# ShaneAO
# Real-Time Control

Donald Gavel

Predictive Control Meeting

May 15, 2013

Revised: Oct 24, 2013

# Acknowledgements

- SPG Software Group
  - Will Deich
  - Kyle Lanclos
  - Steve Allen
  - John Gates
  - Mark Reinig
- Graduate Students
  - Andrew Norton
  - Sri Srinath

- Helpful discussions with

Don Wiberg, UCSC, Dave Palmer, LLNL, (GPI group), Christoph Baranec, Caltech, Reed Riddle, Caltech (ROBO-AO group)

# ShaneAO RTC -

The ShaneAO AO control system is implemented in a hierarchy of support software packages:

- Lowest level – fast computations – “bare minimum” data/parameter-driven program
- Mid level – data and parameter maintenance (diagnostics, calibration, parameter loading, operations modes)
- GUI level – user interface
- Support routines – generate parameters, do simulations and validations
- Code maintenance – cvs repository, Knowledge Tree documentation, on-line documentation

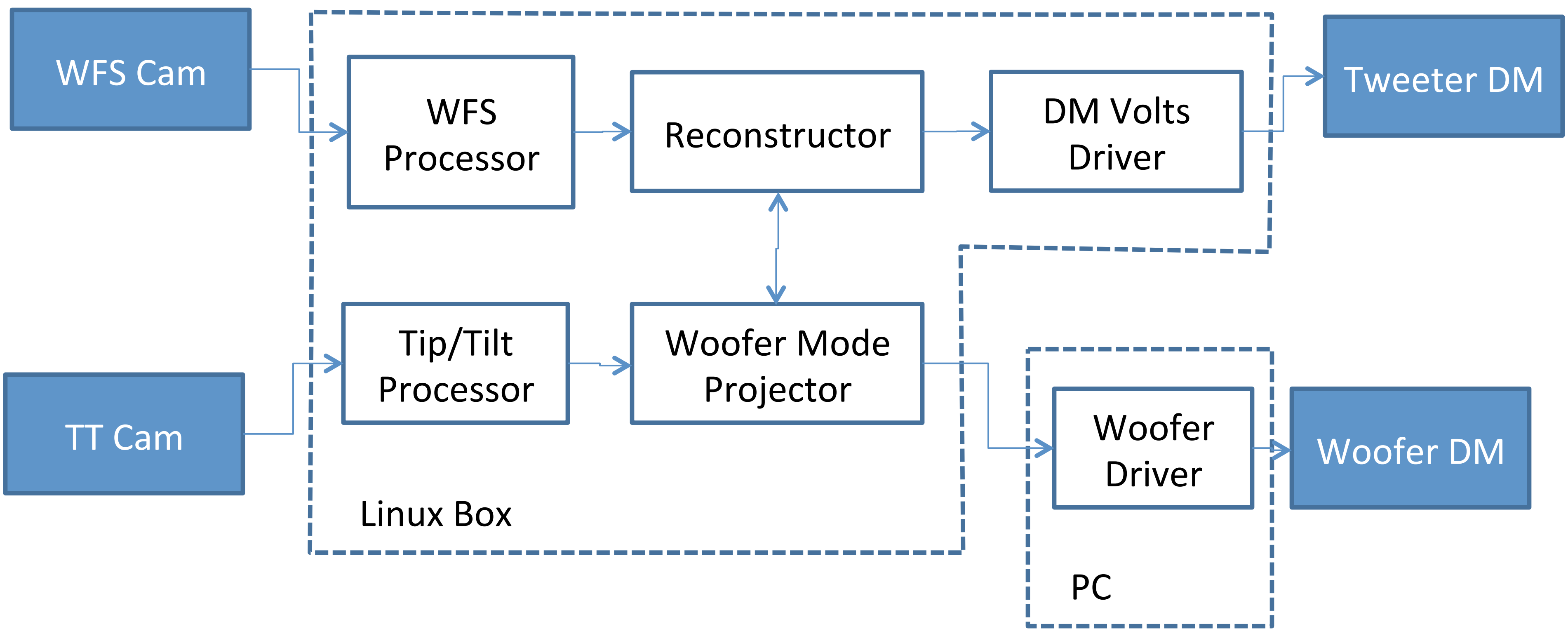
# Requirements Definition Documents

## ShaneAO document server (KnowledgeTree) links

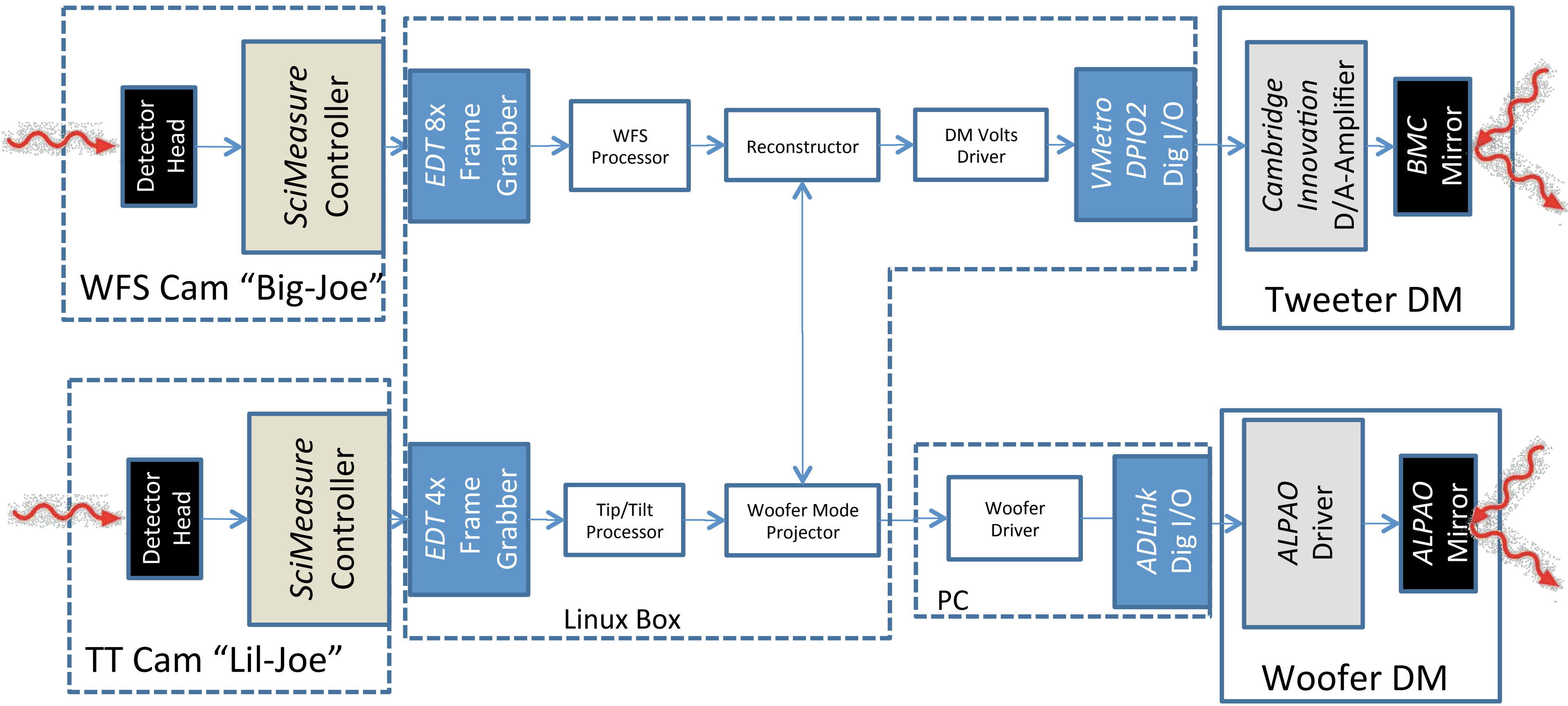
- RTC Software Definition Document 011bu
- RTC Timing Requirements 011bj
- RTC Data Requirements 011bk

