AO Loop Control Software

Overview

Linux-based

Open-source, no closed library

C code (~100k lines) + high-level scripts (baseline control interface using bash scripts provided)

Uses libraries: CUDA & MAGMA (GPU computing, optional), FFTW, FITSIO, GNU scientific library, readline

Source code + example simulated AO system: https://github.com/oguyon/AdaptiveOpticsControl

Hardware

Hardware Requirements / compatibility:

RTS runs on a single multi-core computer. Minimum ~15 cores system, 128GB ram (heavy use of shared memory and shielded processes running on single core)

CPU only or CPU+GPU computing engine. Requires GPU(s) for high speed / high actuator count. Supports NVIDIA hardware (CUDA lib).

Can span multiple computers (for example, camera or DM driven by computer other than main RTS). Software uses and configures fast private low-latency TCP link (eg. 10GbE or 40GbE fibers) for transfers.

Interfaces to hardware through shared memory structure. Hardware already coupled with RTS: BMC deformable mirror, Ocam2k camera, SAPHIRA camera (with UH readout electronics), OwlCam InGaAs Raptor Photonics camera, Andor sCMOS.

Capabilities

Speed

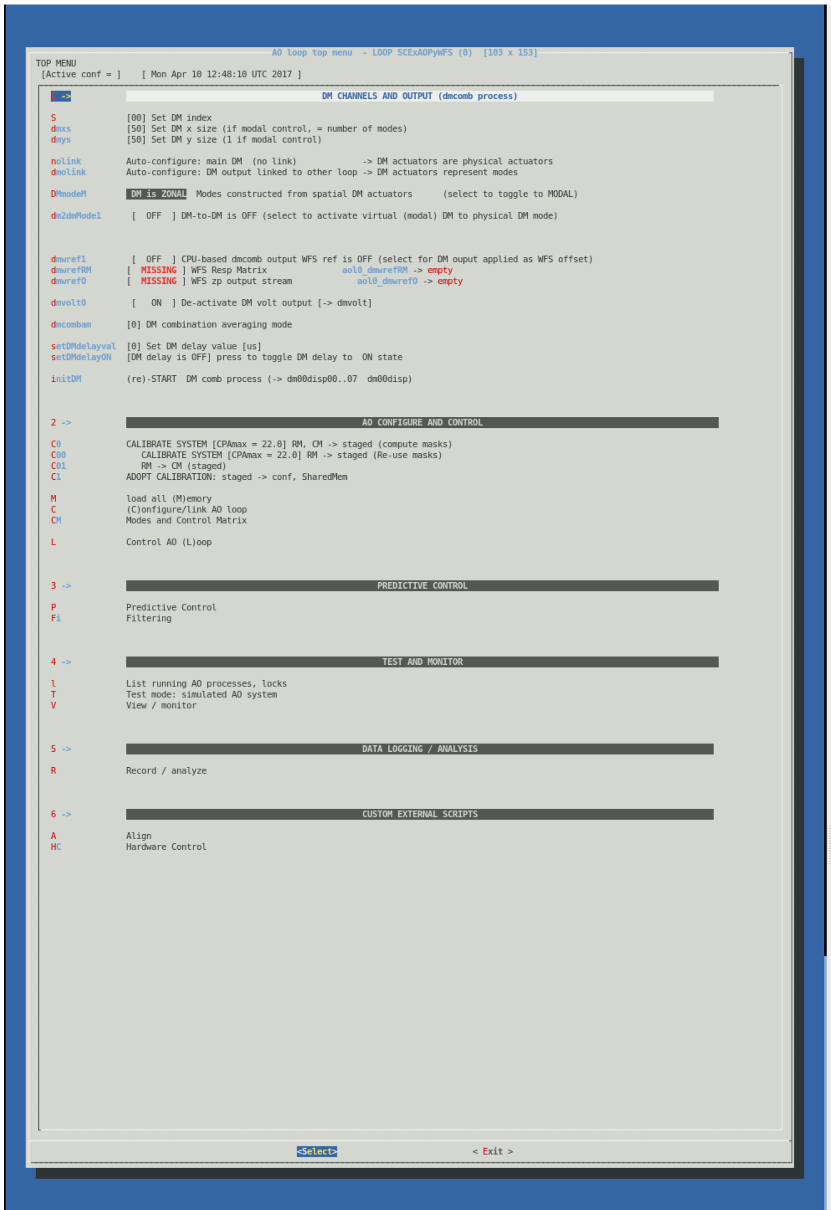
Largely limited by hardware. Fully system timing stable at 10us level, and RTS latency due to IPC, TCP transfers between computers, and GPU transfers is <100us total → can drive ~10 kHz loop on multi-computer system, and ~20 kHz loop on single computer.
SCExAO implementation drives 2000-actuator, 14,400-sensor loop at 3.5kHz, limited by camera readout speed.

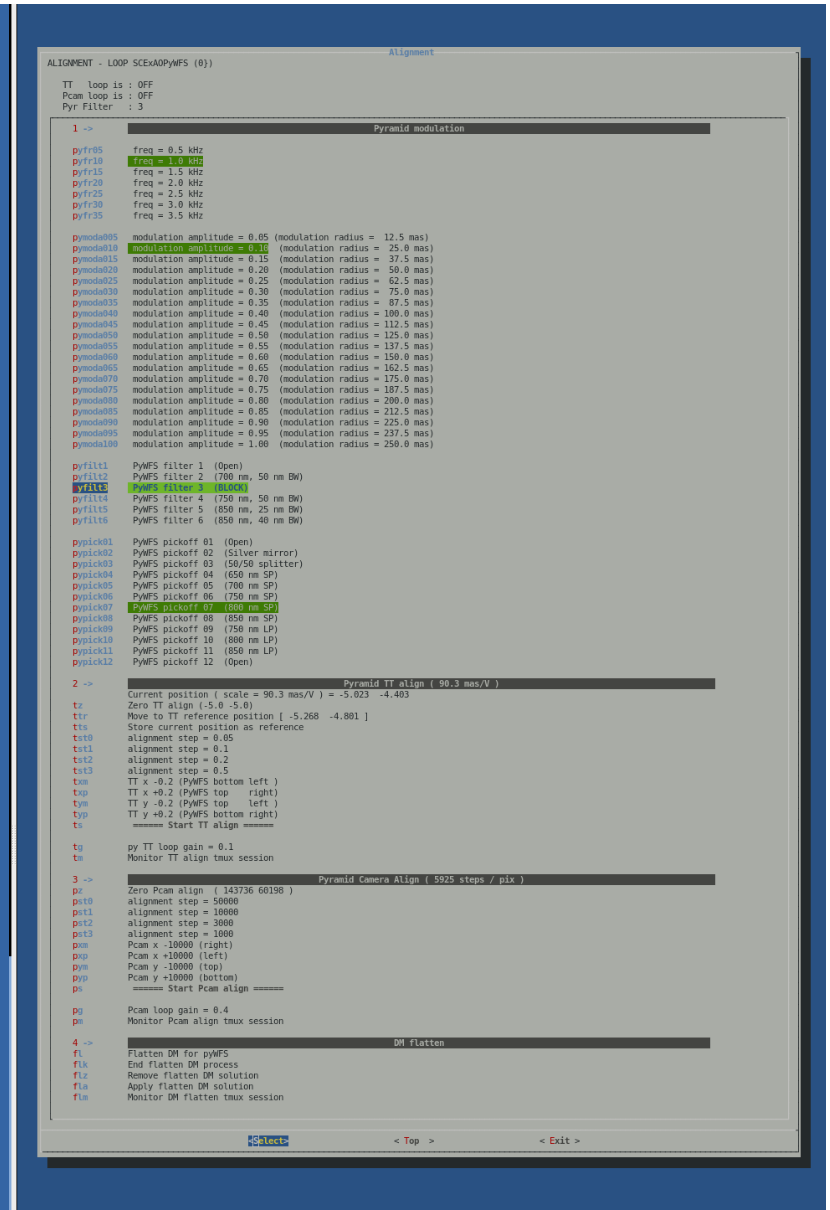
Flexible architecture

All input, output and intermediate data is stored as shared memory. A common format for all shared memory data streams facilitates software development. Multiple processes run simultaneously to perform operations on shared memory streams. Additional processes can be deployed (for example, real-time analysis of an intermediate data stream) without impacting existing processed.

IPC is built in the shared memory structure which contains POSIX semaphores (default of 10 semaphores, more if needed): 10 different processes can run on the same input. Each process waits on input stream(s), and posts output stream(s) semaphore(s) → Real-time operations can be chained, with multiple branches

Example control GUI (bash scripts)





For each file:
conf_<name>_name.txt points to archived file location

conf/conf_<name>_name.txt are read by function ReadConfFile for loading into shared memory and FITS copy to ./conf/aol#_<name>.fits

