Running the Observables codes

- There are 2 observables codes: **HMI_observables** and **HMI_IQUV_averaging**.
  The executables are in:
  /home/jsoc/cvs/Development/JSOC/_linux_x86_64/proj/lev1.5_hmi/apps/ and
  /home/jsoc/cvs/Development/JSOC/_linux_avx/proj/lev1.5_hmi/apps/
  while the sources are in:
  /home/jsoc/cvs/Development/JSOC/ proj/lev1.5_hmi/apps/

  HMI_observables produces the front camera observables (hmi.V_45s, hmi.M_45s, hmi.Ic_45, hmi.Lw_45s, hmi.Ld_45s, hmi.V_720s, hmi.M_720s, etc... and their nrt equivalent), while **HMI_IQUV_averaging** produces the side camera ones (hmi.S_720s and hmi.S_720s_nrt).

  The basic way of calling **HMI_observables** from the directory */home/jsoc/cvs/Development/JSOC/*on n02 is:
  _linux_x86_64/proj/lev1.5_hmi/apps/HMI_observables begin="2015.01.10_00:00:00_TAI" end="2015.01.10_00:20:00_TAI" levin="lev1" levout="lev15" wavelength=3 quic klook=0 camid=1 cadence=45.0 lev1="hmi.lev1" smooth=1 rotational=0 linearity=1 -L

  On solar3 you will need to use the AVX binary rather than the x86_64 one.
  Warning: the code produces slightly different results between the AVX and x86_64 binaries.
  For instance, hmi.M_45s[2015.01.10_00:00:45_TAI] processed with the same code version but once with the AVX binary (on solar3) and once with the x86_64 binary (on n02) returns for DATAMEAN: 0.408243 and 0.408214; and for DATARMS: 48.986858 and 48.986938.
  camid=1 is for the front camera (*camid=0* for the side camera), *quicklook=0* is for definitive observables (=1 for NRT ones), *levin="lev1"* tells the code to start from lev1 data (rather than lev1d or lev1p), *levout="lev15"* tells the code to produce observables (rather than "lev1p" or "lev1d"), *cadence=45.0* is the cadence of the observables sequence, *smooth=1* tells the code to use look-up tables from the MDI-like algorithm that have been corrected for the interference fringes, *rotational=0* tells the code to use the usual pzt flat fields (rather than the rotational flat-fields), *lev="hmi.lev1"* tells the code which input series to use (hmi.lev1_nrt for NRT mode), and *linearity=1* tells the code to apply the non-linearity correction. *wavelength=3* tells the code which of the 6 HMI wavelengths is the reference one: the code needs to locate reference filtergrams to identify the observables sequence that was run and what its properties are. It also needs to know how to organize the lev1 records for the temporal interpolation. From the SDO launch onwards we have always used

wavelength=3, which ground tests have shown results in a lower noise level on the observables (therefore, the value should always be set at 3). Finally, -L forces the code to write a log (This option has sometimes caused issues in the past, resulting in the code hanging after the DRMS was updated). You will notice that the output series (e.g. hmi.V_45s) are not mentioned in the command line: they are hard-coded in the source. **Therefore, be extremely careful if you manually run the observables code, because you will overwrite the published observables records. I think that you should copy HMI_observables.c and HMI_IQUV_averaging.c in your own cvs tree, rename them, and change the output series inside the source (there are many output series, a lot of them are just for test purpose: more than 100, starting from line 1801 for HMI_observables.c, and 7 series from line 1588 in HMI_IQUV_averaging.c). Then, run your own copy of the code to avoid accidentally overwriting existing lev15 or lev1p records.** The .jsd files for all of these series are in /home/couvidat/cvs/JSOC/proj/myproj.

To produce the MDI-like algorithm observables from side camera data (i.e. from hmi.S_720s or hmi.S_720s_NRT), use: *levin=”lev1p” camid=0 cadence=720.0* for the usual mod C sequence. If a mod L sequence was used rather than a mod C, set *camid=3* (to tell the code that combining both cameras was required when producing the Stokes vectors). Theoretically, you can use **HMI_observables** and run it on side camera data to produce level 1.5 observables (i.e. to produce hmi.V_135s etc... for the usual mod C sequence): however, some things are missing in the code to do that (noone ever asked me for 135s cadence Dopplergrams...).

The basic way to run **HMI_IQUV_averaging** from */home/jsoc/cvs/Development/JSOC/* on solar3 is:
_linux_avx/proj/lev1.5_hmi/apps/HMI_IQUV_averaging begin="2015.04.05_19:48:00_TAI" end="2015.04.05_21:15:00_TAI" wavelength=3 camid=0 cadence=135.0 npol=6 size=36 lev1="hmi.lev1" quicklook=0 average=12 linearity=1 rotational=0

you need to specify the number *npol* of polarizations expected (6 for a mod C, 8 for a mod L because we declare two consecutive I+/-V polarizations in a 90s list on the front camera as different polarizations), the size of the framelist (*size*=36 for a mod C --- 6 polarizations at 6 wavelengths --- and 48 for a mod L --- 8 polarizations at 6 wavelengths ---). *average =12* is the number of minutes you want to average (you can use *96* min too, as there is a series called hmi.S_5760 that was used for test purpose). After you run **HMI_IQUV_averaging**, you also need to run two other programs (**hmi_segment_module** and **hmi_patch_module**), prior to running **HMI_observables** to produce the hmi.V_720s (etc...) observables. See with Jeneen or Xudong for details.