# RTC Hardware

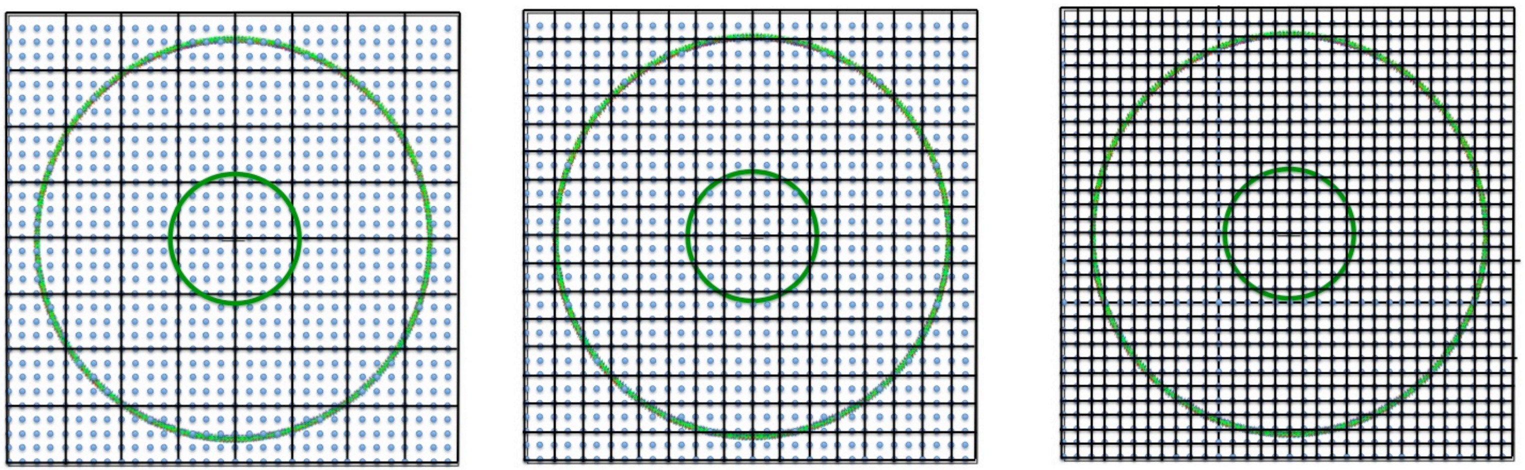
# Hardware/RTC data flow

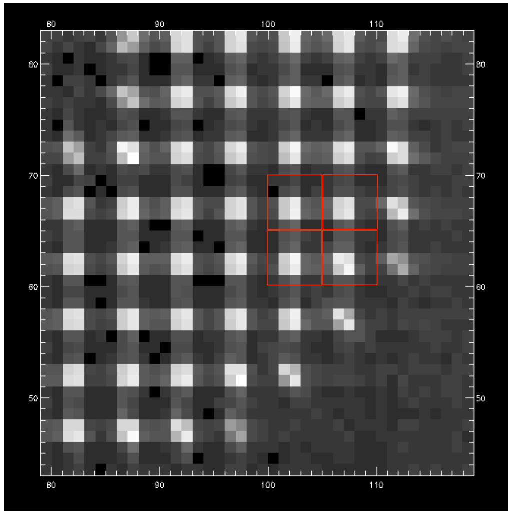


# Detail: hardware pieces in the data flow paths

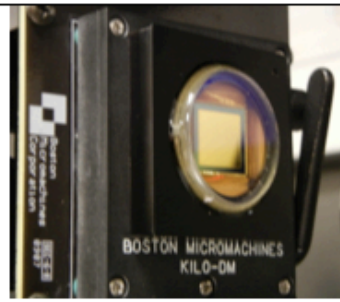


# ShaneAO allows for 3 WFS architectures: 8x, 16x, 30x





# Woofer-Tweeter

|  |  |
|---|---|
| Figure 1. A) Image of the Tweeter DM. The 1024 actuator device from Boston Micromachines Corporation has a reflective gold coating and a working aperture of 9.8 mm. | B) Image of the Woofer DM, the Alpao low-speed DM52-25. This DM will also function as the tip-tilt corrector. The device has a working aperture 15 mm and is coated with protected silver. |

| Characteristic | BMC 1K MEMS DM (Tweeter) | Alpao Low-speed DM52-25 (Woofer) |
|---|---|---|
| Number of actuators | 1024 | 52 |
| Pitch | 340 mm | 2.5 mm |
| Flat shape surface figure error (Measured at the LAO) | Expecting ~ 10 nm RMS* | < 7 nm RMS |
| Wavefront tip/tilt stroke | NA | +/- 50.0 mm Peak-to-Valley |
| Hysteresis | < 1 nm | < %1 |
| Bandwidth | > 60 kHz | > 250.0 Hz |
| Working aperture | 9.8 mm | 15 mm |
| Coating | Gold | Protected silver |

Table 1.0 Deformable mirror characteristics of the woofer and tweeter DMs for the ShaneAO system upgrade. *Several Boston Micromachines Corporation MEMS DMs have been measured at the LAO and were found on average to have an RMS surface flatness of roughly 10 nm. We expect similar results from our new mirror, though we not have yet performed this test with this MEMS DM.

