MARSMASK -- User: urmslopg -- Thu Jul 28 23:52:35 2022 ----
INP=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marsdispcompipe/localcache/NLB_712299404DSR
LF0961766NCAM00353M1.VIC'
OUT=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marsdispcompipe/output/NLB_712299404DSPLF09
61766NCAM00353M1.VIC'
MASK=
'/proj/mslredops-
workspace/opgs/matis/OPS/workarea/marsdispcompipe/output/NLB_712299404MDSLF09
61766NCAM00353M1.VIC'
```

This is just a straightforward application of what we've learned already.

```
$MARSLIB/marsmask inp=NLB_712299404DSRLF0961766NCAM00353M1.VIC
out=NLB_712299404DSPLF0961766NCAM00353M1.VIC
mask=NLB_712299404MDSLF0961766NCAM00353M1.VIC
```

The examples above should be sufficient to turn any history label into the command line needed to re-create the image.

6.9 Coordinate Frames

Every description of geometry in the 3-D world, be it XYZ points, surface normals, mesh vertex coordinates, or slopes, must be measured in a defined coordinate frame, or the value is meaningless.

Coordinate frames in the Mars missions are identified by a name, an index, and (optionally) a solution ID. The name identifies the type of frame, such as Site, Rover, Local Level, Orbital, etc. The index specifies which instance of that frame you are referring to - Site 5, Rover (5,12), Orbital 0. The optional solution id allows for different definitions of the frame, such as happens after rover localization (the default solution id is "telemetry", i.e. what the rover or lander knows).

Most missions have the following basic frame types:

- Rover, Rover Nav, or Lander - a coordinate frame attached to the rover or lander, that moves with it (both in position and orientation). The index is usually the Rover Motion Counter (RMC), which identifies unique positions of the rover.
- Local Level - a coordinate system that moves with the rover, but points north/east/down according to the local gravity vector. The index is again the RMC, which identifies unique instances of the frame corresponding to rover positions.
- Site - a coordinate frame "attached" to the ground, pointing north/east/down like Local Level, but detached from the rover so it is fixed in location. Site frames have a single index. The Site is identical to Local Level at the moment the Site is declared.

Missions often have additional frames, such as Rover Mechanical or frames relating to arm instruments. Only a few of the many possible frames are supported in the PIG library.

Landers, which do not move, still generally have a Site frame, just for compatibility with the software. There is usually only one, but it would be possible to have more than one if the lander moved (for example being dragged by an arm). We often call it the Rover frame rather than Lander for landers out of convenience; the two are generally synonymous.

There are also a number of Orbital frames, which are useful for localizing the rover and appear in the PLACES databases [PLACES] but are not supported in the PIG library.

Coordinate frame definitions appear in the label. They always include the name and index of the frame, the name and index of the reference frame, and the offset and orientation (quaternion) of the frame with respect to the reference frame. They can also be specified using an RSF file.

The PIG library supports conversion among frames, at least for frames that are defined in the image label across all the inputs to a given program (and if given, an RSF file). The set of all

known frames defines a tree structure. The root of that tree is called the “FIXED” frame in PIG; it is almost always the lowest numbered SITE frame in the tree (but can be changed via parameters). PIG traverses that tree, compositing parent transforms to determine the transforms for each frame to and from FIXED, so that frame-to-frame conversions consist of transforming the input frame value to FIXED and then transforming the result to the output frame. The FIXED frame does not appear in the label as such, rather it is a runtime construction of the PIG library. Note that even the root of the tree has a reference frame, but the definition of that reference frame will be unknown to the program. The PLACES database (or the PLACES data in PDS) can be used to get the entire frame tree, if it is needed.

Most VISOR programs have a primary coordinate system frame in which results are computed. This is specified via the `COORD` and `COORD_INDEX` parameters. Usually only `COORD` is needed, thus `-site` or `-rover` or `-local_level` (etc.) specify that the output should be in that frame. If the index is not explicitly specified (as is the norm), it defaults to the `ROVER_MOTION_COUNTER` (RMC) in the image label.

Some programs have more than one coordinate frame. `SURF_COORD` is a common case, defining the coordinate frame in which the surface model is measured. Other programs have more. These values cannot be specified using the -keyword parameter form, but instead must be specified using keyword=value form, and the value cannot be abbreviated (for example, `surf_coord=local_level`).

6.10 Quaternions

Rotations in VISOR and PIG, and indeed throughout much of the Mars missions generally (including the flight software) are represented using quaternions.

A quaternion is a set of 4 numbers, composed of a vector and scalar component, that together represent a rotation in 3D space. The PIG library consistently puts the scalar first, often called “ground order”. The flight software commonly (but not exclusively) puts the scalar last, often called “flight order”. Ground order is used throughout here.

A quaternion is thus:

(s, v_1, v_2, v_3)

where

$s = \cos(\theta/2)$

$v_n = \sin(\theta/2) * a_n$

θ = the angle of rotation (right hand rule)

a = (x,y,z) axis around which rotation occurs

Quaternions are normalized, meaning:

$s2+v12+v22+v32=1$

Rotations are accomplished using quaternion multiplication, using the “JPL convention”. For coordinate system definitions, the sense of rotation can be described as:
Given a vector expressed in the current frame, multiplication by this quaternion will give the same vector as expressed in the reference frame.
Multiplying two quaternions composites their rotations (rotate by Q_2 first and then Q_1):

Q1Q2=s1s2-V1V2 , s1V2+s2V1+V1V2

which uses vector dot and cross products. Rotating a vector by a quaternion is thus:

QV=vectorQ(0,V)Q'

where the central term builds a quaternion with scalar 0 and vector component V, Q' is the quaternion inverse, which negates the vector component, and vector() returns the vector part V of the quaternion (simply discarding the scalar).

6.11 POINT_METHOD parameter

The `POINT_METHOD` parameter is a special parameter. Most parameters get an entry in the PDF. However, there are a set of parameters in the PIG library which are not commonly used, for which it would be inefficient to modify each of the dozens of mars programs whenever one was added. The `POINT_METHOD` parameter provides a mechanism to specify such parameters. It is a comma-separated list of parameters, which are either "keyword=value" or (occasionally) just "keyword". You thus end up with the somewhat unwieldy:

```
point=cm=label
```

with multiple = signs. If you need to specify more than one, you need to quote the string and use commas, for example:

```
point='"cm=label,cahv_fov=min"'
```

For keywords with no value, there is no = , e.g.

```
point=no_lbl_point
point='"force_dark,smear=on"'
```

The table is roughly ordered by usefulness, with the most important ones first.

Keyword	Values	Meaning
cm	label kinematics	Specifies whether to read the camera model from the label, or to recompute it using kinematics and the calibration models (default). Pointing correction via nav files can only be done in kinematics mode. Note: cannot be used with marscheckcm; see the help for that program.
pm	0dof 2dof 3dof scale 6dof 7dof	Overrides the pointing model to use. Values are mission-specific. Generally 2dof is az/el, 3dof adds twist (image rotation), and scale adds a scale term (slight zooming of the image). 6dof and 7dof are for arms and use position + euler angles or position + quaternion respectively. Generic: 0dof, 6dof, 7dof MER: pan/nav: 2dof, 3dof PHX: ssi: 2dof,3dof; rac: 6dof, 7dof MSL: mast/nav/rmi: 2dof, 3dof, scale; mahli: 6dof, 7dof InSIGHT: 6dof, 7dof M20: zcam/nav/rmi: 2dof, 3dof, scale; mcc/wat/aci: 6dof, 7dof
cahv_fov	max union	Specifies how the field of view is computed when linearizing images. Min or intersect (default) uses the intersection of the

Keyword	Values	Meaning
	min intersect linear	cameras. Much overlap will be missing, but there are no black areas. Max or union picks the union of the cameras. All overlapping pixels are preserved, but there may be black areas on the sides, and significant resolution could be lost, especially for fisheye lenses. Linear uses only the CAHV vectors, ignoring higher order terms and usually results in a compromise between MIN and MAX with images that are similar scale-wise to the original.
mission	<name>	Overrides the automatic mission detection and forces the mission to be as specified. Generally only used to force the Generic mission via mission=generic.
border_left border_right border_top border_bottom	<number>	Trims that many pixels off the edge of all inputs to a mosaic. Positive numbers shrink the border; negative numbers expand it.
margin_left margin_right margin_top margin_bottom	<number>	Like border, but the value is adjusted for downsample, so for example you can mix thumbnails and regular images and still trim off a bad edge. (both margin and border may be specified; they're cumulative).
rmc_max_index	<number>	Ignore parts of the RMC (make effectively this long) when building CSs. Lets you put one site/drive value in an RSF file and have it apply to all images at that site/drive regardless of low-level RMC components.
rmc_epsilon	<float>	Epsilon value for comparison of CS values. Even if the RMCs match, a new CS is created if the values differ by more than this value. Default: 1e-3. Higher value means more CSs will be considered equal.
cmod_warp	<number>	Specifies algorithm for cmod warping. 1=old model, 2=new model (2017), which works better for non-traditional stereo. 3 or PSPH = align to and return PSPH model. CAHV supports only 1, CAHVOR 1 or 2, and CAHVORE all 3. 1 is the default for MER, PHX, MSL; 2 is default for InSIGHT and M20.
solar_min_elev	<float>	Zenith correction gets the solar elevation from the label. If it's < a minimum, set to that minimum to avoid issues with the math. Default min is 5 degrees, can be overridden via this parameter (value in degrees).
cm_name	<name>	Specifies which instance of the camera model to read from the label. Looks for a model with a matching CAMERA_MODEL_NAME label.
use_uniqueid2		Forces use of getUniqueId2() rather than getUniqueId() when writing nav or brt files. No effect for missions other than M20. For M20, getUniqueId was modified March 2022 to include msec, as needed by the heli. getUniqueId2 omits the msec, as was done before that. Either type of file can be read;

Keyword	Values	Meaning
		this simply allows writing the non-msec version.
force_lin	cahv cahvor cahvore allow_cahv	Force camera model type for linearization. In reality there is little need for this. However, since default as of Feb 2019 is to change CAHV to CAHVOR in order to use a version of warp_models that is aware of the image FOV, we allow an option to restore the old behavior. cahv/cahvor/ cahvore: force to that model; allow_cahv: doesn't force CAHV to CAHVOR. Default: select CAHVOR/E as needed, force CAHV to CAHVOR.
no_lbl_point		Normally the image is pointed using the pointing model described in the label (although this rarely exists). NO_LBL_POINT disables the label pointing, using default pointing instead.
sherloc_point	turret	For M20 Watson/ACI, get camera pose from the turret CS rather than MODEL_TRANSFORM_VECTOR/QUATERNION
nodark		Turn off PHX dark current subtraction
force_dark		Force on PHX dark current subtraction
dark	shaw lemmon lemmon_nowt fast_lemmon fast_lemmon_nowt	PHX dark current method, see RadiometryPHX.cc
rad_frame	dark	For PHX, return dark frame only, without image or flat field
smear	on off	Turns on or off PHX smear correction (default is auto)
nobinning		Turns off PHX radiometric binning compensation
force_binning		Forces on PHX radiometric binning compensation
notemp		Turns off PHX temperature compensation
fido	4dof joints	Specifies the pointing model to use for the FIDO testbed for the nav/pan cameras. Was never retrofitted to use pm instead.
generic	<pm_name>	Specifies the generic pointing model to use. Obsolete, use pm= instead.
mer	label kinematics	Obsolete, use cm= instead

6.12 Linearization

Linearization of images and camera models is an important step in earlier missions (MER, PHX,

MSL) but is rarely used in later missions (InSIGHT, M20).

Linearization does two things:

1. It removes the radial distortion from an image, and the effect of fisheye lenses, resulting in a linear (pinhole) camera model. Practically speaking, this converts CAHVOR and CAHVORE models to CAHV. This also has the effect of making straight lines in the scene appear as straight lines in the image.
2. It epipolar-aligns stereo pairs of images. This means that image features appear on the same line (within noise) in both of the image. This facilitates viewing the images in stereo (e.g. with red/blue glasses or dedicated display hardware) because the stereo disparity is only in the horizontal direction, matching how human eyes work.

On earlier missions, linearization was used to facilitate stereo correlation. Because the disparity is only in the sample direction, a 1D correlator (marsjplstereo) can be used as a first stage. This is then refined with a 2D correlator (marscor3), but the 2D correlator requires a good starting point. On MER and PHX, this linearized result is what was used to generate XYZs, meshes, and downstream products. On MSL, we started moving away from this, by taking the linearized disparity, unlinearizing it (converting it back to the raw geometry), and doing another correlation on the raw images. This provides a better result, with less interpolation noise. On InSIGHT, we stopped using linearized XYZ values entirely, computing them only as an intermediate step. On M20, we completed the process, eliminating linearization entirely from the processing chain. Instead, marsecorr does the first-stage correlation directly on the raw images, taking into account the epipolar curves for each pixel. This works much better for nontraditional stereo geometries, some of which cannot be represented in linearized form.

On M20, linearized images are still produced, but only for purposes of humans viewing the images in stereo.

There are two types of linearization: nominal, and actual. Actual uses a specific image as the stereo partner, they are linearized to each other. Nominal predicts what the stereo partner will look like without necessarily needing the image. This can only be done with traditional stereo camera pairs, and cannot be done at all on M20 due to the complexity of the mast-mounted cameras (zoom and focus for zcam, tiles and downsampling for the navcams).

There's actually a third type, which is used for mono images: linearize an image with itself. There is no epipolar alignment, but the camera distortion is still removed.

6.13 RSF files

RSF files are Rover State Files (sometimes also called Rover Vector Files, RVF), and provide a mechanism by which the rover location expressed in the label can be overridden. For example, when doing mosaic pointing correction, one can also adjust the rover's location (if the mosaic spans locations); the RSF file would be used to supply this to the programs. Or, the PLACES database or PDS delivery thereof [PLACES] could be used to provide updated rover localizations.

The use of RSF files is a bit of a black art, unfortunately. They are not used often, and thus we do not have a wealth of experience in using them. Take one of the examples below and modify it as needed to match your use case.

Below is an example RSF file that redefines site 31 in terms of site 26 (using completely arbitrary numbers).

```
<?xml version="1.0" encoding="UTF-8"?>
<rmc_file mission="M20" variant="Master_RVF" index1="26">
    <priority>
        <entry solution_id="idso-rgd-site"/>
        <entry solution_id="telemetry"/>
    </priority>

    <solution solution_id="idso-rgd-site" name="SITE_FRAME" add_date="2022-
12-15T18:28:00Z" index1="31">
        <reference_frame name="SITE_FRAME" index1="26"/>
        <offset x="100.0" y="200.0" z="30.0"/>
        <orientation s="1.0" v1="0.0" v2="0.0" v3="0.0"/>
    </solution>
</rmc_file>
```

The key point here is to provide a solution ID and to give it a higher priority than the default "telemetry" solution id. You can have multiple `<solution>` elements.

Here's another example, defining two ROVER_FRAME's with respect to their SITE:

```
<?xml version="1.0" encoding="UTF-8"?>
<rmc_file mission="MER2" variant="Master_RVF" index1="133">
        <priority>
        <entry solution_id="mipl-rgd-np_combo"/>
        <entry solution_id="telemetry"/>
    </priority>

    <solution solution_id="mipl-rgd-np_combo" name="ROVER_FRAME"
add_date="2004-
03-04T19:30:19Z" index1="133" index2="0" index3="0" index4="3" index5="0">
        <reference_frame name="SITE_FRAME" index1="133"/>
        <offset x="0.0" y="0.0" z="0.0"/>
        <orientation s="0.149767" v1="0.10667" v2="-0.0505445" v3="-
0.98165"/>
    </solution>
    <solution solution_id="mipl-rgd-np_combo" name="ROVER_FRAME"
add_date="2004-
03-04T19:30:19Z" index1="133" index2="0" index3="76" index4="231"
index5="507">
        <reference_frame name="SITE_FRAME" index1="133"/>
        <offset x="0.0" y="0.0" z="0.0"/>
        <orientation s="0.149767" v1="0.10667" v2="-0.0505445" v3="-
0.98165"/>
    </solution>
</rmc_file>
```

Note that you must generally provide all the RMC elements for a given frame. This can be quite annoying, as there tends to be lots of individual RMCs as the mast moves around, for example. However, the rmc_max_index option to the POINT_METHOD parameter allows you to ignore everything past, say, the second index, so your RSF file need only have the first two indices.

Many programs have a -debug parameter, which dumps out the internal coordinate system database. This option is very helpful when trying to work with RSF files. RSF files are supplied to most Mars programs via the RSF= parameter.

6.14 Parallelized Programs

There are two types of parallelization in VISOR: OMP and MPI.

OMP (Open MultiProcessing) is a standard system supported by most C/C++ compilers to do parallel processing in different threads on a single machine. OMP is easy to use, and automatic at a user level: if you don't have OMP on your compiler, or are on a single-threaded machine, the program runs the same (just slower).

OMP is used by many of VISOR programs; they can be identified via

```
#define LIB_OMP
```

In their imake files.

In most cases, there is a keyword parameter to the program that can turn OMP processing on or off (`-OMP_ON` or `-OMP_OFF`). The default is on.

We have observed that running more than one OMP-enabled job on a single machine at the same time can sometimes cause inordinate slowdowns to the jobs. The root cause of this is unknown, but we often attempt to avoid doing this in the pipelines. Nevertheless, the speed advantages of OMP on modern processors is tremendous and well worth the trouble.

By default, the number of threads used equals the number of cores on the machine where the program is being run. However, this can be controlled by setting an environment variable, e.g.

```
setenv OMP_NUM_THREADS 5
```

We have observed that setting the number of threads to one less than the number actually available (the command-line utility `nproc` tells you this) seems to run faster than using all threads in many cases. The reason for this is unknown, but we suspect it has something to do with thread affinity; when you leave one core open to do OS-type functions, thread affinity and thus performance is improved.

The `OMP_NUM_THREADS` environment variable is a standard part of OMP and is not VICAR-specific. There are other OMP environment variables that can be set as well, but they are likely less useful. Consult the OMP documentation.

The other type of parallelization in VISOR is MPI (Message Passing Interface). It is used only on one commonly-used program (marsmap), and is also in the lesser-used (nearly deprecated) marsint and marscor2. MPI is cross-machine parallelism, where parts of the program run on completely different nodes.

It should be noted that MPI has not been tested in marsmap in some number of years, and may or may not work well any more. Instructions for using it are in the marsmap PDF. Marsmap is also parallelized using OMP, and we have found this significantly more useful (and easier to

use) than the MPI style.

6.15 Generic Mission

VISOR supports many different missions, as described earlier. The supported missions include specific support for parsing and writing labels specific to each mission.

There also exists a "generic" mission, which is used if the image is not recognized as belonging to a specific mission, or if the "`point=mission=generic`" keyword is used.

The generic mission takes advantage of the fact that the VICAR labels are very similar across missions. It will read a subset of labels, generally the bare minimum.

The camera model is the most important of these. The generic mission will read a standard GEOMETRIC_CAMERA_MODEL label group. It also supports, if there is no camera model in the label, a separate camera model file using the same basename as the image but with an extension of `.cahv`, `.cahvor`, or `.cahvore`. These files look like, for example:

```
# M2020_CAL_001_SN_112-FHRB-FLIGHT_RMECH.cahvore
#   translated by 0.090020,0.000000,-1.133380

Model = CAHVORE2 = fish-eye

Dimensions = 5120 3840

C =      1.101777      0.161586     -0.728891
A =      0.885010      0.174004      0.431833
H =   1933.277158   2565.195137    943.363432
V =    758.905175    334.564216   2762.271480
O =      0.884584      0.174836      0.432370
R =      0.000001      0.007410     -0.005557
E =      0.014002      0.007392     -0.005572
```

The calibration camera models use this same format, use those as examples. Note that the parts below this (which you will see in the calibration files) are not needed. The O/R/E terms are only needed if using CAVHOR or CAHVORE.

7. MOSAICS

The following is adapted/summarized from a presentation at the 2012 Planetary Data Workshop [Deen2012], which can be found here [https://ntrs.nasa.gov/citations/20120018099]. Consult the original for more details.

Simply put, a mosaic is a single larger image that is made by combining many individual smaller frames. The hard part is to transform and match the images so they look like a unifie whole. Mosaics provide much better situational awareness and geologic context than the individual component images. PDS contains mosaics generated from virtually every end-of-drive location for every rover and lander supported by VISOR. However, there are many cases where users will want to make their own.

In order to work with VISOR mosaic tools, images must have a calibrated camera model, and the pointing of the camera must be known (approximately - adjustments can be made).

A fundamental design feature of the mosaic tools is that traceability of every image to the source image is maintained. This maintains scientific integrity - quantitative measurements are possible. The tools do not do unconstrained warping or seam blending, and any warping is done in an algorithmic way, based on camera models and surface models.

There are six mosaic projections supported by VISOR - cylindrical, perspective, cylindrical-perspective hybrid, polar, vertical, and orthorectified. See a description of these in Section 4.15.

7.1 Parallax Effects

Parallax is the primary source of error when making mosaics. Parallax occurs when you view a 3-dimensional scene with different depths from two different points in space. If you hold your finger in front of your face, and close your eyes alternately, your finger “moves” with respect to the background. Nearby objects have greater apparent motion than distant objects. This is the fundamental basis of stereo vision, however this causes problems for mosaics. For example, foreground objects can hide different parts of the background in different views, causing areas of no data called “occlusion”. When this occurs, it means some objects are visible in one image but not another. Additionally, for near-field objects, different images may show different sides of the same object.

For in-situ missions, stereo camera heads consist of two cameras mounted to either side of a mast head, with the pivot in the middle. As the head is moved in azimuth, the cameras describe a circle in space - and thus each frame is taken from a different position. This creates parallax – imaging the scene from different points of view.

The fundamental challenge of mosaics is to transform the images so they share a common point of view – that of the output projection. This could be done perfectly if the camera pivoted about its entrance pupil. This is because images naturally share the same point of view, so no transform is needed and there is no parallax. As a matter of fact, this is how the rover “selfies” from MSL (MAHLI) or M2020 (Watson) work. The arm goes through a lot of gyrations to keep that entrance pupil constant is it looks in different directions. However, this is impractical for stereo-vision cameras. A “flagpole” mounting could pivot one camera about its entrance pupil, but its stereo partner would have twice as much parallax, and this mounting method has not been used to date.

Distance matters - parallax is caused by translation of the camera, so at infinity there is no

parallax. The horizon thus has effectively no parallax, whereas an arm workspace in front of the rover or lander can have extreme parallax. Thus the issue is even more severe with arm-mounted cameras, such as on InSight, because the camera moves farther, and the workspace is closer.

Parallax is managed by transforming the point of view of the individual camera frames to share a common point of view (that of the mosaic). This is accomplished by projecting the image back into 3D space and then looking at the result from a different POV.

If the 3D shape of the scene is known, this projection can be done exactly. This is the basis of orthorectified projections, which do not suffer from parallax. Because the 3D coordinates of each point in the scene are known, objects can be placed properly and viewed from the mosaic point of view (usually overhead, for orthorectified mosaics). Objects are not distorted, but holes or gores appear in the image due to occlusions.

7.2 Surface Models

The orthomosaic technique requires stereo analysis of the terrain, but stereo is not always available, and when it is, it does not necessarily cover the entire mosaic. For the (more common) case where the 3D shape is not known, an assumed shape - called a surface model - is used instead. Each image is projected to the surface model, creating a 3D virtual scene. This is then viewed from the point of view of the mosaic, by projecting the virtual scene back into the mosaic.

Using a surface model works well if the surface model closely matches the actual surface, and if the point of view is not moved too much. Deviations from the model cause distortions due to parallax, but these deviations are usually much less than the parallax in the raw images.

There are several types of surface model available:

- Infinity - essentially, no surface model. Everything is projected to infinity, and there is no parallax correction. However, this works well for distant scenes.
- Flat plane - this models the rover sitting on top of a flat plane, a.k.a. the ground. This works well for most scenes, but breaks down for nearby cliff faces or large boulders. The plane is described by a surface normal (it may be tilted), and a ground intercept point. Generally the plane is tilted with the rover (made easy by defining it using Rover frame), which works well if the rover is on a slope. The plane surface model is by far the most common model used for mosaics.
- Sphere - The surface model is a sphere, and the rover can be inside or outside the sphere. There are two subtypes, SPHERE1 and SPHERE2; see the help for `marsmap` for descriptions. They are rarely used.
- Mesh - this uses a terrain mesh as a surface model, instead of a flat plane. With an accurate surface model, this provides some of the benefits of the orthorectified mosaic (reduced parallax) while having an easier time filling gaps. However, not every location has a suitable mesh. Fortunately, stereo is more accurate closer in, which is where parallax is the worst; when the projection goes off the edge of the mesh, it's typically close enough to infinity not to matter.

7.3 Seam Correction

Much of this is adapted from the 2015 Planetary Data Workshop [Deen2015].

Perfect parallax correction is not possible. Othorectified projection can be nearly so if the XYZ data is good enough, but holes or gores appear due to occlusions. Surface model-based projections do not have holes, but deviations from the surface model create distortion, evident as discontinuities at the seams. Finally, image warping can provide an illusion of no seams, but the unconstrained geometry leads to pretty pictures that are unsuitable for scientific interpretation.

In addition to parallax, we have imprecise knowledge of the camera pointing, as well as camera modeling errors. All of these contribute to seams, or visible discontinuities where images overlap.

VISOR seam correction capability is based on bundle adjustment using tiepoints. The pointing of the cameras is adjusted, along with sometimes internal parameters such as image scale. The cameras are corrected as a unit, thus preserving traceability. Parallax error is in principle uncorrectable using these techniques, except by adjusting the surface model to better match the actual surface. However, parallax errors can be reduced somewhat by distributing the errors across the entire mosaic, which makes it harder to see individual seam errors.

There is no absolute best answer when it comes to pointing correction. A solution that works well for a mosaic may not work for a terrain mesh, and vice-versa. Each should be used in the context from which it was derived.

7.4 Pointing Correction Principles

There are several principles we adhere to in VISOR tools:

- Maintaining scientific integrity of the results is paramount. Given a pixel, we can mathematically report the source image it came from, and where in that image. This information is encoded in the IDX/ICM index files, but is also derivable using the projection information in the label.
- No unconstrained warping. We treat the image as a single entity, and point the *camera*, not the *pixels*.
- Make use of *a priori* metadata. This ensures the results do not go “out to lunch”. Some popular stitching programs ignore *a priori* data, with sometimes disastrous results - we have seen cases where frames entirely swap positions using this method. By using the *a priori* metadata, we ensure that the right frame is in the right place.
- No blending of seams. Seam blending can create artifacts or “fuzzy” areas around the seams, which could lead to misinterpretation of the data.
- Try to minimize visible seams. Within the constraints of no blending, we try to hide the seams as much as possible, for example by reordering images to put the seams in less-visible locations (e.g. don’t split a rock). Where seams occur, we prefer a hard-edged, straight seam to something that might be misinterpreted in the image.

7.5 Pointing Correction Process

The pointing correction process proceeds as follows:

- Reorder images to put seams in the least visible places
- Gather tiepoints
- Compute pointing correction (bundle adjustment)

- Apply to mosaic
- Adjust tiepoints and repeat as necessary

7.6 Tiepoints

Tiepoints identifiy features which are the same across two (or more) images. For example, this rock on image A is the same as this rock on image B. Typically tiepoints are generated to subpixel accuracy using a correlation process, but manual tiepoints might be integer-pixel accuracy.

Tiepoints can be gathered manually (program `marstie`), or automatically (`marsautotie` or `marsautotie2`). The `marsautotie` program uses a correlation process to find tiepoints, whereas `marsautotie2` uses a keypoint detection process, using the Affine SIFT (ASIFT) algorithm. The `marstie` program can be used to create tiepoints, or to refine tiepoints gathered by one of the automated programs. Tiepoints can also be created from disparity maps using `disp2tp` (and the reverse using `tp2disp`).

Note that there are some special tiepoint types that tie e.g. an image pixel to an XYZ location, or to a known elevation. These are rarely used; see the help for the `marsnav` program.

7.7 Bundle Adjustment for Mosaics

In the mosaic context, tiepoints are used to correct pointing for the mosaic, a process known as bundle adjustment. Tiepoints are projected from one image to the surface model, then back into the second image. The difference between where that projection hits the second image, and where the tiepoint says it should hit, is the tiepoint error. The sum of this error across all tiepoints is the error metric.

The `marsnav` program attempts to minimize this tiepoint error by adjusting a set of parameters that affect the image geometry. These parameters can be:

- Pointing of individual images (typically az/el of motor joints, could also be XYZ/Euler angles for arm-based cameras). Sometimes includes image scale, or twist (rotation around the camera axis).
- Tilt and location of surface model
- Localization (pose) of rover if it moved during panorama (rover motion magnifies parallax problems considerably)

“Inertia” can optionally be added as an addition component of the error metric. This provides a stay-in-place weight, that increases as images point farther away from their starting position. It can be thought of as a rubber band keeping the images in place. The farther they move, the more the rubber band stretches. Inertial guards against the entire solution drifting (for example, a constant azimuth shift in all images would be undetectable but would compromise the ability to read azimuth off the mosaic). It also minimizes the impact of bad tiepoints, by not letting them “pull” the image as much.

The marsnav process closely mirrors the actual mosaic projection process. For this reason, it is the best bundle adjustment mechanism to use for (non-ortho) mosaics. Tiepoints should generally be gathered close to the ground (e.g. not on the tops of rocks), as they are more likely to be close to the surface model and thus can help determine the optimal surface model.

7.8 Bundle Adjustment for Meshes

Terrain meshes, and XYZ points generally, benefit from a different bundle adjustment mechanism. While marsnav can be used for traditional stereo cameras (by tiepointing just one eye and assuming the other keeps the same pointing), it does introduce errors if the surface does not match the surface model. However, it becomes unusable for arm-mounted cameras such as InSight, for stereo work. Because the "left" and "right" eye images are simply motions of an arm, one with fairly large pointing errors, we cannot use the "left" eye images and apply the results to the right. And if we tiepoint them together, we are essentially forcing the derived XYZs to match the surface model, which is not what we want at all for terrain analysis.

The `marsnav2` program fixes this. It uses the Ceres solver to do a full bundle adjustment, solving for pointings and adjusting ground points simultaneously. It is not as good for standard mosaics, but excels at orthomosaics and terrain meshes, especially from nontraditional cameras.

For `marsnav2`, the tiepoint strategy is different. Rather than keeping them on the surface, you want them distributed throughout the 3D volume as much as possible - bottoms and tops of rocks both. You also want to reuse the same point in different pairs as much as possible (creating "tracks", essentially tying more than two images together at the same point).

There is a "miss distance" mode in `marsnav` which can also be used for mesh-type correction. It looks at how closely the two projected rays (from each half of the tiepoint) come to each other at closest approach. However, the rays are often so close to parallel that the solution does not converge well, and `marsnav2` generally works better.

8. MESHES

The terrain models are generated by triangulating XYZ points clouds. Each points cloud corresponds to a stereo pair of images and triangulation produces a wedge of terrain model with the narrow end of the wedge pointing toward the cameras. The original image is used as a texture map to add detail and color to the polygonal surface representation. Most XYZ images will contain holes, or pixels for which no XYZ values exist. Some of the factors that cause holes are parallax, occlusions, correlation failures, and failure of a result to meet quality checks of the XYZ computation program. While there are mesh generation parameters that will result in generation of polygons covering small holes, the intent is to preserve geometric structure of the input points cloud. SITE frames are used almost exclusively for all terrain products.

9. PROGRAM SUMMARIES

Program	Description
disp2tp	Generates an XML tiepoint file from a disparity map
marsautoloco	To automatically adjust the map coordinates origin (Easting, Northing) of the INP image with respect to the base map image. The adjustment is based on the coregistration of the orthomosaic (INP) to the baseline image.
marsautotie	To automatically gather tiepoints for a set of overlapping images.
marsbrt	This program computes intensity (or "brightness") corrections for a mosaic. These corrections are intended to reduce radiometric seams in the image.
marscahv	To convert images from distorted (CAHVOR/CAHVORE) to linear (CAHV) coordinates.
marscheckcm	Given input file, compare camera models obtained via various means. Currently compares camera model found in the input file's label(LABEL) to camera model obtained by applying kinematics transformation to calibration camera models(KINEM).
marscolor	This program provides the capability of converting an image from source color space into a target color space.
marscor3	Given a stereo pair of images and a low-resolution or low-quality disparity map representing line and sample disparities for every pixel in the scene, this program refines the disparities to create a higher-quality map.
marsdebayer	Removes the Bayer pattern from a color CCD image, such as used with the MSL cameras Mastcam, MAHLI, and MARDI. The result is a three-band color image.
marsdispcompare	This program is meant to be part of processing chain which ends up with L->R disparity image, a R->L disparity image, and a mask indicating which values in the l->R disparity image should be ignored.
marsdispinvert	This program takes a disparity map, such as generated by marscor3 or marsjplstereo, and "inverts" it, or flips the eyes.
marsdispwarp	This program is used to transform an image through a disparity map so the geometry matches that of the partner.
marsecorr	Correlation program that accounts for all camera types and acquisition geometries between two images to perform first-stage disparity map computation.
marsfilter	Computes a mask for an XYZ image, intdended to remove points

Program	Description
	that are not of interest.
marsifoot	Put a footprint of an instrument down in a mask image, given an XYZ.
marsigood	Generates an overall goodness product by looking at the goodness bands of multiple other products (such as tilt and roughness).
marsint	Used for interprocess communication and can be run in parallel on multiple machines. This dramatically improves the wall-clock execution time of the program.
marsinverter	Generates inverse lookup table products
marsirough	Computes a measure indicating the roughness of the surface for each pixel, for the purposes of instrument placement.
marsitilt	Computes the tilt an instrument would have if placed at the given point.
marsjplstereo	Computes disparity maps from a stereo pair.
marsmap	Assemble multiple frames into a mosaic in one of three projections (Cylindrical, Polar, and Vertical)
marsmask	Apply mask to an image
marsmcauley	Generate a Mcauley (hybrid cylindrical-perspective) projection
marsmesh	Generates polygon meshes from XYZ point clouds
marsmos	Assemble multiple frames into a mosaic, using an output camera model (derived from the input)
marsnav	Correct the navigation (pointing, e.g. azimuth and elevation) of a set of overlapping images.
marsortho	Assemble multiple frames into a mosaic in orthographic projection
marsprojfid	Simple program to accept a file list and corresponding tiepoint file, extract all Fiducial tiepoints from the file, and project those through the corresponding image
marsproj	Simple program to accept an XYZ coordinate (expressed in the coordinate frame specified by COORD), project it through the camera model, and report the line/sample coordinates of the point in the supplied image
marsrad	Radiometrically correct an image, usually an EDR.
marsrange	Computes a cartesian distance between a reference point and an

Program	Description
	XYZ position of an input image
marsrcorr	An image matching program that computes regularized disparity maps
marsrelabel	Update certain label groups
marsremos	Quickly make a mosaic using a different type of derived image (XYZ, slope, etc) without having to recompute the geometry
marsrfilt	Range-based filtering on an XYZ image, returning a new XYZ image
marsrough	Computes a measure indicating the roughness of the surface for each pixel, given XYZ and UVW (surface normal) images as input.
marsslope	computes one of the following slope function types (Slope, Slope Heading, Slope Magnitude, Slope Rover Direction, Northerly Tilt, Solar Energy)
marstie	Gather tiepoints for a set of overlapping images, either manually or automatically.
marsunlinearize	Converts a linearized Disparity into an equivalent unlinearized disparity
marsuvw	Computes a surface normal for each pixel, given an XYZ image as input. This surface normal is a unit vector pointing "out" of the surface (up, for an in-situ image of the ground).
marsxyz	Compute, from a disparity map, the XYZ coordinates of each point in a stereo pair
mslrough	Computes a measure indicating the roughness of the surface for each pixel, given XYZ and UVW (surface normal) images or an XYZ image and a single UVW surface normal vector as input
nsytfoot	Put a footprint of an instrument down in a mask image, given an XYZ
nsytgood	Generates an overall goodness product by looking at the goodness bands of multiple other products (such as tilt and roughness)
nsytrough	Uses the UIX and ZIX files to "place" the instrument at that point
nsyttilt	Computes the tilt an InSight instrument would have if placed at the given point
nsytwksp	Computes a mask showing the workspace boundaries for the InSight mission
tp2disp	Generates a set of disparity map images from an input tiepoint file

10. USE CASE SCENARIOS

This section describes how to accomplish various common tasks using VISOR. Many of the tasks you are likely to want to do are covered here, with some variations. However, this is far from a comprehensive list of everything you can do with VISOR. It is intended to give you a flavor of what can be done, as well as show concrete examples of how to do it. Refer to the documentation for the programs (in the `*.pdf` files) for details on any given program. The emphasis here is not on individual programs, but rather showing how the programs work together to accomplish a goal. Feel free to contact MIPL with any questions.

In all cases, the data needed to run these examples is included in the sample data directory of the distribution package (see Section 2.5). You are encouraged to work their way through whichever of these scenarios most interest you, following along by actually running the programs and observing the results (you should copy the data to a working directory first). It is also a good opportunity to try variations, changing parameters to see what happens. Experiment!!

One general note. The focus of VISOR is on functionality, not user friendliness. This is not commercial software. Bad inputs can cause the programs to crash. Values are not always range checked. Simple file-not-found errors sometimes manifest as puzzling crashes (if a program crashes, first make sure all inputs are available and readable). You are welcome to submit fixes to any such issues you find, or just live with them as we do, as the inevitable result of having too few resources to do what we really want.

10.1 Linearization

Programs Used: marscahv
Example Mission: MSL
Example Sol Site Drive: 3546 96 1766
Sample Data Directory: Linearization

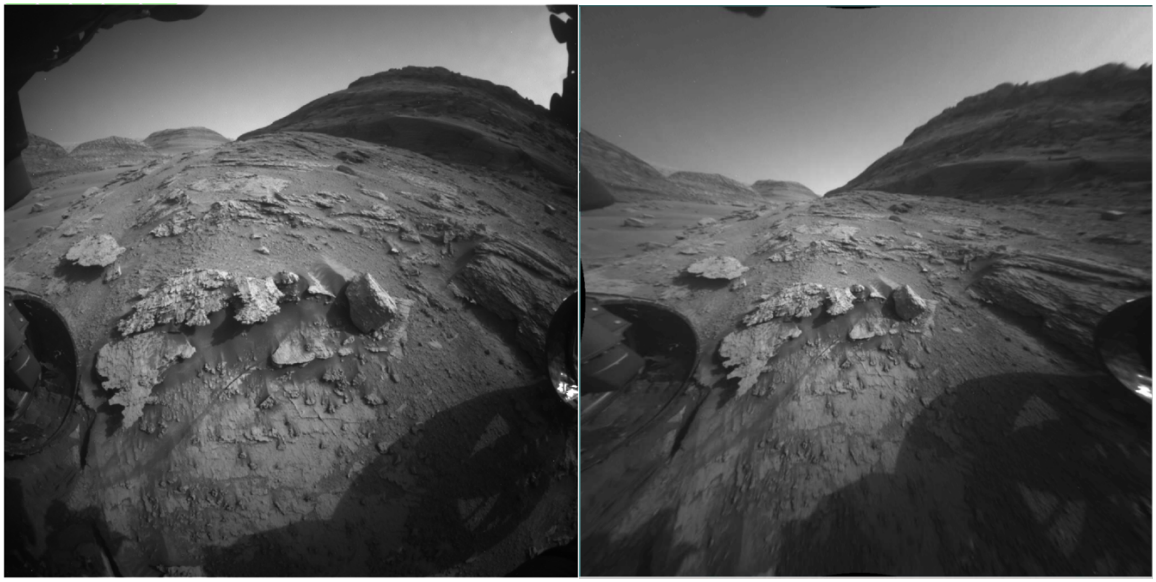
The marscahv program generates a geometrically corrected version of the EDR, applying the C, A, H and V camera model vectors.

EDRs and single-frame RDRs are described by a camera model. This model, represented by a set of vectors and numbers, permit a point in space to be traced into the image plane, and vice-versa. The Navcam, ChemCam, and MMM cameras use a CAHVOR model, while the Hazcams use a more general CAHVORE model. Both model types are nonlinear and involve some complex calculations to transform line/sample points in the image plane to XYZ positions in the scene, in order to deal with lens distortion and fisheye lenses. To simplify this, the images are "warped", or reprojected, in a process often called “linearization”, such that they can be described by a linear CAHV model.

marscahv with stereo partner, which is the “actual” linearization mode:

```
xvd -fit F_712299302EDR_F0961766FHAZ00302M1.vic &
```

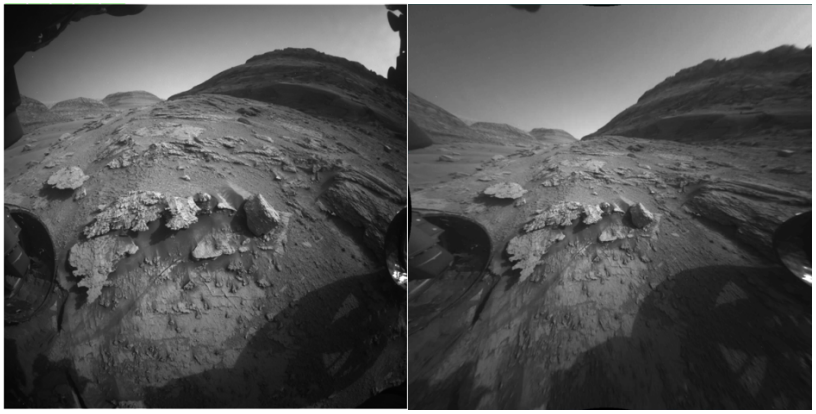
BEFORE: AFTER:



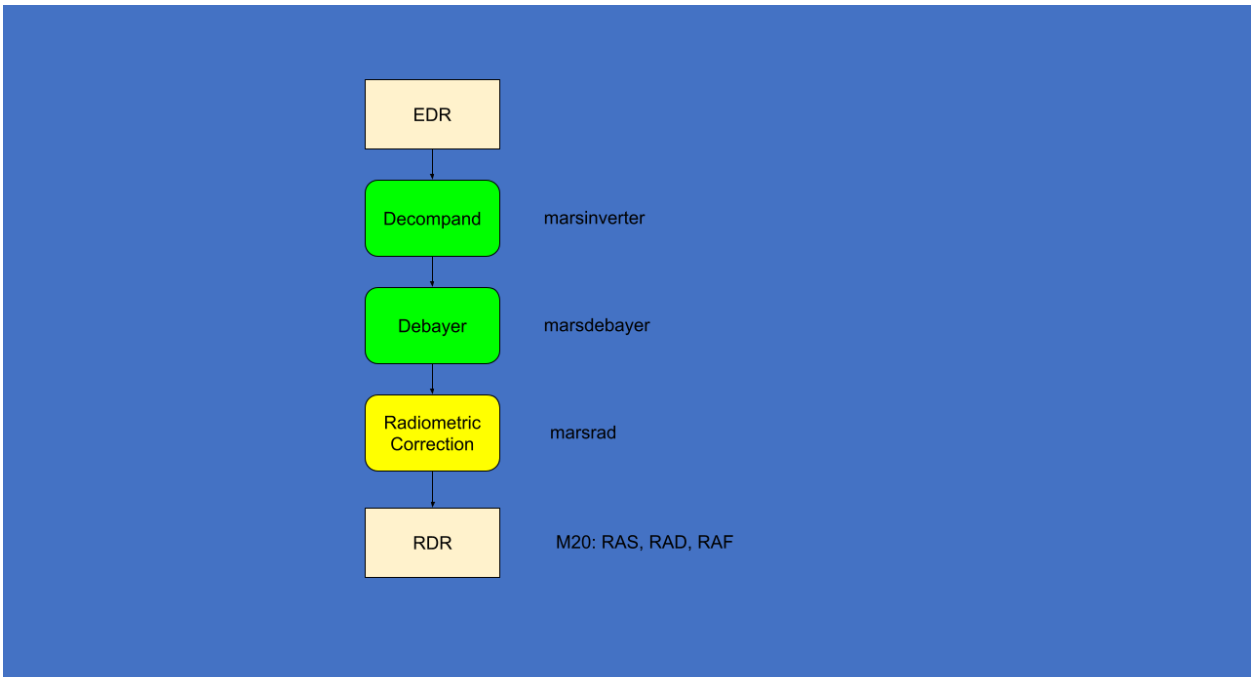
marscahv without stereo partner, which is the “nominal” linearization mode. The stereo partner is constructed by creating a virtual camera using the other eye of the stereo pair with the same pointing. If the image is not a stereo pair, it is simply linearized with itself:

```
$MARSLIB/marscahv inp=FLB_712299302EDR_F0961766FHAZ00302M1.IMG
out=FLB_712299302EDR_F0961766FHAZ00302M1.vic point_method=cahv_fov=min

xvd FLB_712299302EDR_F0961766FHAZ00302M1.vic -fit &
```

BEFORE: AFTER:



10.2 Radiometric Correction

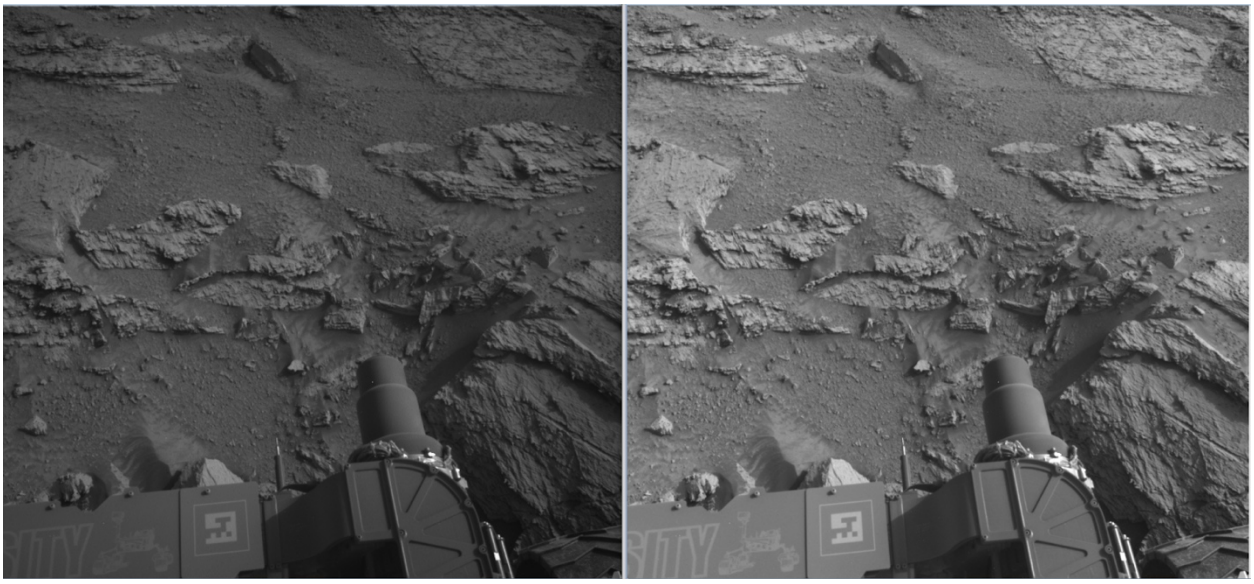


Programs Used: marsrad
Example Mission: MSL
Example Sol Site Drive: 3546 96 1766
Sample Data Directory: RadiometricCorrection

Image radiometric correction can be done using the marsrad VICAR program and is usually done on an edr (or fdr for m20).

```
$MARSLIB/marsrad inp=NLB_712299404EDR_F0961766NCAM00353M1.IMG
out=NLB_712299404RAD_F0961766NCAM00353M1.vic dnscale=100.0 bits=15
```

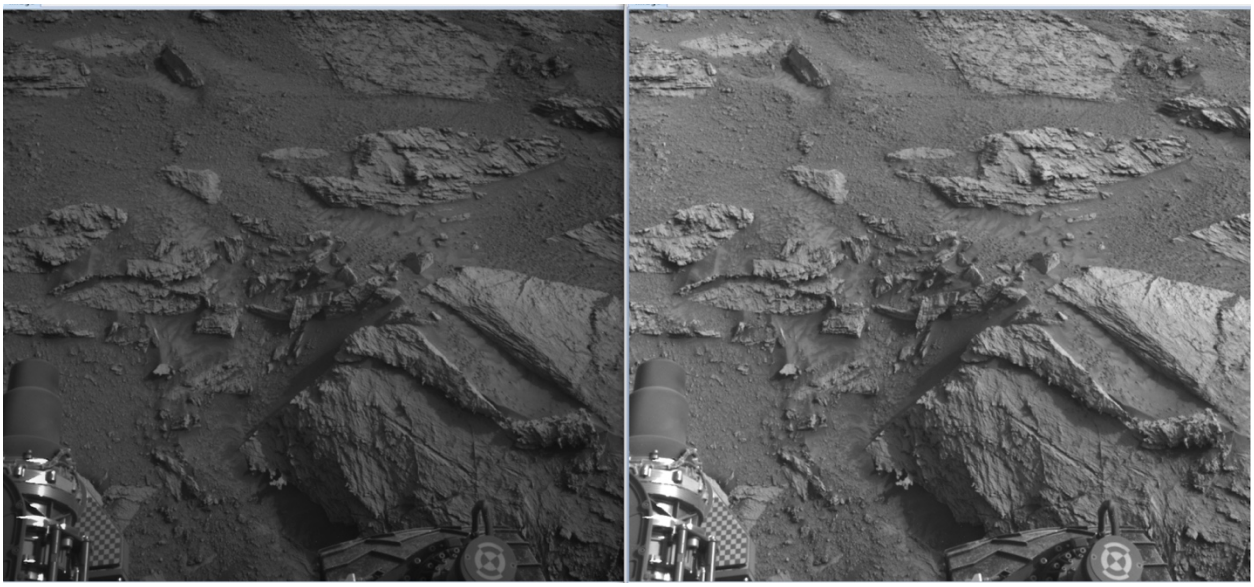
Before: After:



Now repeat for the corresponding right Navcam image.

```
$MARSLIB/marsrad inp=NRB_712299404EDR_F0961766NCAM00353M1.IMG
out=NRB_712299404RAD_F0961766NCAM00353M1.vic dnscale=100.0 bits=15
```

Before: After:



These radiometrically corrected images will be used in a later example to generate xyz and slope overlays.

Note that if the image saturates (or undersaturates) you might want to use the DNSCALE parameter to adjust it. The default is 100, which is a good value for most Mars scenes, but an image near twilight or with very bright areas may need a different DNSCALE.

10.3 Bayer Processing

Programs Used: marsinverter, marsdebayer
Example Mission: M20
Example Sol Site Drive: 295 9 0
Sample Data Directory: BayerProcessing

The MSL Mastcams were the first Mars landed missions to use Bayer-pattern color. InSight followed suit, and many of the M20 cameras are Bayer-pattern. Many of the images are de-Bayered on board, returning color in the form of jpeg or other image type. However, some of the images are returned as a raw Bayer pattern. These must be debayered on the ground in order to be viewed in color. This section shows how to do this using VISOR. Note that VISOR uses the Malvar [Malvar2004] debayering algorithm. Other algorithms are possible, but are not yet implemented in VISOR (although simple interpolation is also available). If you choose to implement a new debayering algorithm, please consider donating the code back to us. See Extending VISOR (Section 12).

First decompand it to the original 12 bits

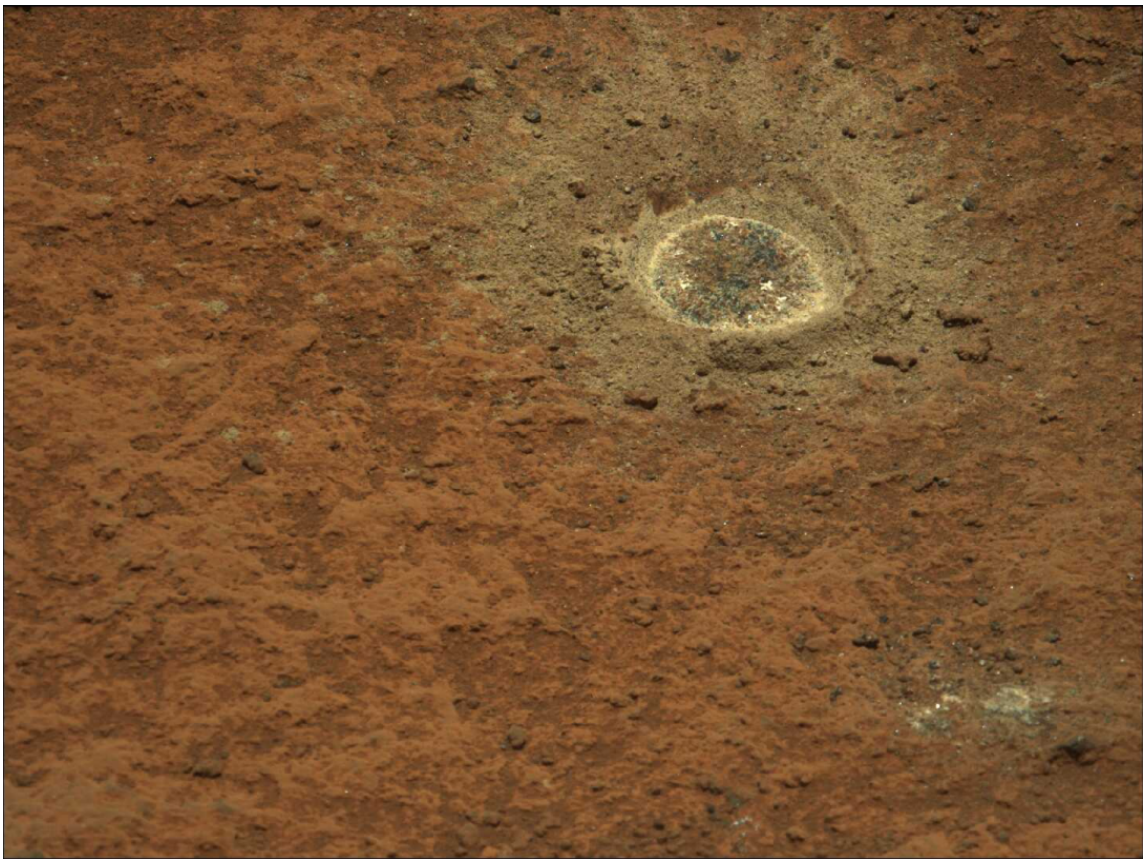
```
$MARSLIB/marsinverter
ZL0_0295_0693125859_803ECM_N0090000ZCAM03273_1100LMJ02.IMG
ZL0_0295_0693125859_803EDR_N0090000ZCAM03273_1100LMJ02.VIC point=cm=label
```

Now run marsdebayer to get color

```
$MARSLIB/marsdebayer
ZL0_0295_0693125859_803EDR_N0090000ZCAM03273_1100LMJ02.VIC
ZL0_0295_0693125859_803EBY_N0090000ZCAM03273_1100LMJ02.VIC -malvar
max_dn=4095 point=cm=label
```

View output

```
xvd ZL0_0295_0693125859_803EBY_N0090000ZCAM03273_1100LMJ02.VIC -fit &
```



10.4 Color Processing Using Filter Wheels

Programs Used: marsrad, spectoxyy, xyy2hdtv, viccub
Example Mission: Phoenix
Example Sol: 9
Sample Data Directory: ColorProcessingUsingFliterWheels

The MER and Phoenix science cameras achieved color by using a color wheel in front of a monochrome sensor. A simple way to convert this to color is to simply take the closest filters available to red, green, and blue, and combining them using `$R2LIB/viccub`. For MER Pancam Left, that's generally filters 2, 5, and 7. That's not exactly correct though, as filter 2 is infrared, not red. And unfortunately, there is no green filter on the Pancam Right.

For cameras like these, approximate true color can be obtained by making use of the wavelength of each available filter. This is done using the `$R2LIB/spec2xyy` and the inaptly named `$R2LIB/xyy2hdtv` (which really converts xyY colorspace to sRGB colorspace).

Let's walk through an example using Phoenix data. For cameras like PHX SSI Left or MER Pancam Right, where there are red (ish) and blue filters, but no green, these programs can be used by repeating the two filters, e.g. (red red blue blue) and bumping the wavelength by 1 for the duplicate. This is needed because the spec2xyy program needs a minimum of 3 inputs. The results of this are not great, but are reasonable for some uses.

Make list file of sub frame color EDRs

```
ls *.img > phx_color.lis
```

The values should be sorted in SCLK order, fix if not (really, in filter order: 1,2,8,a,b,c).

Radiometrically correct each IMG and output new list file. Using a for loop:

```
foreach i (`cat phx_color.lis`)
      set j = `echo $i | sed -e s/ESF/RAD/`
      $MARSLIB/marsrad $i $j
      echo $j >>phx_rad.lis
end
```

Convert data to the xyY colorspace. Get the wavelength values for each filter form the camera SIS (Table 2.1.2 - SSI Spectral Filters (Flight Model)) and use it as input for the **`lamda`** (sic) parameter. The **`range`** parameter is set to **`1.6`** which is the average distance from the sun to Mars measured in Astronomical Units (AU).

```
$R2LIB/spectoxyy \( `cat phx_rad.lis` \) \( phx.x phx.y phx.YY\) lamda=\(672
445 753 604 533 485\) conv=\(.01 .01 .01 .01 .01 .01\) range=1.6
```

Convert data to sRGB for display.

```
$R2LIB/xyy2hdtv \( phx.x phx.y phx.YY \) \( phx.r phx.g phx.b \)
```

Combine RGB data

```
$R2LIB/viccub \( phx.r phx.g phx.b \) phx.rgb
```

View output

```
xvd phx.rgb -fit &
```

10.5 Radiometric Correction, Part 2

Programs Used: marsinverter, marsrad
Example Mission: InSight
Example Sol: 8
Sample Data Directory: CalibratedColorProcessing

This section shows a broader range of radiometrically corrected products, using InSight as an example. They look much the same here, but the various forms have different uses. For example, the 12-bit RAS is useful for meshes where you want a constant (0-4k) data range, RAS is useful for a wider dynamic range with less chance of saturation, RZS is useful when making mosaics of data acquired at different times of day, and RAY is useful to maximally preserve the input dynamic range when you are not intending to mosaic the results.

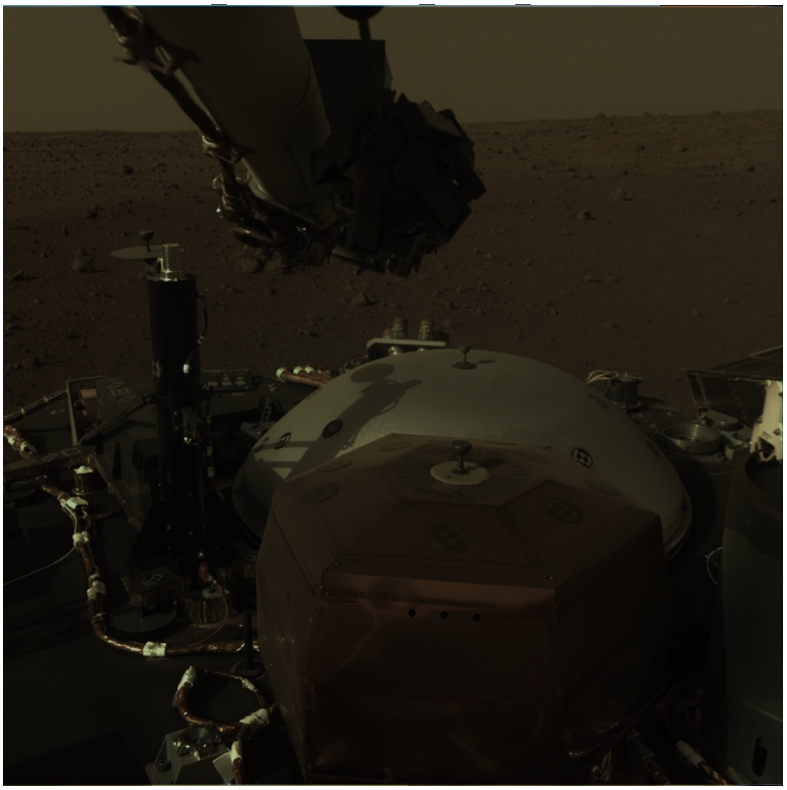
With the exception of RZS, this is all about different uses of DNSCALE, either explicitly or implicitly, to scale the fundamentally floating-point rad result to integer for easier use.

First, run marsinverter to convert from byte to 12-bit data:

```
$MARSLIB/marsinverter D000M0008_597250940EDR_F0000_0133M2.VIC
D000M0008_597250940ILT_F0000_0133M2.VIC
```

Create a 12-bit radiometrically correct image:

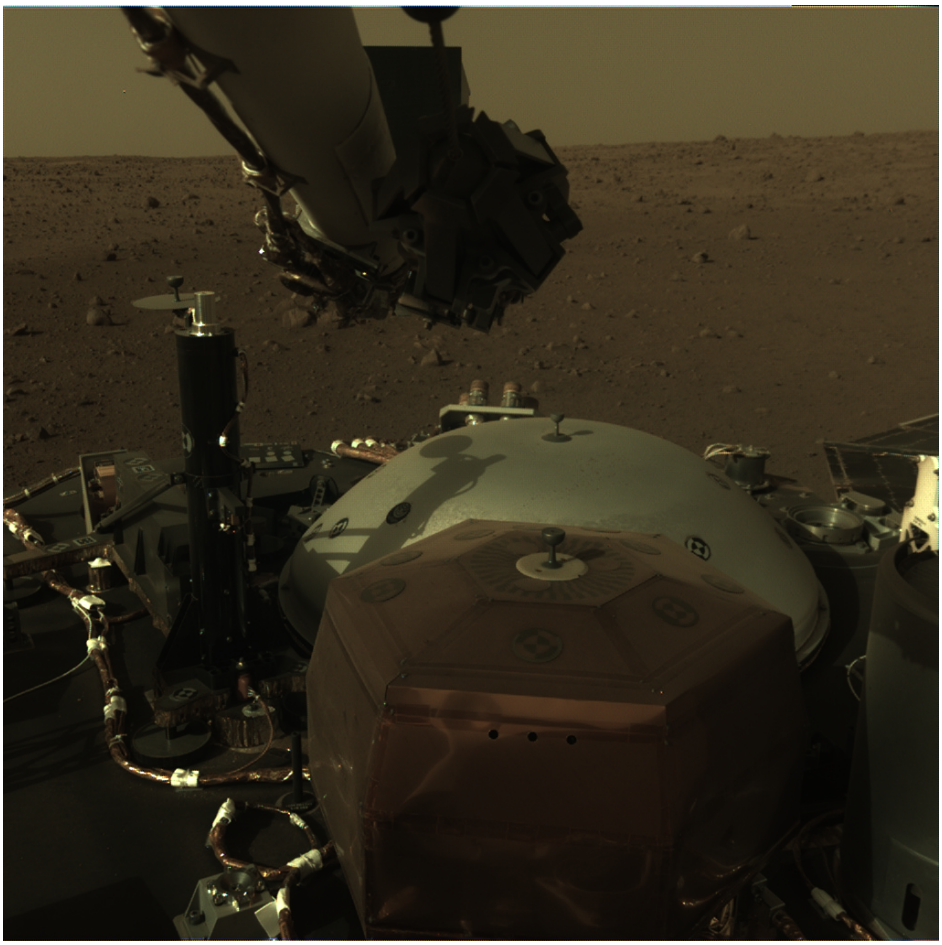
```
$MARSLIB/marsrad D000M0008_597250940ILT_F0000_0133M2.VIC
D000M0008_597250940RAS_F0000_0133M2.VIC dnscale=100 bits=12

xvd D000M0008_597250940RAS_F0000_0133M2.VIC -fit &
```



Create a 15-bit radiometrically correct image. Note the dnscale is bigger to use more of the

available dynamic range:

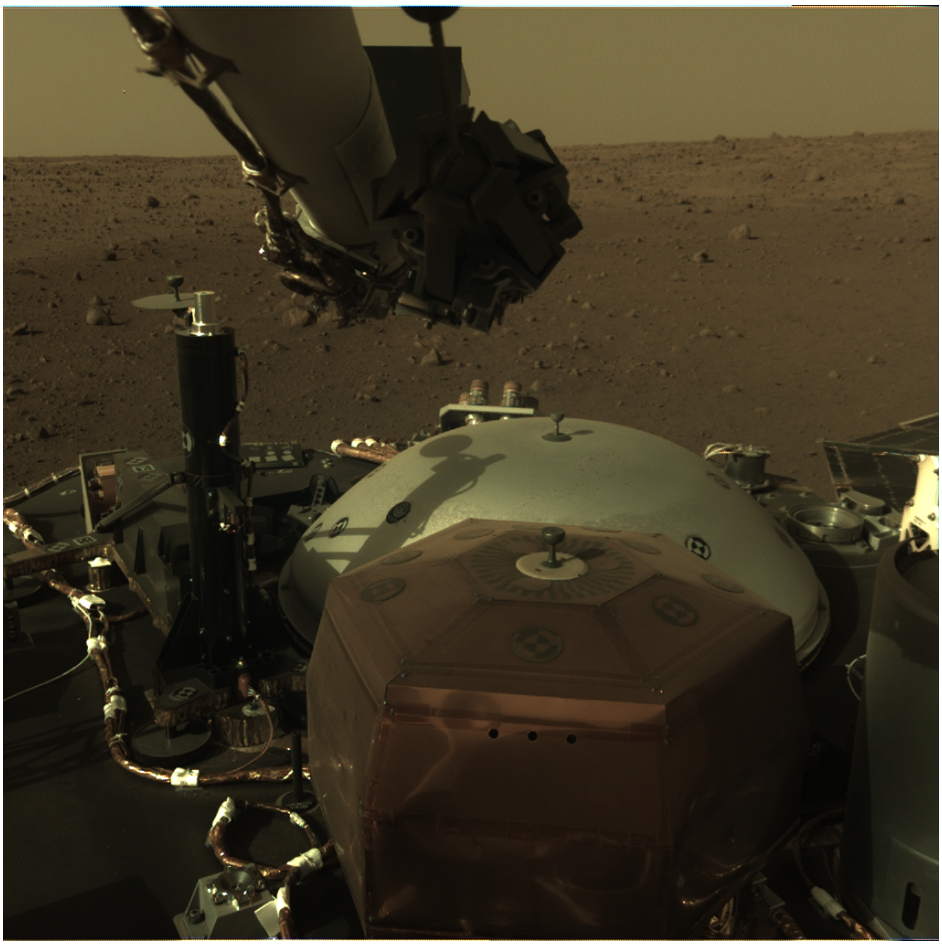
```
$MARSLIB/marsrad D000M0008_597250940ILT_F0000_0133M2.VIC
D000M0008_597250940RAD_F0000_0133M2.VIC dnscale=400 bits=15

xvd D000M0008_597250940RAD_F0000_0133M2.VIC &
```



You may need to adjust the Data Range on xvd (say, 0 to 4095). Edit/Data Range from the menu or ctrl-D.

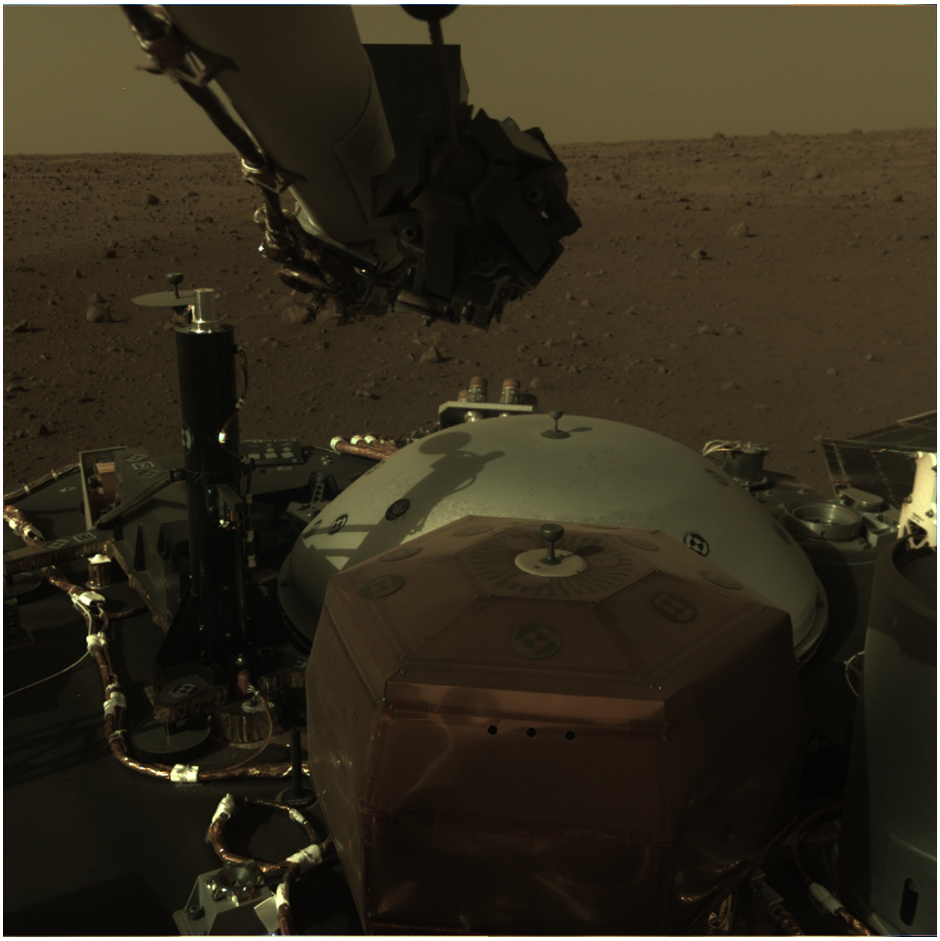
Zenith-corrected image, which partially corrects for lighting conditions:

```
$MARSLIB/marsrad D000M0008_597250940ILT_F0000_0133M2.VIC
D000M0008_597250940RZS_F0000_0133M2.VIC dnscale=100 bits=12 -zenith

xvd D000M0008_597250940RZS_F0000_0133M2.VIC &
```



Dynamically scaled radiometrically corrected image, which preserves the optimal dynamic range in the output but is not consistent between frames (so you would not want to make a mosaic out of these). The effective difference is that the DNSCALE is automatically determined based on the exposure time.

```
$MARSLIB/marsrad D000M0008_597250940ILT_F0000_0133M2.VIC
D000M0008_597250940RAY_F0000_0133M2.VIC -dynamic

xvd D000M0008_597250940RAY_F0000_0133M2.VIC &
```



10.6 Calibrated Color Processing

Programs Used: marsinverter, marsrad, marscolor, gamma
Example Mission: InSight
Example Sol: 8
Sample Data Directory: CalibratedColorProcessing

This section talks about how to run the various color programs to get various flavors of calibrated color, using InSight IDC as an example. See the InSight SIS for details on what all the various color spaces are.

Note that all of these should be considered "approximate" true color. Color processing is a very tricky topic and could be the subject of entire papers. We make no representation here as to the quality of the color calibration parameters, consult the appropriate mission SIS for that. Rather, this discusses the mechanics of how to call the programs. See [Maki2020b] for details on the color spaces.

As of this writing, only M20 and InSight have a full set of color calibration parameters. See for example NSYT_flat_fields.parms in the NSYT calibration directory (under param_files).

First, run marsinverter to convert from byte to 12-bit data:

```
$MARSLIB/marsinverter D000M0008_597250940EDR_F0000_0133M2.VIC
D000M0008_597250940ILT_F0000_0133M2.VIC
```

This time, run rad to a float image, so scaling is not a factor, and use that for the subsequent steps.