With **HMI_observables** you can also produce intermediate lev1d and lev1p records. Lev1d are level 1 images that have been gap filled (cosmic ray hits and bad pixels), de-rotated, un-distorted, interpolated in time at the same T_REC, re-registered for the same Sun center coordinates, etc. They are stored in DRMS series **hmi.HMISeriesLev1d45** (and 45Q for the NRT mode) for front camera data, and in **hmi.HMISeriesLev1d135** (and 135Q) for side camera data. These data take an enormous amount of memory, so they are not meant to be saved on a regular basis. They are computed at run-time by the code, but are discarded at the end of the run. For a given T_REC of the front camera observables, 12 lev1d records are produced. Lev1p records are lev1d records that have been calibrated in polarization (polcal() has been run on them), i.e. they are true I+V and I-V polarizations at the 6 wavelengths for the front camera data. They are saved in the **hmi.HMISeriesLev1pb45** (and 45Q for the NRT mode) series. A single lev1p record will contain all 12 data segments for the front camera.

For side camera data, lev1p records are the Stokes vector, but NOT averaged over 12 minutes (**HMI_observables** performs a temporal interpolation at T_REC, while **HMI_IQUV_averaging** performs a temporal averaging). So, with the standard mod C sequence, it's the Stokes vector at a 135s cadence. They are stored in **hmi.HMISeriesLev1pa135** (and 135Q). Again, lev1p and lev1d data were meant only to be tools to debug the observables codes, but because they give the user access to the Fe I line profiles at different polarizations, they can be used for scientific purpose (e.g. to study the line during flares). However, because they require so much disk space, they can only be produced on an on-demand basis. Also, **HMI_observables** requires all polarizations and wavelengths to be available in memory to compute the observables (unlike **HMI_IQUV_averaging**): this takes a lot of RAM at run-time. For side camera data (which have many more polarizations than front camera ones) **HMI_observables** with *levout="lev1p"* will simply not run on n02 or solar3 (well, it will run until a segmentation fault occurs because you exceed the amount of memory available) but will run fine on the cluster.

To run the observables codes on the cluster, here is what I do (taken from Keh-Cheng's email): first, make sure you have source ~kehcheng/cluster/sge2.csh in your ~/.cshrc file. Then, e.g. on $JSOCROOT on solar3, create a text file (say, *script.txt*) that contains the following (this example is if you want to produce lev1p data from front camera images for 20 minutes on January 10, 2015):

```
setenv OMP_NUM_THREADS 8

/home/jsoc/cvs/Development/JSOC/_linux_avx/proj/lev1.5_hmi/apps/HMI_observables
begin="2015.01.10_00:00:00_TAI" end="2015.01.10_00:20:00_TAI" levin="lev1" levout="lev1p" wavelength=3
quicklook=0 camid=1 cadence=45.0 lev1="hmi.lev1" smooth=1 rotational=0 linearity=1 -L
```

Make sure there is an empty line after the *-L*, otherwise (for whatever reasons unknown to me) the job will die on the cluster. Finally, just type the following in the Linux shell:
qsub -pe smp 8 script.txt

I put *8* after *smp*, but I'm not sure what the optimal number is. You can type *qstat -u* followed by your user name to see if your job is running.

- Some things to remember when you run **HMI_observables** and **HMI_IQUV_averaging**: if the *begin* and *end* times you provide are in a time interval where NO level 1 record is available (e.g. due to an instrument issue, or the sequencer was stopped for various reasons) then the observables code does not know whether this lack of lev1 is normal or is a lev1 processing issue that may be fixed later. Consequently, the code will just exit without producing any observables record, thus leaving a gap in the lev1.5 records. To fill this gap, if it turns out that no lev1 will ever be created for the T_REC in the time interval requested, then just run the observables code with a *begin* and *end* interval that includes at least 1 lev1 record. Then the code will produce empty lev1.5 records, with the QUALITY keyword set appropriately for the times where no lev1 is available.

- Another thing: if the observables sequence is changed (from the usual FTSID=1021 to, say, a mod L with FTSID=1022) you may need to change the command line parameters. For instance, if you go from a 1021 to a 1022, **HMI_IQUV_averaging** needs to have *npol* changed from 6 to 8, *size* changed from 36 to 48, and *cadence* from 135 to 90. For that specific example (1021 to 1022) there is no change required for **HMI_observables** when computing the front camera observables at a 45s cadence. However, for computing the 12-min average observables with **HMI_observables** (from hmi.S_720s) you will need to change the *camid* command line parameter from 0 to 3 (because CAMERA is set to 3 for hmi.S_720s(_nrt) when both cameras are combined, even though HCAMID remains set to the usual 2). Also, say you run **HMI_IQUV_averaging** with the command-line parameters for a FTS=1021: then the code will still process the T_REC during which the 1022 was taken, but will complain that the command-line parameters do not match the actual observables sequence, and consequently it will create empty observables records with the QUALITY keyword set accordingly. So you will have to reprocess these T_REC with the correct command-line parameters. Also, for the specific case of FTS=1022, the hmi.S_720s(_nrt) records produced by **HMI_IQUV_averaging** will have CAMERA=3, but if there was a problem and the code created an empty record then this record has CAMERA=1: so you might end up with several records for a given T_REC (CAMERA being a prime key). Finally, when running **HMI_IQUV_averaging** on a mod L, you might have to modify (increase) the