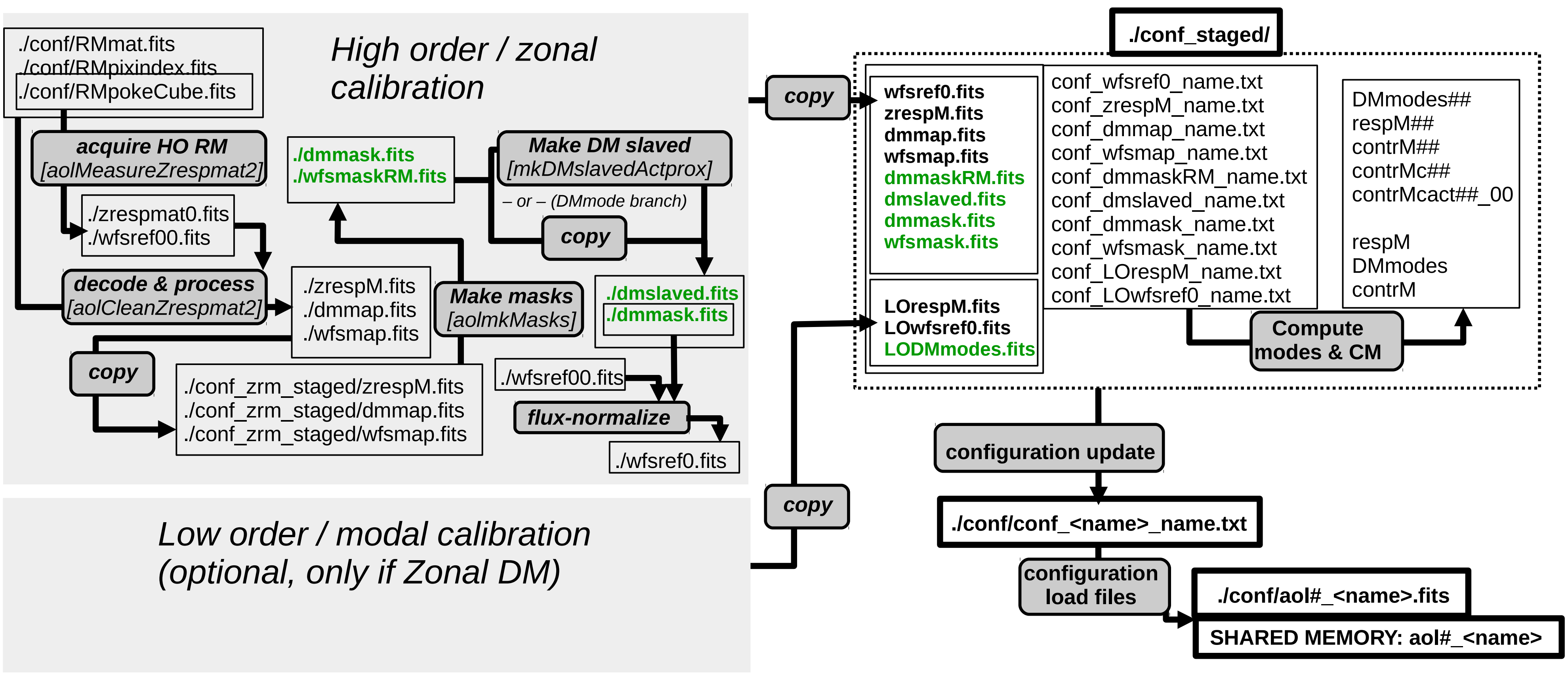
Calibration Work Flow

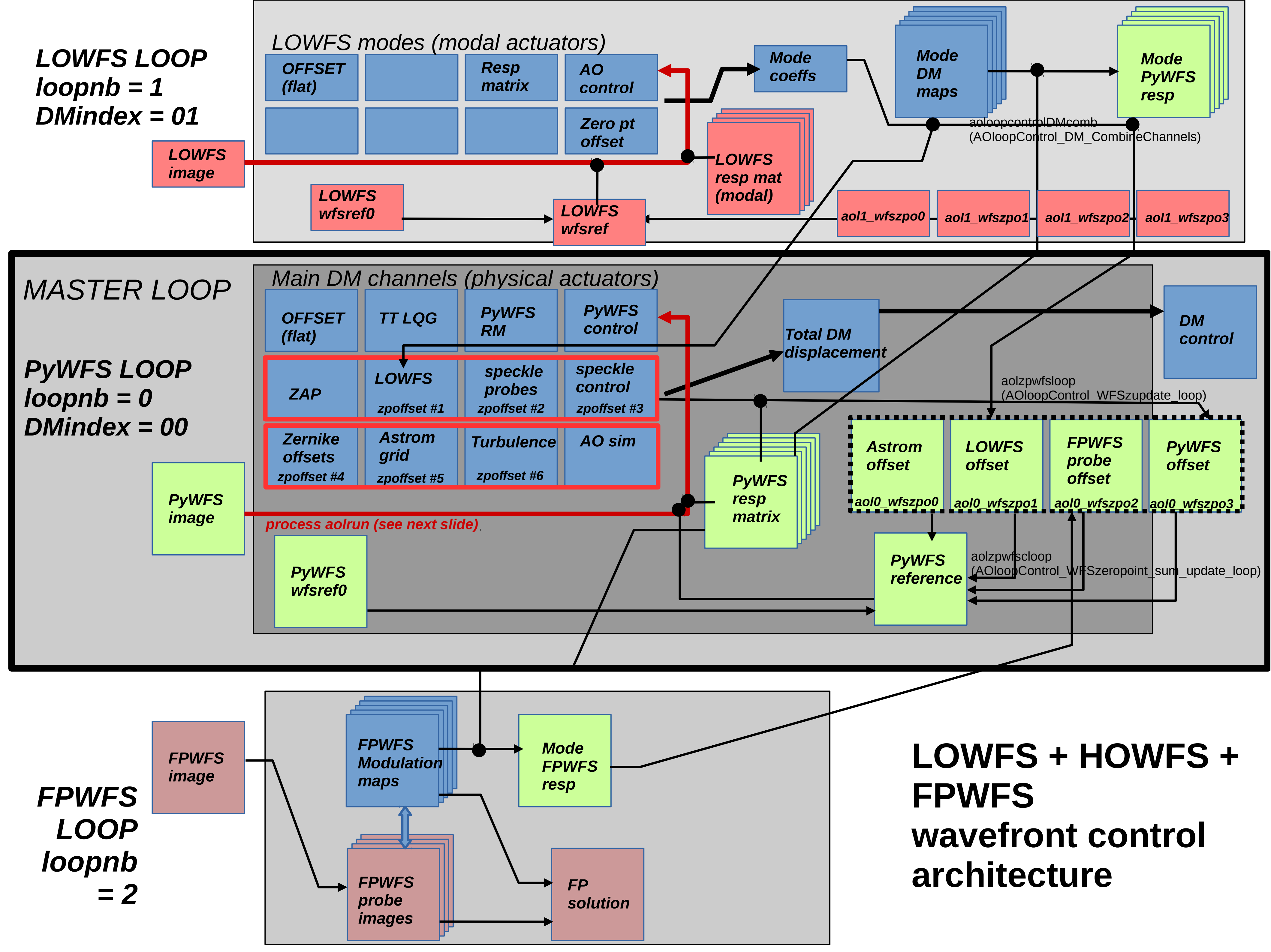
Conventions :

Modal DM: “actuators” indices have no spatial meaning
→ No spatial filtering options
→ “Direct write” CM and “Modal” CM are the same (1 mode = 1 actuator)

Zonal DM: actuator indices correspond to spatial coordinates
→ Need linear transformation between mode coefficients and actuators

If re-using masks, keep from previous calibration




LOWFS LOOP
loopnb = 1
DMindex = 01
LOWFS modes (modal actuators)
OFFSET (flat)
Resp matrix
AO control
Zero pt offset
Mode coeffs
Mode DM maps
Mode PyWFS resp
aoloopcontrolDMcomb
(AOloopControl_DM_CombineChannels)
LOWFS image
LOWFS resp mat (modal)
LOWFS wfsref0
LOWFS wfsref
aol1_wfszpo0
aol1_wfszpo1
aol1_wfszpo2
aol1_wfszpo3
MASTER LOOP
Main DM channels (physical actuators)
OFFSET (flat)
TT LQG
PyWFS RM
PyWFS control
ZAP
LOWFS zpoffset #1
speckle probes zpoffset #2
speckle control zpoffset #3
Zernike offsets zpoffset #4
Astrom grid zpoffset #5
Turbulence zpoffset #6
AO sim
Total DM displacement
DM control
PyWFS LOOP
loopnb = 0
DMindex = 00
PyWFS image
process aolrun (see next slide)
PyWFS wfsref0
PyWFS resp matrix
aolzpwfsloop
(AOloopControl_WFSzupdate_loop)
Astrom offset aol0_wfszpo0
LOWFS offset aol0_wfszpo1
FPWFS probe offset aol0_wfszpo2
PyWFS offset aol0_wfszpo3
PyWFS reference
aolzpwfscloop
(AOloopControl_WFSzeropoint_sum_update_loop)
FPWFS LOOP
loopnb = 2
FPWFS image
FPWFS Modulation maps
Mode FPWFS resp
FPWFS probe images
FP solution
LOWFS + HOWFS + FPWFS
wavefront control architecture

Control Matrix Computation Modes

WFSnorm *(./conf/conf_WFSnormalize.txt)* ***WFS normalization mode*** C code: AOconf[loop].WFSnormalize

0: Do not normalize WFS images

1: Normalize WFS images

WFSnorm should be left unchanged between RM acquisition and Loop control

DMprimaryWrite (./conf/conf_DMprimWriteON.txt) ***DM primary write*** C code: DMprimaryWrite_ON

0: DM primary write is off

1: DM primary write is on

CMmode *(./conf/conf_CMmode.txt)* ***Combined Control matrix mode*** C code: MATRIX_COMPUTATION_MODE

0: not combined: control matrix is WFS pixels → modes

→ Linking aol#_DMmode_meas ↔ aol#_modeval

→ modesextractwfs reads from aol_DMmode_meas instead of computing

1: combined: control matrix is WFS pixels → DM actuators

DMMODE *(./conf/conf_DMMODE.txt)* ***DM mode (zonal vs. modal)*** Bash script only, only affects bash scripts and options

ZONAL: pixel coordinates correspond to DM actuators physical location

→ spatial filtering enabled for DM modes creation

→ blocks built by spatial frequencies, user can set independent gain values for mode blocks

MODAL: DM pixels correspond to abstract modes

→ no spatial filtering, setting 1 block only

Note: **DMMODE**=ZONAL → **CMMODE**=MODAL (CPA-splitting of modes into blocks)