Andrew Norton, Don Gavel, Renate Kupke, Marco Reinig, Srikar Srinath, and Daren Dillon, "Performance assessment of a candidate architecture for real-time woofer-tweeter controllers: Simulation and experimental results," **SPIE Photonics West**, 2013.

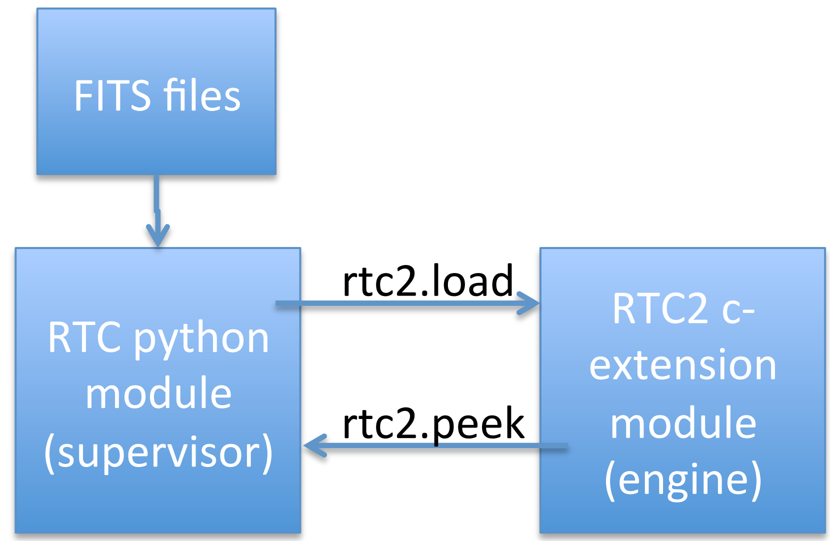
# RTC Software

# Modules…

- `RTC2` – the rtc engine, written in C, dynamically linked to `RTC`
- `RTC` – the supervisor, written in Python
- File System
- `WFS` – definition of mode sets, creation of reconstructor matrix
- IDL scripts - `paramgen.pro`
- GUI

# Parameter Preparation and Handling

- "Parameters" (matrix, offsets, and limits,…) are prepared outside the RTC.
- All calibration operations (flat, dark, refcent…) are done outside the RTC
- Parameters stored in FITS files
- Parameters loaded through the Python-C extension
- Python-C extension "peeks" at the RTC data via pointers, and displays diagnostics as you like



# Preparing the Reconstructor

# A Mathematical Framework for the Reconstructor

- Assume the wavefront is “fittable” by a set of modes

$$\phi_b(x) = \sum_i c_i b_i(x) \qquad e_\phi = \phi(x) - \phi_b(x)$$

- $\{c_i, b_i(x)\}$ is any vector space. These “internal” modes (basis functions $b_i(x)$) don’t have to be orthonormal, can mix pieces of mode sets (Zernike, Fourier, DM modes, etc.). Solution is restricted to Hilbert subspace spanned by the basis functions
- The Shack-Hartmann wavefront sensor responds to the wavefront as

$$s_j = \int w_j(x) \nabla\phi(x) dx + n_j; \quad i \in \{subaps\}$$

- And thus is related to the mode coefficients as

$$\mathbf{s} = \mathbf{Hc} + e_s$$

- Finally, we assume that the deformable mirrors can produce the mode set, with some fitting error, where $r_i(x)$ are actuator influence functions

$$\phi_b(x) = \sum_i a_i r_i(x) + e_{fit}$$

- The actuator command vector is related to the mode coefficients by

$$\mathbf{a} = \mathbf{Ac}$$

# Internal Mode Space Formulation is General

- **Fourier reconstructor** is in this formulation:
  - Fourier-equivalent matrix form:
  - Fourier "reconstructor" is (mathematically) subsumed in the internal mode space concept
  - Modal weights are also subsumed
  - -> An alternative *implementation* is needed to get at *signals* in Fourier space

$$\mathbf{s} = \mathbf{F}\left[i\mathbf{k}\right]\tilde{\phi} = \mathbf{Hc}$$

$$\mathbf{a} = \mathbf{F}\tilde{\phi} = \mathbf{Ac}$$

$$\mathbf{F} = \left[e^{i\mathbf{kx}}\right]$$

- **Poke-matrix reconstructor** is in this formulation.
  - The basis set can be the actuator influence functions (c=a; A=I, and H = the familiar "poke" matrix). Not recommended for ShaneAO 8x and 16x modes.
- **Zernike mode reconstructors** are in this formulation