RSUNerr variable inside the source (and check into cvs and compile the new version): mod L requires combining both cameras for the side-camera observables, but the code might complain because CRPIX1, CRPIX2, and R_SUN are too different between front and side cameras. Increasing *RSUNerr* will increase the tolerance the code has on these parameters. If you change *RSUNerr*, please also modify the line *strcpy(COMMENT,"De-rotation: ON; Un-distortion: ON; Re-centering: ON; Re-sizing: OFF; correction for cosmic-ray hits; RSUNerr=1.0 pixels; dpath=");* to reflect the new value (this way the *COMMENT* keyword of the observables will have the correct tolerance).

- To produce the lev1p data with **HMI_observables** and from front camera data, here is how to call the observables code from */home/jsoc/cvs/Development/JSOC* on n02:
  _linux_x86_64/proj/lev1.5_hmi/apps/HMI_observables begin="2015.01.10_02:00:00_TAI" end="2015.01.10_02:2 0:00_TAI" levin="lev1" levout="lev1p" wavelength=3 quicklook=0 camid=1 cadence=45.0 lev1="hmi.lev1" smooth=1 rotational=0 linearity=1 - L
  The special option is *levout="lev1p"* rather than the usual *levout="lev15"*. -L may or may not be necessary (usually we add -L for definitive data, but discard it for NRT ones). Again, the front-camera lev1p data are saved in the series: hmi.HMISeriesLev1pb45 (definitive mode) and hmi.HMISeriesLev1pb45Q (NRT mode). If you are running the observables code on side camera data (*camid=0*), run it on the cluster, and the output is in hmi.HMISeriesLev1pa135(Q) for the standard observables sequence (FTSID=1021). Some people, might be asking for quite a lot of these lev1p: again, be careful because they really take a lot of space (and the definitive version is backed on tapes)...

- If you run the observables codes on older level 1 records, they are likely offline and have to be retrieved from tape, which can take time. Retrieve the records from tape prior to running the observables code, or the code might hang for hours. To retrieve level 1 from tapes, you can either ask Hao directly, or do a *show_info -p*

- Finally, if I manually run the codes on n02 or solar3 I usually type *setenv OMP_NUM_THREADS 8* before, since both code use OpenMP. I also type *unlimit stack*.

Calibration Codes To Run on a Regular Basis

To monitor the instrument and for trending purpose we take a lot of calibration sequences. Most are processed by Rock. Here I only mention the few I process:

- Every other week we take a detune sequence (currently, HFTSACID=3027) that has 60 images. To analyze this detune sequence, the first thing to do is to locate the FSN of the first record of the detune (e.g., type *show_info ds="hmi.lev1[2015.1.5_TAI/1d] [? HFTSACID = 3027 ?]" key=FSN,T_OBS*).
  Then, run the **phasemaps_test_voigt** code (see **hmiphasemaps_script** in */home/couvidat/cvs/JSOC* for examples). Type *unlimit stack* on n02 or solar3 prior to running the code, or it's going to result in a segmentation fault.
  Here is an example for the detune of March 25, 2015 (assuming I run the code on n02 from */home/couvidat/cvs/JSOC)*:
  ./_linux_x86_64/proj/lev1.5_hmi/apps/phasemaps_test_voigt input_series="hmi.lev1[2015.03.25/24h][86968038/60]" phasemap_series="hmi.phasemaps" hcamid=0 reduced=3 FSRNB=0.1689 FSRWB=0.33685 FSRE1=0.695 FSRE2=1.417 FSRE3=2.779 FSRE4=5.682 FSRE5=11.354 shift=0.0 center=2.7 thresh=750000. cal=2

  you only have to change the *T_REC* interval, the range of *FSN*s, *hcamid* (0 for side camera and 1 for front camera), and *cal* (=2 for the calibration used after 2014/01/15). Note, the keyword *hcamid* will change to 2 and 3 respectively in the output records for the phasemap. In these records, the keyword *hcamid* is defined as `` *0 = DARK SIDE CAMERA, 1 = DARK FRONT CAMERA, 2 = LIGHT SIDE CAMERA; 3 = LIGHT FRONT CAMERA".* You have to run **phasemaps_test_voigt** twice: once for the front camera, once for the side camera. During the run, the code will print on the screen the average OBS_VR during the sequence (e.g.: *VELOCITY = 22.286509*): it should always be close to 0 within a 100 m/s or so. If not, then it means that the detune was not taken at the usual time: ask Rock what the reason was. *reduced=3* is used to produce 128x128 maps (the current size). You can produce phase maps from hmi.lev1, or hmi.lev1_cal (the records stay online with hmi.lev1_cal, but they are a copy of hmi.lev1 and the series sometimes has some catching up to do with hmi.lev1).