GPUmode *(./conf/conf_GPUmode.txt)* ***# of GPUs to use for CM multiplication*** C code: AOconf[loop].GPU

0: use CPU

>0: number of GPUs

if CMmode=1 and GPUmode>0:

GPUallmode *(./conf/conf_GPUall.txt)* ***Use GPU for all computations*** C code: AOconf[loop].GPUall = COMPUTE_GPU_SCALING

0: Use CPU for WFS reference subtraction and normalization

→ WFS reference subtraction and normalization done by CPU (imWFS0→ imWFS1→ imWFS2)

→ CM multiplication input is imWFS2 (GPU or CPU)

1: Use GPU for all computation

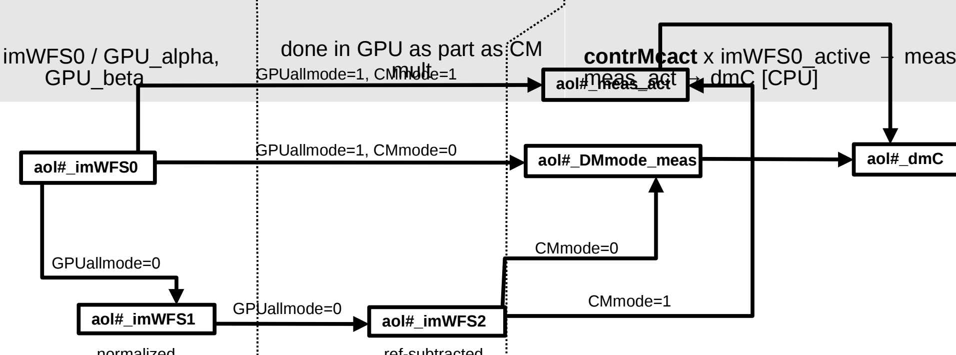
→ WFS reference subtraction and normalization done by GPU

→ GPU-based CM multiplication input is imWFS0

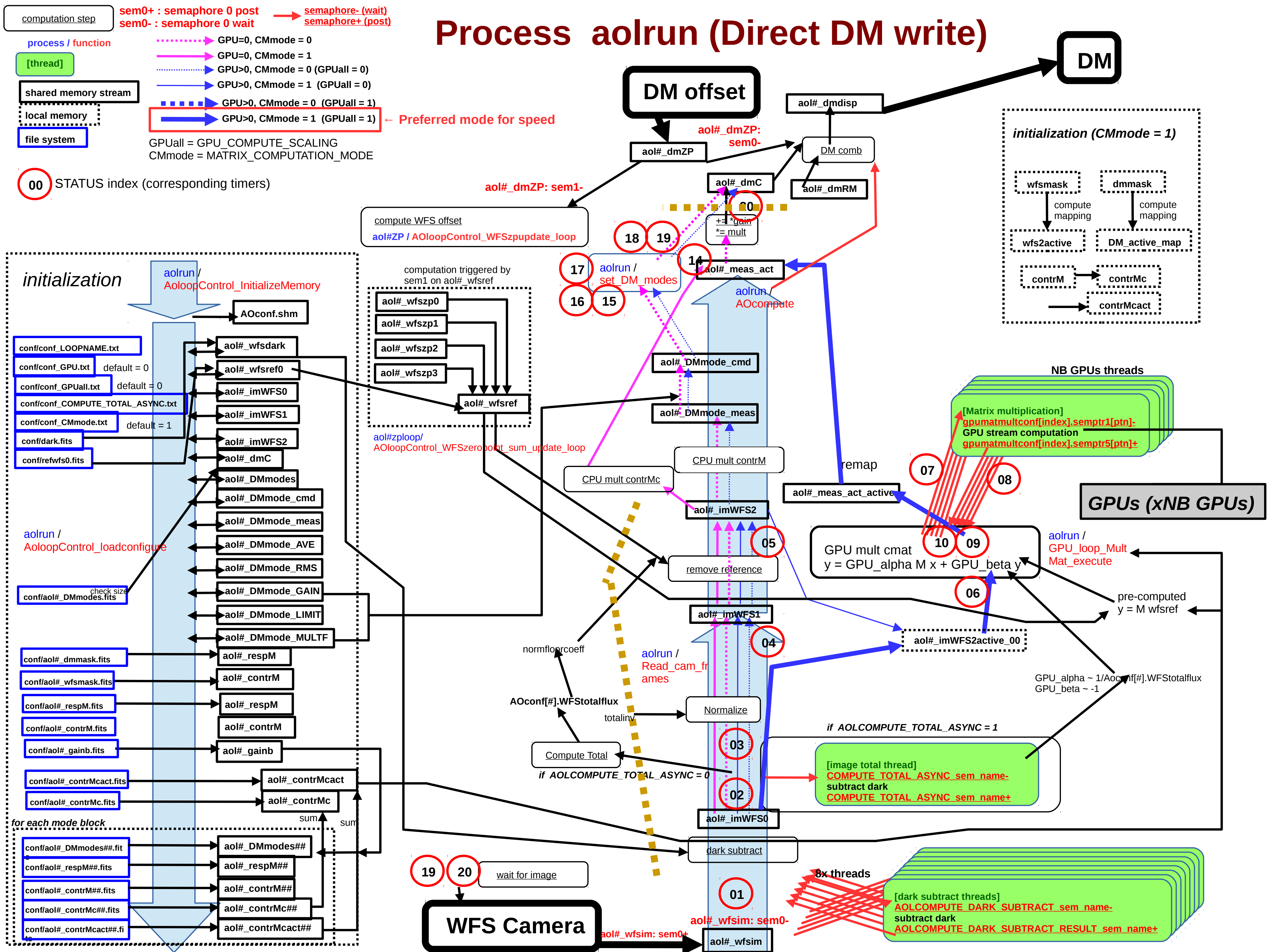
Control Matrices

Matrix	Description	Input→ output	Gain control (primary write)	Notes
contrM (CMmode=0)	***Full modal control matrix*** *Split in multi-GPU*	WFSpix → DMmodes	0.0<**loopgain**<1.0 0.0<**DMmodes_GAIN[m]**<1.0	gainMB has no effect and will not update contrM
contrMc (CMmode=1, GPUmode=0)	***Full combined control matrix*** *Split in multi-GPU*	WFSpix → DMactuators	0.0<**gainMB[k]**<1.0 0.0<**loopgain**<1.0	contrMc re-built for each change of gainMB If DM is MODAL: gainMB has no effect and will not update contrM
contrMcact (CMmode=1, GPUmode=1)	***Combined control matrix, only active pixels*** *Split in multi-GPU*	Active WFS pixels → Active DM actuators	0.0<**gainMB[k]**<1.0 0.0<**loopgain**<1.0	contrMcact re-built for each change of gainMB If DM is MODAL: gainMB has no effect and will not update contrM

CMmode MATRIX_COMPUTATION_MODE	GPUmode	GPUallmode COMPUTE_GPU_SCALING	Camera read output (Read_cam_frame)	WFS reference subtraction	Control Matrix operation(s)
0	0	0	→ imWFS1	CPU subtraction → imWFS2	**contrM** x imWFS2 → DMmode_meas [CPU] DMmode_meas → cmd_modes [CPU] DMmodes x cmd_modes →dmC [CPU]
0	>0	0	→ imWFS1	CPU subtraction → imWFS2	**contrM** x imWFS2 → DMmode_meas [GPU] DMmode_meas → cmd_modes [CPU] DMmodes x cmd_modes →dmC [GPU]
0 [to be done]	>0	1	→ imWFS0 / GPU_alpha, GPU_beta	done in GPU as part as CM mult	**contrM** x imWFS0 →DMmode_meas [GPU] DMmode_meas → cmd_modes [CPU] DMmodes x cmd_modes →dmC [GPU]
1	0	0	→ imWFS1	CPU subtraction → imWFS2	**contrMc** x imWFS2 → meas_act [CPU] meas_act → dmC [CPU]
1	>0	0	→ imWFS1	CPU subtraction → imWFS2	**contrMcact** x imWFS2_active → meas_act_active [GPU] meas_act → dmC [CPU]
1	>0	1	→ imWFS0 / GPU_alpha, GPU_beta	done in GPU as part as CM mult	**contrMcact** x imWFS0_active → meas_act_active [GPU] meas_act → dmC [CPU]

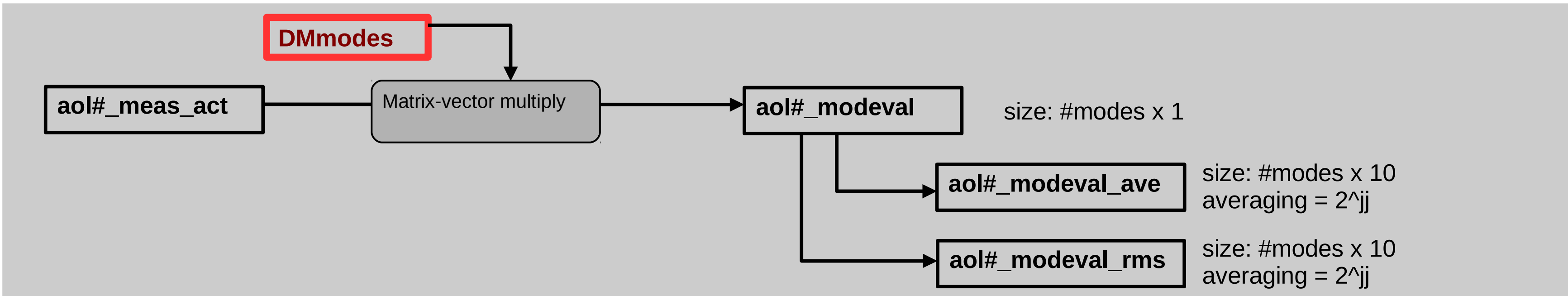


Process aolrun (Direct DM write)