```
$MARSLIB/marsrad D000M0008_597250940ILT_F0000_0133M2.VIC
D000M0008_597250940RAF_F0000_0133M2.VIC -identity -real
```

Now run marscolor to convert it to XYZ colorspace (still as a float). Note that the filename is CNF_T not CNF_F to indiciate Tristimulus colorspace.

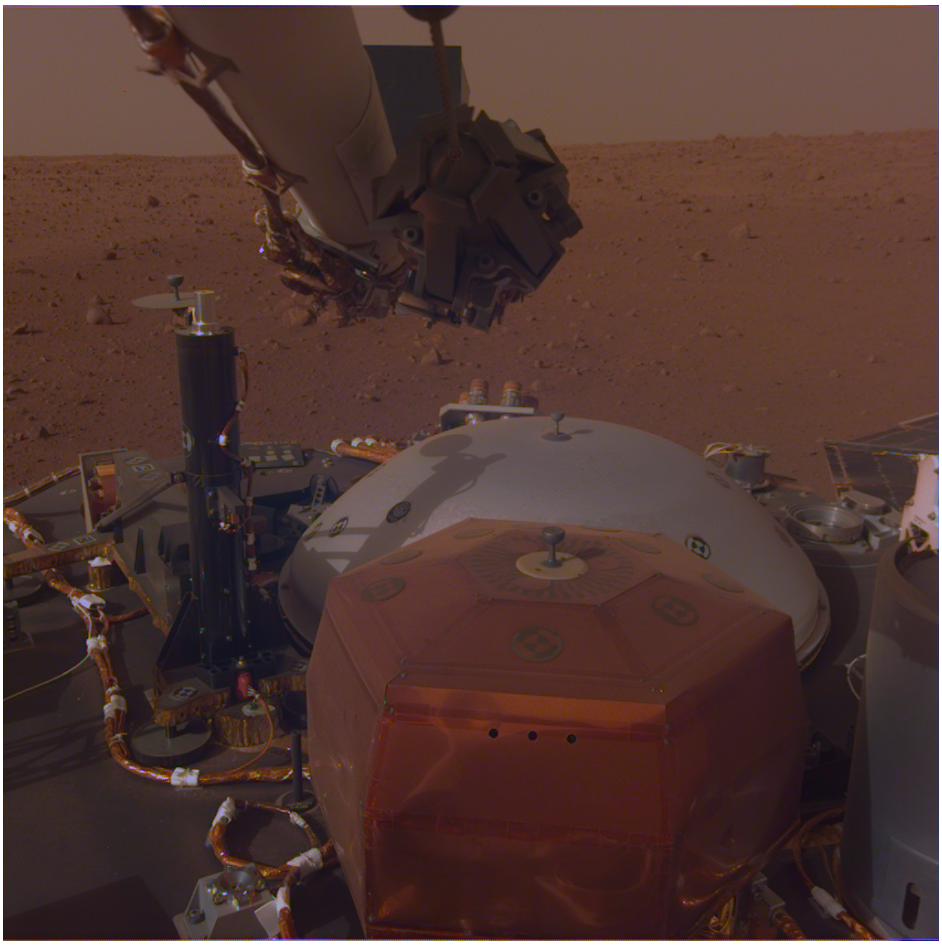
```
$MARSLIB/marscolor D000M0008_597250940RAF_F0000_0133M2.VIC
D000M0008_597250940CNF_T0000_0133M2.VIC -xyz -real -identity
```

Convert that to sRGB color space:

```
$MARSLIB/marscolor D000M0008_597250940CNF_T0000_0133M2.VIC
D000M0008_597250940CSS_F0000_0133M2.VIC -srgb dnscale=.4 -half bits=12
```

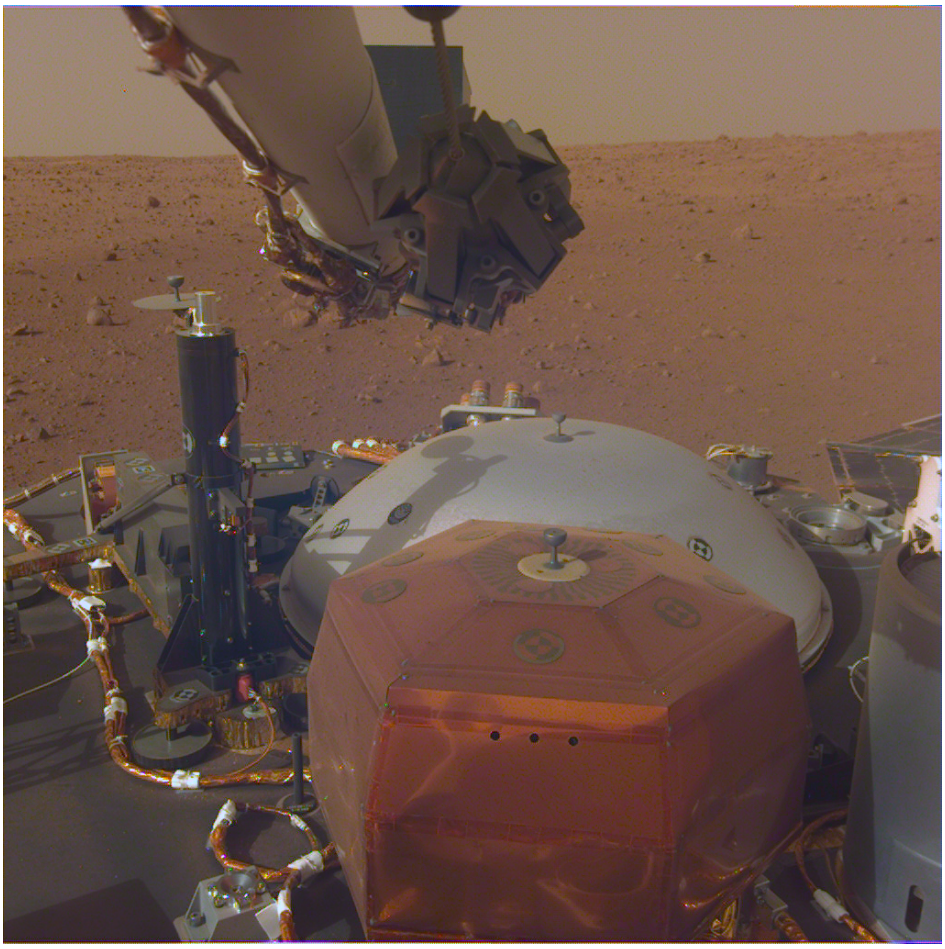
And convert to byte using a gamma encoding so it looks natural on most displays:

```
$R2LIB/gamma D000M0008_597250940CSS_F0000_0133M2.VIC
D000M0008_597250940CSG_F0000_0133M2.VIC -srgb

xvd D000M0008_597250940CSG_F0000_0133M2.VIC &
```



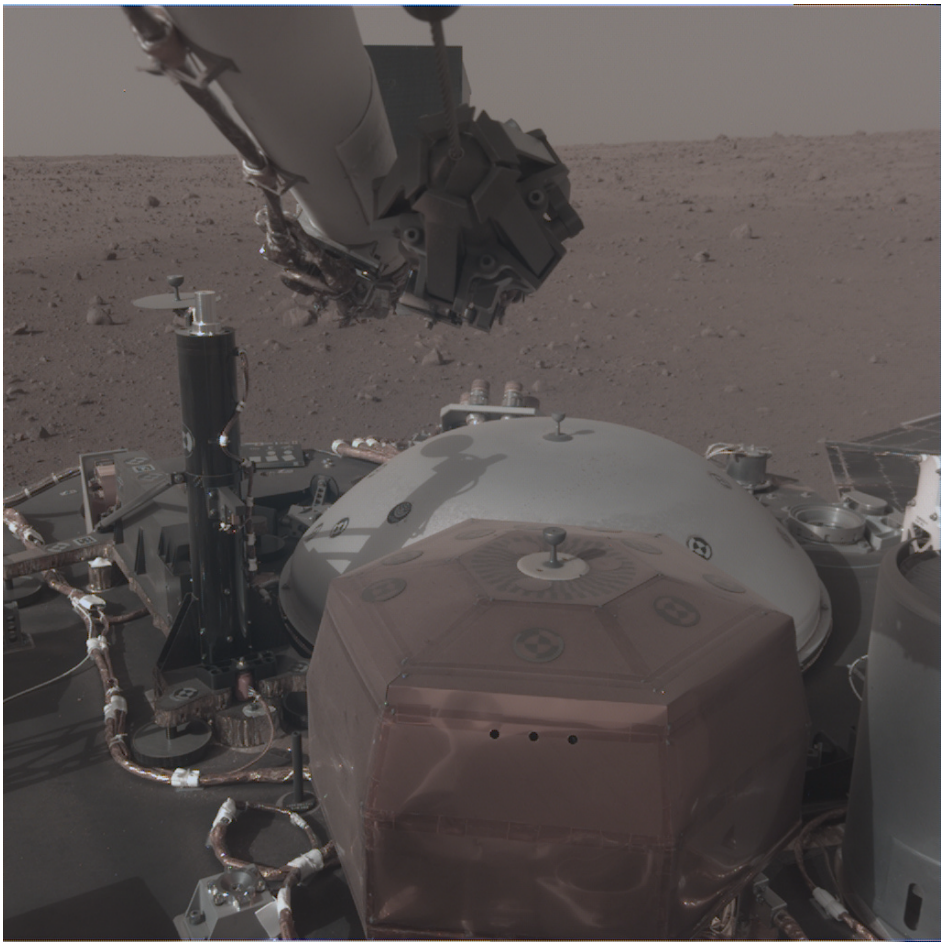
Now we convert to white-balanced sRGB using a Mars illuminant, and gamma correction. Note that this starts from the CSS produced above.

```
$MARSLIB/marscolor D000M0008_597250940CSS_F0000_0133M2.VIC
D000M0008_597250940CPS_F0000_0133M2.VIC -prgb dnscale=1 -half bits=12

$R2LIB/gamma D000M0008_597250940CPS_F0000_0133M2.VIC
D000M0008_597250940CPG_F0000_0133M2.VIC -srgb

xvd D000M0008_597250940CPG_F0000_0133M2.VIC &
```

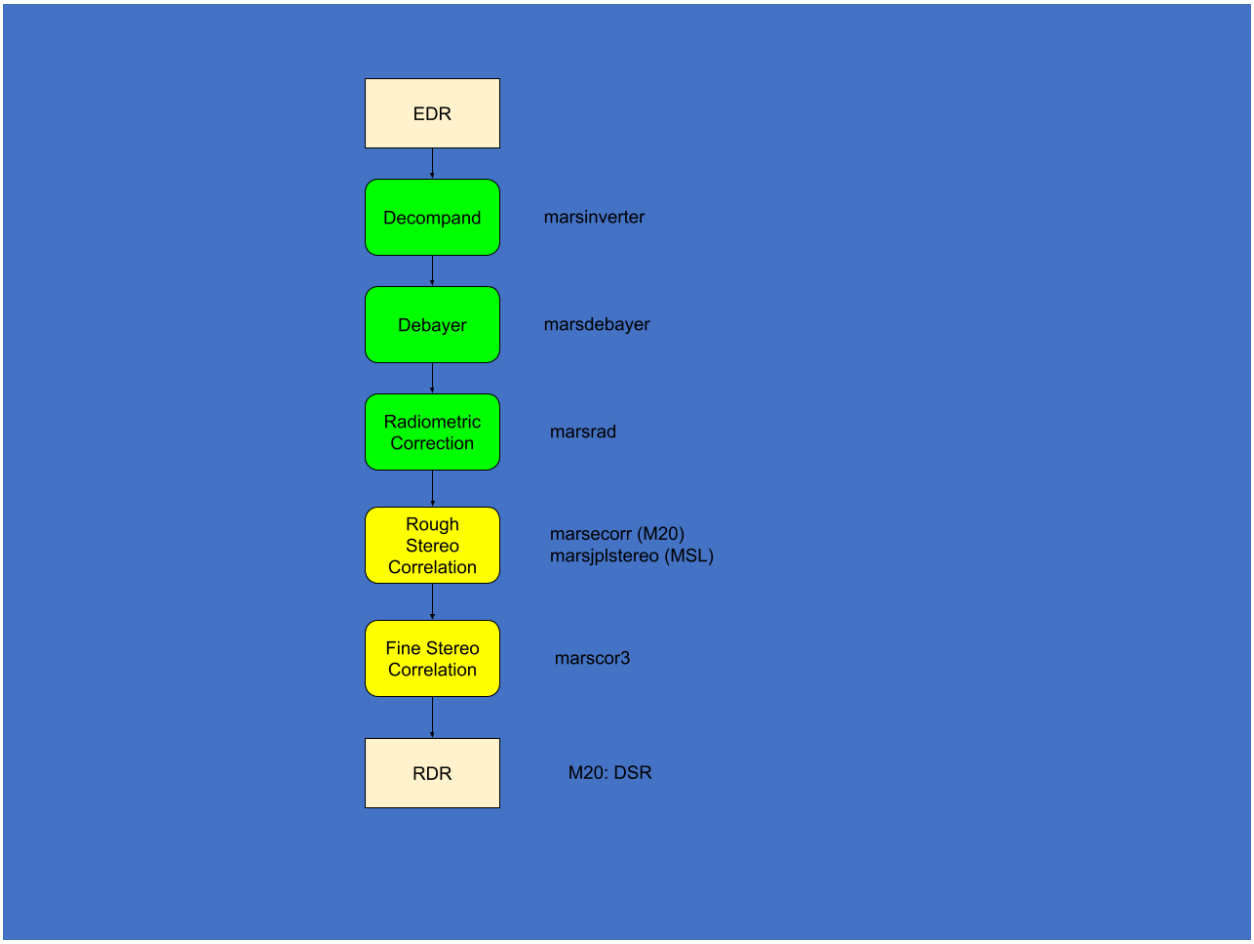


Finally, we convert to a white-balanced iRGB space, with gamma correction:

```
$MARSLIB/marscolor D000M0008_597250940RAF_F0000_0133M2.VIC
D000M0008_597250940CWS_F0000_0133M2.VIC -wrgb dnscale=1 -half bits=12

$R2LIB/gamma D000M0008_597250940CWS_F0000_0133M2.VIC
D000M0008_597250940CWG_F0000_0133M2.VIC -srgb

xvd D000M0008_597250940CWG_F0000_0133M2.VIC &
```



10.7 Stereo Correlation – Script



Programs used: marsecorr, marscor3, marsdispcompare, marsmask, marsxyz, marsrfilt, transcoder (optional)
Example Mission: M20
Example Sol Site Drive: 175 6 1752
Sample Data Directory: StereoCorrelationWithScript

Running stereo correlation is one of the most important jobs for VISOR. The full process is rather complex so instead of going through each step manually, we'll use a processing script (but see below for another example without the script). Please feel free to modify the script as needed. The intent of the script is to show an example of how the data are processed. You may very well want to adjust parameters to the various programs, either based on history labels for the PDS data for the mission you are working with, or to tweak the results in some way. The script is just a starting point.

The script is in the sample data tarball in the Scripts directory. Note that there's a run_corr which we use here, but also a run_corr_rcam which is specifically tuned for the m20 hazcams by using the coefficients file to communicate between marsecorr and marscor3.

There are two processing chains: linearized and non-linearized. Prior to InSight, the linearized chain was used. The nonlinearized chain developed for InSight and M20 is superior though and

should be usable in earlier missions as well. That is the chain we exercise here.

For reference, the linearized chain operates as follows.

- Start with an image pair, generally RAS or RAD (radiometrically corrected) but could be anything
- Linearize the images with marscahv. If they are not nominal stereo pairs, use stereo_partner parameter to marscahv to linearize them to each other
- Run the first stage correlator, marsjplstereo, on both L->R and R->L (DFFL)
- Run the second stage correlator, marscor3, on both L->R and R->L (DSRL)
- Run marsdispcompare to reconcile the L->R and R->L runs, and marsmask to mask off the bad disparities (MDSL, DSPL)
- Run marsxyz to create linearized XYZ (optionally) (XYZL)
- Run marsunlinearize to create a non-linearized initial correlation (DFF)
- Run the second stage correlator (marscor3) again on both L->R and R->L (DSR)
- Run marsdispcompare and marsmask to reconcile the runs (MDS, DSP)
- Run marsxyz to create XYZ images (XRZ)
- Run marsrfilt to range-filter those (optionally) (XYZ)

Not all those steps are done on every mission, for example on MER the R->L correlation and reconciliation was not done. As a reminder, specific program parameters can be extracted from the PDS product history labels, if you want to reproduce this chain.

The nonlinearized chain used on InSight and M20 and discussed here is similar but uses a different first-stage correlator and skips the linearization steps.

- Start with an image pair, generally RAS or RAD (radiometrically corrected) but could be anything
- Run the first stage correlator, marsecorr, on both L->R and R->L (DFF)
- Run the second stage correlator, marscor3, on both L->R and R->L (DSR)
- Run marsdispcompare to reconcile the L->R and R->L runs, and marsmask to mask off the bad disparities (MDS, DSP)
- Run marsxyz to create XYZ images (XRZ)
- Run marsrfilt to range-filter those (optionally) (XYZ)

Skipping linearization leads to much better flexibility on the stereo correlation, as certain geometries are not amenable to epipolar alignment (for example, motion toward or away from the target).

Run stereo script, setting $path to wherever the sample data scripts are.

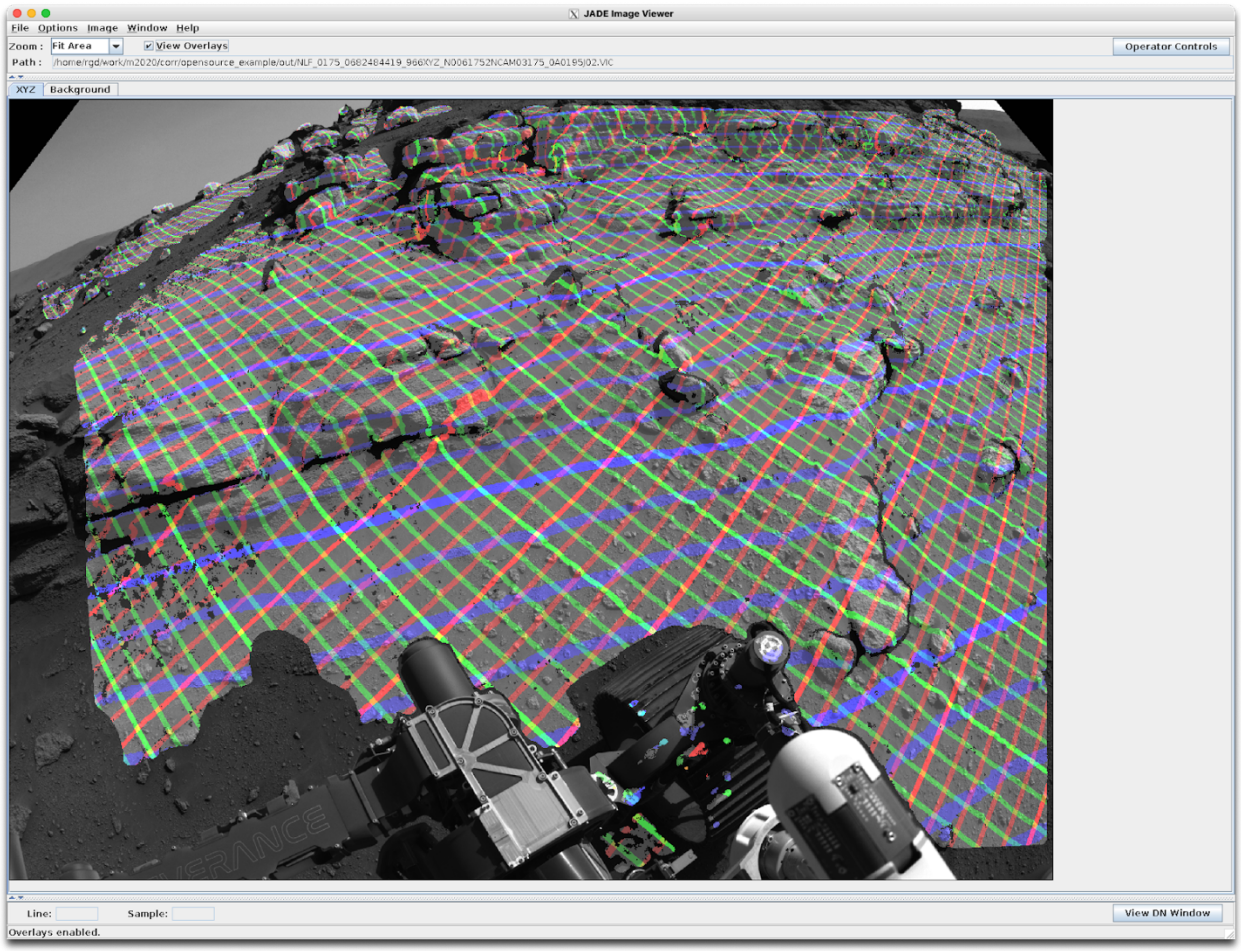
```
set path = (/path/to/sample/data/Scripts $path)

run_corr NLF_0175_0682484419_966RAS_N0061752NCAM03175_0A0195J02.IMG
NRF_0175_0682484419_966RAS_N0061752NCAM03175_0A0195J03.IMG
```

It will run the transcoder to convert all the VIC images to dual-labeled IMG form. If the transcoder is not in the path, these calls will fail, but the script will continue to run just fine (transcoding is optional).

Run jadeviewer to see L-side XYZ overlaid on the image:

```
java jpl.mipl.jade.viewer.ImageViewer
NLF_0175_0682484419_966XYZ_N0061752NCAM03175_0A0195J02.VIC XYZ
NLF_0175_0682484419_966RAS_N0061752NCAM03175_0A0195J02.IMG &
```

Select the View Overlays button to turn on the background, and (optionally) Fit Area.

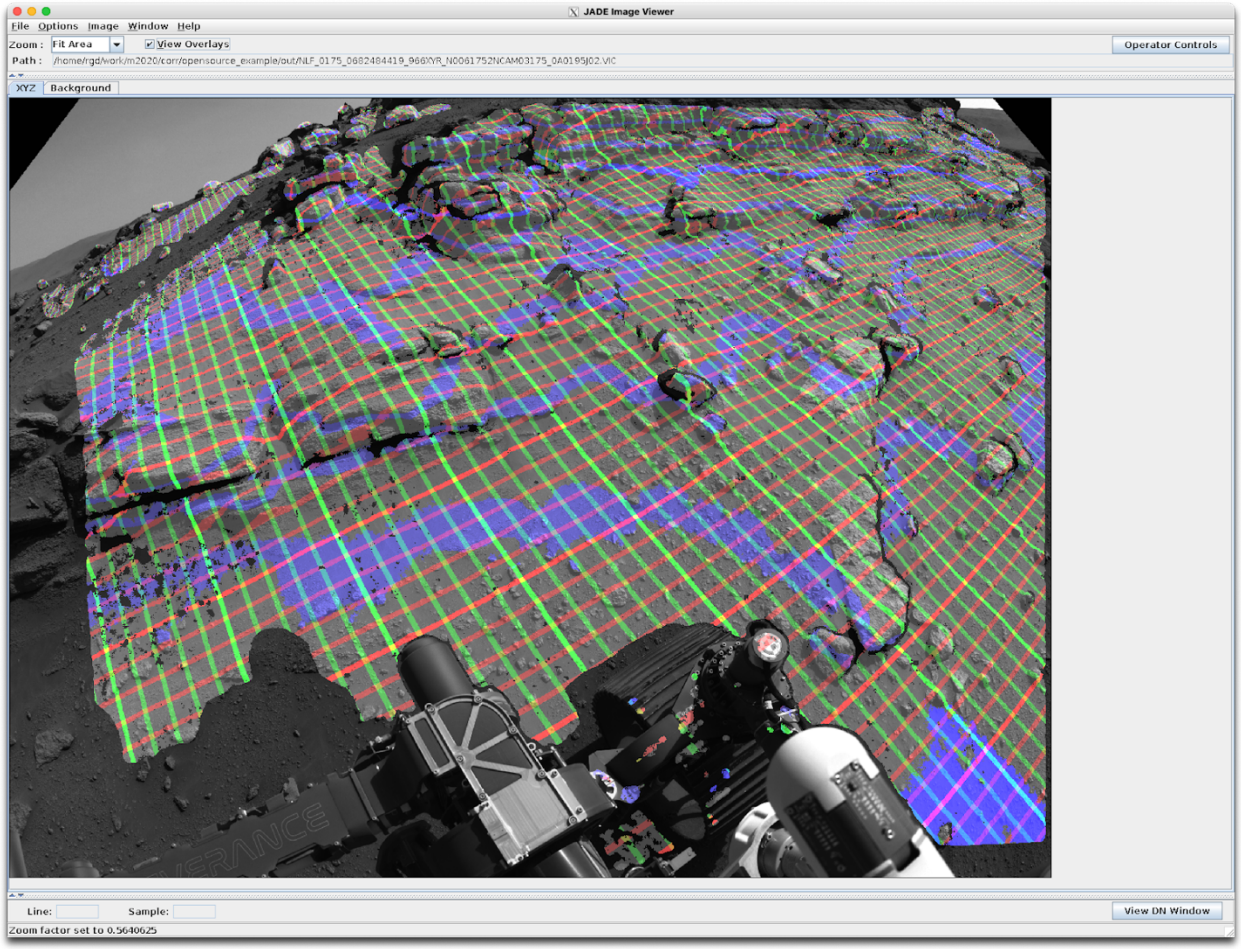


Note that the overlays show the axis orientation. Red are lines of constant X (0.1m spacing by default), green are lines of constant Y, and blue are constant Z. The red and green lines show the XY coordinate grid while the blue lines are elevation contours. In this case, we are close to a hillside so the contours indicate a significant slope going up and away. The Operator Controls button in the upper left can be used to change the grid spacing. We have found over the years that it looks best when the contour interval is 10x the contour width, although other ratios are possible.

The XYZ image shown has coordinates expressed in Site frame. The XYR image expresses the values in Rover frame. Note how the X and Y grid aligns with the arm (which is square to the front of the rover). Also the Z contours are more meandering, because the rover is actually sitting on the slope and is thus tilted along with the terrain, so from the rover's perspective the terrain is "flatter" in Z than it is in the gravity-aligned Site frame.

```
java -Dsun.java2d.xrender=false jpl.mipl.jade.viewer.ImageViewer
NLF_0175_0682484419_966XYR_N0061752NCAM03175_0A0195J02.VIC XYZ
NLF_0175_0682484419_966RAS_N0061752NCAM03175_0A0195J02.IMG &
```

As noted earlier, the -D option may be needed when rendering on a Mac using X-windows, and should probably be omitted in other cases.



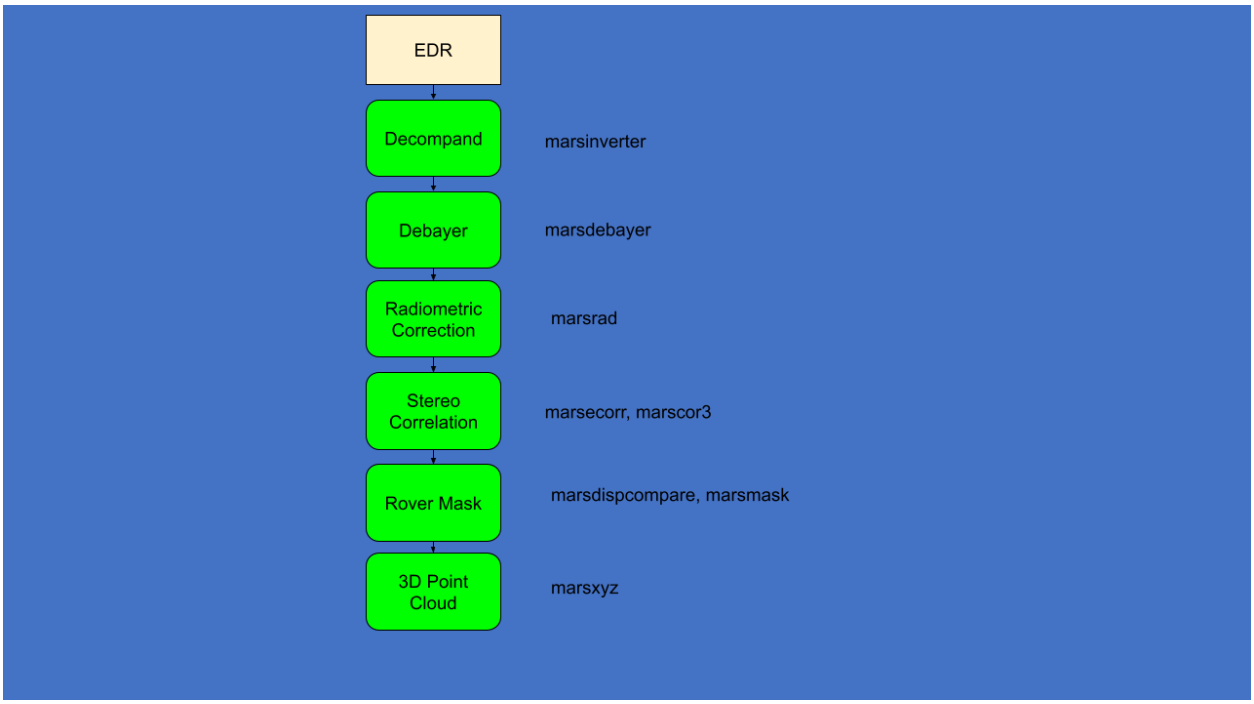
In order to interpret the difference between these two, let's look at the rover orientation.

```
$R2LIB/label -list NLF_0175_0682484419_966XYZ_N0061752NCAM03175_0A0195J02.IMG
| grep -A10 ROVER_COORDINATE_SYSTEM | grep QUAT

ORIGIN_ROTATION_QUATERNION=(0.138727, 0.161278, 0.0603893, -0.975242)
QUATERNION_MEASUREMENT_METHOD='FINE'

java jpl.mipl.mars.rmc/QuaternionTranslation "(0.138727, 0.161278, 0.0603893,
-0.975242)"

*** Quaternion ************************************
  (0.138727, 0.161278,
   0.0603893, -0.975242)
***************************************************
```

```
*** Euler Angle ************************************
  ROLL  : -4.439924343492795 degrees
  PITCH : 19.34925036789371 degrees
  YAW   : -164.56539283029315 degrees
*****************************************************

*** Axis-Angle *************************************
  AXIS : (0.1628526776357653, 0.060978925864343066,
          -0.9847640164365817)
  ANGLE: 164.0516201518735 degrees
*****************************************************
```

This shows that the rover is tilted up 19.3, almost 20 degrees, which explains the Z contour differences. It also shows a yaw of -164 degrees, which is 16 degrees off from 180. The Site frame image indeed shows about a 16 degree rotation of the red lines with respect to the arm (compare to the Rover frame image). (The grid lines look the same at azimuths 0 and 180.)

10.8 Stereo Correlation - Separate Programs



Program used: marsinverter, marsrelabel, marsrad, marscahv, marsjplstereo, marsdispinvert, marscor3, marsdispcompare, marsmask, marsxyz
Example Mission: MSL
Example Sol Site Drive: 3546 96 1766
Sample Data Directory: StereoCorrelation

This section goes through the stereo correlation process step by step, using MSL data as an example, and starting earlier in the processing chain. The script above is recommended, but the hope is that this section might help explain the critical correlation process better. This section also uses the linearized flow.

This example starts from the EDR, the first image in the processing chain. The first step is to decompand the image, which will expand the image back to the original 12 bits if it was compressed to 8 bits. In this particular case, this is a no-op, but in general the step is needed. Note, for M20 this goes ECM -> EDR rather than EDR -> ILT.

Decompand left and right ncam EDRs

```
$MARSLIB/marsinverter inp=NLB_712299404EDR_F0961766NCAM00353M1.IMG
out=NLB_712299404ILT_F0961766NCAM00353M1.VIC

$MARSLIB/marsinverter inp=NRB_712299404EDR_F0961766NCAM00353M1.IMG
out=NRB_712299404ILT_F0961766NCAM00353M1.VIC
```

Update the camera model in the label. This is an optional step, which you probably do not need. It serves as a useful example though. This step will replace the camera model in the image with the camera model in the calibration directory (properly pointed for this image). If you

had your own camera models, or if camera models were updated during the mission and you have data from before the replacement, this step would be useful to replace those models. This is also the most reliable way of incorporating the results of bundle adjustment (via a .NAV file) back into the image in order to reprocess XYZ data, in which case add a nav=xxx.NAV parameter to the marsrelabel call.

Note that on most missions, such a label replacement is done by the pipeline for the EDR, because the ground-based camera models are more up to date than the flight ones.

```
mv NLB_712299404ILT_F0961766NCAM00353M1.VIC
NLB_712299404ILT_F0961766NCAM00353M1.tmp

$MARSLIB/marsrelabel inp=NLB_712299404ILT_F0961766NCAM00353M1.tmp
out=NLB_712299404ILT_F0961766NCAM00353M1.VIC -cm

mv NRB_712299404ILT_F0961766NCAM00353M1.VIC
NRB_712299404ILT_F0961766NCAM00353M1.tmp

$MARSLIB/marsrelabel inp=NRB_712299404ILT_F0961766NCAM00353M1.tmp
out=NRB_712299404ILT_F0961766NCAM00353M1.VIC -cm
```

If you replace the camera model with the cal model, nothing special need be done. If, however, you apply a bundle adjustment result (nav file) to change the pointing, then every subsequent program invocation needs to have the parameter `point=cm=label` , which tells it to use the camera model from the label rather than recomputing it. If there are other point= options, such as for marscahv below, they can be combined: `point='"cm=label,cahv_fov=min"'` (that's double quotes inside single quotes).

Next, apply radiometric correction

```
$MARSLIB/marsrad inp=NLB_712299404ILT_F0961766NCAM00353M1.VIC
out=NLB_712299404RAD_F0961766NCAM00353M1.VIC dnscale=100 bits=15
$MARSLIB/marsrad inp=NRB_712299404ILT_F0961766NCAM00353M1.VIC
out=NRB_712299404RAD_F0961766NCAM00353M1.VIC dnscale=100 bits=15
```

Linearize the images. This converts the images from the raw CAHVOR/CAHVORE camera models, which include radial distortion and fisheye terms, to linear CAHV (pinhole camera) models. Importantly, it also epipolar-aligns the images, meaning that matching features appear on the same line on the left and right images (within camera model error). The method used here assumes the images are a "natural" stereo pair. If they are not, such as for long-baseline stereo or cross-camera stereo, use the stereo_partner parameter on each marscahv call to linearize to that specific partner.

```
$MARSLIB/marscahv inp=NLB_712299404RAD_F0961766NCAM00353M1.VIC
out=NLB_712299404RADLF0961766NCAM00353M1.VIC point_method=cahv_fov=min

$MARSLIB/marscahv inp=NRB_712299404RAD_F0961766NCAM00353M1.VIC
out=NRB_712299404RADLF0961766NCAM00353M1.VIC point_method=cahv_fov=min
```

Recall the discussion in the script-based stereo section. There are situations where linearization will not work. For that reason, it is better to use the non-linearized chain if at all possible. However, because so much data has been generated using the linearized chain (all of MER, Phoenix, and MSL), we continue with the linearized chain here as an example.

Generate the first-stage disparity map, using the linearized correlator marsjplstereo. For the right side, we cannot run marsjplstereo (it works only with positive disparities), so we invert the disparity map to turn L->R into R->L. This inverted map should never be used directly but works fine as a seed for marscor3.

```
$MARSLIB/marsjplstereo
inp=\(NLB_712299404RADLF0961766NCAM00353M1.VIC,NRB_712299404RADLF0961766NCAM0
0353M1.VIC\) out=NLB_712299404DFFLF0961766NCAM00353M1.VIC pyrlevel=3
windowsize=13 maxdisp=2032 blobsize=50

$MARSLIB/marsdispinvert
inp=\(NRB_712299404RADLF0961766NCAM00353M1.VIC,NLB_712299404RADLF0961766NCAM0
0353M1.VIC\) out=NRB_712299404DFFLF0961766NCAM00353M1.VIC
in_disp=NLB_712299404DFFLF0961766NCAM00353M1.VIC disp_pyramid=3
```

Refine the first-stage disparity map to make a higher-quality second-stage map

```
$MARSLIB/marscor3
inp=\(NLB_712299404RADLF0961766NCAM00353M1.VIC,NRB_712299404RADLF0961766NCAM0
0353M1.VIC\) out=NLB_712299404DSRLF0961766NCAM00353M1.VIC
in_disp=NLB_712299404DFFLF0961766NCAM00353M1.VIC template=\(7,11\) search=25
quality=0.5 -gores gore_quality=0.6 gore_passes=3 -gore_reverse
disp_pyramid=3 mode=amoeba4 ftol=0.004 -multipass -filter

$MARSLIB/marscor3
inp=\(NRB_712299404RADLF0961766NCAM00353M1.VIC,NLB_712299404RADLF0961766NCAM0
0353M1.VIC\) out=NRB_712299404DSRLF0961766NCAM00353M1.VIC
in_disp=NRB_712299404DFFLF0961766NCAM00353M1.VIC template=\(7,11\) search=25
quality=0.5 -gores gore_quality=0.6 gore_passes=3 -gore_reverse
disp_pyramid=3 mode=amoeba4 ftol=0.004 -multipass -filter
```

Cross-compare left and right disparity maps to get rid of bad matches. The L point is mapped through the L->R map to a R point, then back through the R->L map to an L point. If the transformed L point doesn't match the original within tolerance, it is discarded.

```
$MARSLIB/marsdispcompare
\(NLB_712299404DSRLF0961766NCAM00353M1.VIC,NRB_712299404DSRLF0961766NCAM00353
M1.VIC\) out=NLB_712299404MDSLF0961766NCAM00353M1.VIC

$MARSLIB/marsdispcompare
\(NRB_712299404DSRLF0961766NCAM00353M1.VIC,NLB_712299404DSRLF0961766NCAM00353
M1.VIC\) out=NRB_712299404MDSLF0961766NCAM00353M1.VIC
```

Apply the mask generated above to create the final disparity map.