  Here is an example of an output on the screen (I only show the last lines, because the code outputs many more things before that):

SPATIALLY AVERAGED PHASES (IN DEGREES)
NB Michelson: -119.416300
WB Michelson: 57.350394
Lyot E1: -135.894386

SPATIALLY AVERAGED WIDTH, DEPTH, AND CONTINUUM
WIDTH: 0.075712
DEPTH: 569.935947
CONTINUUM: 996.775439

COTUNE TABLE

E1	WB	POL	NB
7	50	0	80
10	44	0	68
13	38	0	56
16	32	0	104
19	26	0	92
22	20	0	80
25	74	0	68
28	68	0	56
31	62	0	104
34	56	0	92
37	50	0	80
40	44	0	68
43	38	0	56
46	32	0	104
49	26	0	92
52	20	0	80
55	74	0	68
58	68	0	56
61	62	0	104
64	56	0	92

The spatially averaged phases are in degrees. The width, depth, and continuum are the values fitted for a Voigt model of the Fe I line (I use an analytical expression for the Voigt profile, which is an approximation). I usually note all of these values somewhere (in a file or a note book), so that I can see how they change with time. The code also outputs a table showing the 27 intensities --- 30 for a given camera minus 3 dark frames ---- of the detune (left column) and the predicted values (right column) resulting from the fit, under the label *SPATIALLY AVERAGED INTENSITIES*: this allows you to derive the goodness of fit.
Under the label *COTUNE TABLE*, the code provides the optimal cotune table corresponding to the phases computed.

When the code runs you'll see plenty of warnings for: *"Unable to open the series hmi.cosmic_rays"*; it's normal, just ignore these warnings. I also run **phasemaps_test_voigt** on the definitive lev1 records rather than lev1_nrt, so you have to wait a few days after the detune is taken to process it.

The output cotune table includes all of the 20 possible tuning positions (but HMI only uses 6). The optimal tuning (the one that centers the HMI optical filter on the 6173.3433 A wavelength) corresponds to position number 11 (assuming the first line is index 1). For the detune of March 25, 2015, the best tuning is 37 50 0 80 where these numbers corresponds to steps in the hollow-core motors of the tunable

elements. The tuning polarizer is not used, so its position is always 0. For the position 37 50 0 80, 37 is for the Lyot element E1, 50 is for the WB Michelson, and 80 is for the NB Michelson. Due to the drift of the two Michelsons, it is necessary to re-tune HMI on a regular basis (the last returning was performed in April 2015, the next retuning will probably not be necessary until the end of 2016 or the beginning of 2017) .
The optimal tuning (in HCM steps) is computed from the phases (in degrees) this way:
for NB = -NBphase/6.0+60
for WB = -WBphase/6.0+60
for E1 = E1phase/6.0+60 (note the lack of a minus sign!)
since the steps must be integers, you have to round these numbers.

The current (as of May 2015) tuning positions are 37 50 0 80 (you can find all of the tuning positions used in the past by typing: *show_info ds="hmi.lookup_corrected_expanded[]" key=T_REC,FSN_REC,HWL1POS,HWL2POS,HWL3POS,HWL4POS*).
When a re-tuning is needed, you have to produce a new cotune table and give it to Rock who will update some files.
The IDL code **cotunetable.pro** (in */home/couvidat/cvs/JSOC*) will produce such a table. You have to first edit the source to enter the correct phases of the tunable elements (in degrees). A way to see when a retuning will be needed is to plot the polynomial coefficients of the series hmi.coefficients. COEFF0 keeps increasing with time, until a retuning is operated (the re-tunings are easy to spot on the plot of COEFF0 with time): you can extrapolate COEFF0 at future times to estimate when the retuning will have to be performed. Another way is to extrapolate the phases of the tunable elements at future times and to derive the optimal cotuning from these phases: you can estimate at which date the current cotuning ceases to be optimal.

- Another useful code is **lookup.c** in */home/couvidat/cvs/JSOC/ proj/lev1.5_hmi/apps/lookup.c* . This code produces the look-up tables for the MDI like algorithm, from the tuning element phase maps.
The script **lookup_script** in */home/couvidat/cvs/JSOC/* provides examples of how to run this code. Currently, we produce look-up tables that are corrected for interference fringes, and that are expanded to 90 pixels off the solar limb (rather than the mere 50 pixels used at the start of the mission). So you primarily want to populate the hmi.lookup_corrected_expanded DRMS series.
For example, to produce look-up tables valid after the retuning of April 8, 2015 (assuming you are on n02 and in the directory */home/couvidat/cvs/JSOC/*) type:
./_linux_x86_64/proj/lev1.5_hmi/apps/lookup phasemap="hmi.phasemaps_corrected[86968067]" lookup="hmi.lookup_corrected_expanded" HCME1=37 HCMWB=50

HCMPOL=0 HCMNB=80 NUM=6 hcamid=1 cal=2 &

Don't forget to type *unlimit stack* prior to running the code, or you'll get a segmentation fault. **Be careful not to overwrite existing look-up tables when you run this code!** *HCME1*, *HCMWB*, *HCMPOL*, and *HCMNB* are the hollow core motor positions to cotune HMI. *NUM* is the number of wavelengths you want: HMI takes 6 in normal operations, but we also have test sequences with 8 and 10 wavelengths, so it's useful to produce 6 look-up tables (front and side cameras, for 6, 8, and 10 wavelengths). *hcamid=1* is for the front camera, =0 for the side camera.