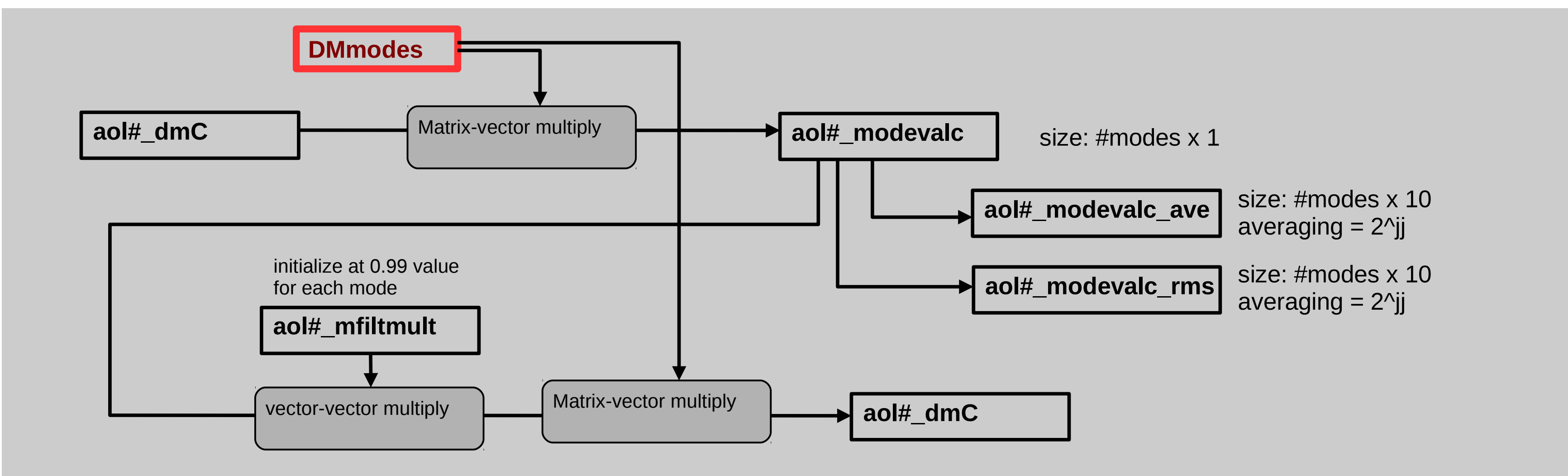


Auxillary processes

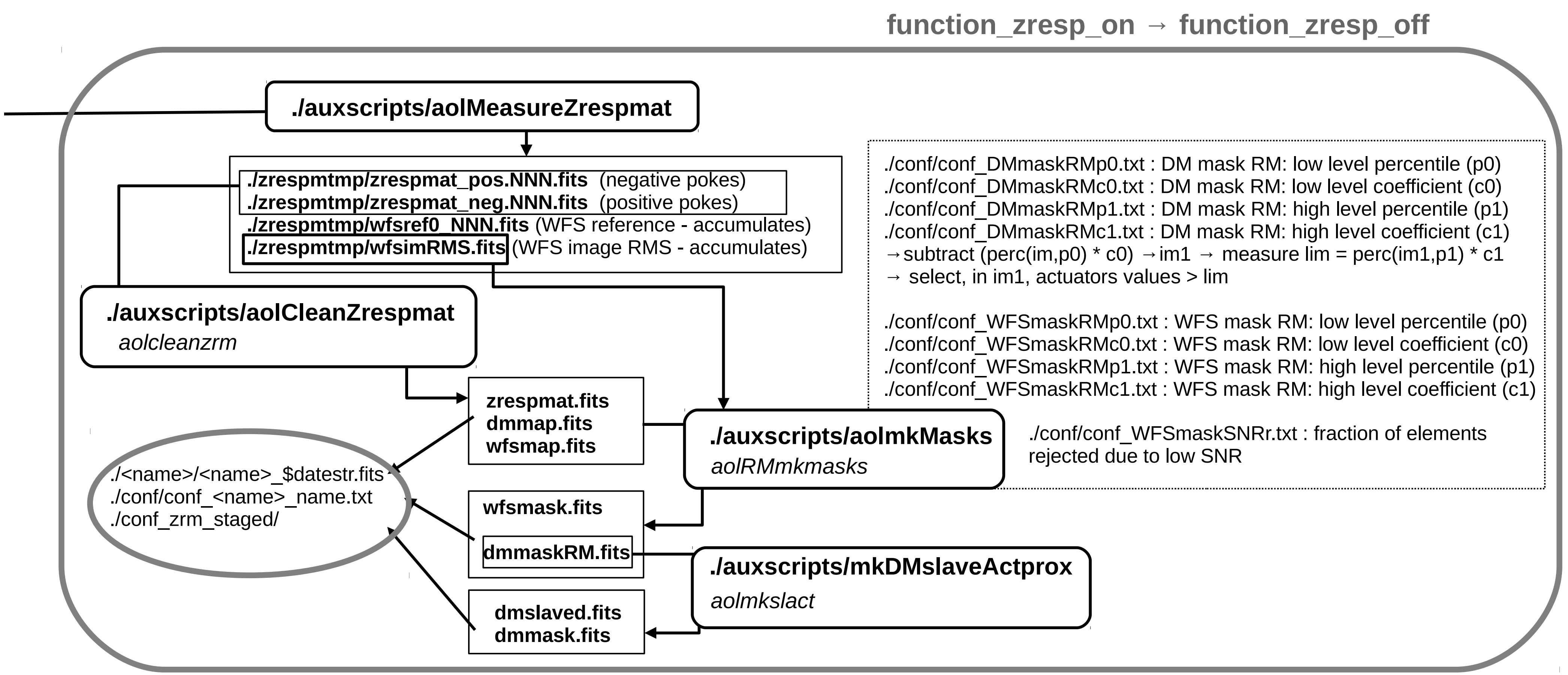
Decompose WFS measurements in modes

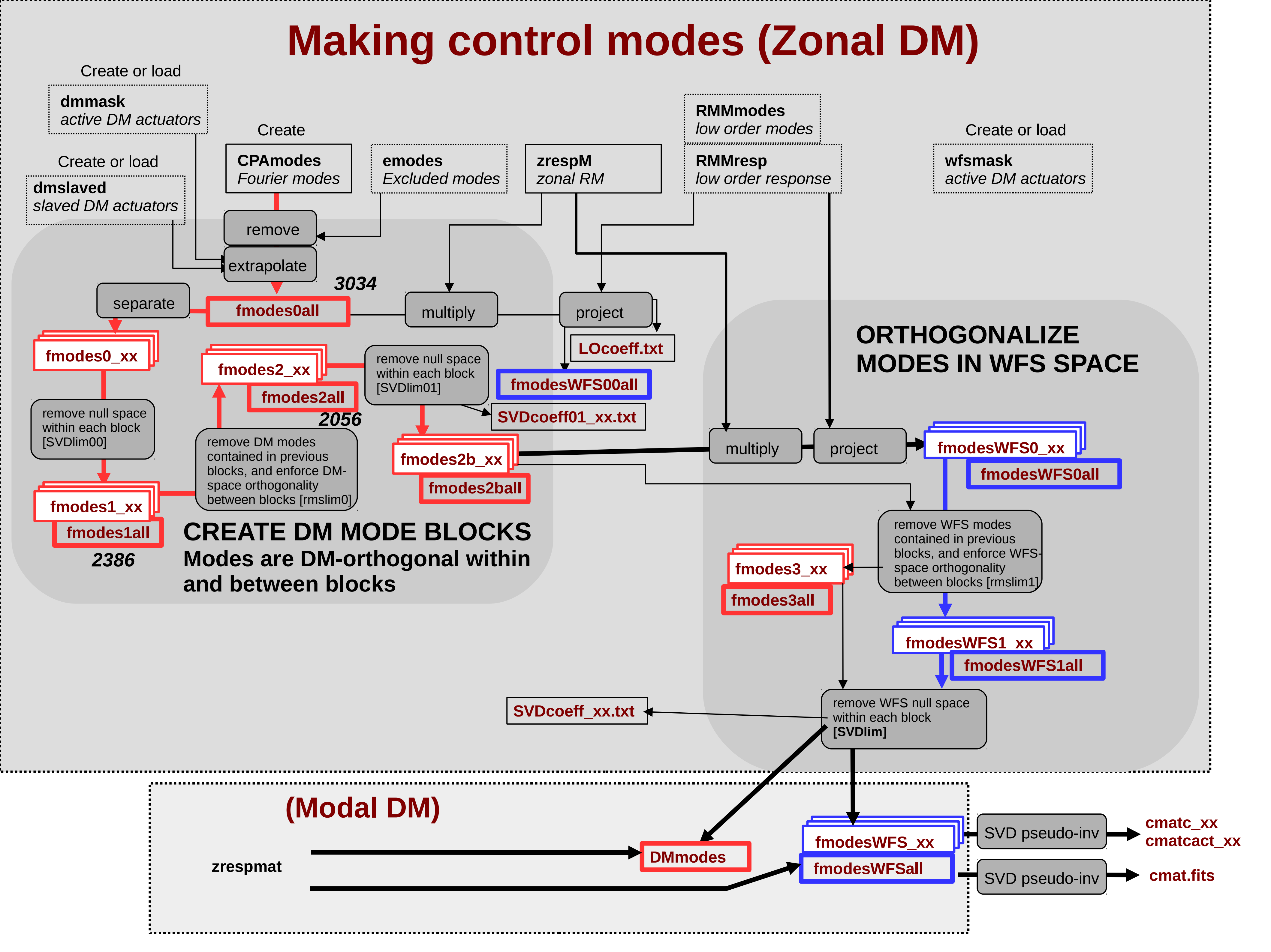


Decompose DM commands in modes + apply modal mult gains



Zonal response matrix acquisition → masks

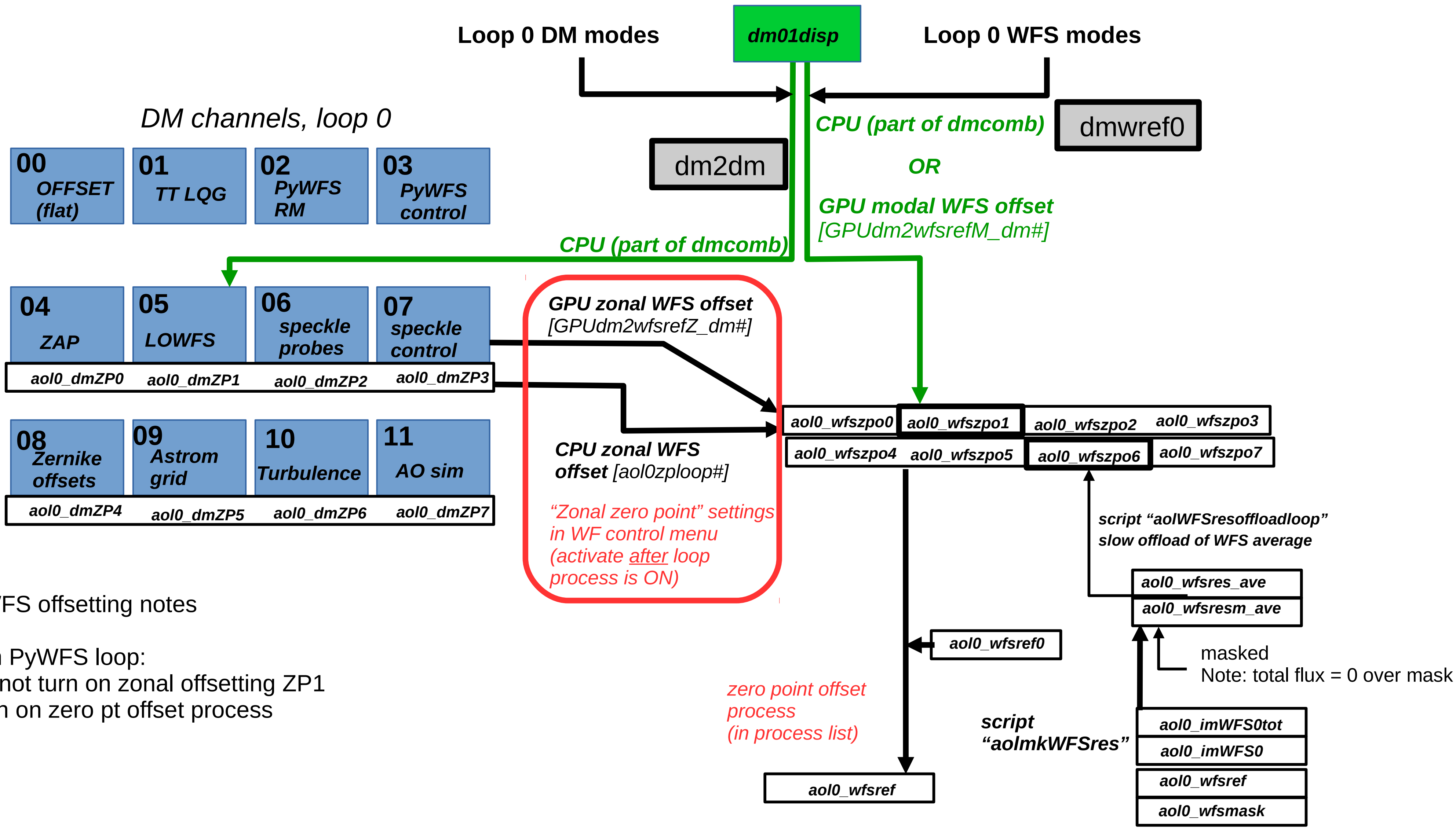




Making control modes (Zonal DM)
Create or load
dmmask
active DM actuators
Create or load
dmslaved
slaved DM actuators
Create
CPAmodes
Fourier modes
emodes
Excluded modes
zrespM
zonal RM
RMMmodes
low order modes
RMMresp
low order response
Create or load
wfsmask
active DM actuators
remove
extrapolate
3034
separate
fmodes0all
multiply
project
LOcoeff.txt
fmodesWFS00all
fmodes0_xx
fmodes2_xx
fmodes2all
remove null space within each block [SVDlim01]
SVDcoeff01_xx.txt
remove null space within each block [SVDlim00]
2056