```
$MARSLIB/marsmask inp=NLB_712299404DSRLF0961766NCAM00353M1.VIC
out=NLB_712299404DSPLF0961766NCAM00353M1.VIC
mask=NLB_712299404MDSLF0961766NCAM00353M1.VIC
```

```
$MARSLIB/marsmask inp=NRB_712299404DSRLF0961766NCAM00353M1.VIC
out=NRB_712299404DSPLF0961766NCAM00353M1.VIC
mask=NRB_712299404MDSLF0961766NCAM00353M1.VIC
```

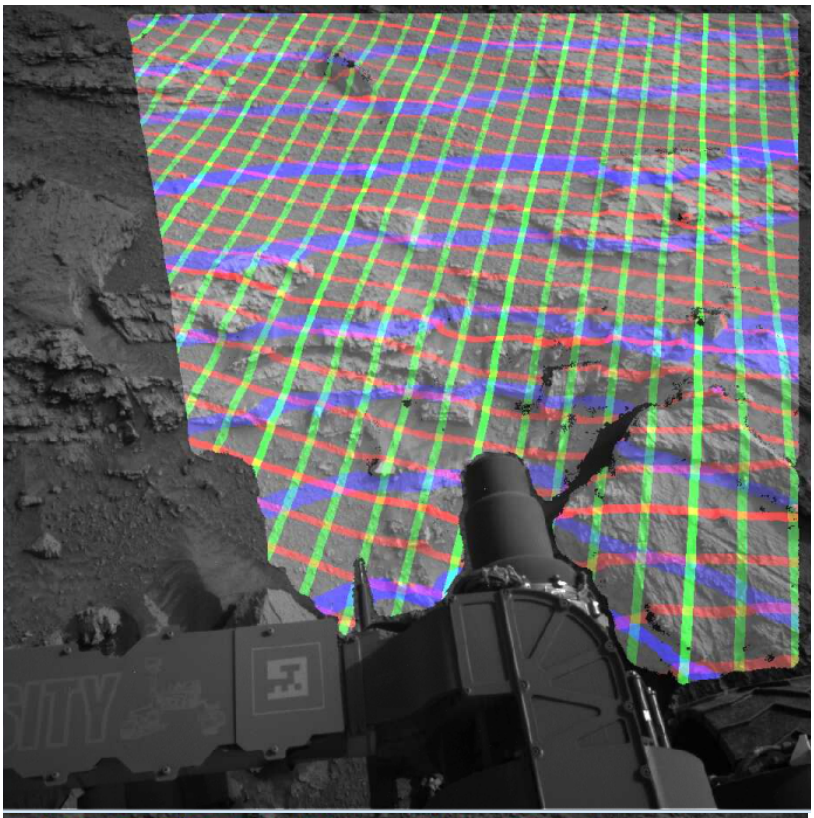
Generate the XYZ image.

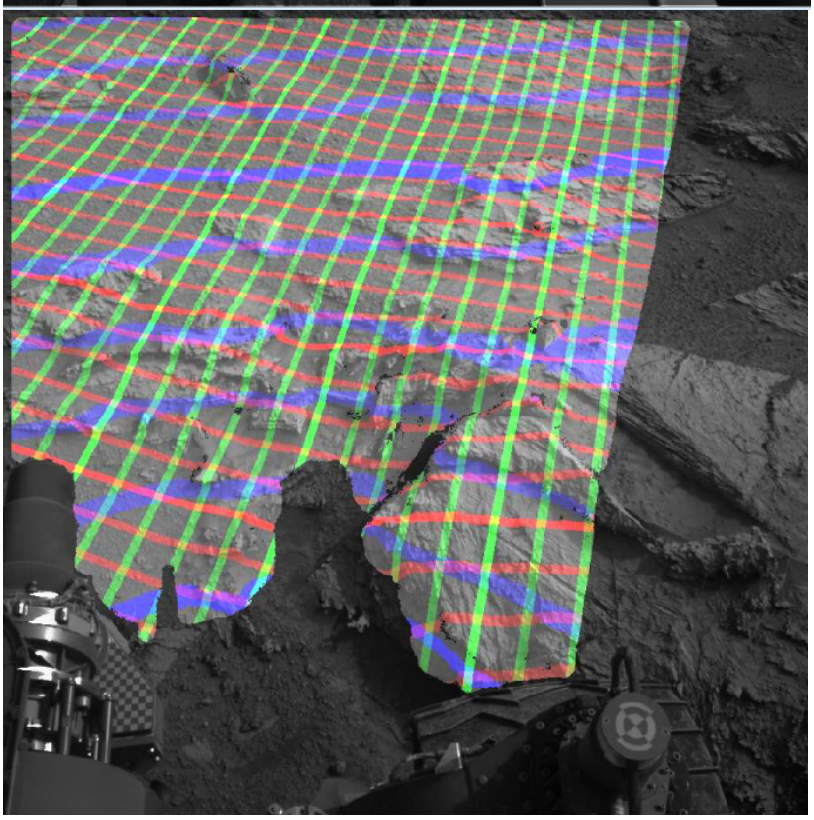
```
$MARSLIB/marsxyz
\(NLB_712299404RADLF0961766NCAM00353M1.VIC,NRB_712299404RADLF0961766NCAM00353
M1.VIC\) out=NLB_712299404XYZLF0961766NCAM00353M1.VIC
dispar=NLB_712299404DSPLF0961766NCAM00353M1.VIC error=0.005 abserror=0.15
linedisp=4.0 avglinedisp=0.75 zlimits=\(-50.0,50.0\) spike_range=0.04
outlier=0.5

$MARSLIB/marsxyz
\(NRB_712299404RADLF0961766NCAM00353M1.VIC,NLB_712299404RADLF0961766NCAM00353
M1.VIC\) out=NRB_712299404XYZLF0961766NCAM00353M1.VIC
dispar=NRB_712299404DSPLF0961766NCAM00353M1.VIC error=0.005 abserror=0.15
linedisp=4.0 avglinedisp=0.75 zlimits=\(-50.0,50.0\) spike_range=0.04
outlier=0.5
```

At this point, marsrfilt should be run to refine the XYZ results. However, since marsrfilt is not currently in the MSL pipeline we are mimicking here, we do not illustrate it. However, it is highly recommended for use with all marsxyz results, even on MSL.

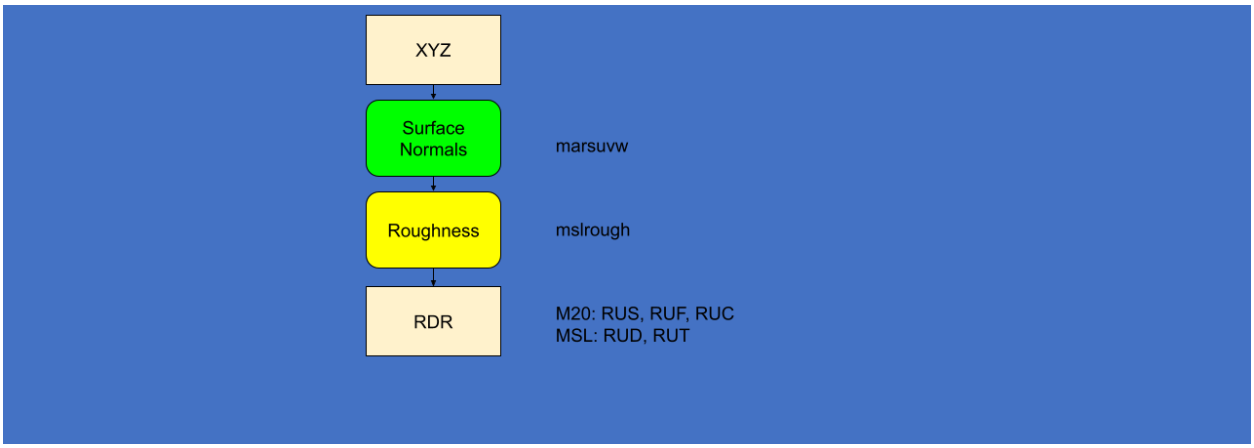
As described earlier, the results can be viewed using Marsviewer, this time using the MSL_FLAT file finder:

```
java -Dsun.java2d.xrender=false jpl.mipl.mars.viewer.MarsImageViewer &
```





10.9 Surface Normals and Slope Maps



Programs used: marsuvw, marsslope
Example Mission: M20
Example Sol Site Drive: 175 6 1752
Sample Data Directory: SurfaceNormalsSlopeMaps

In this example, we create a slope map for an image. We continue with the above Stereo Correlation example, so if you have not run it, please do so. The outputs of that process are the inputs to this one.

Slope maps are derived from surface normals. Surface normals are computed by analyzing a patch of XYZ pixels and fitting a plane to those pixels. So it is important to choose an appropriate size for this patch. A small patch size will pick up minute variations, which might be good for rock analysis but also picks up noise. A patch the size of the rover will show the kind of tilt to expect if the rover was at that position, without all the noise. In addition, if you are looking in the distance, a small patch size will become nothing but noise. Geologists may have other constraints based on the size of features they are examining.

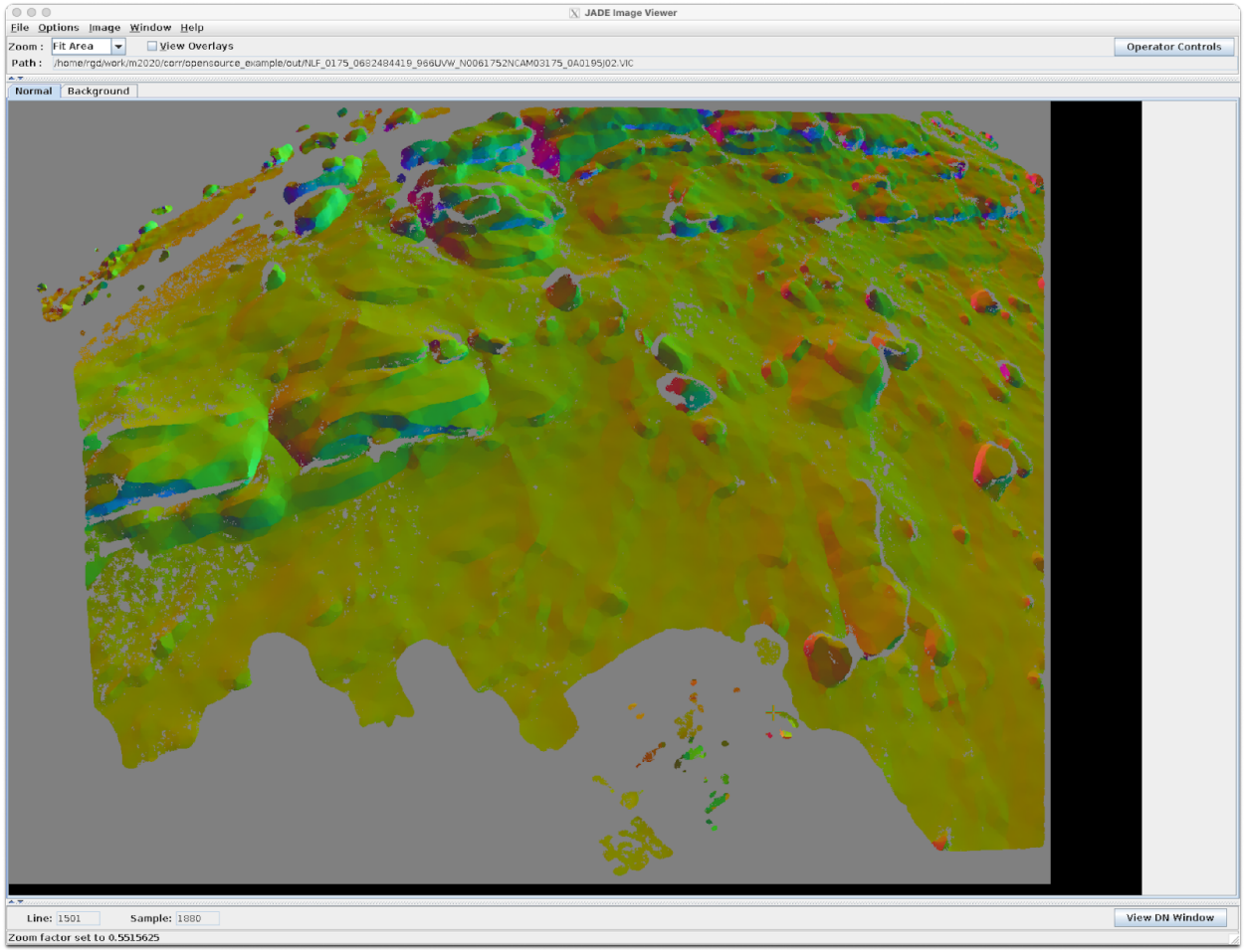
In general, the UVW products in PDS use a small patch size (often a size related to the arm instruments, depending on mission) while UVS products use a large patch the size of the rover, for analysis of traversability.

Once the surface normal is created, various kinds of slope products can be created from that.

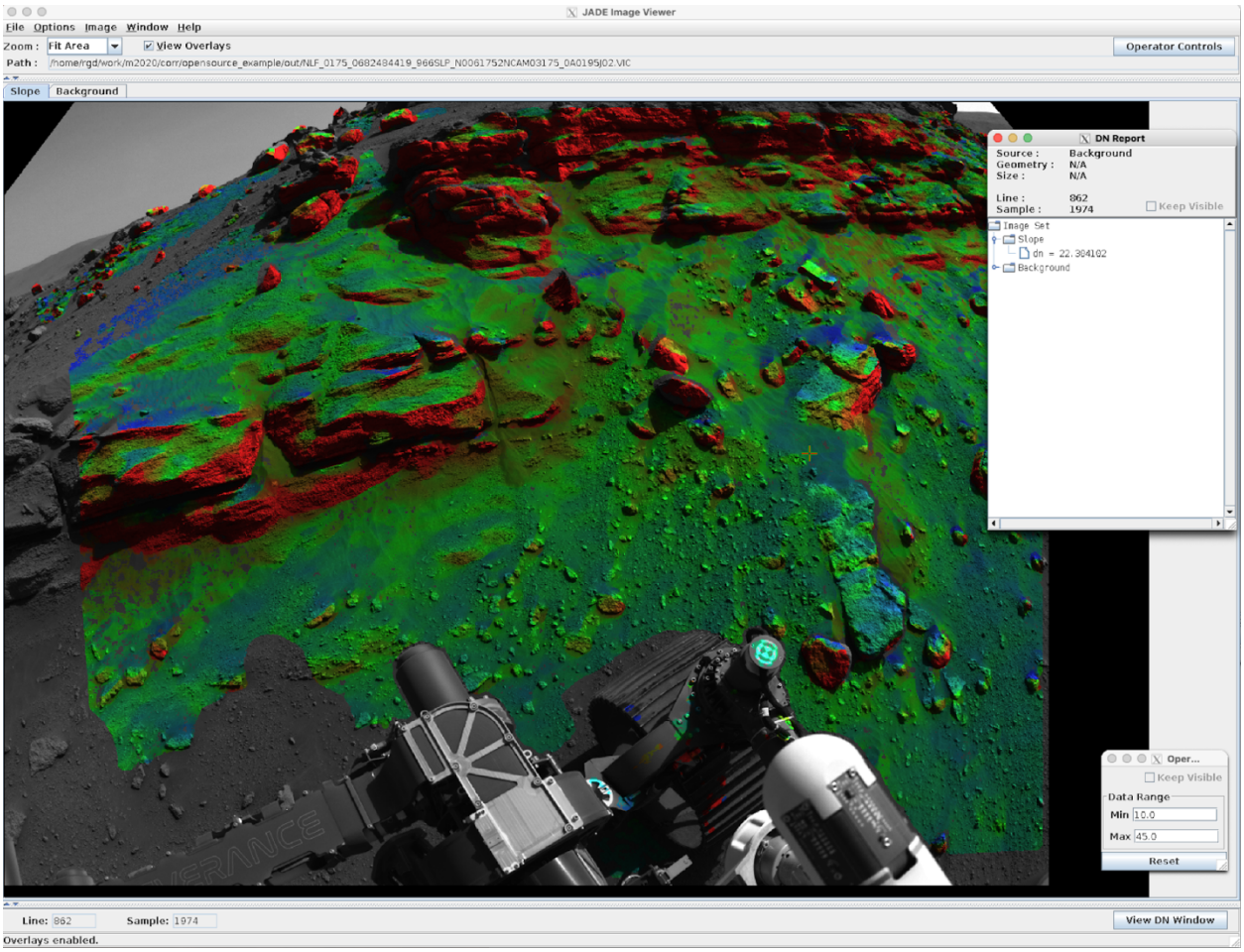
Any of the XYZs can be input to marsuvw; it looks at the label to determine if the XYZs are expressed in Site, Rover, or other frame, and adjusts accordingly. Likewise, the output frame of the surface normals may be selected independently. By convention the UVW product is generally produced in Rover frame (because it is used for arm work) while UVS is produced in Site frame (because it is used for driving in operations). The marsuvw program can set the patch size either in pixels or meters; generally, both are used but the size in meters (`separation` parameter) is primary (the pixel size, `radius`, is set much larger than needed, but can help limit runtime when close in). For this example, we choose 5cm radius and 30 pixels radius.

```
$MARSLIB/marsuvw NLF_0175_0682484419_966XYZ_N0061752NCAM03175_0A0195J03.IMG
NLF_0175_0682484419_966UVW_N0061752NCAM03175_0A0195J03.VIC box_radius=50
radius=30 separation=.05 point=cm=label

java -Dsun.java2d.xrender=false jpl.mipl.jade.viewer.ImageViewer
NLF_0175_0682484419_966UVW_N0061752NCAM03175_0A0195J03.VIC UVW
NLF_0175_0682484419_966RAS_N0061752NCAM03175_0A0195J03.IMG &

$MARSLIB/marsslope
inp=NLF_0175_0682484419_966XYZ_N0061752NCAM03175_0A0195J03.IMG
out=NLF_0175_0682484419_966SLP_N0061752NCAM03175_0A0195J03.VIC
uvw=NLF_0175_0682484419_966UVW_N0061752NCAM03175_0A0195J03.VIC -slope

java -Dsun.java2d.xrender=false jpl.mipl.jade.viewer.ImageViewer
NLF_0175_0682484419_966SLP_N0061752NCAM03175_0A0195J03.VIC SLP
NLF_0175_0682484419_966RAS_N0061752NCAM03175_0A0195J03.IMG &

$MARSLIB/marsslope
inp=NLF_0175_0682484419_966XYZ_N0061752NCAM03175_0A0195J03.IMG
out=NLF_0175_0682484419_966SHD_N0061752NCAM03175_0A0195J03.VIC
uvw=NLF_0175_0682484419_966UVW_N0061752NCAM03175_0A0195J03.VIC -heading

java -Dsun.java2d.xrender=false jpl.mipl.jade.viewer.ImageViewer
NLF_0175_0682484419_966SHD_N0061752NCAM03175_0A0195J03.VIC SHD
NLF_0175_0682484419_966RAS_N0061752NCAM03175_0A0195J03.IMG &
```

The UVW image in Marsviewer is shown below. The surface normal unit vector is directly converted to the R,G,B components of the color display, so the color indicates the orientation of the surface. Note how the color change on the rocks: the front side of rocks are green while the left side tends to red. It is easier to see this with View Overlays off, but please experiment. The actual numeric value of the normal (or XYZ or any other product) can be obtained using the View DN Window button.



The Slope overlay is best seen with the overlays on. It presents a rainbow (a common color scheme in Marsviewer/Jadeviewer) where red is “bad” and blue is “good”. The Operator Controls button is important here, as it sets the range for the slope (and thus what is “bad”). The default is 0-30 degrees, meaning anything >= 30 is red and anything <= 0 is blue. Because the terrain is so highly tilted here, much of it is red. Changing the range to 10-45 results in the image below. The value under the cursor (in degrees) is shown in the DN Report window. Note that you can ctrl-click in the image to “plant” the cursor (disconnect it from the mouse) allowing measurements to continue to be displayed while you move the mouse elsewhere (say, to cut-and-paste the numbers). Ctrl-click again reattaches the cursor to the mouse.



The slope heading provides the direction of the slope in degrees.

The surface normal can also be generated in site frame (below). Note that box_radius was expanded tremendously. That's because box_radius is centered around the coordinate system origin, and the Site frame coordinate system origin is about 150m away from the rover for this image. Surface normals of either frame can be provided to marsslope; it automatically converts the inputs as needed.

```
$MARSLIB/marsuvw NLF_0175_0682484419_966XYZ_N0061752NCAM03175_0A0195J03.IMG
NLF_0175_0682484419_966UVS_N0061752NCAM03175_0A0195J03.VIC box_radius=50000
radius=30 separation=.05 -site
```

As a final note, if you are using a very large window on marsuvw, it can get very slow. This is mitigated using `-slope` mode. See the history label for UVS products and the PDF help or marsuvw for details.

10.10 Using Marsviewer Locally

Programs used: Marsviewer
Example Mission: M20
Example Sol Site Drive: 175 6 1752
Sample Data Directory: StereoCorrelation, plus results from Cases 10.7 and 10.9.

The preceding sections created a lot of files. It is somewhat cumbersome to visualize each of them separately using Jadeviewer or xvd. That's where Marsviewer comes in - it is a convenient way to see all of the RDR products for an image, as well as a large collection of images.

Marsviewer requires that files strictly follow the naming convention for the mission (this is why the rather cumbersome names were used in the examples above). It also *sometimes* requires a directory structure.

Marsviewer works using a "file finder" (FF). The File Finder is a module that interprets each mission's naming convention and directory structures. When you start Marsviewer, you select the FF you want to use, and the directory to point to.

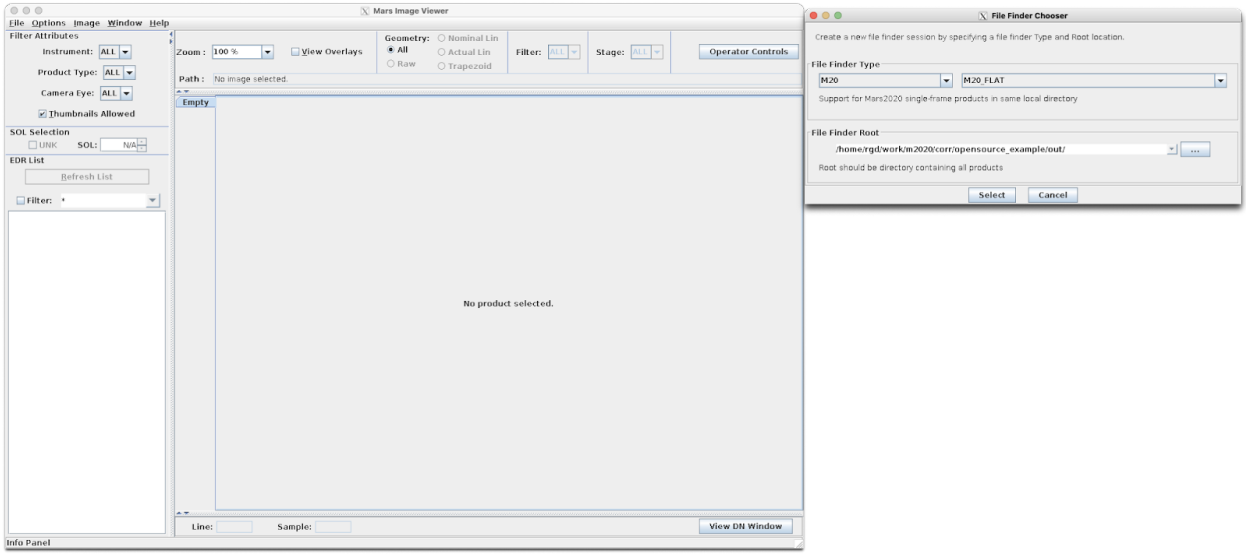
The easiest method is to use the "Flat" file finder. Each mission has a flat FF. This interprets the filename convention but assumes all files are in the current directory.

Start up Marsviewer:

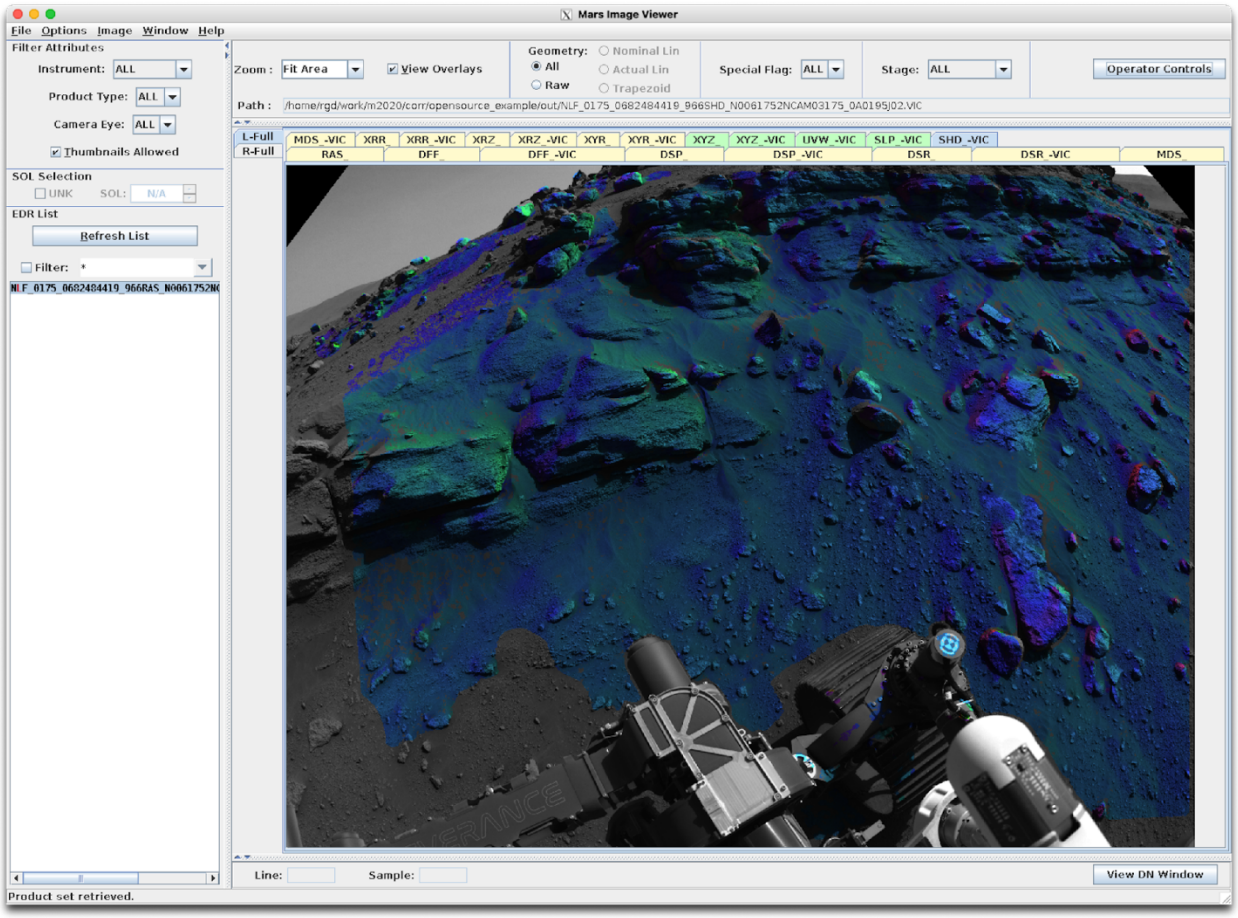
```
java -Dsun.java2d.xrender=false jpl.mipl.mars.viewer.MarsImageViewer &
```

Again, the -Dsun… clause should only be needed when using X-windows to a Mac display.

Set the file finder mission to M20, select the M20_FLAT finder, and put in the directory name to your data under File Finder Root.



Hit Refresh List if needed, then select the image in the list on the left. All the various products will come up in tabs across the top. Select the one of interest, Slope Heading in this case.



If you have more than one image in the directory, they will show up in the list on the left. Note that you must have a "base" type image for it to appear on the left. Those depend on the mission but are generally various EDR types, FDR, or radiometrically corrected (RAS/RAD/RZS) images.

You can also set up a full directory structure to cover more than one sol, which matches the directory structure of the operations data or PDS data. This structure varies per mission but in general is something like …`/sol/nnnnn/<ids|opgs|mipl>/<edr|rdr>/<inst>/`. Consult the SIS or bundle SIS for the mission. Some missions have file finders that match the bundle/data-set structure in PDS.

10.11 Roughness

Program used: marsuvw, mslrough
Example Mission: MSL
Example Sol Site Drive: 3546 96 1766
Sample Data Directory: Roughness

The roughness maps, RUD (for the Drill) and RUT (for the Dust Removal Tool (DRT) contain surface roughness estimates at each pixel in the image, along with a "goodness" flag indicating whether the roughness meets certain criteria. For each pixel, the surface normal defines a reference plane. XYZ pixels in the area of interest are gathered, and the distance to the plane is computed. Outliers are thrown out. For the remainders, the minimum and maximum distances from the plane are found. Roughness is defined as the distance between this min and max (thus, is peak-to-peak variation within the area along the normal vector).

Generate UVW:

```
$MARSLIB/marsuvw inp=NLB_712299404XYZLF0961766NCAM00353M1.IMG
out=NLB_712299404UVWLF0961766NCAM00353M1.VIC separation=0.05 error=0.003
radius=15 box_radius=50

$MARSLIB/marsuvw inp=NRB_712299404XYZLF0961766NCAM00353M1.IMG
out=NRB_712299404UVWLF0961766NCAM00353M1.VIC separation=0.05 error=0.003
radius=15 box_radius=50
```

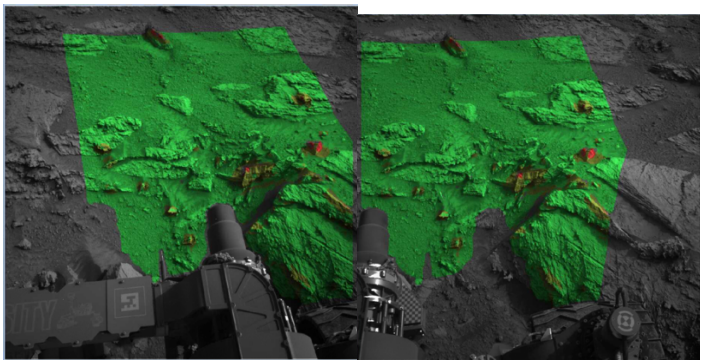
Generate roughness map:

```
$MARSLIB/mslrough inp=NLB_712299404XYZLF0961766NCAM00353M1.IMG
out=NLB_712299404RUDLF0961766NCAM00353M1.VIC
uvw=NLB_712299404UVWLF0961766NCAM00353M1.VIC

$MARSLIB/mslrough inp=NRB_712299404XYZLF0961766NCAM00353M1.IMG
out=NRB_712299404RUDLF0961766NCAM00353M1.VIC
uvw=NRB_712299404UVWLF0961766NCAM00353M1.VIC
```

As described earlier, the results can be viewed using Marsviewer, this time using the MSL_FLAT file finder:

```
java -Dsun.java2d.xrender=false jpl.mipl.mars.viewer.MarsImageViewer &
```

Note that you'll need to copy a linearized image of the same SCLK, such as RASL, from a previous use case, if you want a background image.



10.12 Cylindrical Mosaic

Programs used: marsmap
Example Mission: MSL
Example Sol Site Drive: 3546 96 1766
Sample Data Directory: CylindricalMosaic

A frequent scenario for generating a simple-cylindrical projected mosaic is when rover post-drive imaging has occurred. These mosaic products are ideal for depicting the terrain from a wider perspective.

Make List file

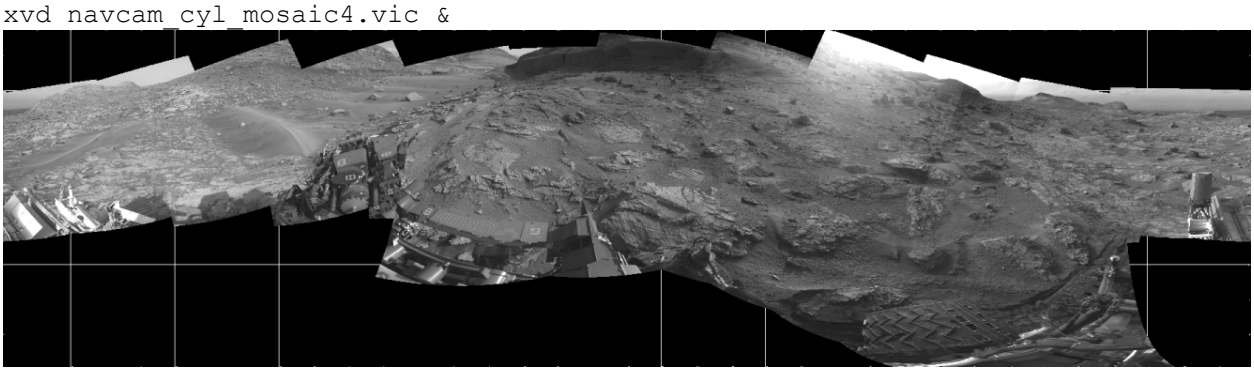
```
cd to directory where Navcam images are located
ls *.IMG >mosaic.lis
```

Generate the Navcam Cylindrical mosaic:

```
$MARSLIB/marsmap inp=mosaic.lis out=navcam_cyl_mosaic4.vic -cyl
surf_coord=local_level brtcorr=mosaic.brt
```

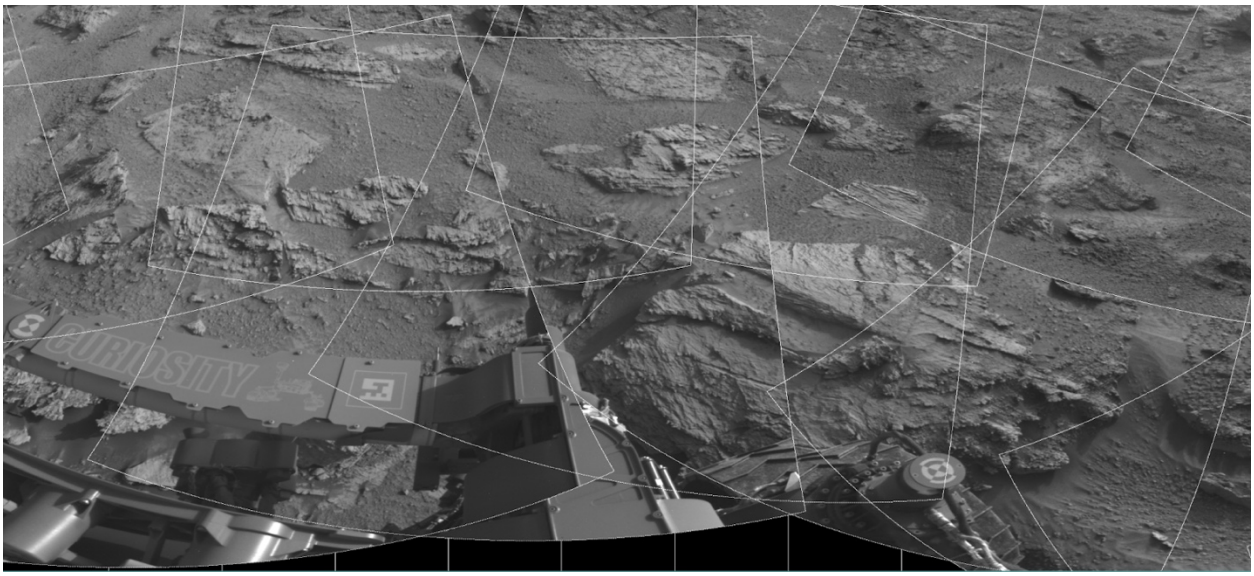
Use xvd to view output mosaic