- In /home/couvidat/cvs/JSOC there are a few IDL codes (.pro) of interest: **analyze_Iripple.pro**. This code follows the temporal evolution of the tunable element I-ripples. You must first run the scripts in **hmiphsemaps_script** (starting from the line: *./_linux_x86_64/proj/lev1.5_hmi/apps/phasemaps_test_voigt_Iripple2 input_series="hmi.lev1_cal[2010.05.06/24h][4875773/60]" hcamid=0 reduced=3 FSRNB=0.1689 FSRWB=0.33685 FSRE1=0.695 FSRE2=1.417 FSRE3=2.779 FSRE4=5.682 FSRE5=11.354 shift=0.0 center=2.7 thresh=750000. cal=1 > Iripple_results.txt*). *phasemaps_test_voigt_Iripple2* does not compute phase maps: it performs a spatial averaging of the detune-sequence intensities across the CCD and then fits the phases of the tunable elements, the Fe I line characteristics, and the I-ripple characteristics. It outputs the result on the screen, e.g.:
4875778 -131.915283 13.736845 -139.697708 0.067878 0.923974 1.365605 -0.006369 0.049829 -0.114849 -0.006080 -0.061444 0.043307 11.580346
where the first number is the *FSN* of the first image used by the code, the 3 following numbers are average phases (in degrees) of the NB Michelson, WB Michelson, and E1 (respectively), the following 3 numbers are the width, depth, and continuum intensity of the Fe I line (fitted by a Voigt profile), the following 6 numbers are the I-ripple parameters, and the last number is a measure of the goodness of fit (the square root of the total residual error). The I-ripple parameters are 2 for E1, 2 for WB, and 2 for NB. With the example provided here, -0.006369 is the I-ripple coefficient for the cosine of E1, while 0.049829 is the coefficient for its sine. The I-ripples for each tunable elements are modeled as: $I=I*(1+[A*\cos(phase)+B*\sin(phase)]^2)$ where phase is a combination of the tuning phase plus the "initial" phase of the element, and *I* is the intensity transmitted by the element.

  In hmiphasemaps_script, you will notice that I ran **phasemaps_test_voigt_Iripple2** on all of the detunes (up to a certain date): the goal is to build the *Iripple_results.txt* file. This file is then used to follow the temporal evolution of the I-ripple. To create a nice plot of the temporal evolution I run **analyze_Iripple.pro**. Instructions to run this code are in the header of the source.

- Another code (in IDL): **sun_lin.pro**. This code processes the calibration sequences to measure the non-linearity in the CCD gains. It's Jesper's code that I modified slightly to return a nice figure and to run on lev0 images (note that normally you run it on lev1 data, but no lev1 is available for some sequences taken during commissioning. In that case you need to use hmi_ground.lev0). We take non-linearity calibration sequences occasionally (maybe once a year. See with Rock). The FTSID of these sequences has changed with time (5003, 5004, 3004, 3005, 3034, etc...).

- **bad_datamin.pro** computes the number of daily images with DATAMIN<0 from hmi.lev0a and draws a plot as a function of time. Rock pointed out that it's now part of the health and monitoring website, so this IDL code is not needed anymore.

- **HMI_CCD_temperature.pro** computes the temperature dependence of CCD gains.

Procedure for an HMI retuning

prior to the retuning:

– Compute the phase maps for the tunable elements by running the **phasemaps_test_voigt** program (see examples in hmiphasemaps_script in */home/couvidat/cvs/JSOC*, and a description in a previous section of this document). I use the most recent phase maps computed from a detune sequence to produce the new look-up tables (for the MDI-like algorithm) that will be required by the new instrument co-tuning. Since detunes are taken every other week, the most recent set of phase maps might be a couple of weeks old: this is not an issue, as the phases change only slowly with time. The look-up tables that will be used by the new cotuning need to be ready prior to the date and time of this retuning.
Again, here is a standard way of running the phase map code:
/home/couvidat/cvs/JSOC/_linux_x86_64/proj/lev1.5_hmi/apps/
phasemaps_test_voigt input_series="hmi.lev1[2015.03.25/24h][86968038/60]"
phasemap_series="hmi.phasemaps" hcamid=0 reduced=3 FSRNB=0.1689
FSRWB=0.33685 FSRE1=0.695 FSRE2=1.417 FSRE3=2.779 FSRE4=5.682
FSRE5=11.354 shift=0.0 center=2.7 thresh=750000. cal=2
where *cal=2* is used to indicate which calibration to use (we have changed the calibration twice as of May 1, 2015), *reduced=3* is used to indicate you want 128x128 phase maps, *hcamid=0* refers to the side camera (=1 for the front), and the FSN in hmi.lev1 refers to the beginning of a detune sequence (HFTSACID=3027). After producing the phase maps (in hmi.phasemaps) for both front and side cameras,

you should look at them (with **fitsio_read_image** in idl or with ds9 for instance) to make sure they look smooth and that there are no bad pixels. If the phase maps have an issue, use the most recent set that is fine.