remove DM modes contained in previous blocks, and enforce DM-space orthogonality between blocks [rmslim0]
fmodes2b_xx
fmodes2ball
fmodes1_xx
fmodes1all
2386
CREATE DM MODE BLOCKS
Modes are DM-orthogonal within and between blocks
ORTHOGONALIZE MODES IN WFS SPACE
multiply
project
fmodesWFS0_xx
fmodesWFS0all
remove WFS modes contained in previous blocks, and enforce WFS-space orthogonality between blocks [rmslim1]
fmodes3_xx
fmodes3all
fmodesWFS1_xx
fmodesWFS1all
SVDcoeff_xx.txt
remove WFS null space within each block [SVDlim]
(Modal DM)
zrespmat
DMmodes
fmodesWFS_xx
fmodesWFSall
SVD pseudo-inv
SVD pseudo-inv
cmatc_xx
cmatcact_xx
cmat.fits

OFFSETTING
LOWFS (loop #1, dm01) → PyWFS (loop #0, dm00)

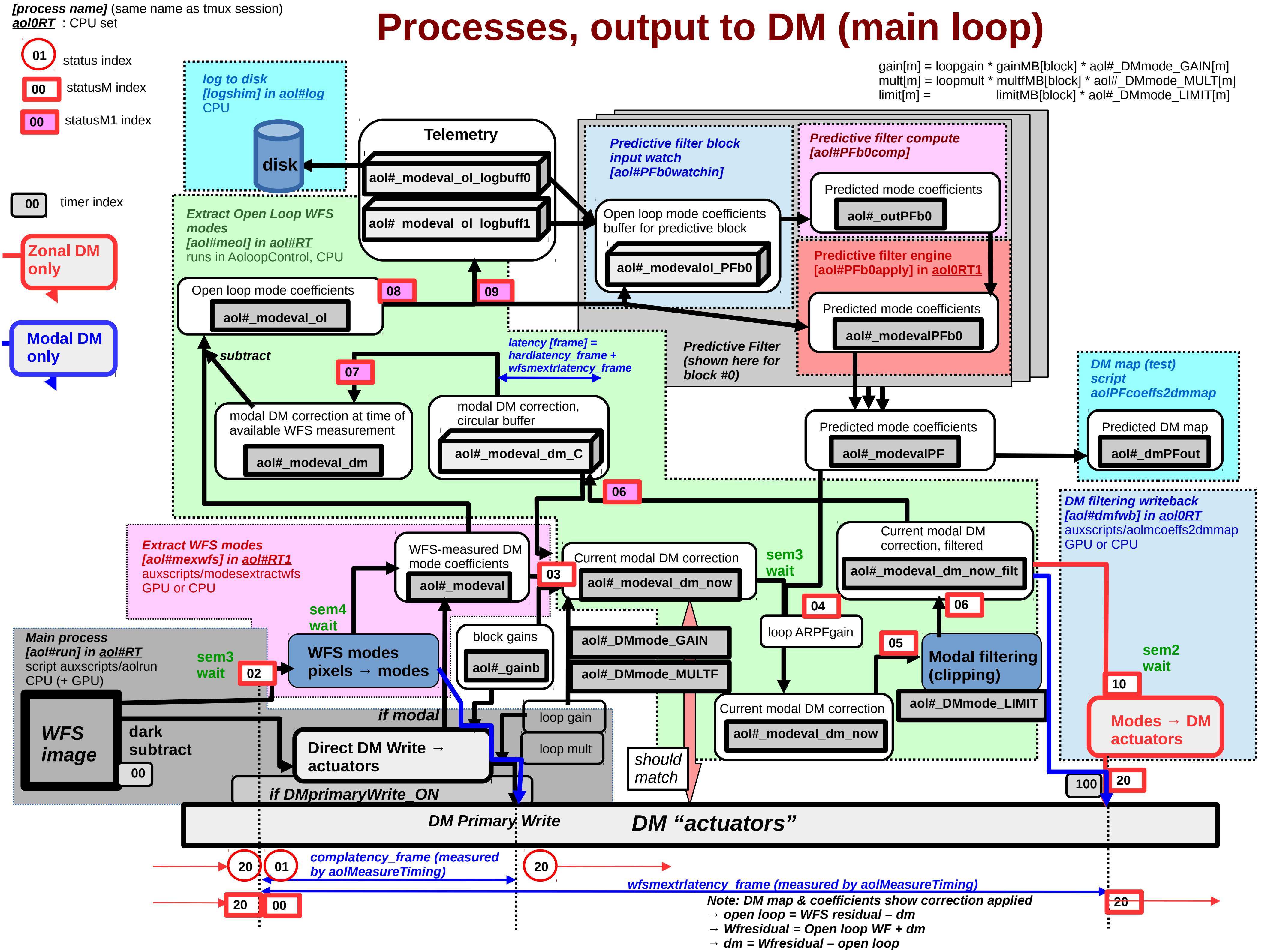
Green color: process is part of loop #1



LOWFS offsetting notes

From PyWFS loop:
- Do not turn on zonal offsetting ZP1
- Turn on zero pt offset process

Processes, output to DM (main loop)