```
xvd navcam_cyl_mosaic4.vic &
```



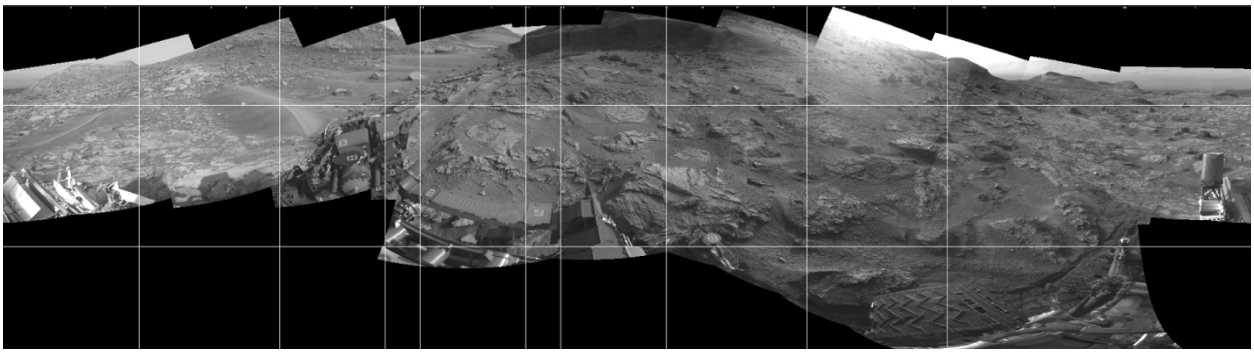
Individual image footprints and image numbers can be added by using the footprt and foot_dn can be used to change the appearance of the footprints.

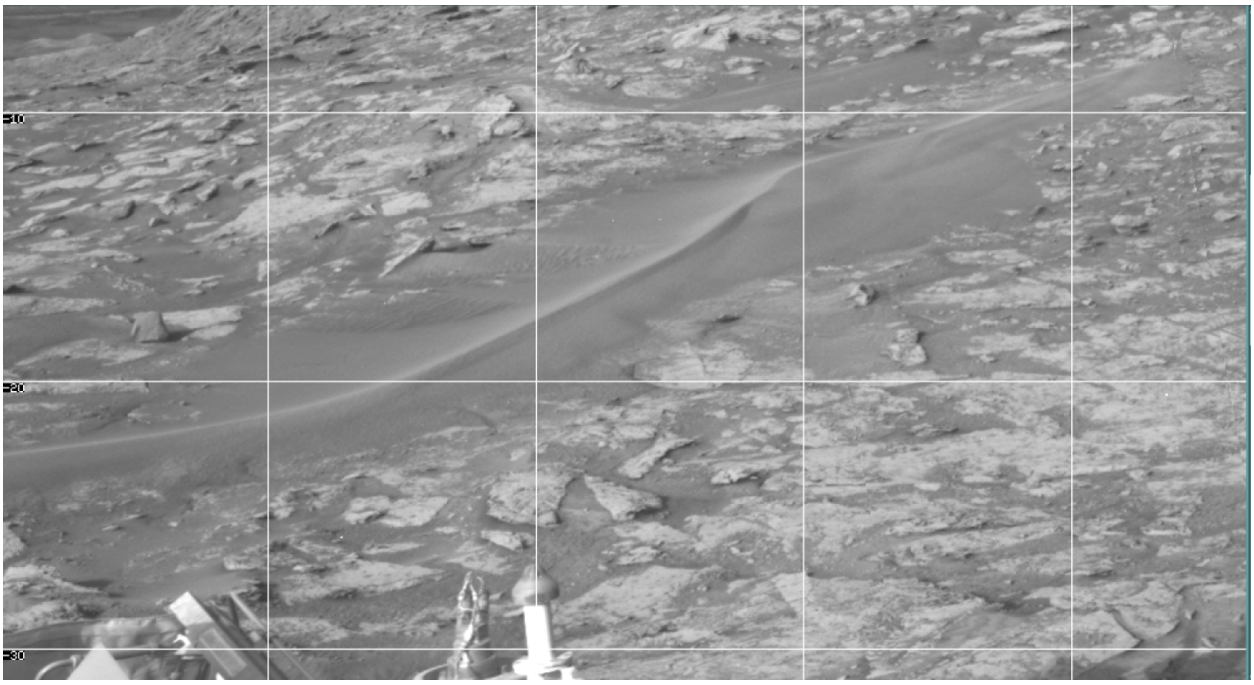
```
$MARSLIB/marsmap inp=mosaic.lis out=navcam_cyl_mosaic5.vic -cyl
surf_coord=local_level footprt=overlap foot_dn=10000

xvd -min 0 -max 4095 navcam_cyl_mosaic5.vic &
```



By default, gridlines are generated behind the mosaic and can be brought to on top of the mosaic with the grid argument. Use grid argument to generate a grid on top of the image so that it shows everywhere. Additionally the grid zoom level and spacing parameters can be adjusted as desired.

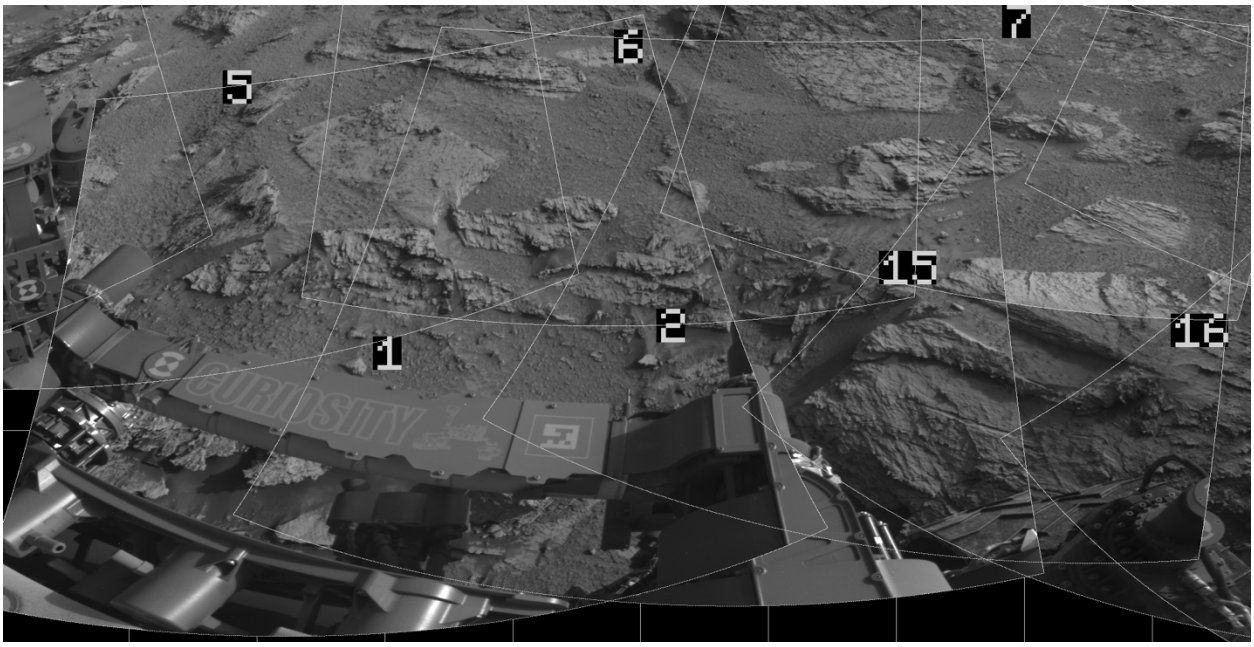
```
$MARSLIB/marsmap inp=mosaic.lis out=navcam_cyl_mosaic5.vic -cyl
surf_coord=local_level grid=grid_overlay

xvd -min 0 -max 4095 navcam_cyl_mosaic5.vic &
```





Mosaics are built with the first image from the list file appearing on top. So where images overlap, the first file “wins”. We can take advantage of that to improve this mosaic. First thing is to run the mosaic with image numbers so we can see what is what. Then zoom in on the center bottom where the arm elbow is.

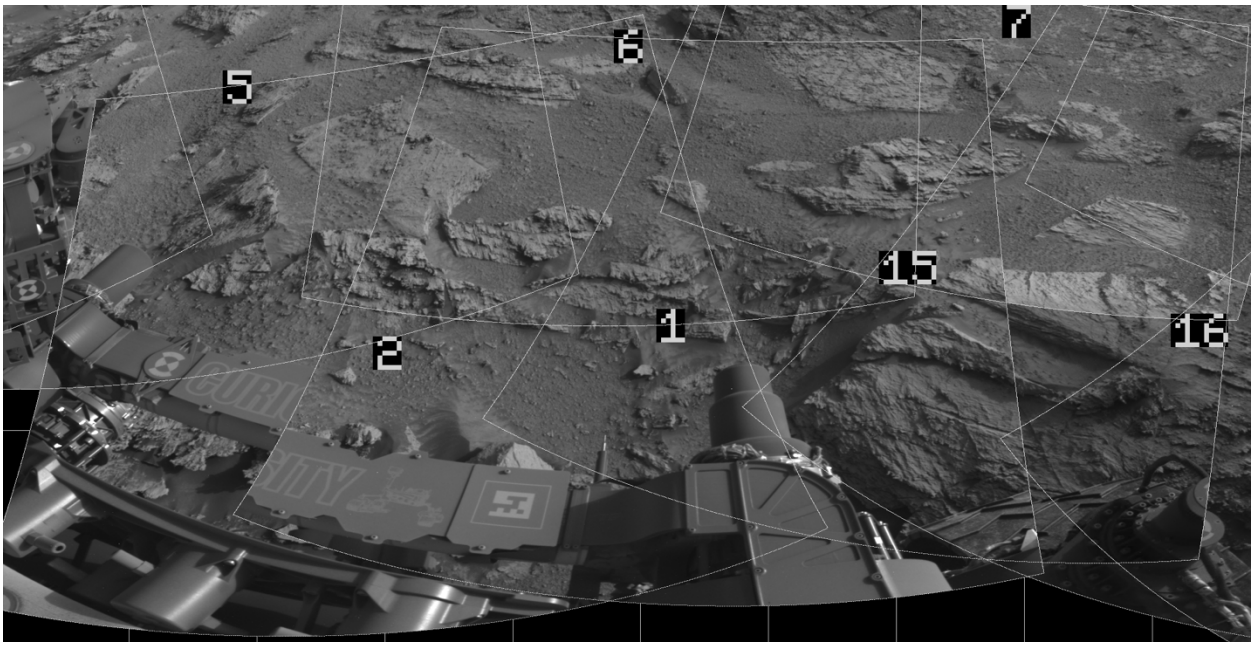
```
$MARSLIB/marsmap inp=mosaic.lis out=navcam_cyl_mosaic5.vic -cyl
surf_coord=local_level -number number_zoom=8 footprt=overlap

xvd -min 0 -max 4095 navcam_cyl_mosaic5.vic &
```



It’s a little hard to visualize, but the image numbers are in the center of the frames. So you can

see image 1's outline covers most of the left half of the image, and image 2's outline covers the center. The arm's elbow is right at the edge of image 1, and parallax issues mean image 2 does not line up with it at all on the arm (even though it's close on the ground). That's because the surface model is at ground level, not elbow level.

But what if we wanted to see the elbow more clearly? We can simply reorder the images and put #2 on top of #1. Use a text editor on the `mosaic.lis` file and swap the order of the first two images. Run the exact same command, and you get:



The elbow is now whole. Unfortunately, the arm itself is split in two. This illustrates the problems with parallax … it is not possible to get rid of all seams in a mosaic like this due to parallax issues.

If you look closely though, image 15 contains the elbow but only a fraction of the arm. So it could possibly be moved on top of the original 1 to accomplish the goal. It will split the arm in a different place, but that might be less objectionable. It depends on what you want to achieve – there is some artistry to making a good mosaic.

Once you're happy, rerun the mosaic with the numbers and footprints removed to get a final mosaic:

```
$MARSLIB/marsmap inp=mosaic.lis out=navcam_cyl_mosaic6.vic -cyl
surf_coord=local_level
```

10.13 Seam Corrected Mosaic

Programs used: marsautotie, marsnav, marstie, marsmap
Example Mission: MSL
Example Sol Site Drive: 3546 96 1766
Sample Data Directory: CylindricalMosaic, plus list file from Case 10.12.

This section shows how to use the tiepoint/bundle adjustment process to correct seams in a mosaic.

First, make a list file or use the one already created from the above section (recommended). Then run marsautotie to get tiepoints between all the images:

```
$MARSLIB/marsautotie inp=mosaic.lis out=navcam_cyl_mosaic_drive.autotie
surf_coord=local_level busy=25 dens=100 center_weight=500
```

The marsnav program does the actual bundle adjustment, which attempts to change the pointing of images in order to minimize the tiepoint error.

```
$MARSLIB/marsnav mosaic.lis
navcam_cyl_mosaic_drive.nav  navcam_cyl_mosaic_drive.autotie out_sol=rgd
surf_coord=local_level point=pm=3dof -do_surf inertia=\(.1 .1 .1\)
```

The inertia parameter provides some “stay in place” weight that prevents the images from drifting too far from their starting point. The out_sol parameter is just the user’s initials or name, for documentation purposes. The point=pm=3dof parameter causes it to use a 3 degree of freedom pointing model for the mast, which adds twist (rotation in the image plane) to the standard azimuth and elevation parameters (it is optional).

Look at “Commanded mean pixel error” and “Solution mean pixel error” in the log. Those are a good indication of how the bundle adjustment is doing. There should be a significant decrease in error for the first recycle, with diminishing returns after that. A good mosaic generally will have less than 2 pixels of error.

Then run the mosaic to see how we did:

```
$MARSLIB/marsmap mosaic.lis navcam_cyl_mosaic_drive.vic
navcam_cyl_mosaic_drive.nav -num number_zoom=7 -tight
```

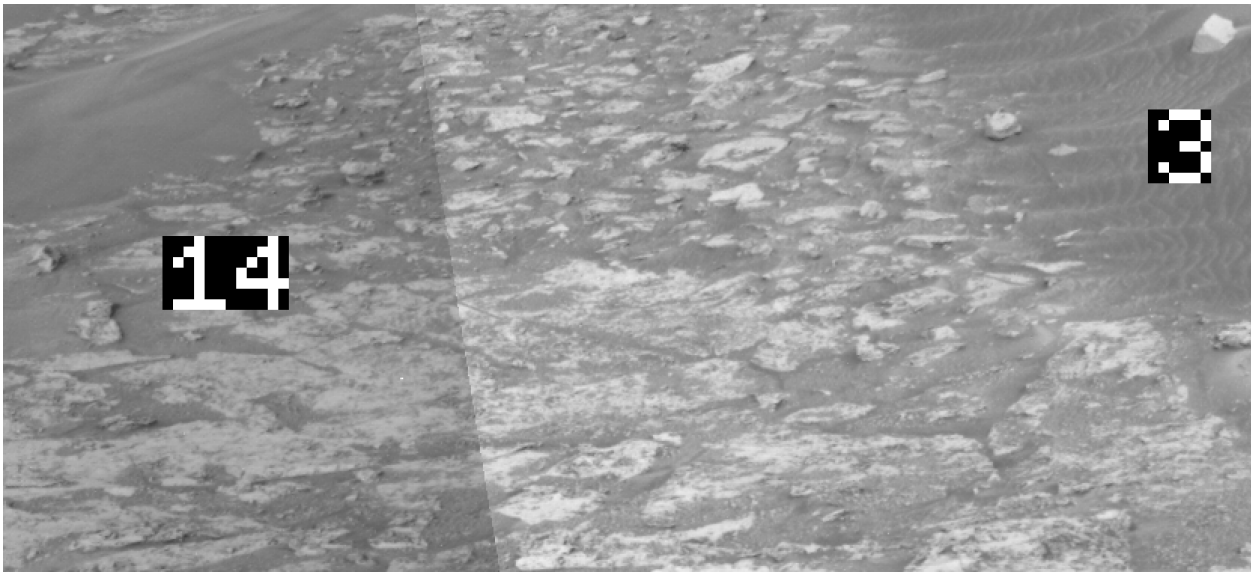
Note, added the nav file and REMOVED surf_coord (the surface model is in the nav file). REMOVING surf_coord IS CRITICAL HERE, otherwise it will override the new surface model in the nav file. Also critical is the -tight parameter. There is a bug currently where the nav file will not be properly read without it.

```
xvd -min 0 -max 4095 navcam_cyl_mosaic_drive.vic -fit &
```

Inspect the mosaic paying close attention to all the seams (zoom x2 to really see any offsets). Most of the time the seams between tiers and horizontal seams are fine, but should still be checked. It is primarily the vertical seams between images that exhibit the most offsets and need the most attention.

After inspecting the mosaic, marsautotie did well to autocorrect for offsets however there are

some small offsets at the image seams of 3 and 14. In reality, it is borderline if this is worth fixing manually, but it serves as an example.



The hard part in the current software setup is to get the interactive marstie program to edit just that pair of tiepoints. The simple but tedious way (which works for small mosaics) is to let it bring up all pairs, and hit ctrl-X to exit as each window comes up, until you get to the one you want.

An easier way, especially for larger mosaics, is to use the `sed` command to set `interactive="1"` for each entry in the tiepoint file, and then set the tiepoint of interest to `interactive="0"`. With the `-new` option to marstie, that will bring up just that pair.

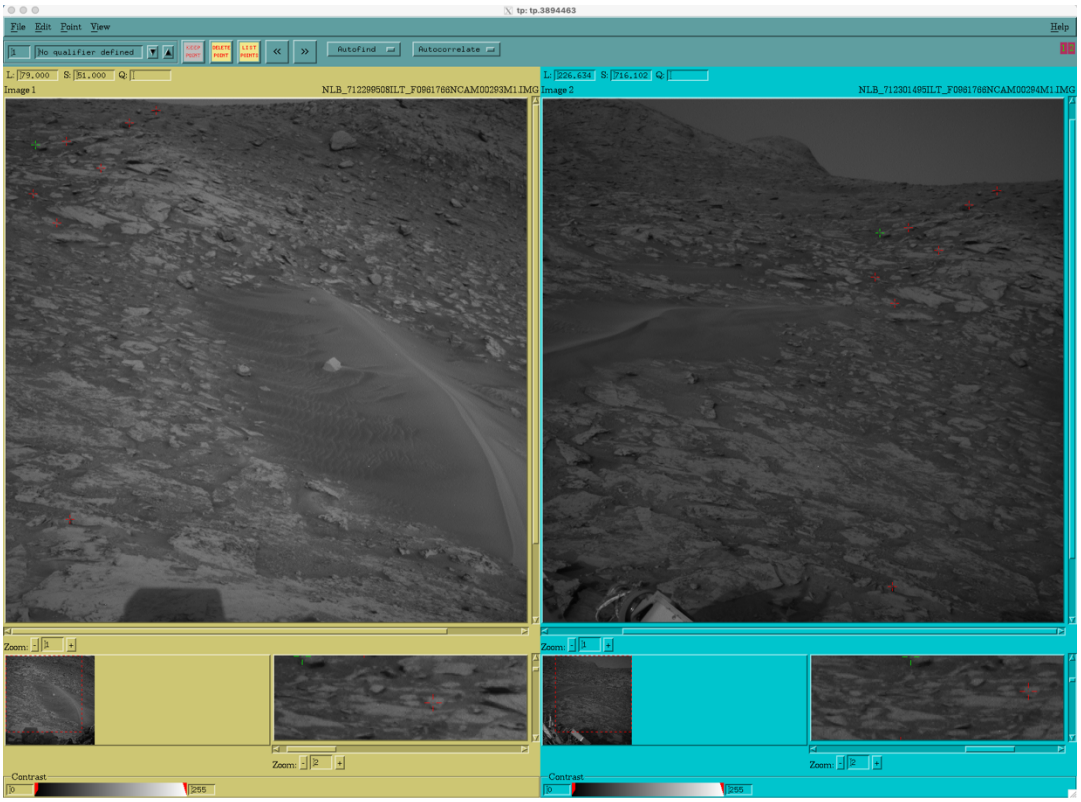
```
sed -e 's/interactive="0"/interactive="1"/' < navcam_cyl_mosaic_drive.autotie
> navcam_cyl_mosaic_drive.autotie2

vi navcam_cyl_mosaic_drive.autotie2
```

In the editor, search for `left_key="2" right_key="13"` and change any one of those tiepoints to `interactive="0"`. Note that the numbering starts at 0 in the tiepoint file, so images 3 and 14 are really 2 and 13 in the file. Only one tiepoint needs to be changed, that will bring up all tiepoints for that pair.

```
    <tie type="0" left_key="2" right_key="13">
      <left      line="79.000000" samp="51.000000"/>
      <projected line="225.717105" samp="717.573233"/>
      <right     line="226.634002" samp="716.101951"/>
      <flags quality="0.953816" interactive="0"/>
    </tie>
```

Then run marstie:

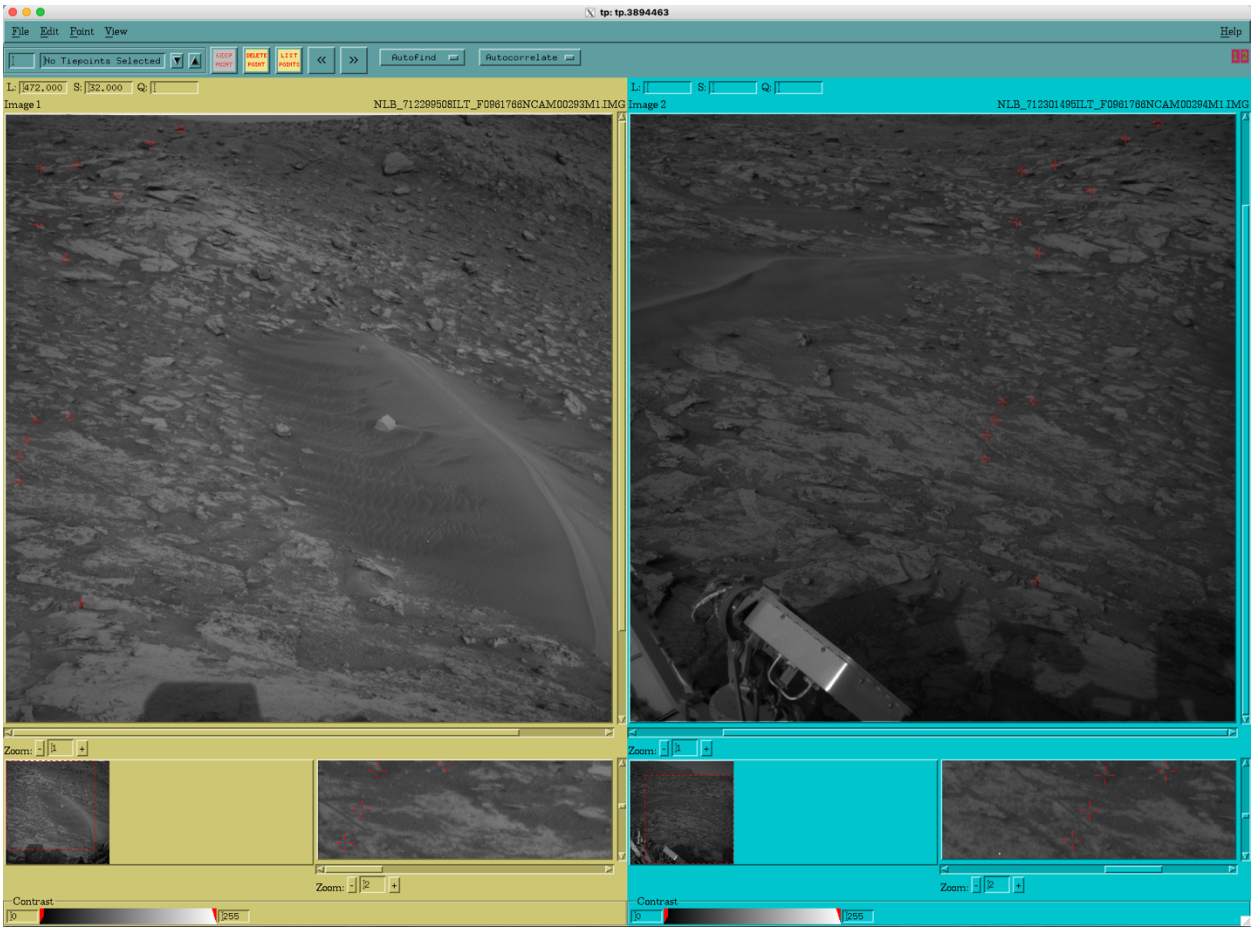
```
$MARSLIB/marstie mosaic.lis navcam_cyl_mosaic_drive.tie
navcam_cyl_mosaic_drive.autotie2 -new
```



Using the gui requires a 3 button mouse where: left click is to select/deselect a tiepoint and middle click is to add or move a selected tiepoint. On a Mac using Xquartz, option-click is usually the middle button, although you might have to select Emulate three button mouse under Preferences. Stretch image, pan, and zoom to find matching features in both images to use for tiepointing. Use ctrl-A to turn on or off the correlator as needed. Once a feature is identified, use middle mouse button to place tiepoint on that same feature in both images. Use Ctrl-S to save each tiepoint before moving onto the next. The interface is admittedly a bit clunky and takes some getting used to.

In this case, the automatic tiepoints are clustered at the top of the image, with one at the bottom, and none in the area of the seam where we identified our problem. So, add several more tiepoints in that area to make it “work harder” to match that particular seam. The left image in marstie is the one that comes first in the list file, so it is on top in the mosaic. For that reason, you generally want your tiepoints around the edge of the left image, as that corresponds to the actual seam location.

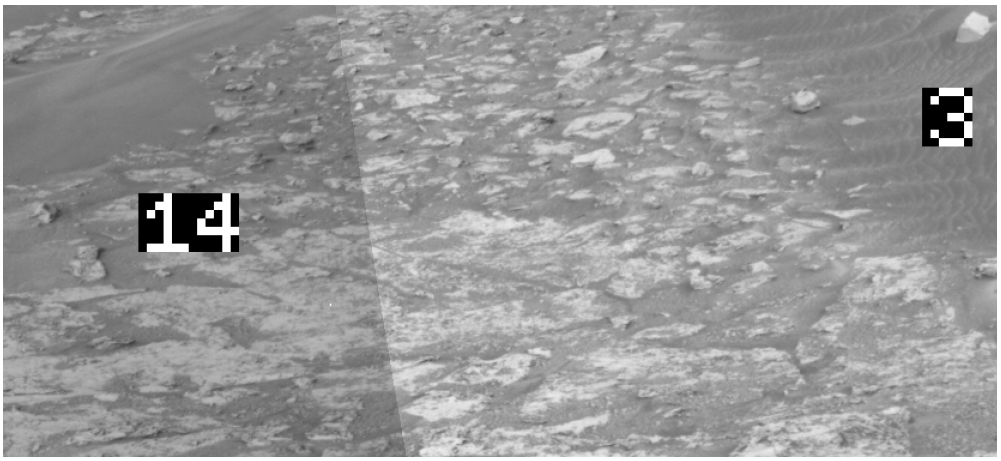
Here, we’ve added 5 more tiepoints in that middle section. Once happy, use ctrl-X to save and exit.



Effective use of marstie could be a book on its own. Experiment and practice. When you click somewhere new to add a new tiepoint, it attempts to predict the matching location on the other image using an affine transform. This is often helpful, but needs a couple points to get seeded properly. You generally want tiepoints to be on the ground (matching the surface model) and definitely not on the rover hardware, since generally you want to correct the ground and won't care about the rover. Of course, if you use miss-distance tiepoints or marsnav2, you want a wide variety of Z values for your tiepoints. See the help for marstie, marsautotie, marsautotie2, marsnav, and marsnav2.

Once done with the editing, run marsnav with the edited tiepoint file and marsmap again to see the changes made.

```
$MARSLIB/marsnav mosaic.lis
navcam_cyl_mosaic_drive.nav navcam_cyl_mosaic_drive.tie out_sol=rgd
surf_coord=local_level point=pm=3dof -do_surf inertia=\(.1 .1 .1\)

$MARSLIB/marsmap mosaic.lis navcam_cyl_mosaic_drive2.vic
nav=navcam_cyl_mosaic_drive.nav -num number_zoom=7 -tight

xvd -min 0 -max 4095 navcam_cyl_mosaic_drive.vic &
```



The difference is subtle, but it is there and is a slight improvement.

Often it takes several iterations of edit/run to get the highest quality mosaic.

10.14 Mosaic Brightness Adjustment

Programs used: marsmap, marsbrt
Example Mission: MSL
Example Sol Site Drive: 3546 96 1766
Sample Data Directory: CylindricalMosaic, plus list file from Case 10.12.

Sometimes variations in brightness or contrast between frames become very visible in a mosaic. Ideally, radiometric correction should solve this, but it is not perfect. Brightness correction attempts to match the radiometric seams. Results are no longer radiometrically calibrated, but correction factors are retained so we know what was done to each image. This process uses a bundle-adjustment algorithm similar to pointing correction that gathers image brightness/contrast statistics in image overlap areas and adjusts brightness/contrast of each image (overall multiplicative and additive factor) to minimize global error in statistical match.

The output mosaic exhibits a few brightness differences among some of the individual images. The marsbrt program is used to make a brightness correction file (*.brt) that can then be applied to the mosaic when running marsmap. The output .brt can then be used as an input in marsmap to apply this color correction to the final mosaic.

Start with the same mosaic list file from the previous sections.

First marsmap needs to be rerun with the ovr_out argument to create an image overlap file (*.ovr) that then can be supplied as an input in marsbrt. This .ovr file is analogous to tiepoints.

```
$MARSLIB/marsmap inp=mosaic.lis out=navcam_cyl_mosaic1.vic -cyl
surf_coord=local_level ovr_out=mosaic.ovr
```

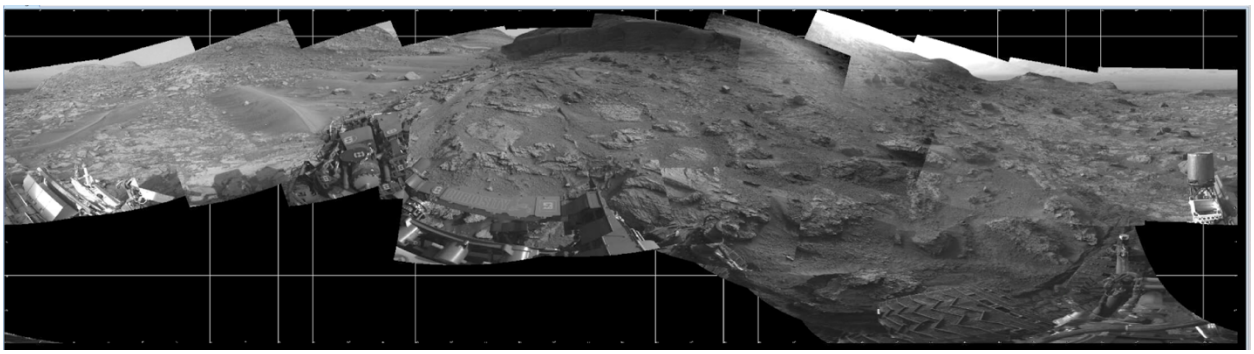
We then run marsbrt to compute the brightness. This is analogous to marsnav in the geometry system (in other words, marsbrt runs the brightness “bundle adjustment”).

```
$MARSLIB/marsbrt inp=mosaic.lis out=mosaic.brt in_ovr=mosaic.ovr out_sol=rgd
outl=1.5
```

Note: The outlier parameter can be adjusted (here it says to reject overlaps that are more than 1.5 standard deviations away from the average). We have found that it helps a lot to keep it low like this, probably because it removes both the rover bits and the sky from consideration.

The output mosaic*.brt file can now be used to rerun marsmap with the BRTCORR argument:

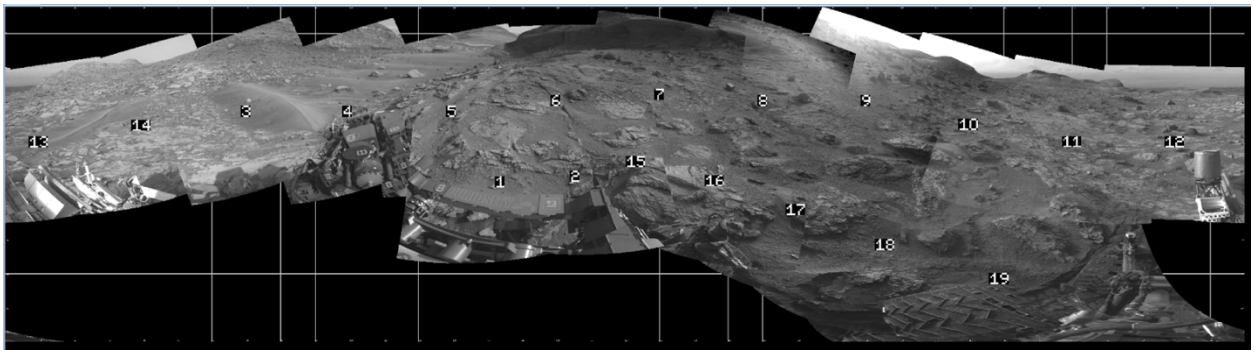
```
$MARSLIB/marsmap inp=mosaic.lis out=navcam_cyl_mosaic2.vic -cyl
surf_coord=local_level brtcorr=mosaic.brt

xvd -min 0 -max 4095 navcam_cyl_mosaic2.vic &
```



The brightness differences have roughly been corrected for, however differences among a few of the images still remain. Some of the images (particularly near the top) could be reordered in the list file to help mitigate these differences.

Run marsmap with number parameter and number_zoom to increase size of numbers

```
$MARSLIB/marsmap inp=mosaic.lis out=navcam_cyl_mosaic3.vic -cyl
surf_coord=local_level brtcorr=mosaic.brt -number number_zoom=10

xvd -min 0 -max 4095 navcam_cyl_mosaic3.vic &
```



Looking at this mosaic, it may be best to reorder some of the images to hide parts of the image that may be affecting the overall brightness correction process (top right corner of image 8 as example). These edits can be done in vi or another text editor.

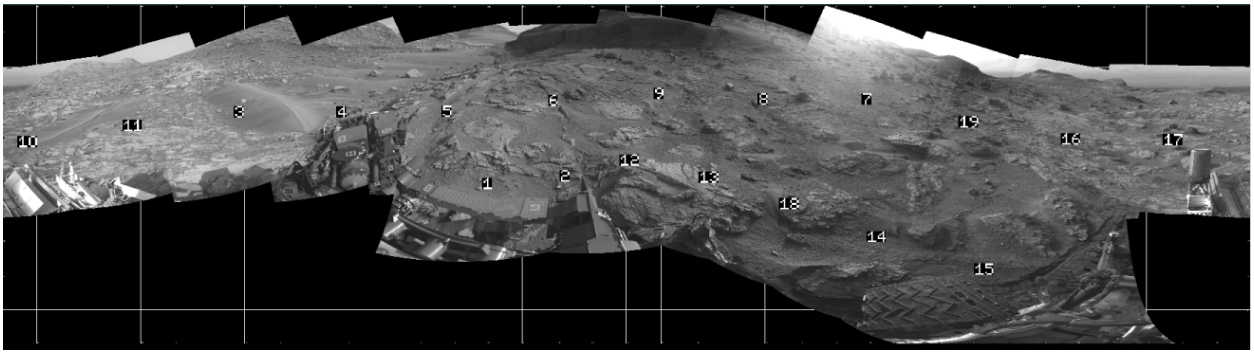
The new order of the images in the list file:

```
NLB_712299372ILT_F0961766NCAM00353M1.IMG
NLB_712299404ILT_F0961766NCAM00353M1.IMG
NLB_712299508ILT_F0961766NCAM00293M1.IMG
NLB_712299539ILT_F0961766NCAM00293M1.IMG
NLB_712299570ILT_F0961766NCAM00293M1.IMG
NLB_712299596ILT_F0961766NCAM00293M1.IMG
NLB_712301270ILT_F0961766NCAM00294M1.IMG
NLB_712301223ILT_F0961766NCAM00294M1.IMG
NLB_712299621ILT_F0961766NCAM00293M1.IMG
NLB_712301464ILT_F0961766NCAM00294M1.IMG
NLB_712301495ILT_F0961766NCAM00294M1.IMG
NLB_712301548ILT_F0961766NCAM00354M1.IMG
NLB_712301580ILT_F0961766NCAM00354M1.IMG
NLB_712301644ILT_F0961766NCAM00354M1.IMG
```

```
NLB_712301676ILT_F0961766NCAM00354M1.IMG
NLB_712301339ILT_F0961766NCAM00294M1.IMG
NLB_712301379ILT_F0961766NCAM00294M1.IMG
NLB_712301612ILT_F0961766NCAM00354M1.IMG
NLB_712301308ILT_F0961766NCAM00294M1.IMG
```

Rerun marsmap with list file changes and number parameters:

```
$MARSLIB/marsmap inp=mosaic.lis out=navcam_cyl_mosaic3.vic -cyl
surf_coord=local_level brtcorr=mosaic.brt -number number_zoom=10

xvd -min 0 -max 4095 navcam_cyl_mosaic3.vic &
```



A few more brightness changes may still need to be made. In principle, these should be done by changing the "tiepoints" (overlaps) and rerunning the brt file. In practice, we have no good tools for this. You can often make some headway by changing the size of the overlaps (generally, reducing them in order to be able to throw out outliers). You can also adjust the borders when creating the ovr files (e.g. `point='"border_left=100,border_right=50"'`) to trim off areas of vignetting that may be throwing off the statistics. Note that those are global settings that affect every image. You can sometimes get there by being creative with the border and refimg. It's a black art.

In this particular case there are only a few images that we are concerned with correcting and this can be done by manually editing the brightness and contrast values in the *.brt for that image. To identify which values correspond to the image that we want to edit, we can run marsmap with the number parameter. This will output a mosaic that displays numbers in the center of each input image according to their order in the list file. We can then use this information to identify the appropriate line in the *.brt file to edit.

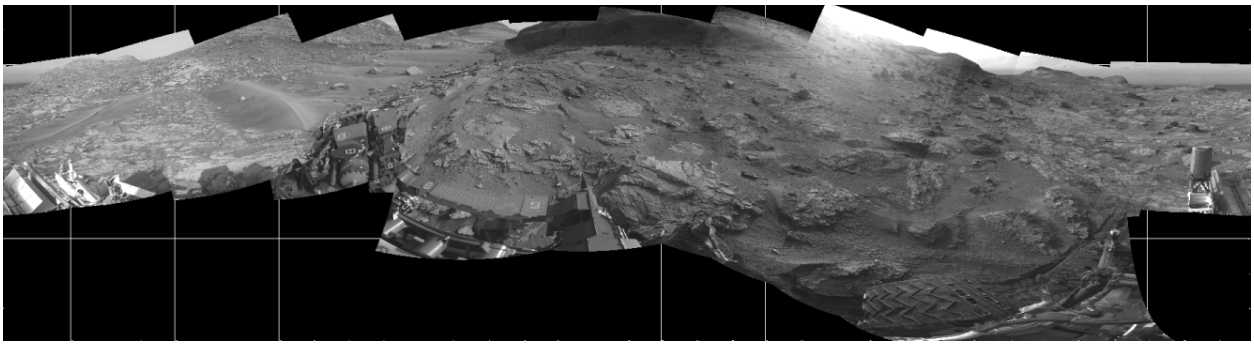
These edits can be done by directly editing the values in the *.brt file using vi or another text editor.

To do this:

1. Open list file in text editor
2. Open *.brt file in text editor
3. In list file find the desired image to color correct
4. Match the unique id of the identified image in the list file to the unique id in the *.brt file
5. Increase or decrease the ADD (brightness) or MULT (contrast) values as desired
6. Save *.brt file and rerun

To achieve optimal brightness correction, the above steps usually need to be done a few times. Rerunning the same command will overwrite the output, so be careful. Once satisfied with the result, rerun marsmap with the numbers and number_zoom parameter removed to generate the final mosaic.