- Update the interference fringe correction with the IDL **cal_fringes.pro** code in */home/couvidat/cvs/JSOC* (this code was developed by Jesper). The fringe correction uses as many phase maps from detune sequences as possible, and is usually updated only prior to a retuning of HMI. For the last retuning (April 8, 2015), I used 132 detune sequences (from May 1, 2010 onwards) to compute the correction. All of the phase maps need to have been computed with the same calibration (the *cal* command-line option of **phasemaps_test_voigt**). Because the phase maps in hmi.phasemaps have been produced with different *cal* settings (since we changed the calibration over time and phase maps for hmi.phasemaps should be computed with the current calibration at the time of the detune), I created other phase maps series whose only purpose is to be used to compute the fringe correction: hmi.phasemaps_cal11, hmi.phasemaps_cal12, and hmi.phasemaps_cal13. Only the latter will be of use for future retuning.
  To run **cal_fringes**, you first need to create 2 text files listing all of the FSNs and paths to the phase maps obtained with the front or side cameras and that will be used to compute the fringe correction. An example of such a file is *cal_fringes13_updated_side.txt* in the same directory as the code. It has lines like:

  4649722 /SUM10/D692211818/D516927333/S00000
  4875802 /SUM10/D692211818/D515284944/S00000
  5198362 /SUM10/D692211818/D515287278/S00000

  The first column is the *FSN_REC* of the phase maps, and the second column is their path. The actual path to the phase maps will change with time because their online retention time is only 90 days (but they are archived). To produce *cal_fringes13_updated_side.txt* I typed:

  show_info ds="hmi.phasemaps_cal13[][][2][128]" key=FSN_REC -p -q > cal_fringes13_updated_side.txt
  show_info ds="hmi.phasemaps[][2013.12.19_TAI-2015.4.1][2][128]" key=FSN_REC -p -q >> cal_fringes13_updated_side.txt

  You can see that I combined phase maps from hmi.phasemaps_cal13 with phase maps from hmi.phasemaps that were computed with *cal=2* (which corresponds to the calibration number 13... sorry for the weird numbering scheme: the calibration used after the 2nd change since the launch of SDO ---- *cal=2* --- is the 13th calibration I had tested since I started working on HMI).
  For a future retuning, you should run the same thing, except change the 2015.4.1 date

into the date of the most recent detune sequence (the last *T_REC* in the hmi.phasemaps series).
Again, the fringe correction is computed separately for front and side cameras, so you need so create two separate files (with names like *cal_fringesXXX.txt* and *cal_fringesXXX_side.txt* where XXX is whatever string you want). You also need to update the **cal_fringes.pro** file so that it will use these new .txt files (it's on the line: *filename='cal_fringes13_updated_side.txt'*). After it runs, the code returns three data arrays (newphases, newphases1, and newphases2) that contain the corrected phase maps for all of the detunes in your input .txt file.
You only need to save 1 corrected set of phase maps in a fits file (use the newphases1 array for the corrected phase maps): you will need to locate the position in the newphases1 array that corresponds to the most recent phase maps you computed (the array newphases1 has 4 indexes, the last one corresponds to the detune sequence index), and then run the following in your IDL session:
fits_write,'phases_86968067_side.fits',float(newphases1[,*,*,132])*
in this example 86968067 is the *FSN_REC* of the latest phase maps, corresponding to the index 132 in the newphases1 array, and for the side camera. newphases1 is preferred because it includes correction for the small-scale and large-scale fringes, while newphases only includes the large-scale ones, and newphases2 is doing another correction that is not as good.

– Once the corrected phase maps (2 sets, one for front camera and one for side camera) have been saved as fits files on your local directory, they need to be ingested into the hmi.phasemaps_corrected series using the **ingest_corrected_phasemaps.c** program (in */home/couvidat/cvs/JSOC/proj/lev1.5_hmi/apps/*). Run it for both the front and side cameras. This code is not user friendly: you have to edit the source to provide the correct *FSN_REC* of the phase maps you are trying to ingest. Here is what the relevant part of the source looks like:

```
char *inRecQuery ="hmi.phasemaps[86968067][][3][128]";
//char *inRecQuery ="hmi.phasemaps_cal13[65714753][][2][128]";
//char *inRecQuery ="hmi.phasemaps[51564722][][3][128]";
char *dsout      ="hmi.phasemaps_corrected";
//char *inFilename ="phases_65714753_side.fits";
//char *inFilename ="phases_51564722.fits";
char *inFilename ="phases_86968067.fits";
```

in *inRecQuery*, you provide the info about where the original phase maps come from (so their *FSN_REC* and *HCAMID*). The code will copy the keywords of this record into the hmi.phasemaps_corrected record. In *inFilename* you provide the name of the fits file where the corrected phase maps are stored. A note about the camera convention: hmi.phasemaps is using *HCAMID*, not *CAMERA*, as a prime key. With *HCAMID*, the front camera is 3 and the side one is 2.