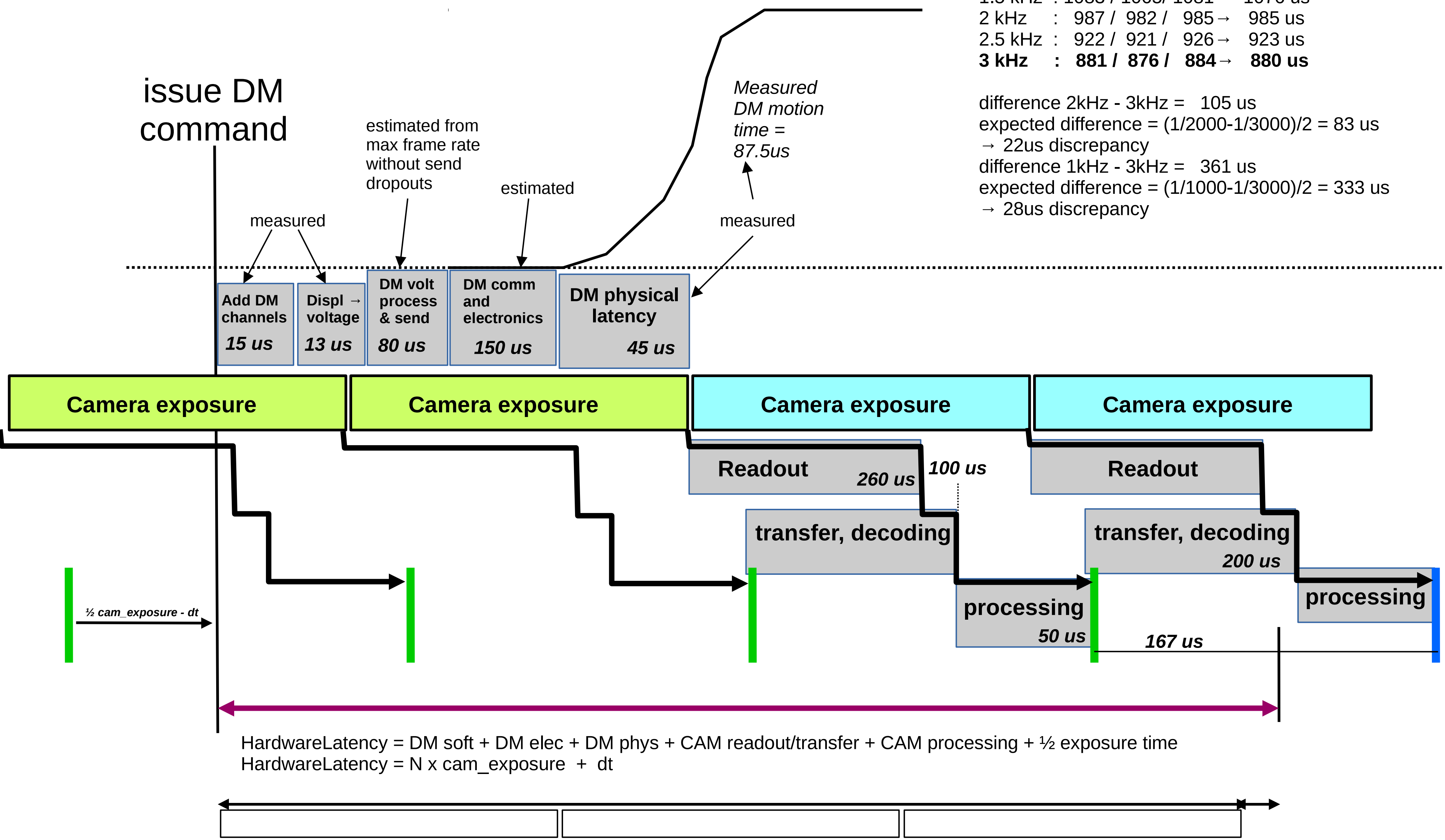
Hardware Latency

Definition:
Time offset between ***DM command issued****, and* ***mid-point between 2 consecutive WFS frames with largest difference***

SCExAO measured hardware latencies:

1 kHz : 1253 / 1260/ 1269 → 1261 us
1.5 kHz : 1083 / 1065/ 1081 → 1076 us
2 kHz : 987 / 982 / 985 → 985 us
2.5 kHz : 922 / 921 / 926 → 923 us
3 kHz : 881 / 876 / 884 → 880 us

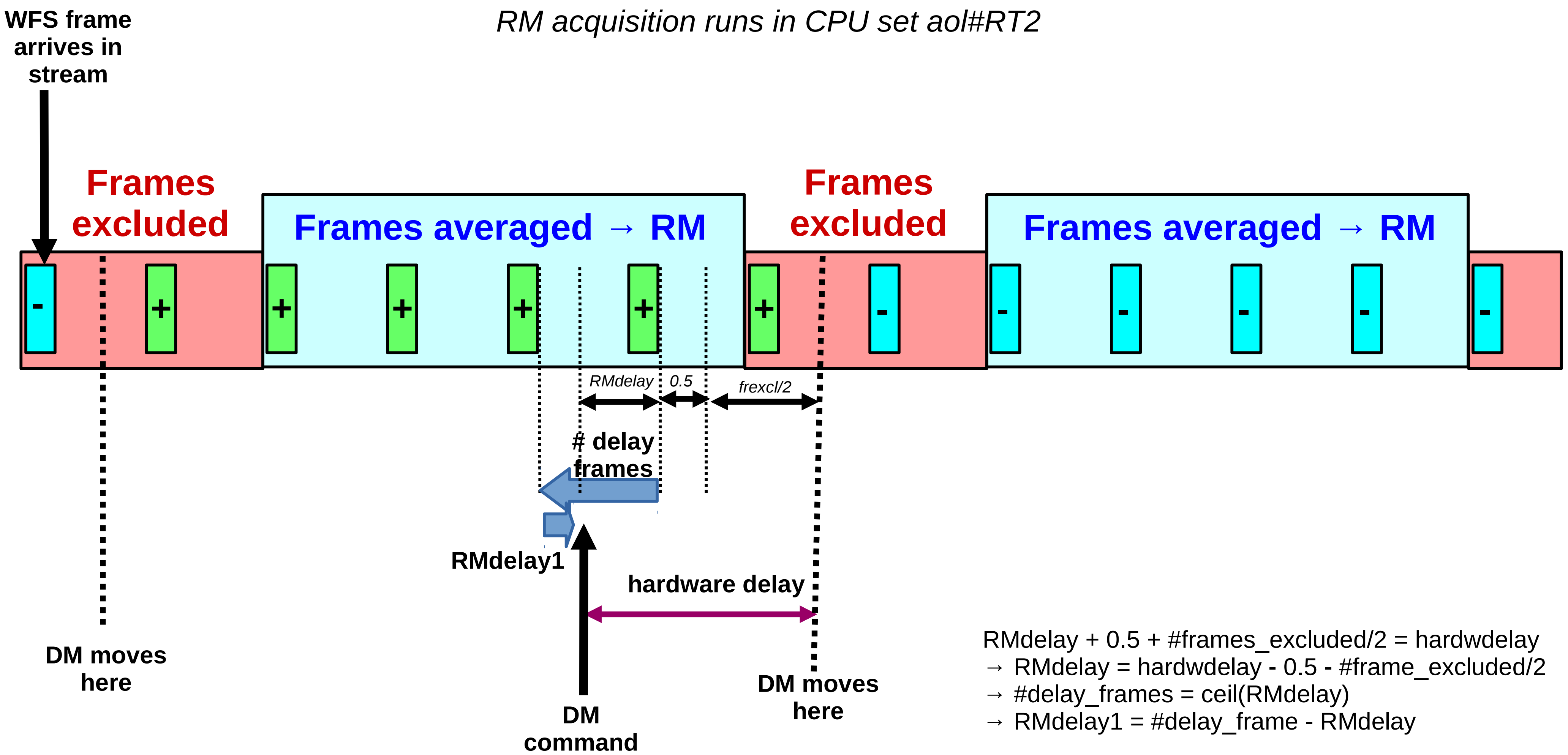
difference 2kHz - 3kHz = 105 us
expected difference = (1/2000-1/3000)/2 = 83 us
→ 22us discrepancy
difference 1kHz - 3kHz = 361 us
expected difference = (1/1000-1/3000)/2 = 333 us
→ 28us discrepancy



HardwareLatency = DM soft + DM elec + DM phys + CAM readout/transfer + CAM processing + ½ exposure time
HardwareLatency = N x cam_exposure + dt

RM acquisition - Timing

RM acquisition runs in CPU set aol#RT2



RMdelay + 0.5 + #frames_excluded/2 = hardwdelay
→ RMdelay = hardwdelay - 0.5 - #frame_excluded/2
→ #delay_frames = ceil(RMdelay)
→ RMdelay1 = #delay_frame - RMdelay

Loop:

Wait on and read WFS frame → allocate WFS frame to appropriate frame block

If poke required: wait RMdelay1, then poke

Predictive Control Implementation Notes

Predictive control implementation…

… **must be compatible with multi-WFS architecture**. See slide #6 for SCExAO multi-loop architecture, showing how master loop (Pyramid WFS) is offset by secondary loops (LOWFS and speckle control)

… **must be able to smoothly transition from/to conventional zonal DM write**. Slide #9 shows the details of the “direct fast DM write” mode, which is a GPU-based matrix multiplication going from WFS pixels to DM actuators (no modal step). Slide #9 shows the red arrow in the center-left part of slide #6

… **operates on modes that are constructed from the master WFS response**. Slide #6 shows how the modes are constructed. Slides #19-23 show actual reconstruction performance in real-time (in lab)

… **requires accurate and stable (at ~10us level) timings** to allow pseudo open-loop telemetry reconstruction. Slide #15 shows the timing diagram for SCExAO