```
$MARSLIB/marsmap inp=mosaic.lis out=navcam_cyl_mosaic4.vic -cyl
surf_coord=local_level brtcorr=mosaic.brt

xvd -min 0 -max 4095 navcam_cyl_mosaic4.vic &
```



Note: the time of day in which the images were taken and the position of the Sun can make the brightness correction process much more challenging to perfect. It is highly recommended that you use zenith correction before embarking on a brightness correction campaign. Especially if the images are taken at different times of day. For purposes of brightness correction, this should be done using RZS images as input. Note that you can do zenith correction directly within the mosaic program using the -zenith option and FDR (M20) or ILT (other missions) as input. However, this is not recommended if you do brightness correction, because it is hard to ensure that every program along the way does the zenith correction.

Brightness correction loses some radiometric accuracy, but typically looks much better.

10.15 Cylper (stereo) Mosaic

Programs used: marsmcauley, viccub
Example Mission: MSL
Example Sol Site Drive: 3546 96 1766
Sample Data Directory: CylperMosaic

In addition to generating a simple-cylindrical projected mosaic when rover post-drive imaging has occurred, a Cylindrical-Perspective (Cylper) stereo mosaic is often generated for the same images. This can be done using the marsmcauley VICAR program.

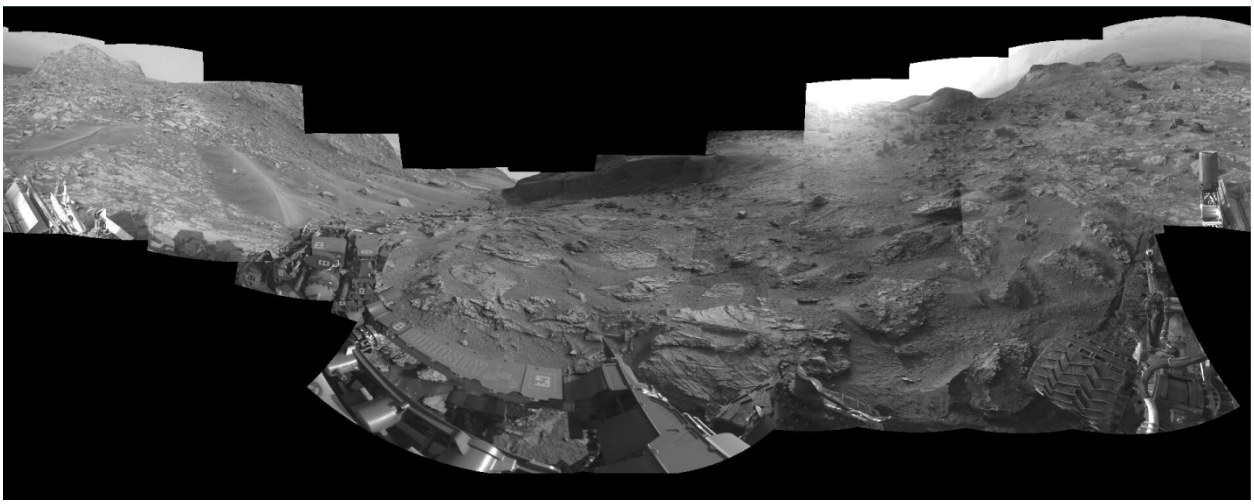
The marsmcauley program generates a hybrid cylindrical-perspective projection made by pointing the output camera at the center of each vertical column of pixels, and projecting that column only. Each output pixel column thus represents the center column (only) of a camera pointed in exactly that direction. This has the advantage of allowing stereo viewing (epipolar lines are preserved) in a panoramic format, without the extreme distortion seen on the edges of a perspective mosaic. Stereo separation is maintained because the output cameras describe a ring in space, which mimics the input camera baseline and preserves the stereo disparity.

Make two list files for input. One with all left ncams and one with all right ncams:

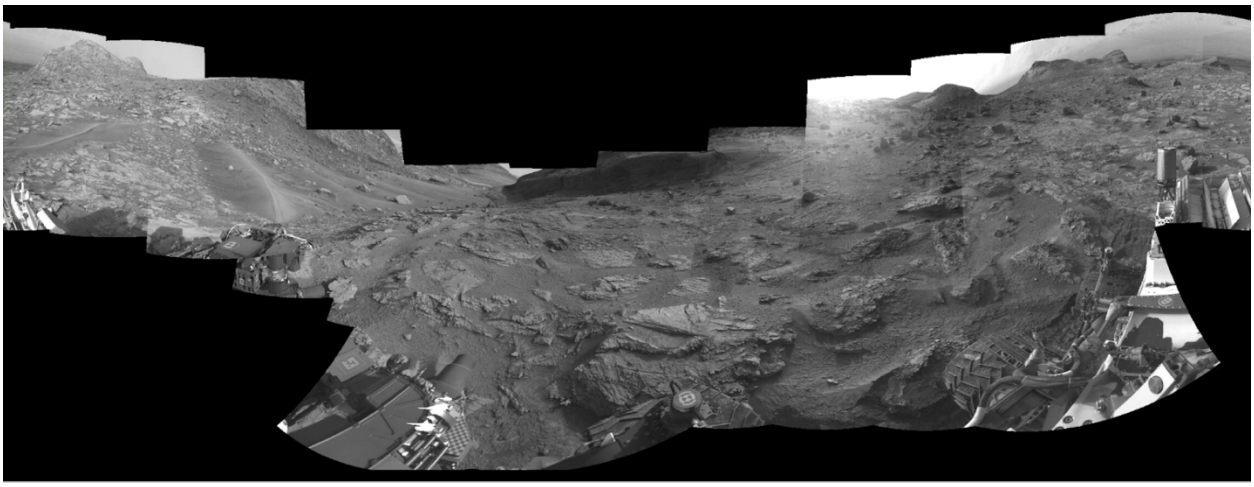
```
ls NLB*.IMG >left.lis
ls NRB*.IMG >right.lis
```

Run marsmcauley two times. Once to generate the left mosaic and again to generate the right mosaic.

```
$MARSLIB/marsmcauley inp=left.lis rinp=right.lis out=left_cylper_mosaic.vic
brtcorr=left_mosaic.brt surf_coord=local_level -left

$MARSLIB/marsmcauley inp=left.lis rinp=right.lis out=right_cylper_mosaic.vic
brtcorr=right_mosaic.brt surf_coord=local_level -right

xvd -min 0 -max 4095 left_cylper_mosaic.vic -fit &
```

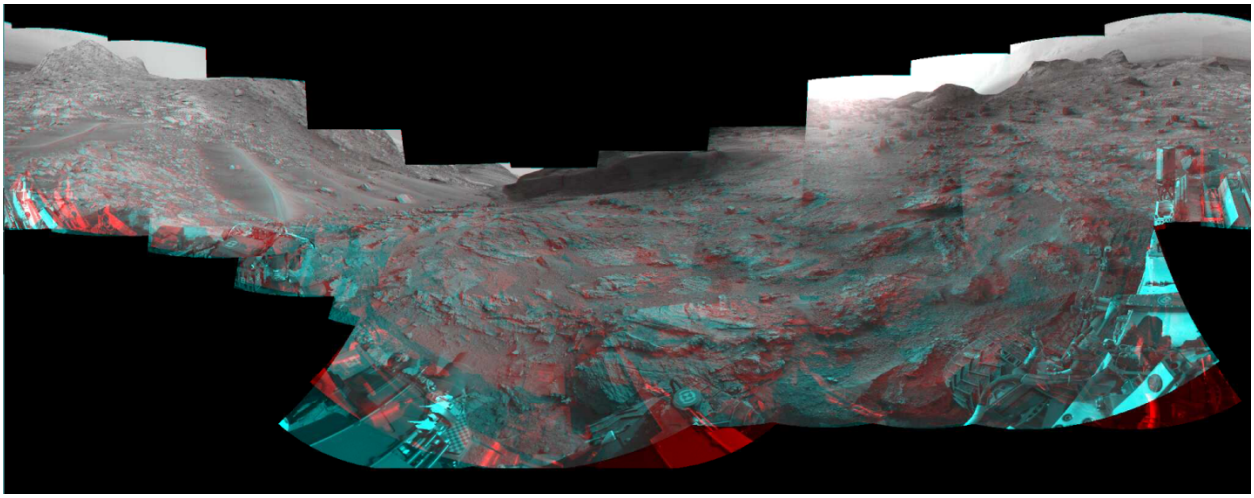


```
xvd -min 0 -max 4095 right_cylper_mosaic.vic -fit &
```



A cylper projected anaglyph mosaic can be generated using viccub. To make a black and white anaglyph run:

```
$R2LIB/viccub \(left_cylper_mosaic.vic right_cylper_mosaic.vic
right_cylper_mosaic.vic\) bw_cylper_anaglyph_mosaic.vic
xvd -min 0 -max 4095 bw_cylper_anaglyph_mosaic.vic -fit &
```



Stereo vision works based on disparity, which is the difference in what the left and right eyes see. The red and blue color fringing are what creates depth when the image is viewed with red/blue glasses (red on the left eye). If the fringing goes one way, the image is in front of the screen. If it goes the other way, the image appears behind the screen. Where there is no fringing, the image appears at screen level. This can be easily identified by looking at the image without stereo glasses. Where the image is purely grayscale with no color fringing, that's the screen level.

What's wrong with this image? Well, humans are good at looking at things behind the screen. This mimics what happens when we look out a window, for example. We are not good at looking at things in front of the screen. It works, but weird things happen along the edges, and if it gets too close you have to go cross-eyed to see it, which can be painful. Looking at the gray line, it is on the horizon, meaning that the entire image is in front of the screen. This is going to

be a very difficult mosaic to view.

What we want is to move that gray line down the image. Experience shows something like 2/3 or 3/4 down from the top is good. That puts most of the image behind the screen, with just a little sticking up (a little is good).

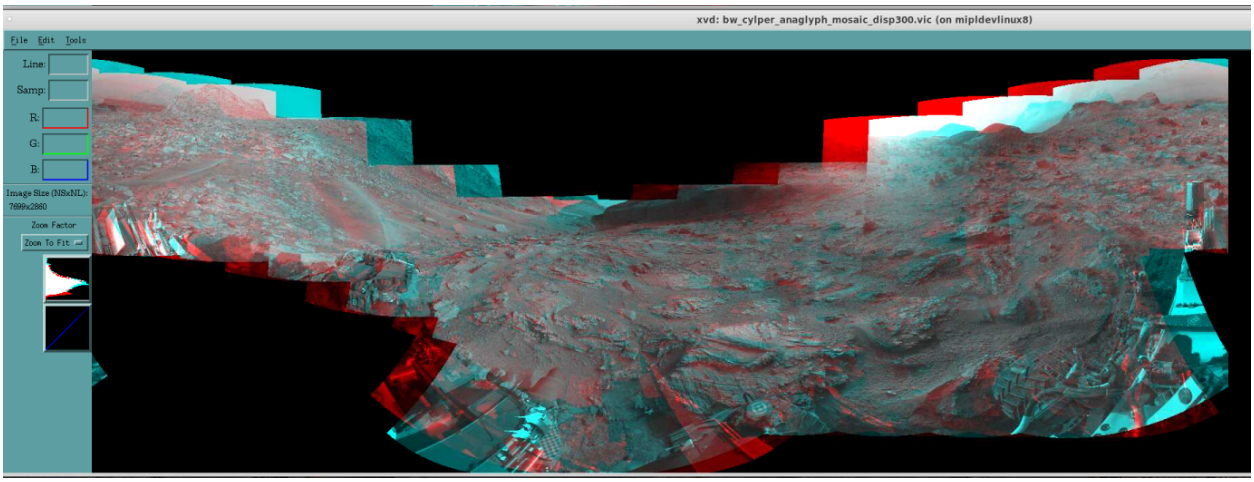
We can adjust the gray line using the disp_pix parameter. This shifts the entire image left or right by the given number of pixels. By convention, we usually adjust the right image, but that is not mandatory. If we use a value of 300, that moves the gray line down into a better place for viewing. Adjust as needed. It is important to note that if the distance between the red and blue features exactly matches the physical distance between your eyes, the image will appear to be at infinity. That would be perfect for a mosaic, if the horizon were set that way. However, it is a very sensitive parameter. Just a little too far, and you have to go wall-eyed to view the image, which most people can't do. So, you have to edge it as far back as you can for good viewing, without exceeding that distance. Unfortunately, when making a mosaic we usually do not know what hardware it will be viewed on. For that reason, we tend to be a bit conservative, and put the horizon closer than it ideally should be. Experiment.

Rerun the right mosaic (only) with disp_pix parameter:

```
$MARSLIB/marsmcauley inp=left.lis rinp=right.lis
out=right_cylper_mosaic_disp300.vic brtcorr=right_mosaic.brt
surf_coord=local_level disp_pix=-300 -right
$R2LIB/viccub \(left_cylper_mosaic.vic right_cylper_mosaic_disp300.vic
right_cylper_mosaic_disp300.vic\) bw_cylper_anaglyph_mosaic_disp300.vic
```

The new mosaic has the gray line toward the bottom, and is much easier to view.

```
xvd -min 0 -max 4095 bw_cylper_anaglyph_mosaic_disp300.vic -fit &
```



10.16 Orthorectified Mosaic

Program used: marsortho
Example Mission: MSL
Example Sol Site Drive: 3546 96 1766
Sample Data Directory: OrthorectifiedMosaic

To assemble multiple frames into a mosaic in an orthorectified projection, it takes both XYZ data to define where each point is, and skin data associated with the XYZ to provide the imagery. In order to generate an ortho output, a one-to-one correspondence is required between XYZ and skin data in their respective list files. Moreover, every entry in the skin list should be geometrically registered to every entry in the XYZ list.

Generate list file for Navcam ILT files.

```
ls *ILT*.IMG > mosaic.lis
```

Generate list files for Navcam XYZ files.

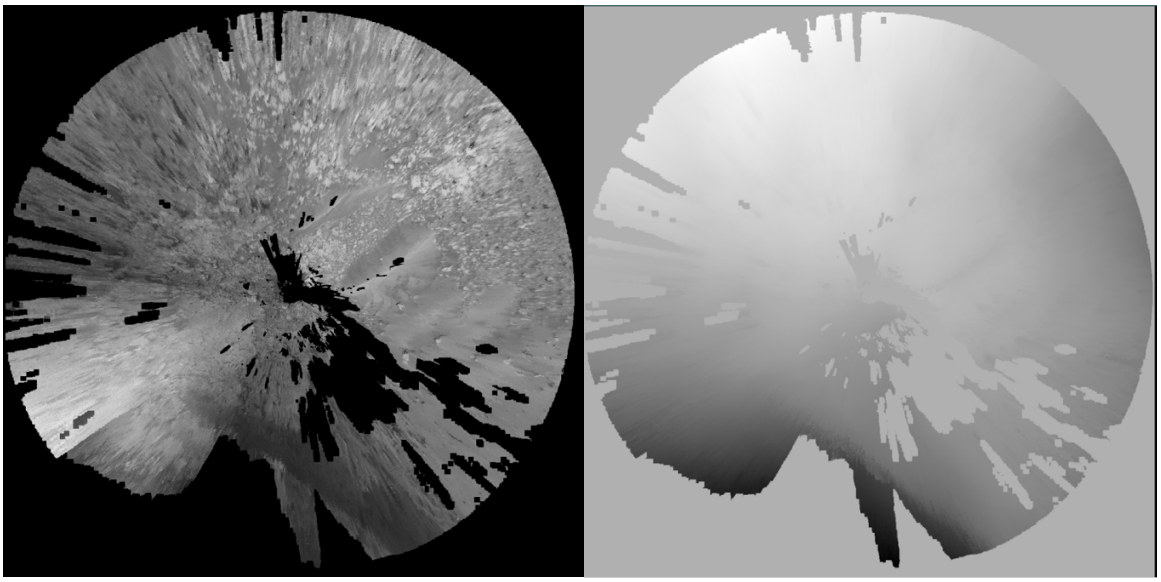
```
ls *XYZ*.IMG > xyz.lis
```

The data in both list files needs to have a 1 to 1 order to each other. Reorder the list files if necessary, although they should naturally match due to sorting by the “ls” command.

We now generate the ortho mosaic and dem mosaic at 45M range using xyz.lis, and mosaic.lis (you can optionally use the .brt file from the earlier section if you want). Specifying a range will override the automatically determined min/max x/y, so for example saying range=45 will set the output to +/- 45 meters in both directions. The scale parameter for both DEM and ORTHO output mosaic is measured in meters per pixel. If it is defined, the mosaic will be set exactly at that resolution.

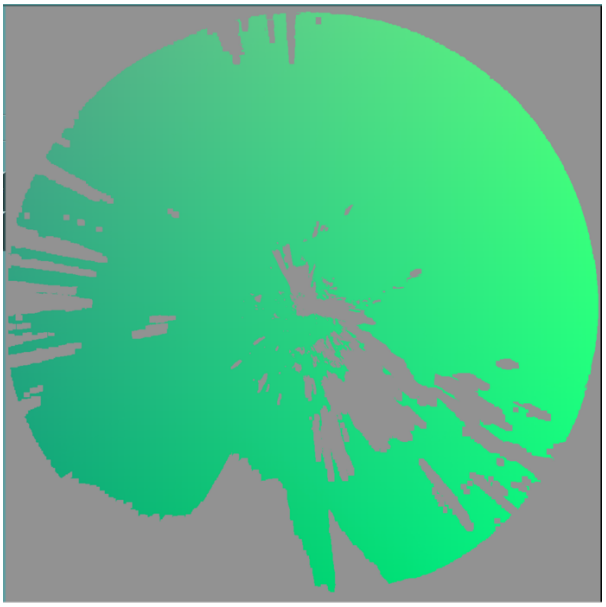
Generate a 45M range ortho mosaic and DEM with a scale of 0.01 m/px.

```
$MARSLIB/marsortho inp=mosaic.lis in_xyz=xyz.lis
out=navcam_ortho_mosaic_45M.vic out_dem=navcam_dem_mosaic_45M.vic
dem_nodata=0 fill_factor=4 overlay=first -weighted -erosion erode_factor=1
coord=local_level write_coord=rover scale=.01 range=45

xvd -min 0 -max 4095 navcam_ortho_mosaic_45M.vic -fit &

xvd navcam_dem_mosaic_45M.vic -fit &
```



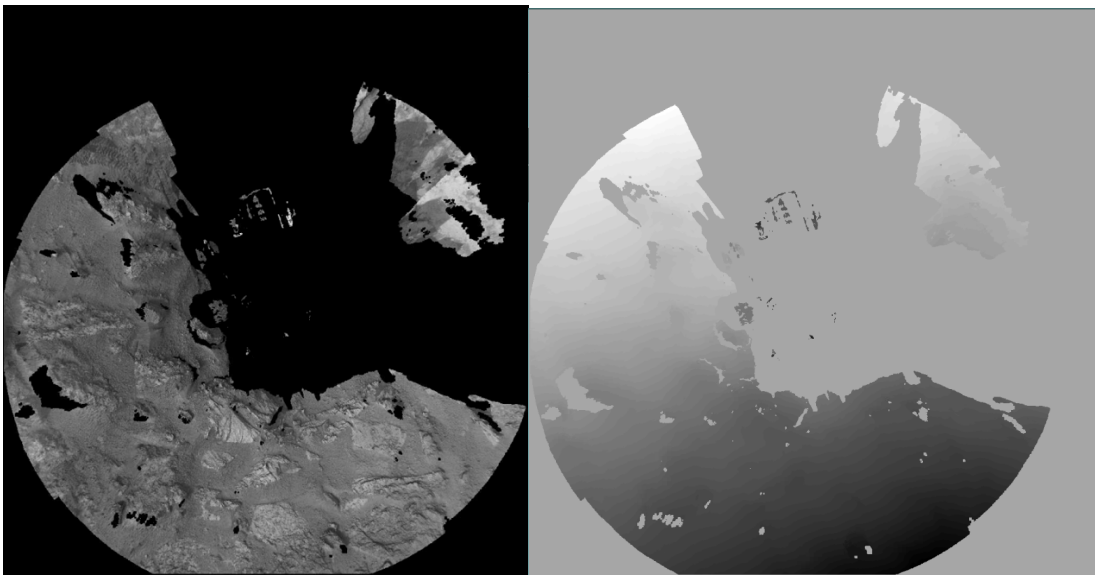
Generate a 45M range XYZ ortho mosaic with a scale of 0.01 m/pixel. Note that we already generated the DEM above so there is no need to redo it; suppress that by leaving off the out_dem parameter.

```
$MARSLIB/marsortho inp=xyz.lis in_xyz=xyz.lis out=navcam_xyz_mosaic_45M.vic
vic dem_nodata=0 fill_factor=4 overlay=first -weighted -erosion
erode_factor=1 coord=local_level write_coord=rover scale=.01 range=45

xvd navcam_xyz_mosaic_45M.vic -fit &

xvd navcam_xyzdem_mosaic_45M.vic -fit &
```



Generate a 5M range ortho mosaic with a scale of 0.001 m/pixel:

```
$MARSLIB/marsortho inp=mosaic.lis in_xyz=xyz.lis
out=navcam_ortho_mosaic_5M.vic out_dem=navcam_dem_mosaic_5M.vic dem_nodata=0
fill_factor=4 overlay=first -weighted -erosion erode_factor=1
coord=local_level write_coord=rover scale=.001 range=5

xvd -min 0 -max 4095 navcam_ortho_mosaic_5M.vic -fit &

xvd navcam_dem_mosaic_5M.vic -fit &
```



An XYZ ortho can be generated similarly. Note that you can supply any image type as the skin file, so instead of an XYZ mosaic you could make a slope mosaic, or a CPG color mosaic, or anything else.

10.17 Overlay mosaics (XYZ etc)

Programs used: marsmap
Example Mission: MSL
Example Sol Site Drive: 3546 96 1766
Sample Data Directory: OverlayMosaic

Mosaics need not be made just from images. The inputs can be any RDR type that is in image format. So, you can make mosaics of just about anything. In this example, we make a mosaic from the XYZ images in the ortho example.

Make list file of XYZ images

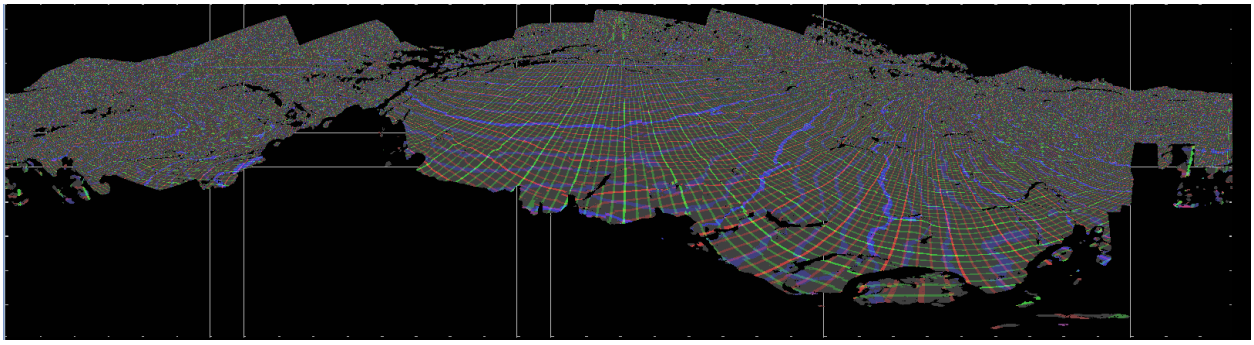
```
ls *XYZ*.IMG > xyz.lis
```

Generate the Navcam Cylindrical mosaic:

```
$MARSLIB/marsmap inp=xyz.lis out=navcam_xyz_mosaic.vic -cyl
surf_coord=local_level
```

This can be displayed with JadeViewer to better show the XYZ data. Note that if you make the cylindrical image mosaic as well, you can include it as a third argument below, select View Overlays, and see the image data on top of the mosaic. In this case, the image is a bit dark, so go to the Background tab, then Operator Controls, and adjust the data range to 0-4095.

```
java -Dsun.java2d.xrender=false jpl.mipl.jade.viewer.ImageViewer
navcam_xyz_mosaic.vic  XYZ &
```



10.18 Using IDX/ICM Outputs to Make XYZ Mosaics

Programs used: marsmap, marsremos
Example Mission: MSL
Example Sol Site Drive: 3546 96 1766
Sample Data Directory: OverlayMosaicIDXICM

A new concept in mosaics was introduced with M20 and MSAM, although it will work with any mission.

IDX files contain a (1-based) integer which specifies the image number the pixel in the mosaic came from. This is the same as the line number in the associated .LIS file. Where more than one input image overlaps, the one actually shown in the mosaic ("on top") is the one listed in the index file.

ICM files contain a pair of floating-point numbers which contain the image coordinate in the specified input file from which this pixel came. Image coordinates are fractional; mosaics are bilinearly interpolated around the given coordinate. The first band of the image is the line number; the second is the sample number (this is the exact same format as disparity files). Coordinates are 1-based and integers fall on the center of the pixel, so (1.0,1.0) is the center of the upper left pixel in the input image. That pixel thus extends from +0.5 to +1.5 in pixel space.

If you generate IDX/ICM files along with your mosaics, they can be used to precisely determine which source pixel was used for any point in the mosaic. This allows you to extract values from the RDR products associated with the mosaic, even if you don't have a mosaic made from that RDR type. For example, you can extract a slope value from the SLP product.

This works as follows: Given a mosaic pixel, find the same pixel in the corresponding ICM and IDX mosaics. The value in the ICM is the image number; it corresponds to the line number in the list. The value in the IDX is the image coordinate (line/sample, 1-based) from which the pixel came. The desired RDR can be brought up in xvd or Marsviewer, open the DN display window, and then move the cursor to the given coordinate.

These files make it possible to create mosaics of any of the RDR data types (for example, slopes). A program named marsremos goes through the mosaic pixel by pixel, uses the IDX and list file to find the image in question, pulls the pixel value from it using the ICM, and puts it in the output file. If the output is named properly, and put in an appropriate directory structure, Marsviewer will visualize it as an overlay just like it does for image RDRs.

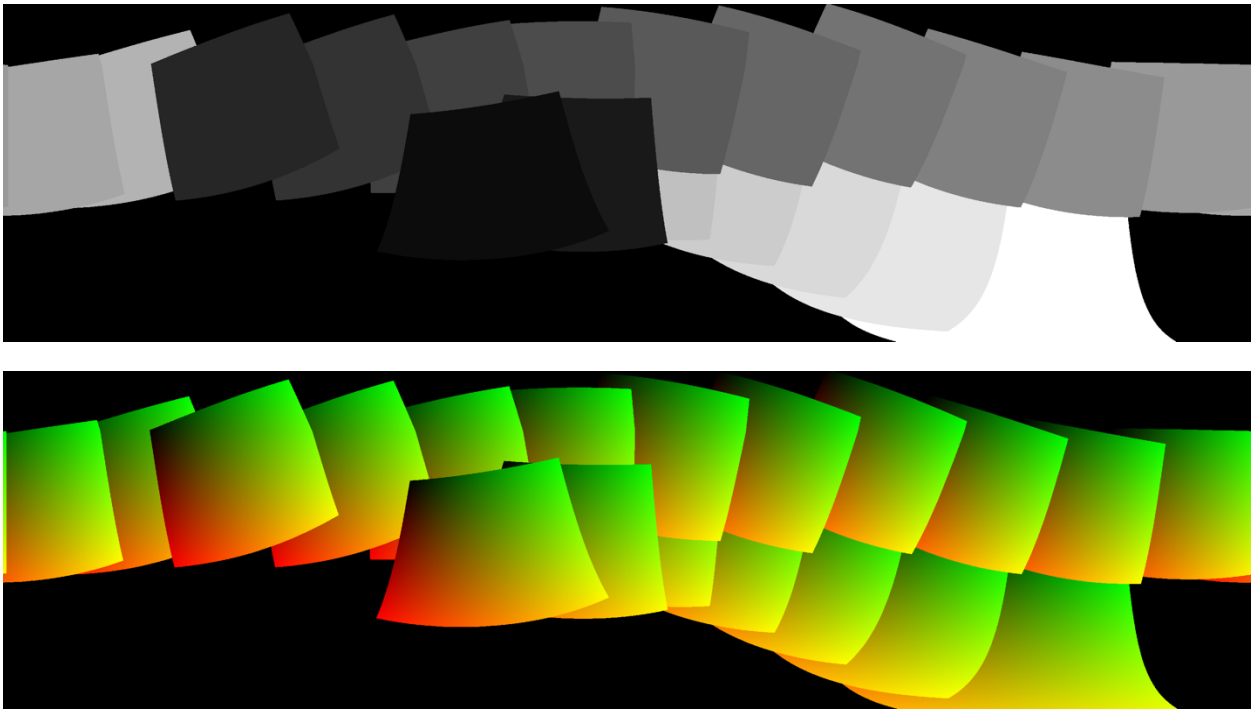
Make list file of image and XYZ products (or use list file from previous section)

```
ls *ILT*.IMG > mosaic.lis
ls *XYZ*.IMG > xyz.lis
```

Generate a standard mosaic, but create the IDX and ICM outputs:

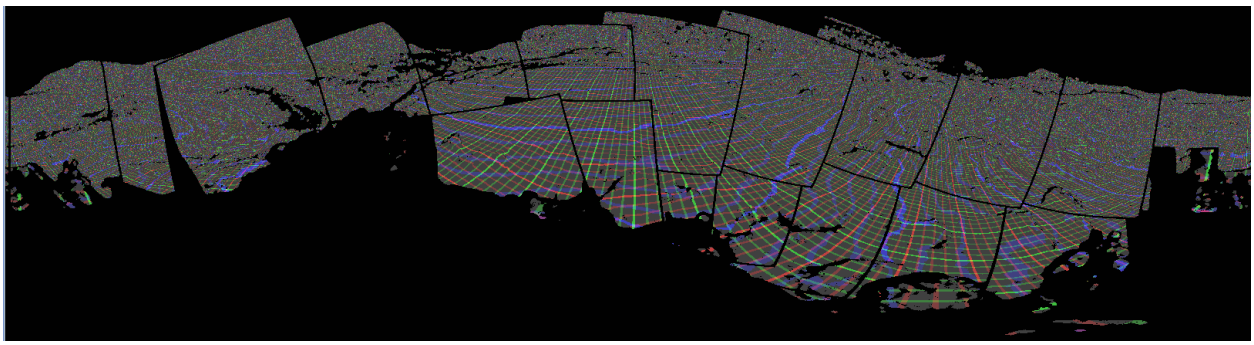
```
$MARSLIB/marsmap inp=mosaic.lis out=navcam_img_overlay_mosaic.vic -cyl
IDX_OUT=navcam_idx_overlay_mosaic.vic ICM_OUT=navcam_icm_overlay_mosaic.vic
surf_coord=local_level

xvd navcam_idx_overlay_mosaic.vic &
```

```
xvd navcam_icm_overlay_mosaic.vic &
```



Now we can generate the XYZ mosaic using the IDX and ICM as inputs:

```
$MARSLIB/marsremos inp=xyz.lis out=navcam_xyz_overlay_mosaic.vic
idx=navcam_idx_overlay_mosaic.vic icm=navcam_icm_overlay_mosaic.vic
coord=local_level

java -Dsun.java2d.xrender=false jpl.mipl.jade.viewer.ImageViewer
navcam_xyz_overlay_mosaic.vic XYZ &
```



There is an important distinction to be made between the marsremos XYZ mosaic and the direct XYZ mosaic from the previous section. Notice that the marsremos version has gaps between the images. That's because it takes XYZ data from the same pixels as the image data, and that normally extends all the way to the edge of the image. The edges of the image never have XYZ data because correlation is an area-based approach, and images also do not have perfect overlap.

So, what's the advantage, and why is M20 using the marsremos version to make XYZ mosaics operationally despite the direct mosaic looking "better"? There are a couple factors. First, the mosaic extents are re-created in the direct mosaic. If for example the XYZ images were linearized, this would mean the mosaic was slightly different from the image mosaic, unless the projection parameters were all specified (normally they are determined automatically). If there's any difference in nav files, that will also make a different mosaic. But the primary issue is that in the places where the marsremos version shows gaps, the direct mosaic shows the imagery from one input image, and the XYZ data from a *different* image! That's because the image that is supposed to be "on top" in the gap area is actually transparent around the edges, and that allows lower images to show through. That creates a mistaken impression, and any geometric seams appear to be in different places.

Because the marsremos version uses the exact same projection, the exact data is shown in the right places (assuming of course the list file is correct). Thus, the XYZ overlay is guaranteed to be perfectly coregistered with the image.

10.19 Making Movies/Blinks

Program used: average, f2, cform, transcoder, convert (part of ImageMagick. https://legacy.imagemagick.org)
Example Mission: MSL
Example Sol Site Drive: 3395 93 2164
Sample Data Directory: MoviesBlinks

Images provided capture dust devil activity on sol 3395. Dust devils on Mars usually do not contain much 'dust' and are not easily visible in the original images. Preprocessing of the EDRs is usually required prior gif movie creation to enhance the dust devil activity. Usually the best way to process a dust devil (or cloud movie) is to generate an image that represents the mean of all the frames in the movie (one image with all individual frames stacked and averaged), generate an image that represents the difference between each individual frame and the mean, and then stretch the difference. The below steps walk through these preprocessing steps.

Make a list file of Navcams

```
ls *.IMG > ddm.lis
```

Generate averaged image of all frames

```
$R2LIB/average \( `cat ddm.lis` \) mean.vic
```

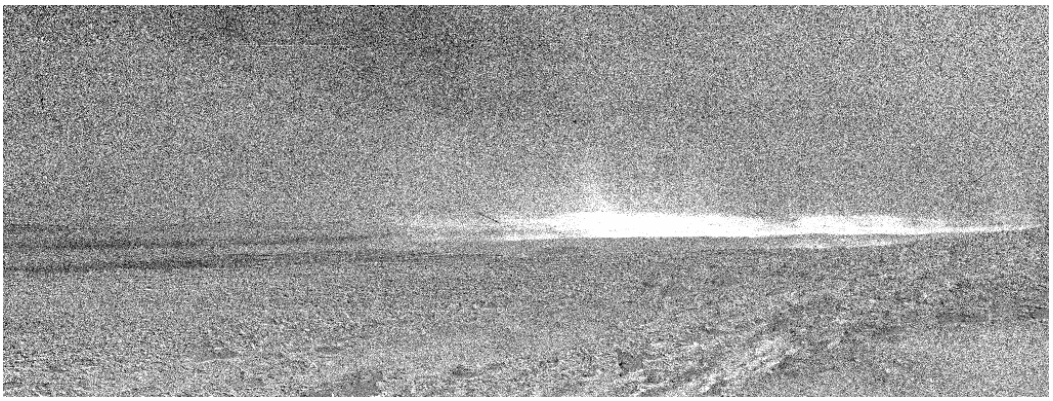
Using a for loop, iterate through each image to 1) generate a difference image by subtracting each frame from averaged image (**`mean.vic`**); 2) restretch the differenced images; 3) convert from **`.vic`** to **`.png`**

```
foreach i (`cat ddm.lis`)
  $R2LIB/f2 \( $i mean.vic\) ${i:r}.diff func='"in1-in2"' -half
  $R2LIB/cform ${i:r}.diff ${i:r}.vicb irange=\(-30 30\) orange=\(0 255\) -
byte
  $V2JDK/bin/java -Xmx3072m jpl.mipl.io.jConvertIIO inp=${i:r}.vicb
out=${i:r}.png format=png 2rgb=true oform=byte ri=true
end
```

Make the movie with ImageMagick's **`convert`** routine (or any other method you prefer)

```
convert -delay 50 -loop 0 *.png 3395_NCAM00596_DDM.gif
```

Below is one frame from that movie.



10.20 Multi-Instrument Mosaic

Programs used: marsautotie, marsnav, marsmap, cform, transcoder, convert (part of ImageMagick. https://legacy.imagemagick.org)
Example Mission: M20
Example Sol Site Drive: 441 24 4358
Sample Data Directory: MultiInstrumentMosaic

Mosaics can be made with more than one instrument. You do have to worry a little more about parallax in this situation. This example works through a Mars 2020 example of Supercam RMI over top of Mastcam-Z. This shows you the context around the RMI image, which is often hard to interpret because of its very narrow field of view. It also shows the use of automated tiepointing.

First make a list file from the *.IMG files with output list name as:

```
ls *.IMG >LZLRGB_0441_FDR_0244358_CYL_S_RZBALDFJ01.LIS
```

The list should be in the order RMI, RMI, ZCAM (but this is automatic from “ls” sorting).

For marsautotie, the **ground** and **normal** parameters provide an approximate surface model to help in finding the proper tiepoints. They are both expressed in rover frame, and can be extracted by looking at Navcam images of the same spot (or anything that has stereo coverage). Ground comes from the XYR product (XYZ in Rover frame), while normal comes from the UVW (surface normal in rover frame) product. These images are not supplied with the sample data, but can be obtained from PDS, or generated using the tools above. The XYZ and surface normal coordinate are in the command line below. Note that it only need be approximate.