- Once the corrected phase maps are available, you need to use them to produce new look-up tables for the MDI-like algorithms, with the new co-tune setting.
  Use **lookup.c** (in */home/couvidat/cvs/JSOC/proj/lev1.5_hmi/apps/*) for that purpose. The script lookup_script in */home/couvidat/cvs/JSOC/* shows how to run the code. For the retuning of April 2015, here is what I did (on n02 and from */home/couvidat/cvs/JSOC*):

```
./_linux_x86_64/proj/lev1.5_hmi/apps/lookup phasemap="hmi.phasemaps_corrected[86968067]"
lookup="hmi.lookup_corrected_expanded" HCME1=37 HCMWB=50 HCMPOL=0 HCMNB=80 NUM=6 hcamid=1
cal=2 &
./_linux_x86_64/proj/lev1.5_hmi/apps/lookup phasemap="hmi.phasemaps_corrected[86968067]"
lookup="hmi.lookup_corrected_expanded" HCME1=37 HCMWB=50 HCMPOL=0 HCMNB=80 NUM=6 hcamid=0
cal=2 &
./_linux_x86_64/proj/lev1.5_hmi/apps/lookup phasemap="hmi.phasemaps_corrected[86968067]"
lookup="hmi.lookup_corrected_expanded" HCME1=37 HCMWB=50 HCMPOL=0 HCMNB=80 NUM=8 hcamid=1
cal=2 &
./_linux_x86_64/proj/lev1.5_hmi/apps/lookup phasemap="hmi.phasemaps_corrected[86968067]"
lookup="hmi.lookup_corrected_expanded" HCME1=37 HCMWB=50 HCMPOL=0 HCMNB=80 NUM=8 hcamid=0
cal=2 &
./_linux_x86_64/proj/lev1.5_hmi/apps/lookup phasemap="hmi.phasemaps_corrected[86968067]"
lookup="hmi.lookup_corrected_expanded" HCME1=37 HCMWB=50 HCMPOL=0 HCMNB=80 NUM=10 hcamid=1
cal=2 &
./_linux_x86_64/proj/lev1.5_hmi/apps/lookup phasemap="hmi.phasemaps_corrected[86968067]"
lookup="hmi.lookup_corrected_expanded" HCME1=37 HCMWB=50 HCMPOL=0 HCMNB=80 NUM=10 hcamid=0
cal=2 &
```

  So I called the code 6 times, to produce look-up tables for the front and side cameras, and for the cases where HMI uses 6, 8, or 10 wavelengths (the cases 8 and 10 wavelengths are for special observables sequences that we may run occasionally). I actually ran the code more than that because here I only showed how to populate the hmi.lookup_corrected_expanded series, which is the one currently used, but there are other series you may want to populate (hmi.lookup, hmi.lookup_corrected, and hmi.lookup_expanded). Running the code once on n02 takes half an hour or so (with 8 OpenMP threads), so running it 6 times will take roughly 3 hours. The new cotuning of the instrument (in terms of hollow core motor positions so that the instrument is centered on the Fe I line at 6173.3433 A) is provided on the command line by *HCME1*, *HCMWB*, *HCMPOL*, and *HCMNB*: you need to use the correct values. *NUM* is the number of wavelengths you want, and *cal* is the calibration used (should be 2).

- Finally, you need to update the std_flight.w file if necessary (to add the new tuning positions). The file is in */home/couvidat/cvs/JSOC/proj/tables/hmi_mech/std_flight.w* and should be checked into cvs. Make sure the file in the development tree (*/home/jsoc/cvs/Development/JSOC/proj/tables/hmi_mech*) has been updated (I usually ask Art to do it). Rock is the one who creates std_flight.w. You have to be careful because there are several versions available (one with and one without comments).

After the retuning occurs:

- Update the file filePhaseMaps.txt (in */home/couvidat/cvs/JSOC/proj/lev1.5_hmi/apps/*), check it into cvs, and make sure that the file in */home/jsoc/cvs/Development/JSOC/proj/lev1.5_hmi/apps* has been updated (ask Art). This file tells VFISV which phase maps to use for *T_REC* larger than a certain value. Therefore, it needs to be updated for VFISV to process the hmi.S_720s(_nrt) records obtained after retuning. The first column in this file is the *T_REC* (as a double) at which the retuning occurred, and the right column is the *FSN_REC* of the look-up table to use for *T_REC* larger than the retuning *T_REC*. You need to know the precise time at which retuning occurs to update this file, so it is usually done shortly after retuning. However, if the retuning is done during a calibration maneuver, no observable sequences are taken for hours, so it does not matter if the *T_REC* you use is not exact: in that case, you can update the file prior to retuning, once you have a good enough estimate of the time at which retuning will occur (ask Rock). The *T_REC* in filePhaseMaps.txt are listed as doubles, not as timestamps. To convert timestamps into double I use **time_convert.c** (in */home/couvidat/cvs/JSOC/proj/lev1.5_hmi/apps/*), but I believe there is an actual DRMS function that does that. You have to edit **time_convert.c** to provide the *FSN* of a lev1 record (the record at the time of retuning for instance), and the code returns the *T_REC* as a double corresponding to this *FSN*.