# Generating the Reconstruction Matrix

- The reconstructor strives to find the mode coefficients given the sensor readings, then set the actuators accordingly

$$\mathbf{H}^{\dagger} = \mathbf{P}\mathbf{H}^{T}(\mathbf{Q} + \mathbf{H}\mathbf{P}\mathbf{H}^{T})^{-1}$$

$$\mathbf{R} = \mathbf{A}\mathbf{H}^{\dagger}$$

- Regularized pseudo-inverse of H (Waffle Suppression, Minimum Variance Estimation)
- *Only* **R** *is used by the rtc* (#3 on viewgraph 19)
- Matrix sub-blocks are used to incorporate woofer mode de-projection and filtering…

# ShaneAO DMs Mode Spaces


Figure 1. A) Image of the Tweeter DM. The 1024 actuator device from Boston Micromachines Corporation has a reflective gold coating and a working aperture of 9.8 mm.

B) Image of the Woofer DM, the Alpao low-speed DM52-25. This DM will also function as the tip-tilt corrector. The device has a working aperture 15 mm and is coated with protected silver.

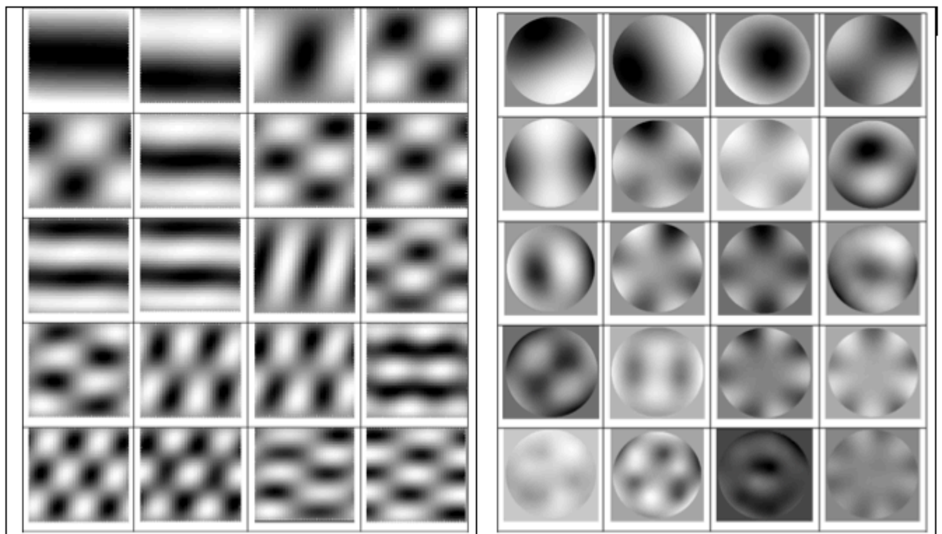


Figure 6. A) First 20 orthonormal tweeter modes as viewed over the 9.8 mm working aperture of the MEMS 1K DM. Mode 1 is shown in the upper left, and the mode structure continues left to right, top to bottom.

B) First 20 orthonormal woofer modes of the woofer DM. Mode 1 is shown in the upper left. The nodes of each mode trend to the perimeter as the mode number increases leaving less correction over the full pupil.

Andrew Norton, Don Gavel, Renate Kupke, Marco Reinig, Srikar Srinath, and Daren Dillon, "Performance assessment of a candidate architecture for real-time woofer-tweeter controllers: Simulation and experimental results," **SPIE Photonics West**, 2013.