```
$MARSLIB/marsautotie LZLRGB_0441_FDR_0244358_CYL_S_RZBALDFJ01.LIS
LZLRGB_0441_RAS_0244358_CYL_S_RZBALDFJ01.TIE templ=21 linear_q=0.5 q=0.6
border=20 dens=50 busy=10 fov=60 surf_coord=ROVER ground=\(3.541 1.801 -
0.202\) normal=\(0.002 0.007 -1.000\)
```

For marsnav, -remove lets it take out bad tiepoints, whose residual errors are above max_resid (a tunable parameter). Look at the stdout output for clues in tuning these parameters. The tiepoint list output by marsnav shows the residual error of each tiepoint in the last column. The ref=3 parameter tells it to not let the ZCAM image move – rather, the RMIs will move to match it.

```
$MARSLIB/marsnav LZLRGB_0441_FDR_0244358_CYL_S_RZBALDFJ01.LIS
LZLRGB_0441_RAS_0244358_CYL_S_RZBALDFJ01.NAV
LZLRGB_0441_RAS_0244358_CYL_S_RZBALDFJ01.TIE out_sol=rgd inertia=\(.1 .1 .1
.1\) point=pm=scale surf_coord=ROVER  ref=3 -do_surf -remove max_resid=5
```

Run marsmap to create mosaics. The first shows the overlay at full RMI resolution. The second two create a blink at half resolution. The topel and bottomel parameters were determined by trial and error to trim the output to the desired size.

```
$MARSLIB/marsmap LZLRGB_0441_FDR_0244358_CYL_S_RZBALDFJ01.LIS
LZLRGB_0441_RAS_0244358_CYL_S_RZBALDFJ01.VIC
nav=LZLRGB_0441_RAS_0244358_CYL_S_RZBALDFJ01.NAV fov=\(60 60\) -tight topel=-
27.9 bottomel=-33.9
```

```
$MARSLIB/marsmap LZLRGB_0441_FDR_0244358_CYL_S_RZBALDFJ01.LIS
LZLRGB_0441_RAS_0244358_CYL_S_BLBALDFJ01.VIC
nav=LZLRGB_0441_RAS_0244358_CYL_S_RZBALDFJ01.NAV fov=\(60 60\) -tight -nogrid
topel=-27.9 bottomel=-33.9 zoom=0.5

$MARSLIB/marsmap LZLRGB_0441_FDR_0244358_CYL_S_RZBALDFJ01.LIS
LZLRGB_0441_RAS_0244358_CYL_S_BZBALDFJ01.VIC
nav=LZLRGB_0441_RAS_0244358_CYL_S_RZBALDFJ01.NAV fov=\(60 60\) -tight -nogrid
input_range=\(3 4\) topel=-27.9 bottomel=-33.9 zoom=0.5
```

Stretch the images to byte. The input range of 0 to 6000 is again determined by trial and error and can change as needed for any given mosaic. Note that there's no rule the output be called .VIC or .vic - here, .vicb is used to indicate it's a byte version.

```
$R2LIB/cform LZLRGB_0441_RAS_0244358_CYL_S_RZBALDFJ01.VIC
LZLRGB_0441_RAS_0244358_CYL_S_RZBALDFJ01.vicb irange=\(0 6000\) orange=\(0
255\) -byte

$R2LIB/cform LZLRGB_0441_RAS_0244358_CYL_S_BLBALDFJ01.VIC
LZLRGB_0441_RAS_0244358_CYL_S_BLBALDFJ01.vicb irange=\(0 6000\) orange=\(0
255\) -byte

$R2LIB/cform LZLRGB_0441_RAS_0244358_CYL_S_BZBALDFJ01.VIC
LZLRGB_0441_RAS_0244358_CYL_S_BZBALDFJ01.vicb irange=\(0 6000\) orange=\(0
255\) -byte
```

Using a for loop, convert from **`.vicb`** to **`.png`**

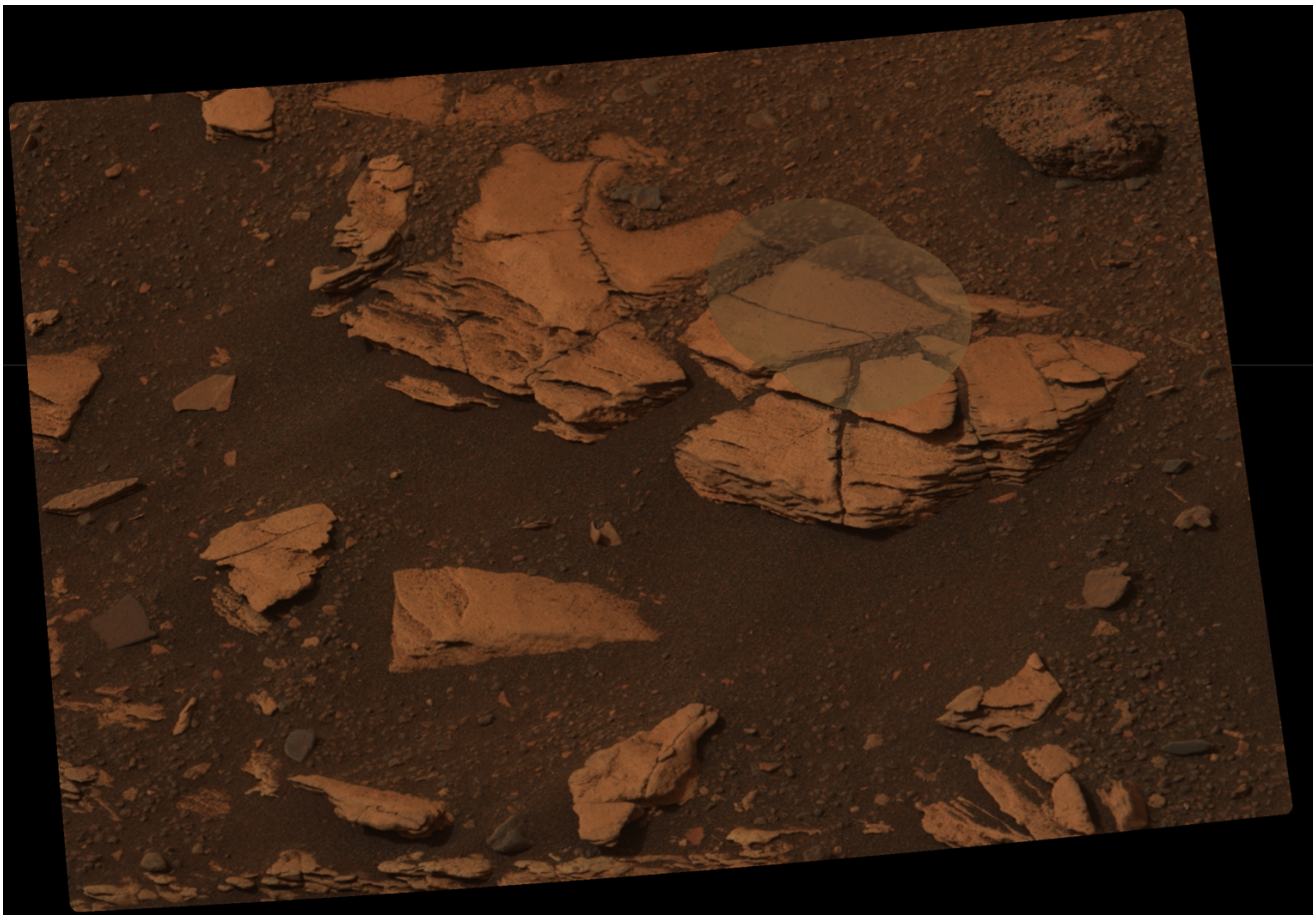
```
foreach i (*.vicb)

  $V2JDK/bin/java -Xmx3072m jpl.mipl.io.jConvertIIO inp=${i:r}.vicb
out=${i:r}.png format=png 2rgb=true oform=byte ri=true

end
```

Make a blink gif with ImageMagick's **`convert`** routine (https://legacy.imagemagick.org)

```
convert -loop 0 -delay 100 LZLRGB_0441_RAS_0244358_CYL_S_BLBALDFJ01.png
LZLRGB_0441_RAS_0244358_CYL_S_BZBALDFJ01.png sol0441_RZBALDF_blink.gif
```

Outputs are a full-res combined mosaic, and half-res mosaics of the Mastcam-Z background, with and without the RMI on top. These are combined into an animated gif blink movie, to better show the coregistration of the images.

```
xvd LZLRGB_0441_RAS_0244358_CYL_S_RZBALDFJ01.VIC &
```



10.21 In-Situ Mesh Creation

Programs used: marsmesh, cform, transcoder
Example Mission: M20
Example Sol Site Drive: 175 6 1752
Sample Data Directory: StereoCorrelation (plus results of Case 10.7)

Given an XYZ resulting from stereo analysis and an image, a mesh can be created. These terrain meshes convert the XYZ point cloud into a set of geometry triangles in the “obj” format, suitable for loading into 3D viewers such as Meshlab (http://www.meshlab.net).

This section presumes you have run the Stereo Correlation example from Section 10.7, as it uses the XYZ output from that process. The results from Surface Normal are not needed, although they don’t hurt to be present.

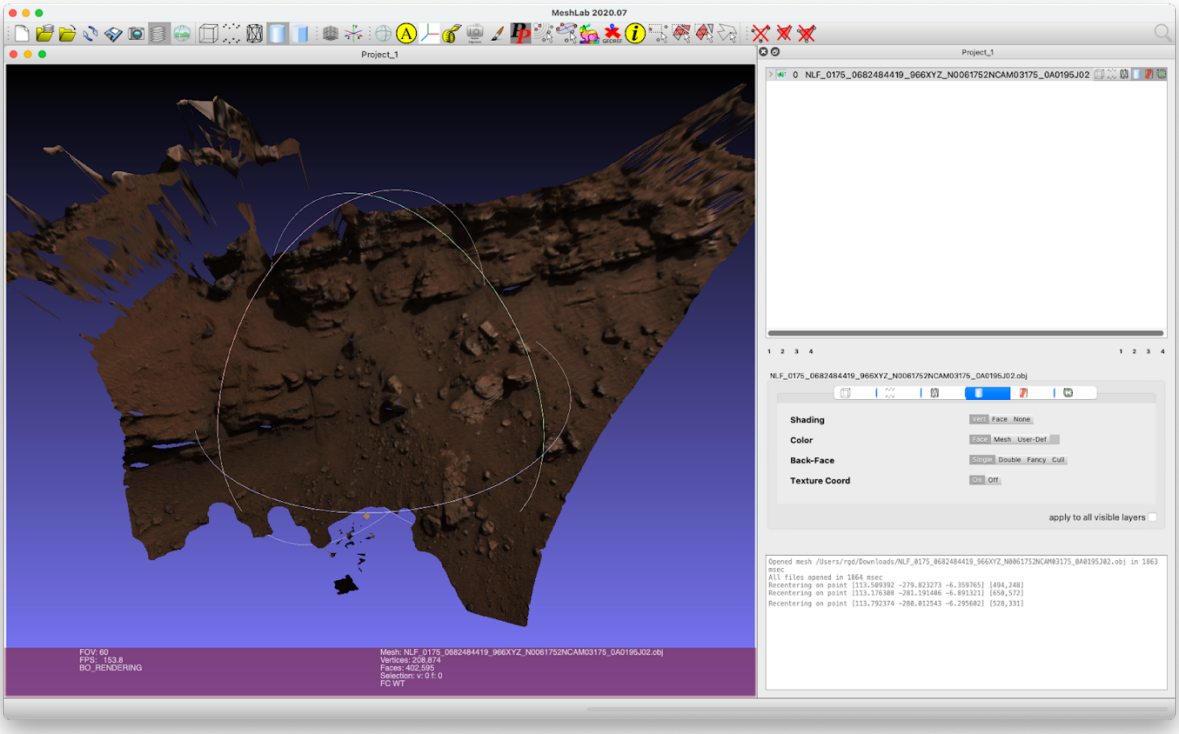
First, we convert the image to PNG format for use as the skin. This is done using $R2LIB/cform to convert it to byte, followed by the transcoder to convert it to PNG. Note that you may wish to adjust the data range on the cform call if the mesh is too bright or dark. Then marsmesh does the actual work. Also note the .VICb extension, for VICAR Byte. There is no requirement that an image file end with either .VIC or .IMG from the VICAR point of view - any name is acceptable. PDS has rules, and some tools like Marsviewer do too, but as a rule VICAR programs - including the Mars programs - are prohibited from analyzing the filename or

requiring any particular filename pattern.

```
$R2LIB/cform NLF_0175_0682484419_966RAS_N0061752NCAM03175_0A0195J02.IMG
NLF_0175_0682484419_966RAS_N0061752NCAM03175_0A0195J02.VICb irange=\(0 4095\)
orange=\(0 255\) -byte

java jpl.mipl.io.jConvertIIO
inp=NLF_0175_0682484419_966RAS_N0061752NCAM03175_0A0195J02.VICb
out=NLF_0175_0682484419_966RAS_N0061752NCAM03175_0A0195J02.png format=png

$MARSLIB/marsmesh NLF_0175_0682484419_966XYZ_N0061752NCAM03175_0A0195J02.VIC
NLF_0175_0682484419_966XYZ_N0061752NCAM03175_0A0195J02.obj
NLF_0175_0682484419_966RAS_N0061752NCAM03175_0A0195J02.VIC range_max=30
res_max=500000 lod_levels=1 max_angle=87 res_min=3000 -adaptive
point=cm=label
```

Note that the skin file is given as a .VIC - this name is converted internally to a .png (one of the rare exceptions where extension matters). The range_max species the maximum range for the mesh; it is cut off beyond this point. 30m is quite close, but is okay for this scene. The rest of the parameters are generally constants.

The marsmesh program outputs a `.obj` and `.mtl` file. These go together, and along with the `.png` file constitute the mesh. The `_LOD` outputs can be ignored (different levels of detail) as can the `.lbl` file (used to help create a PDS4 label for the mesh) and the `.iv` file (binary Inventor format of the mesh). The three mesh files can be loaded into Meshlab or other 3D viewing program. Because +Z is down for Mars data, the mesh often loads in Meshlab looking at the bottom, where there is no texture. Flip it around to see the top.



10.22 Making an orbital mesh

Programs used: marsmesh, f2, viccub, cmodgen
Example Mission: M20
Example Sol Site Drive: 175 6 1752
Sample Data Directory: OrbitalMesh

The process starts from a Digital Elevation Map (DEM) and corresponding visual image. Note that the DEM and the visual image are expected to cover the same extent but may not necessarily have the same scale factors. While there are a number of ways to convert the DEM to a polygonal mesh, for compatibility reasons with our mesh generation software, we choose to convert the DEM image into XYZ and then use the same marsmesh program as we use for in-situ meshes. We use metadata that describes the DEM, specifically coordinates of the upper left corner, pixel scale, as well as definition of the mesh output desired coordinate system.

Usually, the DEM is described in Northing/Easting geographic Cartesian coordinates and conversion to the coordinate system in which output mesh is generated is done using PLACES queries. For this reason, both the DEM and corresponding visual image are fully defined in PLACES as "orbital" frames. Most commonly the output mesh is generated in the current SITE frame.

Because there's a bit of math involved here, we first show what the commands look like in the abstract using variables (in blue) and then later go to a concrete example (in black).

Once the coordinate conversion is done, the XYZ image could be constructed by generating a regular grid for the X/Y bands and using the DEM as the Z band. For example, the X band is computed for a given LINE as:

dem_scale*(upper_left_value_of_x-LINE)

Y band is computed for a given SAMPLE as:

dem_scale*(upper_left_value_of_y+SAMPLE)

The original values in the DEM are simply offset and usually reversed to form a Z-down band. The UL_line and samp below are the coordinates of the upper left corner of the tile with respect to the origin, measured in pixels (which, if you are centering the tile on the origin, is line=size/2, samp= - size/2).

```
$R2LIB/f2 out=x_band.VIC ns="dem_size" nl="dem_size"
func='"dem_scale*(UL_line-LINE)"' -real

$R2LIB/f2 out=y_band.VIC ns="dem_size" nl="dem_size"
func='"dem_scale*(UL_samp+SAMP)"' -real

$R2LIB/f2 dem.VIC dem_zzz.VIC func='"(-(IN1 - site_elevation))"'

$R2LIB/viccub \(x_band.VIC, y_band.VIC, dem_zzz.VIC\) out=xyz.VIC
```

Once an XYZ image is constructed, it is straightforward to convert it to a polygonal representation, but generating texture map coordinates for the corresponding visual image requires creation of a synthetic Camera Model. The synthetic camera is positioned in the center

of the image in X/Y, 10000m above the image in Z, looking straight down along the +Z axis.

```
FOV_x=2*arctan((dem_scale*(dem_size_x/2.0) / camera_height)
FOV_y=2*arctan((dem_scale*(dem_size_y/2.0) / camera_height)
```

The cmodgen program creates a synthetic camera model for this purpose. Unfortuantely the program is not yet open source, but we have provided a binary for several platforms in the Scripts dir (under a $VICCPU directory).

```
/home/sample/data/Scripts/$VICCPU/cmodgen cahvor
cam_pos_x,cam_pos_y,cam_pos_z 0,0,1 1,0,0
texture_image_size,texture_image_size FOV,FOV synthetic_camera_model.cahvor
```

During generation of texture coordinates, for each vertex on the polygonal mesh, the 3D point is back projected using the synthetic camera model to get the corresponding pixel values that are then converted to u/v texture coordinates.

```
$MARSLIB/marsmesh xyz.VIC output_mesh_name.obj texture_image.VIC
point_method='"mission=generic"' baseline=0.5 -fixed maxgap=0
range_max=100000.0
```

Very large orbital meshes may exceed the capacity of some renderers (~8k pixels on a side is a common limit for the texture map). Solving this requires cutting the mesh into tiles and generating each tile separately. Implementing this is left as an exercise for the reader.

Let's turn all that into a concrete example, using M20. The orbital map and DEM are available from the PLACES PDS delivery. Specifically:

https://pds-geosciences.wustl.edu/m2020/urn-nasa-pds-mars2020_rover_places/data_maps/m20_orbital_map.img

https://pds-geosciences.wustl.edu/m2020/urn-nasa-pds-mars2020_rover_places/data_maps/m20_orbital_dem.img

There is also a CSV file that tells where the rover is at any given point on that map:

https://pds-geosciences.wustl.edu/m2020/urn-nasa-pds-mars2020_rover_places/data_localizations/best_tactical.csv

These maps will change with each delivery, but we provided a snapshot of Release 6 in the sample data directory.

Let's say we wanted to make an orbital mesh centered around Site 31 with a 128 m radius (it is helpful for the size to be a power of 2). Search for SITE,31 in the .csv file and you get:

```
SITE,31,-1,-1,400.692,-2791.658,-
44.437,1093697.077,4351703.836,18.451331221,18.346169209,77.401257148,-
2525.285,3811.69,2495.34,952.92,623.84,0.000,-
0.000,0.000,1.000000000,0.000000000,0.000000000,0.000000000,3394505.682,33919
80.397,723053587,632
```

Fields 14 and 15 are map_pixel_line and map_pixel sample, while 16 and 17 are the same for

DEM, and field 13 is the elevation (which is +z up). That gives us:

map_pixel_line = 3811.69
map_pixel_sample = 2495.34
dem_pixel_line = 952.92
dem_pixel_sample = 623.84
elevation = -2525.285

The map is 0.25 m/pixel and the DEM is 1 m/pixel. So 128 m (radius) is 512 map pixels and 128 DEM pixels. We thus want a cutout of the image 128 m in each direction from the Site 31 location:

skin_radius = 512 (pixels)
dem_radius = 128 (pixels)
dem_scale = 1 (meters/pixels)

For the map, the starting line is map_pixel_line minus the skin_radius, and the number of lines is double the skin_radius (since we're going 128 m each direction). Similar for the others.

```
$R2LIB/copy m20_orbital_map.img map_cutout.VIC sl=3299 nl=1024 ss=1983
ns=1024

$R2LIB/copy m20_orbital_dem.img dem_cutout.VIC sl=824 nl=256 ss=495 ns=256
```

We create a PNG for use as the image skin:

```
java jpl.mipl.io.jConvertIIO inp=map_cutout.VIC out=map_cutout.png format=png
```

Now we can make an XYZ out of the DEM. The upper left is line=128 samp=-128 (remember line counts down from the top but x counts up from the origin). This will put x=0 y=0 (the site origin) in the middle of the cutout, which matches what we did above to make the cutout.

```
$R2LIB/f2 out=x_band.VIC ns=256 nl=256 func='"1*(128-LINE)"' -real

$R2LIB/f2 out=y_band.VIC ns=256 nl=256 func='"1*(-128+SAMP)"' -real
```

The site origin is at elevation = -2525.285m, with positive up. We subtract that from the DEM value (also positive up) to get the delta, then invert it to get +Z down (needed for the coordinate systems we use).

```
$R2LIB/f2 dem_cutout.VIC dem_zzz.VIC func='"-((IN1-(-2525.285)))"' -real

$R2LIB/viccub \(x_band.VIC, y_band.VIC, dem_zzz.VIC\) out=xyz.VIC
```

Next we construct the CAHVOR camera model. The camera model coordinate system is X=east, Y=north, Z=down, and we put the camera looking along +Z. Camera position is in the center of the terrain, a large distance up (cam_pos_z = -10,000 meters)

xfov = -2 * ((180.0 / pi) * arctan(($dem_scale * ($dem_size / 2)) / $cam_pos_z))
yfov = xfov = 1.47 (degrees)

We then plug this into cmodgen. The first cluster of numbers is the camera origin, the 4th is the

image size, and the 5th is the FOVs.

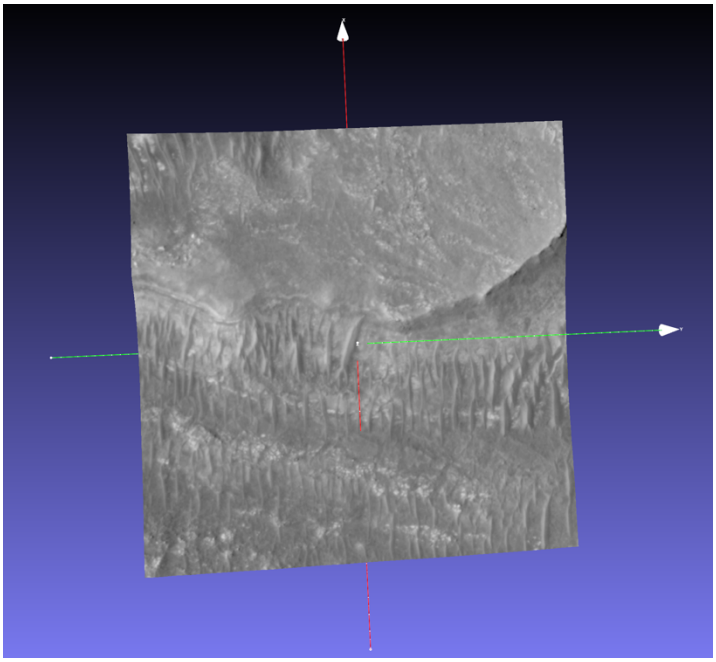
```
/home/sample/data/Scripts/$VICCPU/cmodgen cahvor 0,0,-10000 0,0,1 1,0,0
1024,1024 1.47,1.47 xyz.cahvor
```

The cmodgen program is picky, there must not be spaces where there are commas above in the clusters.

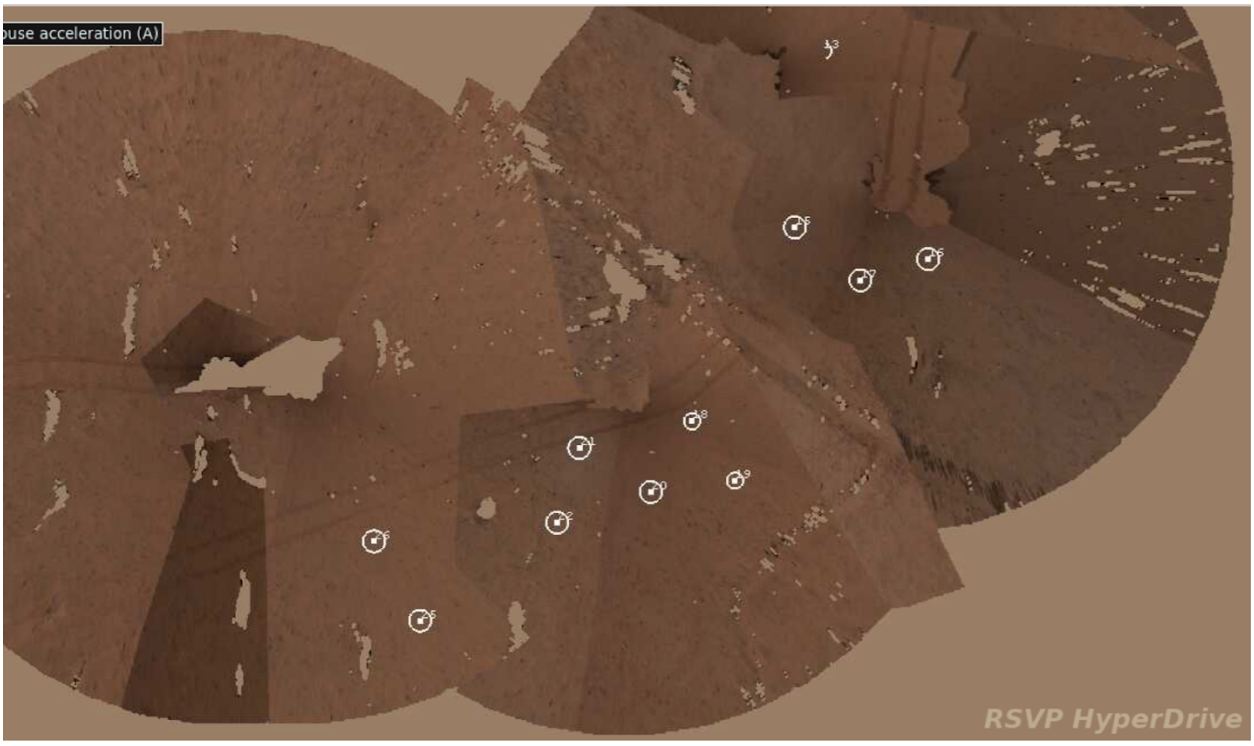
We have now generated a detached camera model for xyz.VIC. The marsmesh program uses the PIG libraries to process that input file. First it looks for a camera model in the label. Finding none, it then looks to see if there is detached camera model, the requirement being that the detached camera model name must match the name of image except for the file extension. Thus if the input XYZ image is named xyz.VIC, corresponding detached camera model file has to be named xyz.cahvor (or generically, .cahv or .cahvore). Also, the skin name is given as "map_cutout.VIC" but it requires there be a PNG of the same base name, i.e. "map_cutout.png".