- Create some "fake" hmi.coefficients entries: the polynomial coefficients records have a keyword named *CAL_FSN* that is used by the observables code **hmi_observables** to determine what look-up table was used to produce the Dopplergrams on which the coefficients are based (*CAL_FSN* is the *FSN_REC* of the corresponding look-up table). Therefore, when we retune HMI and produce new look-up tables, the observables code will not use polynomial coefficients computed with another look-up table. Also, there will be a gap in the observables if no precaution is taken: you need to create entries in hmi.coefficients just prior to and just after the retuning. Of course, you won't have the necessary 24h of data to process these records in the usual way: hence the need to create fake entries.
  I usually just copy (with *set_keys -c*) the last record in hmi.coefficients obtained with 24h of data to a T_REC that is 1s before the retuning, and then I copy the first hmi.coefficients record obtained with 24h of data after retuning to a T_REC that is 1s after retuning (you can do a *show_info ds="hmi.coefficients[]"* for T_REC around the retuning of April 8, 2015 to see what the records look like).
  You compute the hmi.coefficients records after retuning once 24h worth of hmi.V_45s_nrt is available by running the **correction_velocities** code manually (if you wait for the scripts to run automatically, it takes about 3 days after retuning). Example of how to run correction_velocities:

/home/jsoc/cvs/Development/JSOC/_linux_x86_64/proj/lev1.5_hmi/apps/correction_velocities begin=2010.9.30_6:45_TAI end=2010.10.1_6:45_TAI levin=hmi.V_45s_nrt levout=hmi.coefficients

You might have to force computation if the number of hmi.V_45s_nrt records available is less than the 1920 per day expected: if so, add the *forced=1 mindata=1* options on the command line. The polynomial coefficients only change slowly from one day to the next, so if you observe sudden jumps in these coefficients, something went wrong (maybe some Dopplergrams are bad, or maybe there was not enough Dopplergrams to produce reliable coefficients): in that case you probably want to delete the faulty records and copy the closest one to the faulty *T_REC*.

Miscellaneous Codes

- A code that might be useful from time to time is **undistort_lev1.c** (in *proj/lev1.5_hmi/apps*): as the name implies, this code takes level 1 records as an input and gapfills and undistorts it (it was used to process the Venus transit data for instance). This code also run the limb-finder after having undistorted the images, rather than to rely on the ad-hoc values for the solar radius and solar center that are returned by the undistortion routine (this is unlike what is done in the observables codes). You call it this way (from */home/couvidat/cvs/JSOC/* on n02):
_linux_x86_64/proj/lev1.5_hmi/apps/undistort_lev1 begin="2015.05.01_00:00:00_TAI" end="2015.05.01_00:10:00_TAI" in="hmi.lev1" out="su_couvidat.lev1"

  The ouput series is on the command line: make sure you don't overwrite a lev1 record from hmi.lev1!

- Another potentially useful code: **limbfit_sc** (also in */home/couvidat/cvs/JSOC/proj/lev1.5_hmi/apps*): this is a standalone program based on Richard's limb finder. It will run the limb finder on any level 1 record, then will perform the formation height correction, and return the following values:
FSN, X0_LF, Y0_LF, RSUN_LF, CRPIX1, CRPIX2, and *R_SUN*
You run it the following way (on solar3 and in */home/couvidat/cvs/JSOC/*):
_linux_avx/proj/lev1.5_hmi/apps/limbfit_sc dsin="hmi.lev1[2015.5.1_TAI/1m]"

  Even though you can run the limb finder on different series, it crashes if the image on which you run it is not a hmi.lev1 record and has NaNs. I had to replace NaNs by zeroes in some images (like the lev1 with PSF removed by Aimee) to be able to have the limb finder runs smoothly on them.

Ingesting PSF Corrected Level 1 Data

Right now, Aimee's code to produce PSF corrected lev1 records is not running in the DRMS, so you need to first fetch the lev1 data segments (with *exportdata* for instance), then run her code, and then you need to ingest the output lev1 images into a DRMS series. To perform the latter, I use **ingest_Aimee.c** (in */home/couvidat/cvs/JSOC/proj/lev1.5_hmi/apps/*). From */home/couvidat/cvs/JSOC* on n02 you run it like this:
_linux_x86_64/proj/lev1.5_hmi/apps/ingest_Aimee fits="/tmp20/norton/scat/deconvolved/image_lev1_01.fits" out="su_couvidat.lev1" inRec="hmi.lev1[2012.06.06_02:27:00.86_UTC]"

fits is the path to the fits file produced by Aimee, *out* is the series where you want to ingest the lev1 record (make sure not to overwrite production records in hmi.lev1), and *inRec* is the level1 records that Aimee deconvolved (it is used to copy its keywords into the output series).
The code runs the limbfinder on the deconvolved level1 image, to update the *X0_LF, Y0_LF, RSUN_LF, CRPIX1, CRPIX2, CDELT1,* and *R_SUN* keywords.