# Dealing with the Woofer-Tweeter Pair

**Woofer and Tweeter Mode Spaces**

The woofer and tweeter respond to linear combinations of their mode sets according to

$$\phi_w(x) = \sum_i c_{w_i} b_{w_i}(x) \qquad \phi_t(x) = \sum_i c_{t_i} b_{t_i}(x)$$

$$c_w = M_w \int b_{w_i}(x)\phi(x)dx \qquad c_t = M_t \int b_{ti}(x)\phi(x)dx$$

$$M_{\left[\begin{smallmatrix} t \\ w \end{smallmatrix}\right]} = \left[\int b_i(x) b_j(x) dx\right]^{-1}$$

If the tweeter has an arbitrary phase, within its Hilbert supspace, then it can be projected to the woofer:

$$c_w = M_w \int b_{w_i}(x) \sum_j c_{t_j} b_{t_j}(x) dx$$

$$= M_w C_{wt} c_t$$

where

$$C_{wt} = \int b_{w_i}(x) b_{t_j}(x) dx$$

# Woofer and Tweeter mode spaces in matrix form

---

**Hilbert Matrices**

We have various quantities that can be more compactly and simply expressed as Hilbert space matrices (avoiding integral signs)

$$M_w = \left[B_w B_w^T\right]^{-1} \quad M_t = \left[B_t B_t^T\right]^{-1} \quad C_{wt} = B_w B_t^T$$

$$B_t = \begin{bmatrix} b_{t_0}(x) \\ b_{t_1}(x) \\ \ldots \end{bmatrix}$$

and the phases themselves

$$\phi_t(x) = B_t^T c_t \quad \phi_w(x) = B_w^T c_w$$

---

# Least-squares fits, projections, and cross correlations

**Least Squares Fits of Woofer to Tweeter Modes**

$$M_w C_{wt} = \left[B_w B_w^T\right]^{-1} B_w B_t^T = B_w^\dagger B_t^T$$

$$M_t C_{wt}^T = \left[B_t B_t^T\right]^{-1} B_t B_w^T = B_T^\dagger B_w^T$$

$$M_t C_{wt}^T M_w C_{wt} = B_t^\dagger B_w^T B_w^\dagger B_t^T$$

# Projection to woofer modes, followed by de-projection back to tweeter, is kosher

$$M_t C_{wt}^T M_w C_{wt} c_t = B_t^\dagger B_w^T B_w^\dagger \underbrace{B_t^T c_t}_{\phi_t(x)}$$

Then if

$$\phi_t(x) = \phi_w(x)$$

that is,

$$B_t^T c_t = B_w^T c_w$$

(which it is for the woofer-fittable part) then

$$M_t C_{wt}^T M_w C_{wt} c_t = B_t^\dagger B_w^T B_w^\dagger \underbrace{B_w^T c_w}_{\phi_w(x)}$$

Since $B_w^\dagger B_w^T = I$ and $B_t^\dagger B_t^T = I$ this collapses to

$$M_t C_{wt}^T M_w C_{wt} c_t = B_t^\dagger B_w^T c_w = B_t^\dagger B_t^T c_t = c_t$$