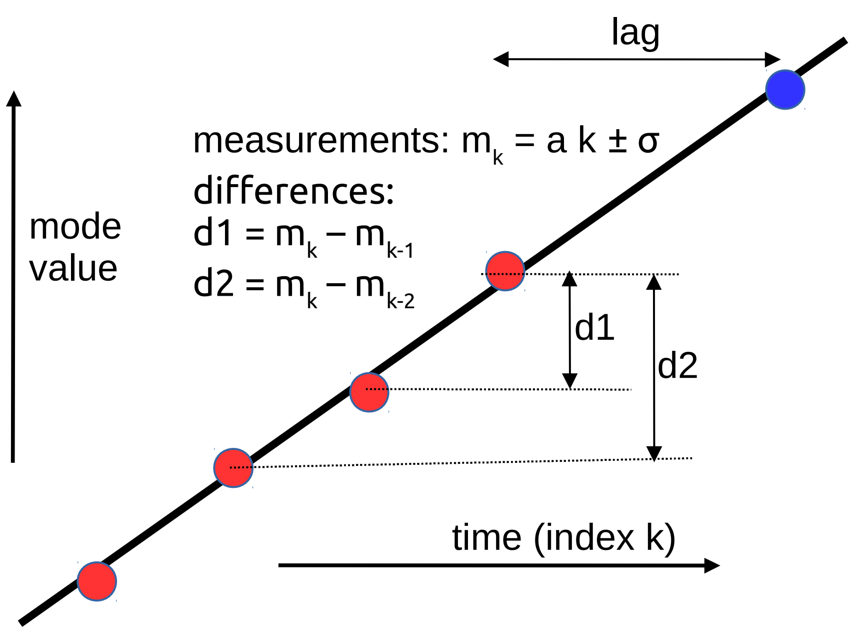
… **requires reliable WFS & DM calibration.** This is achieved by frequently taking on-sky response matrix at full speed modulation (3 kHz) while the loop the closed. Slide #16 shows how timing info allows fast RM acquisition: SCExAO now takes a RM in 2 sec by modulating the DM at 3 kHz (frames excluded = 0, frames averaged = 1)

Slide #14 shows the **overall data flow** with predictive control, including real-time pseudo open-loop reconstruction.

Automatic Gains Setting – Fast Mode

C code: *AOloopControl_AutoTuneGains*   Script: *auxscripts/aolautotunegains*
Goal: **Find optimal gain for each mode in non-predictive mode in bright star regime**. This mode should be very reactive and robust, and able to recompute 1200 optimal gains in < 300 us to allow gain updates @ up to 3 kHz.
Bright star regime: input WF mode evolves linearly with time (control frequency > vibrations)

→ Error is quadratic sum of time lag and measurement noise, which can be expressed as simple functions of recent measurements.



With integrator (gain = g)

Time lag error:
$\sigma_{TL} = a\,(\,lag + 1/g\,)$

Measurement noise propagation:
$\sigma_{MN} = \mathrm{sqrt}(g/(1-g))\ \sigma$

Estimating slope (a) and measurement noise (σ)

$\langle d1^2\rangle = a^2 + 2\,\sigma^2$
$\langle d2^2\rangle = 4\,a^2 + 2\,\sigma^2$

$\mathbf{a^2 = (\,\langle d2^2\rangle - \langle d1^2\rangle\,)\,/\,3}$
$\mathbf{\sigma^2 = (\,4\langle d1^2\rangle - \langle d2^2\rangle\,)\,/\,6}$
$\mathbf{MstdDev^2 = \langle m^2\rangle - \sigma^2}$

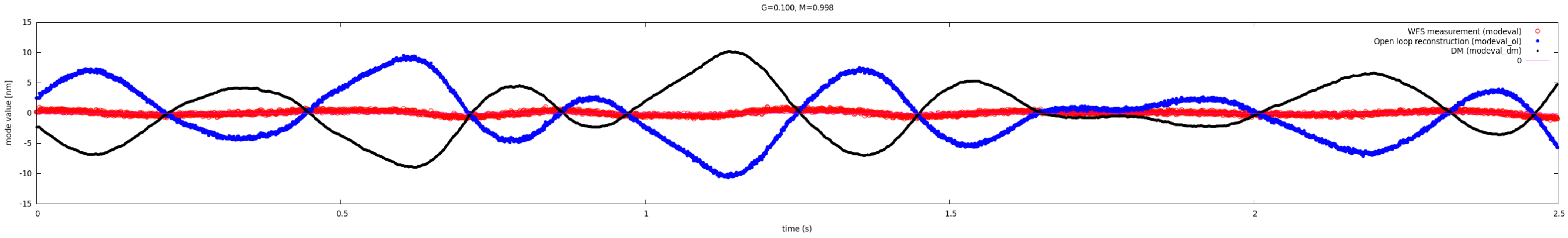
Real time process steps:
- Compute open loop coefficient mode values while loop is closed
- Update slope and measurement noise from running averages of $d1^2$ and $d2^2$
- Optimize $\sigma_{TL}^2 + \sigma_{MN}^2$ as a function of gain → update optimal gain

Open loop reconstruction

Turbulence injected:
8m/s wind speed, slightly filtered Kolmogorov spectrum (LOcoeff=0.1), 50nm RMS on DM (100nm WF), speed = 8 m/s
written to DM every 300us (3.333 kHz frequency)

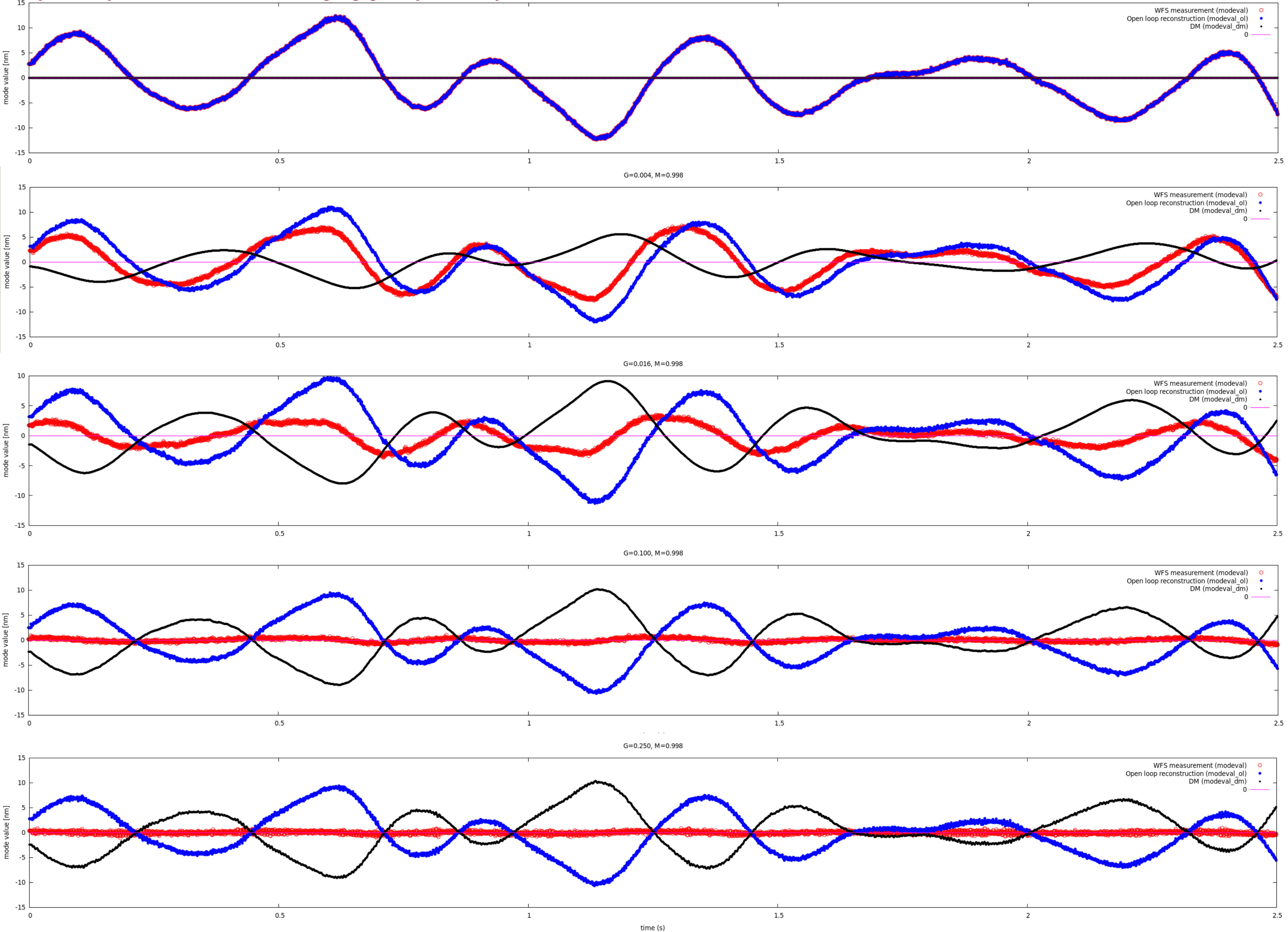
Showing **Open loop (blue)** and **WFS measured (red)** modal coefficient #30
Data acquition start synchronized to turbulence sequence start