```
$MARSLIB/marsmesh xyz.VIC site_31_orbital_mesh.obj map_cutout.VIC
point_method='"mission=generic"' baseline=0.5 -fixed maxgap=0
range_max=100000.0
```

The result, imported into marsmesh:



While we originally developed the process described here for processing HiRISE/CTX and other orbital data, it is straightforward to adapt the process to any DEM. For example, DEMs generated from Helicopter images, or orthographically projected in-situ Engineering or Science cameras. The key is to create an appropriate synthetic camera model so the skin can be properly mapped onto the mesh.
Below is an example of a polygonal mesh generated from M20 Navcam-based DEMs composed of localized orthographic projections from multiple rover positions.


RSVP HyperDrive

11. SOFTWARE ARCHITECTURE

VISOR is designed as a set of independent VICAR programs (>100 and growing) that work together by file-based communication. It is fundamentally a command-line architecture, with building blocks that can be put together in scripts to accomplish larger goals. Most “interesting” operations will involve calling more than one program.

The software is written in C++. The application programs themselves are mostly in a “C style”, i.e. functional rather than object-oriented. However, the PIG library is an object-oriented library that makes extensive use of mission-specific subclasses. There is nothing that says application programs can’t be more object-oriented, it’s just now how most of them are built.

11.1 Metadata (labels)

VISOR makes extensive use of metadata (image labels), in VICAR format. Converters are available to go to/from this format (see Section 5.5). Thus, although VISOR itself requires VICAR labels, the data could be in any of several other formats, converted in and out of VICAR as needed. The most useful other format is ODL (Object Description Language), the file format used by PDS3 for most images and labels. This can be converted easily to and from VICAR using the transcode (see Section 5.5). Other formats may need additional support to transfer the metadata. For PDS4 data, simply ignore the PDS4 label and use the data file itself, which will usually be in VICAR, ODL, or both formats. There is no specific PDS4 -> VICAR converter as of this writing, the converter only does VICAR/ODL -> PDS4. It should be possible to write a PigFileModel subclass that reads the PDS4 label, however, if desired.

The labels used with VISOR originated with Mars Pathfinder. They were significantly altered for the MER project, and all the other current in-situ projects use the MER-style labels. There have been some tweaks with each mission, but the basic structure is very similar since MER.

New missions are well advised to adopt a similar label structure for their data. In addition to being battle-tested across 5 major landed missions, the Planetary Data System (PDS4) data dictionaries were set up with this label structure in mind. Thus, it is straightforward to convert these MER-style labels to PDS4.

There are enough abstraction layers in the PIG library to be able to support different styles of metadata, at least to an extent. There is a PigFileModel that can convert input labels to the internal representation, and PigLabelModel for output labels. However, for most in-situ missions the benefits of changing the label style are dubious compared to the benefits gained by making use of the heritage.

11.2 PIG Library

VISOR is built around the PIG library. PIG (Planetary Image Geometry) is a C++ class library that encapsulates all of the mission-specific code behind a mission-independent set of base classes.

This architecture makes it easy to support new missions, as long as they are generally similar to the existing ones. The wide range of supported missions attests to this.

The PIG library takes care of things like label I/O, pointing cameras, maintaining coordinate frames, camera models, radiometric and color correction, and surface models.

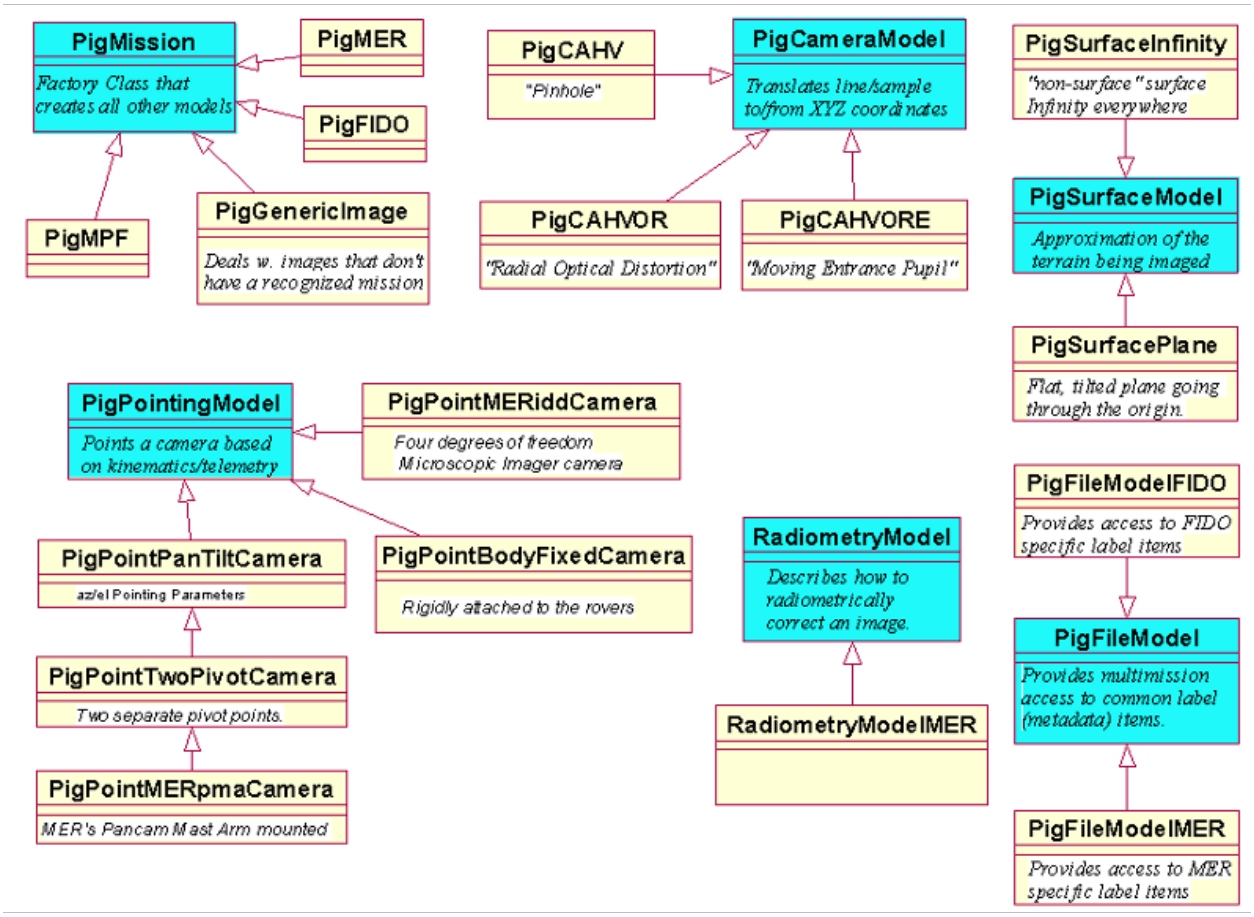
In the case of camera and surface models, the subclasses are not mission-specific, but the same architecture is used to support different kinds of camera and surface models.

11.3 Class Structure

PIG consists of a set of base classes presenting a generic interface to users, and subclases that implement those interfaces. This section will go through each of the primary base classes to get an overview of the design. Potential users are referred to the documentation in the .h and .cc files for specific info.

The code generally uses Javadoc-style comments in the .h file to document the class interfaces and contracts. These are often but not always repeated in the .cc file,which also includes specific implementation comments.

The figure below shows a diagram (from [Deen2003b]) depicting some of the major classes and subclasses in the PIG library. The diagram is representative, not exhaustive. For example, for each subclass named MER, there are analogous subclasses for most other missions. Not every base class is shown.



11.3.1 PigModelBase

This is the base class for all the others. It has functions for printing messages and getting parameters, which have a layer of indirection so the message printer or parameter getter can be replaced. There are also a number of static utility routines for use when instantiated objects are

not available. Notable among these are routines for reading and parsing configuration (calibration) files using $MARS_CONFIG_PATH.

11.3.2 PigMission

The primary base class for PIG is PigMission. This class, which is subclassed for each mission (e.g. PigM20, PigMSL), primarily acts as the factory that creates appropriate subclasses for each of the other base classes. It also provides some other services that are mission-specific but don't fit well elsewhere (e.g. coordinate system management).

The PigMission object (a singleton per mission) is passed around a lot in the code and is the key for accessing most mission-specific services. There are a few static methods for use before subclasses are created, but they should be used sparingly.

11.3.3 PigCameraModel

If there's one class that implements the primary purpose of PIG, it would be PigCameraModel. The camera model is at the core of almost everything VISOR does (see Section 6.5). PigCameraModel supplies a generic interface for camera models (at its core, project a 3D point to an image line/sample, and project line/sample to a vector in 3D space). Subclasses implement specific models, e.g. PigCAHV, PigCAHVOR, PigCAHVORE, and PigPSPH. Although most of the Mars missions use the CAHV family of camera models, there is nothing that mandates this, and the PSPH model is also supported. Support for the standard photogrammetry model is contemplated and should be doable, but this has not been implemented as of this writing.

Note that the camera model subclasses implement the math behind the camera model. Specific numeric parameters are read from calibration files, or from the label. The term "camera model" is often overloaded, sometimes referring to the math model, and sometimes the set of parameters for a specific camera.

11.3.4 PigPointingModel

PigPointingModel is where much of the mission-specific code is. It is responsible for taking a calibration camera model and "pointing" it, i.e. transforming it for use with a specific image. This generally means reading image labels to get joint positions (az/el motors on a mast, joint angles for an arm), computing kinematics, and moving the camera model to match that pose. Note that most arms don't actually use kinematics but rather take a final position and orientation resulting from the flight software's kinematics.

Pointing models have another very important function - adjusting the pointing. This is required for bundle adjustment, to reduce seams in mosaics and meshes. Each pointing model exposes a certain number of "pointing parameters". These are the free variables that can be used to tweak the camera position. They often correspond to available degrees of freedom (e.g. az/el for a mast camera), but not always. For example, many missions implement a "3dof" pointing model that adds a "twist", or rotation in the image plane, which is quite helpful for reducing mosaic seams but does not correspond to a specific physical motion (twist could be caused by thermal effects for example). Pointing models also generally adjust external camera parameters - position and orientation of the camera. But they can adjust internal parameters; for example the "scale" model that many missions implement treat image scale (aka zoom) as a free variable, which is helpful in cases where e.g. zoom or focus are not modeled, or are imprecisely modeled.

It is important to realize that these pointing parameters are intended to be opaque to callers - although names are available for user printouts, client programs should not interpret the meaning of these parameters, but rather treat them as simple knobs in the pointing black box, with no particular knowledge of what the effect would be.

The PigPointingModel inheritance tree actually has both functional and mission-specfic subclasses. For example, PigPointBodyFixedCamera, PigPointCamera[6|7]dof, PigPointPanTiltCamera, PigPointTwoPivotCamera implement specializations for that type of camera, while mission and camera specific classes such as PigPointMSLmastCamera and PigPointM20armCamera are specific to those missions. Often the specificity is minor, like what calibration file to use; in other cases these are direct subclasses of PigPointingModel and implement the entire functionality.

The 6 and 7 dof models deserve some mention. These are fully unconstrained pointing models, divorced from any physical motion constraints. The 6 DOF model implements everything that is needed for this - XYZ and the three Euler angles (roll, pitch, yaw). The 7 DOF model uses the four components of a quaternion rather than Euler angles for camera orientation. This is overconstrained - there are really only 3 degrees of rotational freedom despite having 4 numbers - but is often useful for numerical stability issues. The quaternion representation does not suffer from singularities like the Euler angles do (e.g. at the poles).

11.3.5 PigCoordSystem

PigCoordSystem describes the position and orientation of coordinate systems, and their relationship to each other. Formerly, subclasses were used for each coordinate system defined by a mission, but this has been recently changed to simply use instantiations of PigCoordSystem itself. PigCoordSystem instances are arranged in a tree; the root of the tree is the FIXED coord system. FIXED is generally the earliest Site frame used in any of the input data for the program run in question. CS instances keep track of who their parent is in the tree, as well as the transform (quaternion and offset) from their parent (via PigCSDefinition) and a cached transform to/from Fixed that is used for coordinate system translation.

So for example if a program has data from sites 11 and 12 as input, the Fixed frame will be the same as site 11 (technically speaking, site 11 has a 0-offset identity-quaternion relationship to Fixed). Site 12 has a translation relative to fixed. Then Rover Nav frame (often called just Rover frame) for e.g. site 12 drive 50 will be defined as having some offset and rotation relative to its parent (Site 12). But it also maintains the offset and rotation relative to Fixed, which is a composite of the Rover(12,50)->Site(12) transform and the Site(12)->Site(11) transform. This latter is what is used for coordinate system conversions, which always go from CS object A to Fixed to CS object B.

Another concept in the code is the “natural” frame for instruments. This is simply the frame in which the camera model is defined for that camera. It’s how data from that camera is expressed if there is no explicit conversion.

Almost every function in PIG that works with XYZ coordinate values passes in a PigCoordSystem object that specifies the CS in which the XYZ coordinates are defined. Without a CS definition, XYZ coordinates are meaningless. In a few cases, the CS is implicit (e.g. stored in the object), but there is always one available for any given XYZ (or unit vector) coordinate.

11.3.6 PigFileModel

These represent the input image files and provide high-level access to the metadata, and the image data, contained within them. Subclasses are used to handle the inevitable differences in metadata across missions, but are generally sparse (most of the work is done in the base class).

11.3.7 PigLabelModel

These classe handle writing output labels (metadata) for each type of image product. Subclasses exist per mission, where the metadata format deviates from the “standard” output labels (in general the subclasses are sparse; most of the work is done in the base class).

11.3.8 PigSufaceModel

These describe the ground and provide facilities to intersect view rays with it. Subclasses include flat planes, infinity (no surface), spheres (rarely used), and surfaces based on terrain models (meshes). Surface models are a critical component of mosaics; by ray tracing to a surface that closely approximates the actual ground, parallax is reduced (or eliminated if the match were perfect).

11.3.9 PigSolutionManager

Maintains lists of “solutions” for PigSolutionItem’s, which are: Coordinate System Definitions, Pointing Corrections, and Surface Model Parameters. PigRoverStateManager and PigPointingCorections are subclasses of the solution manager (surface model is directly managed by PigSolutionManager). A rather complex “priority” system exists for finding the item (“solution”) among the alternatives by using solution id’s, but this is rarely used.

The idea behind solutions is that the truth of some value, such as the rover location, is unknown and unknowable. All we have are estimates of the true value derived through various mechanisms. Each of these estimates is called a “solution”. The PLACES database [PLACES], which maintains rover locations, makes use of this concept extensively, but it rarely comes up in the context of the PIG library. Nevertheless, support is there if needed.

11.3.10 PigRoverStateManager

This class (and mission-specific subclasses) maintain a list of all known coordinate systems (as a list of PigCSDefinition objects). The list is created by mining the input data files for coordinate system labels, or by an RSF file (see Section 6.13).

Non-mobile landers often have a PigRoverStateManager simply because they have multiple coordinate systems (e.g. Site vs Local_Level vs Lander). It is somewhat of a degenerate case though, because there is only one instance of each. Nevertheless, it is easier from a code point of view to have this class. It is possible to use this mechanism to track things like a lander moving (e.g. being inadvertently pushed by an arm). While there is some rudimentary support for that for Phoenix and InSight, it is not tested.

11.3.11 PigPointingCorrections

Maintains a list of pointing corrections (bundle adjustment results), which can affect camera pointing, surface models, and coordinate systems. Bundle adjustment is performed via PigAdjustable, which is an interface implemented by several classes that allows opaque

parameters to be adjusted by the bundle adjustment routine. Parameters include things like mast pointing angles, arm camera locations, and surface model normal and offsets.

11.3.12 PigBrtCorrModel

Base class for implementing brightness corrections. These are similar to pointing corrections in that they're the result of a bundle adjustment process, but it matches radiometric seams rather than geometric ones.

The existing brightness correction subclasses do linear correction, meaning a multiplicative and additive factor is applied to the entire image (either in RGB space or HSI space). It is certainly possible to implement other brightness correction mechanisms, where the correction varies across the image, perhaps as a Gaussian spot or by paying attention to the flat field shape, or even by doing some kind of photometric correction. While these have been planned for some time, none have yet been implemented.

11.3.13 RadiometryModel

Should be called PigRadiometryModel; the misnaming is for historical reasons. Implements radiometric correction for each mission. Generally that involves exposure time and responsivity compensation and flat field removal, but can be much more extensive depending on the mission and camera (e.g. dark current, shutter smear, etc). RadiomtetryModel converts the data to physical units, typically W/nm/m^2/sr, but this can vary.

11.3.14 PigColorModel

Implements several forms of color correction and color space conversion, with the help of PigColorConverter. Color correction applies various forms of matrices to the data to correct for differences in responsivity among color bands, making colors appear correct. Color space conversion allows creation of other color spaces such as xyY, XYZ, and sRGB.

11.4 Support Subroutines

A number of subroutines, beginning with "`mars_*`", also exist, which are not technically part of the PIG library but are very useful for VISOR application code. They are multimission subroutines, not intended to be modified or subclassed per mission.

Notable among these is `mars_setup()`, which initializes the PIG library and is called by almost every program. It reads INP file lists, determines the mission and sets up the mission object, creates most of the PIG objects (camera, pointing, surface, file, radiometry, and brightness models, and coordinate systems), and applies pointing correction.

Other subroutines do the low-level work for mars_setup, read image files, and manage tiepoints, among other things.

11.5 Application Programs

Each application program is separate - there is no crosstalk in terms of build products between them. If multiple programs need to use the same routine, that's moved into a subroutine (usually a `mars_*` routine, sometimes even to the general VICAR P2 library).

Programs are typically not object-oriented in design (although a few are and there's no

restrictions against it). They have been written over a long period of time by many developers, so the style and architecture vary greatly.

12. EXTENDING VISOR

VISOR is designed to be extensible. This can range from simple bug fixes or feature additions in existing programs to new application programs to supporting entire new missions.

If you write something interesting, please consider contributing it back to MIPL. We would love to extend the suite, in ways large or small (even just adding a useful parameter to an existing program). Currently we need to vet any incoming code and include it in our internal CM system for the next open source delivery. Eventually we may adopt a true open source model where vetted developers can contribute code directly. This depends on need though - the more contributions we get, the more likely it is we'll set something like that up.

Please try to maintain backward compatibility as much as possible for any changes. We don't want to break existing code. That generally means adding parameters to enable new features rather than turning them on by default. However, this is not absolute, we do sometimes make incompatible changes. Talk to MIPL if you think you need to do this.

12.1 Code Conventions

Appendix A contains a list of coding and style conventions for VICAR generally and the Mars software specifically. Users contemplating adding their own code would be well advised to look at this, especially the Mars-specific ones.

12.2 Augmenting Existing Programs

Changes to applications programs are relatively easy. Copy everything from the program's source directory in `$V2TOP/mars/src/prog/` to your local directory. You can then build it via:

```
vimake programname
make -f programname.make
```

You can learn a lot about the architecture of each program from the PDF help for the program, in addition to code comments. Please make sure the PDF help is kept up to date as changes are made.

12.3 Augmenting Mars Subroutines

If you want to modify the mars_ subroutines, simply copy the directory for that subroutine (from `$V2TOP/mars/src/sub/`) and then build it as above;

```
vimake subname
make -f subname.make
```

Then in the program's imake file, add this (just for development, remove for delivery):

```
#define LIB_LOCAL
#define LOCAL_LIBRARY `ar t sublib.a`
```

and build the program as above, from the *same* working directory.

12.4 Augmenting the PIG library

Building the PIG library is a little more challenging, because changing an include file usually means recompiling everything that is compiled against that include file.

You can of course simply rebuild all of VICAR (see the VICAR Installation Guide [VICAR_IG]). However, the below contains some shortcuts that MIPL uses in development.

In order to create a local buildable PIG, first copy all of the PIG code over to a working directory:

```
cp $V2TOP/mars/src/sub/pig*/* .
cp $V2TOP/mars/src/sub/mars_* .
cp $V2TOP/mars/inc/Pig*.h .
cp $V2TOP/mars/inc/Radiometry*.h .
cp $V2TOP/mars/inc/mars_*.h .
```

Then run imake for all the subroutines:

```
foreach i (pig*.imake mars_*.imake)
   vimake ${i:r}
end
```

Modify the code as needed, then build it:

```
foreach i (pig*.make mars_*make)
   make -f $i
end
```

You may need to remove *.o files manually during the development process.

Note that that also includes the mars_* subroutines.

Programs can then be built against this using the same process as in mars subroutines above (modify the program's imake file and build in the same directory)

12.5 Writing New Programs

The easiest way to write a new program is to copy any of the existing programs as a starting point, rename all the files (and modify the imake, etc), and write your own processing algorithm. Find a source program that's as close as possible to your new program in terms of parameters and input/output files. The structure of the source program helps a lot in getting a new program up and running.

You may also want to choose one of the more recent programs as your model. Coding styles have changed over the years, for example SimpleImage is a relatively new addition and older programs don't use it (it is recommended now but not required). The PDF help generally has a History section that can be used to determine how old the program is.

12.6 Supporting New Missions

Supporting a new mission is a complex undertaking and you would be well advised to consult with MIPL personnel before proceeding. However, it is well within reach for most developers.

The first thing to ask is, does the generic mission suffice? This can be good for one-off types of data or very small missions.

Basically, to create a new mission, start with the existing mission that it is most similar to, copy and rename all the subclasses, and modify as necessary. There is some boilerplate per mission that can be accommodated in this way.

The library can be extended in other ways as well. New camera model types, for example. We have intended to write a photogrammetry-style camera model such as used by OpenCV, but have not gotten to it yet.

12.7 Future Enhancements

There is a long list of things that we at MIPL would like to do but have not yet had the opportunity to implement as of this writing (although we may do so at any time). Contributions for any of these are particularly appreciated, although contributions need not be related to any of these at all. This is far from a complete list!

- Non-linear brightness correction
- Implement photogrammetry-style camera model
- Easier rover pose adjustment (the RSF file mechanism is very cumbersome)
- Alternative correlation mechanisms, such as regularization or Fourier-space correlation
- Alternative/better tiepoint gathering mechanisms
- Improved multi-view (more than two images) mesh reconstruction (we have some of this)
- SLAM (this is in work at MIPL)
- Better error estimates for stereo ranging
- Support for other missions
- Easier-to-use manual tiepointing program
- Tools for computing/measuring strike and dip
- Actual photometric correction
- On-demand services via a web page (would be built on top of VISOR though)

13. ACRONYMS

ACI	Autofocusing Contextual Imager
ASCII	American Standard Code for Information Interchange
ASIFT	Affine SIFT
CADRE	Cooperative Autonomous Distributed Robotic Exploration
CAHV	Camera model – Center, Axis, Horizontal, Vertical
CAHVOR	CAHV plus Optical, Radial
CAHVORE	CAHVOR plus Entrance (pupil)
CCD	Charge-Coupled Device
CLPS	Commercial Lunar Payload Services
CM	Configuration Management
CS	Coordinate System
CTX	Context Camera
CWS	Collaborative Workflow System
DEM	Digital Elevation Map
DN	Data Number
DOF	Degrees Of Freedom
DRT	Dust Removal Tool
EDR	Experiment Data Record
EOL	End Of Dataset Label
FDR	Fundamental Data Record
FIDO	Field Integrated Design & Operations, a JPL rover testbed
FITS	Flexible Image Transport System
FF	File Finder
FSW	Flight Software
GIF	Graphics Interchange Format
GIS	Geographic Information System
HIRISE	High Resolution Imaging Experiment
HP3	Heat and Physical Properties Package
HSI	Hue, Saturation, Intensity
HTML	HyperText Markup Language
ICM	Image Correlation Map
ID	Identifier
IDC	Instrument Deployment Camera
IDX	Index

ILT	Inverse Lookup Table
InSight	Interior Exploration using Seismic Investigations, Geodesy and Heat Transport
I/O	Input/Output
iRGB	Instrument RGB
ISIS	Integrated Software for Imagers and Spectrometers
JPEG	Joint Photographic Experts Group
JPL	Jet Propulsion Laboratory
LIBS	Laser-Induced Breakdown Spectroscopy
LOC	Lines Of Code
MAHLI	MArs Hand Lens Imager
MARDI	MARs Descent Imager
MEDA	Mars Environmental Dynamics Analyzer
MER	Mars Exploration Rovers
MIPL	Multimission Instrument Processing Laboratory
MMM	Mastcam, MAHLI, MARDI
MPI	Message Passing INterface
MPF	Mars Pathfinder
MRO	Mars Reconnaissance Orbiter
MSL	Mars Science Laboratory
MSAM	Mastcam Stereo Analysis and Mosaics
M20, M2020	Mars 2020 project
NASA	National Aeronautics and Space Administration
NSYT	InSight abbreviation
OBJ	Wavefront mesh file format
ODL	Object Description Language
OMP	Open MuiltiProcessing
OpenCV	Open Source Computer Vision library
OPGS	Operational Product Generation Subsystem
ORR	Orthorectified
OS	Operating System
PDART	Planetary Data Archive, Restoration, and Tools
PDF	Parameter Definition File
PDF	Portable Document Format
PDS	Planetary Data System
PDS3	PDS Version 3

PDS4	PDS Version 4
PHX	Phoenix (Mars lander mission)
PIG	Planetary Image Geometry (library)
PIXL	Planetary Instrument for X-ray Lithochemistry
PLACES	Position Localization and Attitude Correction Estimate Storage
PNG	Portable Network Graphics
POV	Point Of View
PSPH	Planospheric
RADAR	Radio Detection And Ranging
RDR	Reduced Data Record
RGB	Red, Green, Blue
RMC	Rover Motion Counter
RMI	Remote Micro Imager
RSF	Rover State File
RTL	Run Time Library (part of VICAR)
RVF	Rover Vector File
SCLK	Spacecraft Clock
SHERLOC	Scanning Habitable Environments with Raman and Luminescence for Organics and Chemicals
SIFT	Scale Invariant Feature Transformation
SIS	Software Interface Specification
SOL	Mars Solar day
sRGB	Standard RGB
SSI	Surface Stereo Imager
TAE	Transportable Applications Environment
TAU	Atmospheric opacity measure (Greek letter, not an acronym)
TBD	To Be Determined/Defined
TIFF	Tagged Image File Format
URL	Uniform Resource Locator
UV	Ultraviolet
VAX/VMS	Virtual Address Extension/Virtual Memory System
VICAR	Video Image Communication and Retrieval
VISOR	VICAR In-Situ Operations for Robotics
Watson	Wide Angle Topographic Sensor for Operations and Engineering
XML	eXtensible Markup Language
XSL	Extensible Stylesheet Language

XVD	X-windows VICAR Display
XYZ	X, Y, Z Cartesian coordinates
ZCAM	Mastcam-Z abbreviation
2D	Two Dimensional
3D	Three Dimensional

14. REFERENCES

[Abarca2019]	Abarca, H., Deen R., Hollins G., Zamani P., Maki J., Tinio A., Pariser O., Ayoub F., Toole N., Algermissen S., Soliman T., Lu Y., Golombek M., Calef III F., Grimes K., De Cesare C., Sorice C., "Image and Data Processing for InSight Lander Operations and Science", Space Science Reviews, Pre-landing InSight Issue, 2019 https://link.springer.com/article/10.1007/s11214-019-0587-9
[Bell2012]	Bell III, J.F. et al., Mastcam-Z Multispectral Imaging on the Mars Science Laboratory Rover: Wavelength Coverage and Imaging Strategies at the Gale Crater Field Site, 43rd Lunar and Planetary Science Conference, 2012
[Bell2017]	Bell III, J.F., A. Godber, S. McNair, M.C. Caplinger, J.N. Maki, M.T. Lemmon, J. Van Beek, M.C. Malin, D. Wellington, K.M. Kinch, M.B. Madsen, C. Hardgrove, M.A. Ravine, E. Jensen, D. Harker,R.B. Anderson, K.E. Herkenhoff, R.V. Morris, E. Cisneros, and R.G. Deen, The Mars Science Laboratory Curiosity rover Mast Camera (Mastcam) instruments: Pre-flight and in-flight calibration, validation, and data archiving, Earth and Space Science, 4, doi:10.1002/2016EA000219, 2017
[Bell2020]	Bell, III, J.F., J.N. Maki, G.L. Mehall, M.A. Ravine, M.A. Caplinger, Z.J. Bailey, S. Brylow, J.A. Schaffner, K.M. Kinch, M.B. Madsen, A. Winhold, A. Hayes, P. Corlies, M. Barrington, R. Deen, E. Cisneros, E. Jensen, K. Paris, K. Crawford, C. Rojas, L. Mehall, J. Joseph, J.B. Proton, N. Cluff, B. Betts, E. Cloutis, A. Coates, A. Colaprete, K.S. Edgett, B.L. Ehlmann, S. Fagents, J. Grotzinger, C. Tate, C. Hardgrove, K. Herkenhoff, B. Horgan, R. Jaumann, J.R. Johnson, M. Lemmon, G. Paar, M. Caballo-Perucha, S. Gupta, C. Traxler, F. Preusker, M. Rice, M.S. Robinson, N. Schmitz, R. Sullivan, and M.J. Wolff, The Mars 2020 Rover Mast Camera Zoom (Mastcam-Z) Multispectral, Stereoscopic Imaging Investigation, Space Science Reviews, in press, http://dx.doi.org:10.1007/s11214-020-00755-x, 2020.
[Bunch2016]	Bunch, W. "VICAR Quick-Start Guide", Version 2.0. May 22 2016. https://www-mipl.jpl.nasa.gov/vicar_os/v2.0/vicar-docs/VICAR_guide_2.0.pdf
[Deen1992]	Deen, R., "The VICAR File Format", September 25, 1992, https://www-mipl.jpl.nasa.gov/external/VICAR_file_fmt.pdf
[Deen2003]	Deen, R., Issues with Linearization, JPL Docushare Collection 2700, File 75670, 2003.
[Deen2003b]	Deen, R.G., "Cost savings through multimission code reuse for Mars image products", in 5th Int'l Symposium on Reducing the Cost of Spacecraft Ground Systems and Operations (RCSGSO), Deep Space Comm. and Nav. Syst. Cent. of Excellence (DESCANSO), Pasadena, California (2003)
[Deen2005]	Deen, R.G. and J.J. Lorre (2005), Seeing in Three Dimensions: Correlation and Triangulation of Mars Exploration Rover Imagery, submitted to 2005 IEEE International Conf. on Systems, Man, and Cybernetics, Waikoloa, Hawaii.
[Deen2012]	Deen, R., "In-Situ Mosaic Production at JPL/MIPL", First Planetary Data Workshop, Flagstaff, Arizona, June 24 2012. https://ntrs.nasa.gov/citations/20120018099

[Deen2015]	Deen, R., et al. “Pointing Corrections for Mars Surface Mosaics,” Second Planetary Data Workshop, Flagstaff, Arizona, June 8-11 2015. https://docs.google.com/spreadsheets/d/13gPNXS-W-jxhaZg-_l5UbP4WWlCcjNJncOBlEa-dgz0/pubhtml#
[Deen2019]	Deen, R., et al, “Multimission Labels in PDS, Parts 1-3”, Fourth Planetary Data Workshop, Flagstaff, AZ, June 2019, Abstracts 7050-7052.
[Deen2023]	Deen, R., et al, “Labelocity: Multimission PDS4 Labels”, 6th Planetary Data Workshop, Flagstaff, AZ, June 2023, Abstract 7071.
[Di2004]	Di, K., and Li, R. (2004), CAHVOR camera model and its photogrammetric conversion for planetary applications, J. Geophys. Res., 109, E04004, doi:10.1029/2003JE002199. https://doi.org/10.1029/2003JE002199
[Duxbury1994]	Duxbury, E., Jensen, D., “VICAR User’s Guide”, October 14 1994 https://www-mipl.jpl.nasa.gov/PAG/public/vug/vugfinal.html
[Labelocity]	https://github.com/NASA-AMMOS/labelocity
[M2020_Cam_SIS]	Ruoff, N., Deen, R., Pariser, O., “Mars 2020 Software Interface Specification”, Version 3.1, February 22, 2022. https://pds-geosciences.wustl.edu/m2020/urn-nasa-pds-mars2020_mission/document_camera/Mars2020_Camera_SIS.pdf
[Maki2018]	Maki, J. N., Golombek M., Deen R., Abarca H., Sorice C., Goodsall T., Schwochert M., Lemmon M., Trebi-Ollennu A., Bannert W.B., “The Color Cameras on the InSight Lander”, Space Science Reviews, Pre-landing InSight Issue, 2018, https://link.springer.com/article/10.1007%2Fs11214-018-0536-z
[Maki2020]	Maki, J.N., et al., “The Mars 2020 Engineering Cameras and Microphone on the Perseverance Rover: A Next-Generation Imaging System for Mars Exploration”, Manuscript Draft for Space Science Reviews, 2020.
[Maki2020b]	Maki, J.N., et al, “ColorProperties at the Mars InSight Landing Site”, Earth and Space Science 8, e2020EA001336. https://doi.org/10.1029/2020EA001336
[Malvar2004]	H.S. Malvar, L. He, R. Cutler, “High-quality linear interpolation for demosaicing of Bayer-patterned color images”, IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 3 (2004), pp. iii–485. https://doi.org/10.1109/ICASSP.2004.1326587
[MER_Cam_SIS]	Chen, A., “Mars Exploration Rover Project Software Interface Specification”, Version 4.4, July 31, 2014 https://pds-imaging.jpl.nasa.gov/data/mer/opportunity/mer1do_0xxx/document/CAMSIS_latest.PDF
[MSL_Cam_SIS]	Alexander D., Deen, R. “Mars Science Laboratory Project Software Interface Specification”, Version 4.1, July 18, 2018. https://pds-imaging.jpl.nasa.gov/data/msl/MSLNAV_0XXX/DOCUMENT/MSL_CAMERA_SIS_latest.PDF

[NSYT_Cam_SIS]	Deen, R., Zamani, P., Abarca, H., Maki, J., "InSight Software Interface Specification", Version 3.3, June 26, 2019 https://pds-imaging.jpl.nasa.gov/data/nsyt/insight_cameras/document/insight_cameras_sis.pdf
[PDSConcept]	PDS4 Concepts, version 1.14.0, May 19, 2020, https://pds.nasa.gov/datastandards/documents/concepts/Concepts_1.14.0.pdf
[PDSIM]	PDS4 Information Model Specification, version 1.14.0, Mar 23, 2020, https://pds.nasa.gov/datastandards/documents/im/current/index_1E00.html
[PDSStd]	Planetary Data System Standards Reference, version 1.14.0, May 22, 2020, https://pds.nasa.gov/datastandards/documents/sr/current/StdRef_1.14.0.pdf
[PLACES]	Rover localization databases MSL: https://pds-imaging.jpl.nasa.gov/data/msl/MSLPLC_1XXX/ M20: https://pds-geosciences.wustl.edu/missions/mars2020/places.htm
[Trebi-Ollennu2012]	Trebi-Ollennu, A., *et al*, "Lunar Surface Operations Testbed (LSOT)", IEEE Aerospace Conference, Big Sky, MT, March 2012.
[VICAR_IG]	VICAR Installation Guide https://github.jpl.nasa.gov/MIPL/VICAR/blob/master/vos/docsource/vicar/VICAR_build_5.0.pdf
[VICAR_QS]	VICAR Quick-Start Guide https://github.com/NASA-AMMOS/VICAR/blob/master/vos/docsource/vicar/VICAR_guide_5.0.pdf

15. APPENDIX A: CODE CONVENTIONS AND GUIDELINES

The following are recommendations for coding style for VICAR generally, and for Mars in particular. They are adapted from an internal Wiki page and are presented here to help readers understand the code, as well as of course if new code is written for contribution back to MIPL. Any rule can be broken with sufficient justification, but you should at least think first to see if there's a reason that it *needs* to be broken.

15.1 General Coding Style

These are about how the code looks and its readability (and thus maintainability).

1. Neatness counts. It's all about readability. Make the code readable by making it look good. Line things up where appropriate, space things out, use blank lines, etc. Take the extra time to get the formatting of the source right.
2. Adapt your coding to the style of the module or subsystem you are in. Whatever is used for brace location, indents, spacing around parens, etc. should be what you use too - unless you reformat the entire module (or subsystem) (and this is not generally recommended). We should especially not mix styles within a single module, it makes the code quickly unreadable and unmaintainable.
3. Make indents consistent. All indents at the same level should line up. It is worth re-spacing an entire section of code if you have to add an "if" or something around it, in order for the indents to be proper.
4. If you use tab characters in the source code, they must be standard Unix 8-char tabs. No exceptions to this one. Because 8 chars is generally too much for indent levels, some combination of tabs and spaces will be needed - or have your editor convert everything to spaces.
5. Dead code should be removed, not commented out. Leaving old code there makes things unreadable, it's all too easy to miss the comment character and think the code is active. It destroys readability. We have a CM system if you need to go find history of the code. If you find you need to retain a section of dead code (it does happen occasionally), comment it specifically saying why it's there. Commented-out nontrivial debug prints that may be needed again are okay if they don't disrupt the flow of the code.
6. If dead code is more than one line, use #if 0 / #endif rather than /* */ around it, as the #if 0 makes it more obvious that the code is dead, and you don't risk an internal /* */ style comment prematurely terminating the dead code block. An alternative is to put // comments on every line (or better yet, //!!!!).
7. Code should be readable on an 80-column window. It is generally more readable to continue on the next line, spaced over appropriately to line things up. It is okay for quoted strings to go over, as it's more trouble than it's worth to do string continuations. But there is no excuse for comments exceeding 80 columns. Note: if you use vi, you can put the following in ~/.vimrc to get it to highlight anything over 80 characters:

```
highlight OverLength term=bold ctermbg=darkred ctermfg=white guibg=#592929
match OverLength /\%81v.\+/
```

8. The most important comments are those at the top of a subroutine, function, or class, that spell out the contract for the function. Think javadoc, even in other languages. Trivial subroutines don't necessarily need them but most should have them. Specifically, any rules about memory management (caller must free, etc.) need to be spelled out.
9. A separator comment (line of //////'s, //------, /********/ etc) extending most of the way

across the line is a good idea for functions so they can be more easily found.
10. Use comments as needed for code clarity, but don't clutter: "a++; // add one to a" is not a good comment. Comments should generally address the *why*, not so much the *what* (which is generally obvious from the code).
11. In C/C++, the function contract should go in the .h file, where users should expect to look if they want to use the function. Implementation details would go in the .c/.cc file above the function. It's okay to put the same comment in both places though.
12. An incorrect or obsolete comment is worse than no comment at all, as it can lead to misinterpretation of the code. Make sure all comments are kept up to date.

15.2 Bob Deen's Coding Style Suggestions

This is the style I use, and thus what's generally in Mars. Your mileage may vary, but note rule #2 above about adapting to the style of the module you are in.

1. I use 3 or 4 character indents (generally I use 4 but some older code is 3).
2. My preferred brace/indent style (yes functions are slightly different from if/while/for/etc):

```
int func(int a, char *b)
{
   if (a) {
      b();
   }
}
```

3. I will occasionally outdent (left margin, no indent at all) debug prints so they stand out for easy removal. Live code should never do this but sometimes commented out debug prints get delivered. Outdenting makes it obvious they're not part of the normal program flow.
4. I mark things that we should think about later, or bad/incomplete code, with four bangs: !!!! . Others use TODO or FIXME for the same thing.
5. I don't like space before the parens on a function call. I do generally space between arguments but not always; in particular that's a good place to squeeze if you're right at the 80-column limit.
6. I line up comments on code paragraphs at the same indent level as the code being commented upon. This helps with readability and allows the starts of functions to stand out better (since they're the only things with no indents, except for possibly debug prints).
7. I use a leading underscore for class member variables. This distinguishes them from local or global variables and parameters.
8. I am inconsistent about CamelCase vs. underscore_separator for names. Generally classes use UpperCamel while variables and functions start with a lower case letter, but otherwise there's not much consistency.
9. I tend to repeat the javadoc-style comments in the .h and the .cc files. At the very least, if modifying a routine where the comments are duplicated, make sure to update both.

15.3 VICAR Coding Guidelines

These apply to code content, not style, for VICAR programs and subroutines (including Mars).

1. All new VICAR code must be 64-bit clean. (most old code is at this point but not all; those should be fixed as they are found).
2. zvmessage() should be used for all stdout prints (directly or indirectly; some other subroutines call it). Don't use printf() directly. This is a holdover from the TAE days but it's still valid. It is okay to use printf() directly for debug prints but they have to all be

removed or commented out before delivery. We don't generally use stderr.

3. Use the RTL parameter passing mechanism (and PDF file) unless there's a really good reason not to. Pretty much the only exception to this is the X-windows programs.
4. If you can process the file a line at a time rather than reading the entire thing into memory, do so. It eliminates any size restrictions. This is not always possible of course.
5. Dynamically allocate memory as much as possible. Large arrays on the stack can run into resource limits.
6. Always use the RTL for accessing VICAR files - don't write your own I/O. This applies to Java code too. Non-image files can use normal file manipulation functions (open, fopen, etc).
7. In general, separate functions into independent executables. VICAR has a modular design, we try to avoid kitchen-sinking programs.
8. VICAR programs should general be agnostic in terms of OS version or platform. If you have to do something platform specific, you're probably doing something you shouldn't be doing in a VICAR program. If it is a must, turn it into a subroutine that can be replaced on other platforms if necessary.
9. VICAR programs must be agnostic to endian-ness. You might be running on a big endian or little endian platform. Generally the rule is "read anything, write native", and the RTL does this for you automatically for images. If a specific (non-image) file is defined with a given endian-ness, code can be written to that, but again realizing you could be running on either type of platform. There is a whole family of x/zvtrans() routines to help with conversion when reading binary files.
10. Try not to be too bleeding-edge in terms of compiler features. Remember we have a lot of ops systems on very old compilers.
11. Avoid using C/C++ exceptions, at least for now. There is no good reason for this other than we haven't used them yet and thus haven't investigated how they might affect the build.
12. C++ templates are okay, but don't go overboard with them. Historically there have been build issues with them, but these seem to generally be resolved in current compilers. If you can completely contain the template code in the .h file (a la SimpleImage), that's preferred.
13. If you mix Fortran and C, use the Fortran bridge mechanism in the RTL.
14. The PDF help should be complete and useful. Every parameter should have both level 1 and 2 help, and the main body should thoroughly describe the algorithms and how to use the program.
15. Try to avoid re-implementing the wheel; use existing subroutines if feasible. Look toward future users and extract reusable bits of code into subroutines.
16. If you change a subroutine API, you are responsible for making sure *all* callers of that subroutine are updated - even code that is not "yours".
17. Contact the MIPL systems and CM folks if you need to add or upgrade an external library ***WELL BEFORE*** you need it. If the first time CM or systems hears about an external library is on delivery day, your code will be rejected. Contact them as soon as you even *suspect* you may need a new library or upgrade.
18. If you need an external library upgraded, you are generally responsible for making sure all existing users of that library work with the new version - even code that is not "yours". Work with MIPL system engineering on this; they can sometimes assign others to help with that for code you're not familiar with.
19. If you need new functionality in vimake, contact CM and SE as soon as the need is identified. Do not wait until delivery day, your code will be rejected.
20. Do not use platform-specific #ifdef's. If you must, use the "feature defines" that are in xvmaininc.h or vmachdep.h. In other words, test only for a given feature, not for a

platform type, as the feature set supported by platforms can change over time (and this way there's only one place to change when that happens).

21. Programs should compile without warnings on all platforms. Add type casts or whatever else is needed to eliminate warnings as much as possible.
22. Code defensively. Programs should not crash due to bad inputs, or running out of memory. That said, realize that the programs are intended for sophisticated users, not the general public, so this principle need not be carried to the extreme. The more "core" the subroutine, the more important defensive coding is.

15.4 Mars/PIG subsystem Rules

These are code content rules specific to the Mars/PIG subsystem. There are "mars*" application programs, "mars_*" general subroutines, the PIG class library, and Mars in general. For things that require approval, that means from Bob, and you'd better have a really good justification!

1. Make the easy things easy, and the hard things possible.
2. Mars software must not interpret, generate, or otherwise parse filenames. All filenames (both input and output) are passed in by the caller (pipeline, user, etc). The *only* exceptions to this are reading config files, and edrgen (because the pipeline cannot know the filename ahead of time).
3. Mars* programs and mars_* subroutines ***must not have any*** mission specific code in them whatsoever. If you need something mission specific, abstract it to a general call and put it in PIG, with the mission-specific code in appropriate PIG subclasses. This rule is inviolate. If mission-specific code is needed (such as reachability code that calls the flight software), it is no longer a mars* program but rather a mission-specfic one (like msl* or nsyt*), and requires approval. But the remainder of the rules apply to these programs too as if they were mars* programs.
4. Mars* programs and mars_* subroutines should use abstraction layers if at all possible. For example, they should never access PigCAVHOR directly, but rather only though the PigCameraModel superclass. In very rare occasions diving into the camera model may be allowed, but only with approval.
5. Programs should use the provided infrastructure as much as possible, for example mars_setup(). Even if you need only one or two things mars_setup() does for you, using the common routine provides consistency across the suite.
6. All vectors, points, quaternions, or any other kind of 3-space coordinate or direction must have associated with it a PigCoordSystem object. Without a coordinate system, a 3-space point or direction is meaningless. This can be explicit or implicit (such as a general CS object used for all projection in a program) but it must be present.
7. Coordinates expressed in different CS's must be converted to a common CS before being used. This includes e.g. reading XYZ coordinates from a file; never assume the CS matches.
8. If you have to add something to PIG, do it. It sometimes takes longer to code this way but it is very much worth the investment.
9. Future-proof as much as possible: always think about how a new feature might work with the next mission or the next camera.
10. Parameters are a good thing. Don't hard code anything that someone might conceivably want to vary. Make sure the default is sensible (see Mars rule #1) but having the flexibility to do things is one of the keys to the power and flexibility of the mars* programs (the second half of #1).
11. Consider the impact on older legacy missions to changes. If it will change their data or processing in some way, talk to those missions to make sure they're okay with it. We

must not put ourselves in a position where older missions can't take the current code.

12. Historically we have avoided direct interfaces with databases, as they interfere with the ability to get the same result given the same input. This *may* change on 2020 but not without approval.
13. Make use of SimpleImage where practical. This is more a guideline than a rule, but it is a helpful class.