PyWFS running at 2kHz, 125mas modulation radius. Total delay = 2.6 frame



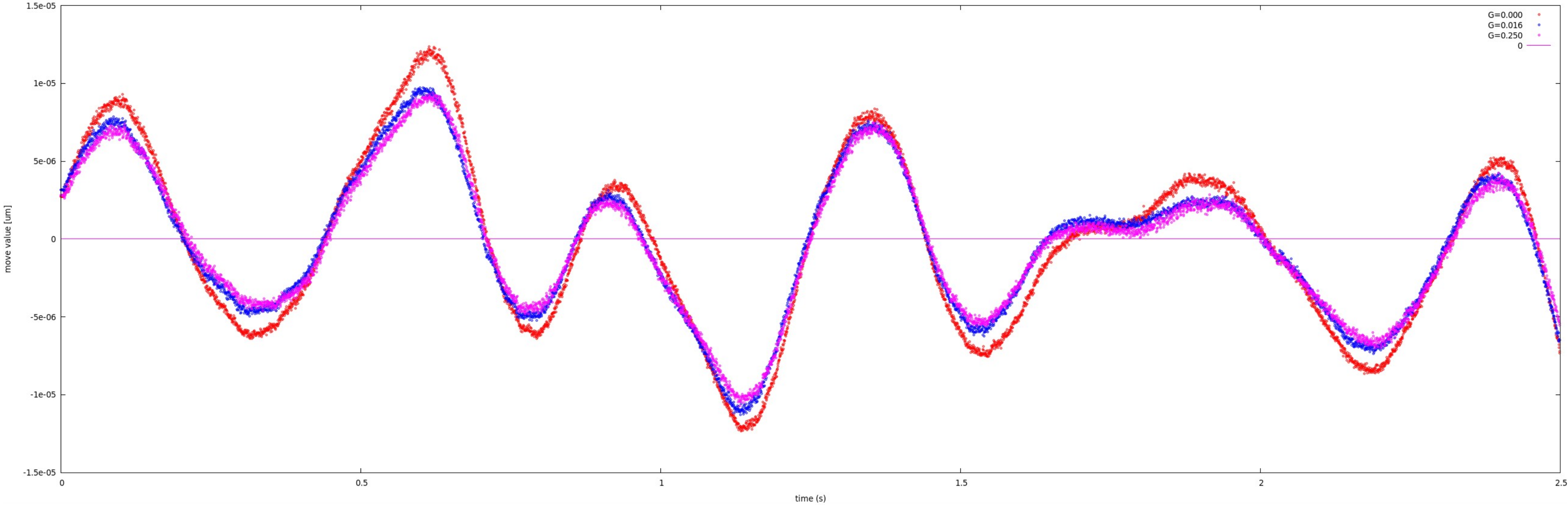
Open loop reconstruction: Changing gain (M=0.998)

G=0.000, M=0.998



Open loop reconstruction
Comparison between gain values

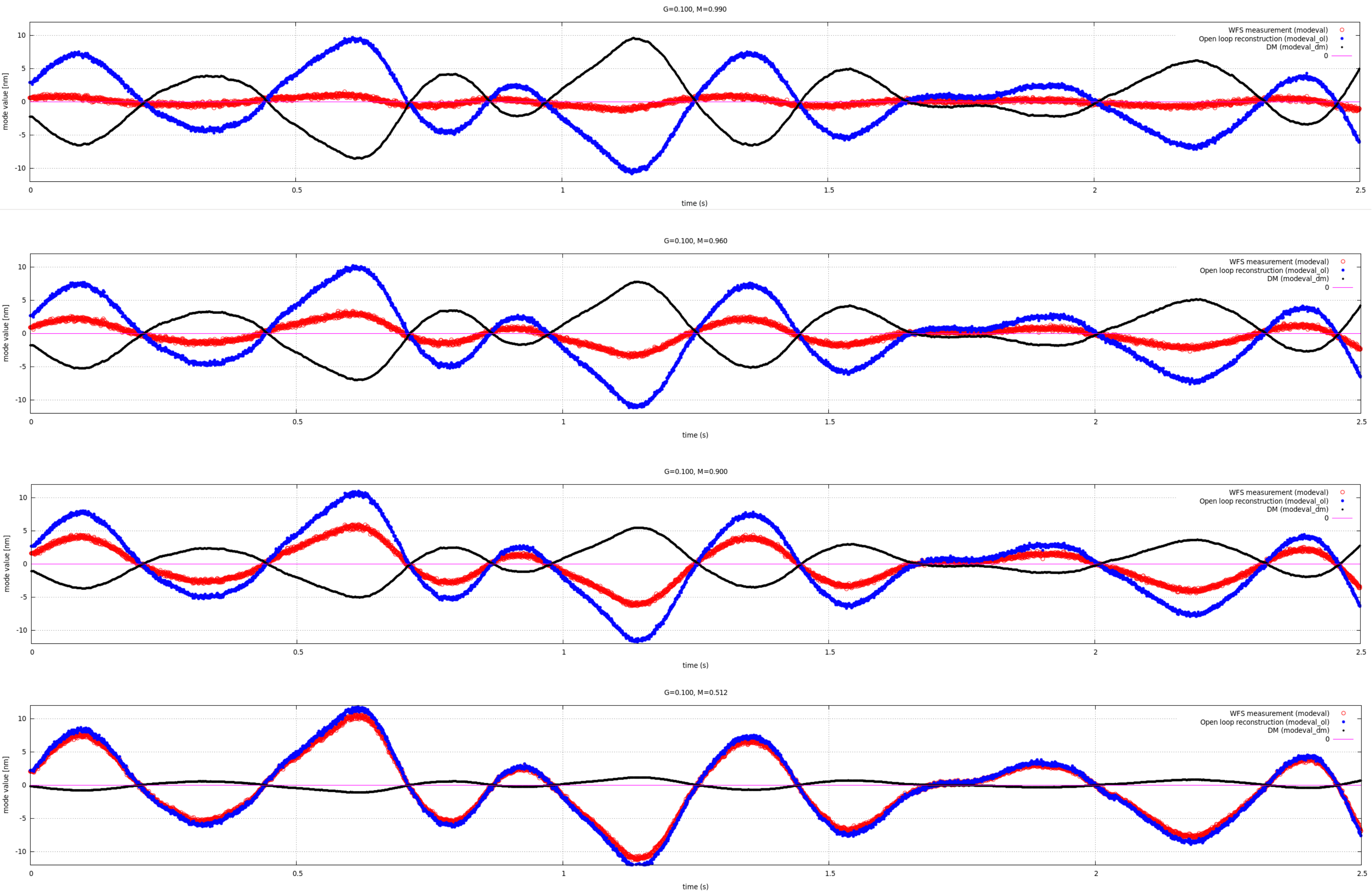
G=0.000 → over-estimates OL values
All G>0.0 reconstructions match at %-level



G=0.000 test relies entirely on WF residuals for OL estimation
G>0.000 tests rely mostly on DM values for OL estimation

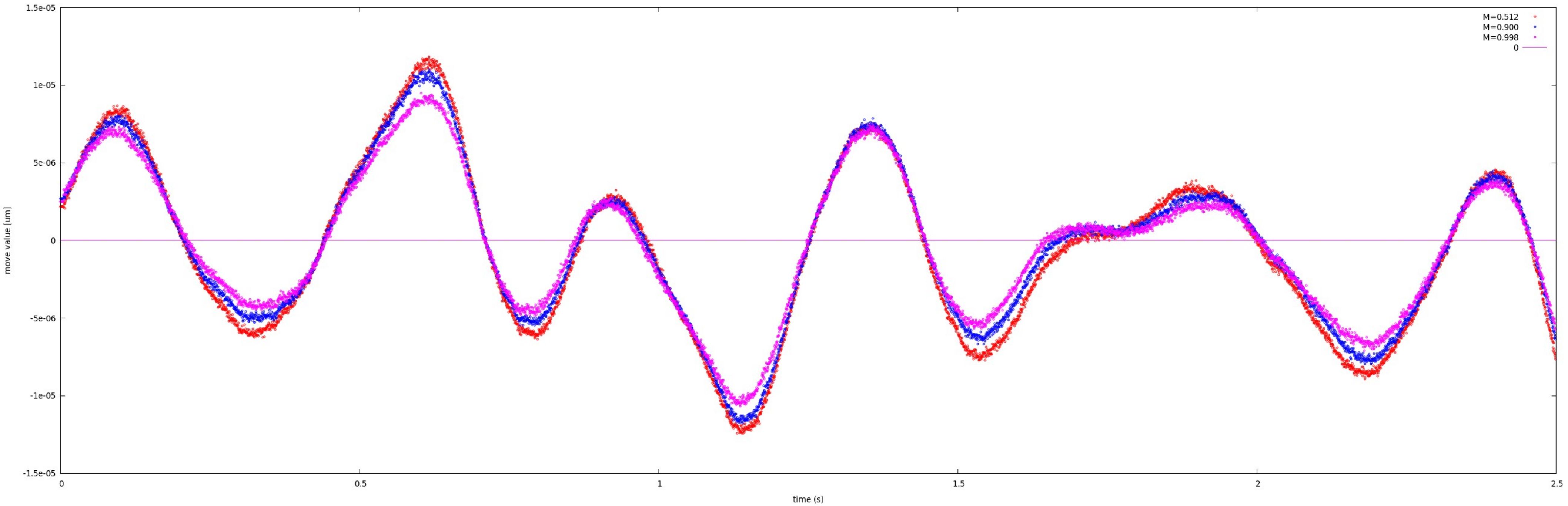
Test shown here uses full speed RM acquisition which underestimates RM by ~15% due to DM time-of-motion → reconstructed WFs from WFS are over-estimated by ~15%

Open loop reconstruction: Changing mult factor (G=0.100)



Open loop reconstruction
Comparison between Mult factor

Small M over-estimates OL values
→ same effect as G=0.0 over-estimation



M<<1 tests rely mostly on WF residuals for OL estimation
M~1 tests rely mostly on DM values for OL estimation

Test shown here uses full speed RM acquisition which underestimates RM by ~15% due to DM time-of-motion → reconstructed WFs from WFS are over-estimated by ~15%

Open loop reconstruction: noise propagation

NOISE = σ x sqrt(1 + M^2 G^2 / (1-M^2))