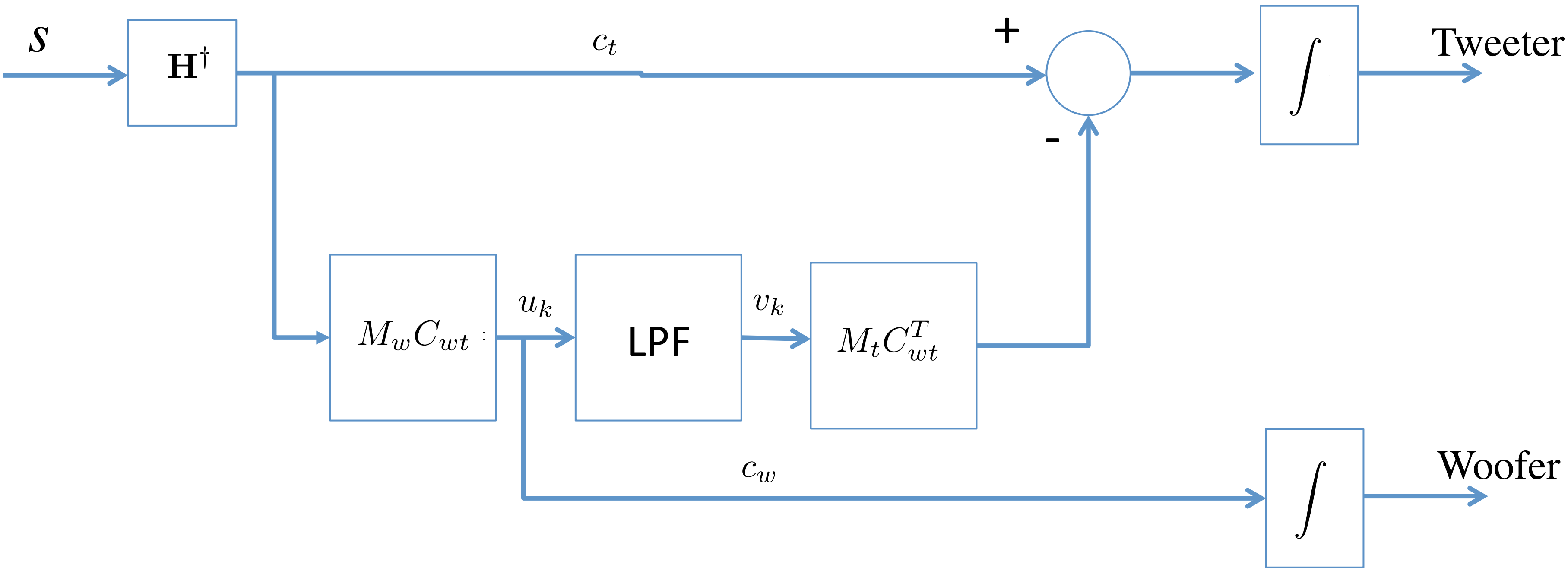
which proves that the projection of woofer modes on the tweeter, followed by projecting them back on to the tweeter again, is an identity process - so long as the woofer modes are in the tweeter Hilbert space.

---

…so long as the woofer modes are in the tweeter's Hilbert space

# Woofer-Tweeter controller

With this in mind, we build up a (conceptual) control flow diagram, where the control is split between woofer and tweeter using projections from tweeter space to woofer space. We don't exactly remove the woofer modes from the tweeter, instead we remove them only after low-pass filtering, because the woofer takes some time to respond.




LPF = $H_L(z^{-1}) = \frac{z^{-1}(1-\alpha)}{1-\alpha z^{-1}}$

$$\int = \frac{1}{1-z^{-1}}$$

# The Woofer modes on the Tweeter are Low-Pass Filtered – so we need a filter expressed in state space

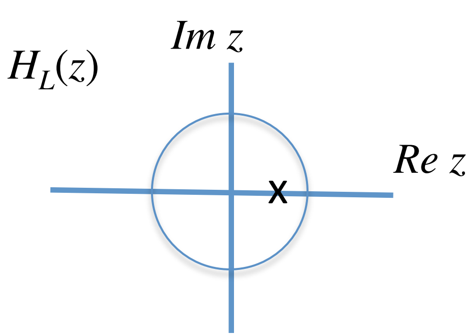
The simplest filter is single-pole:

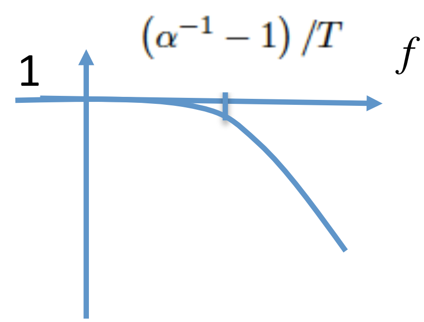
$$v_k = \alpha v_{k-1} + (1-\alpha)u_{k-1}; \quad |\alpha| < 1$$

This has a $z$-transform transfer function

$$H_L(z^{-1}) = \frac{z^{-1}(1-\alpha)}{1-\alpha z^{-1}}$$

which has a pole at $z = \alpha$. The magnitude of the transfer function vs real frequency is shown in the figure, where we've substituted $z = e^{sT} = e^{i2\pi fT}$



# The whole woofer-tweeter control law written in state-space form

We write the negative-feedback control loop in its state-space form:

$$v_k = \alpha v_{k-1} + (1-\alpha)u_{k-1} = \alpha v_{k-1} - (1-\alpha)M_w C_{wt} H^\dagger s_k$$

$$a_{t_k} = a_{t_{k-1}} - R s_k - A_t M_t C_{wt}^T v_{k-1}$$

$$a_{w_k} = a_{w_{k-1}} - A_w M_w C_{wt} H^\dagger s_k$$

Or, in matrix form:

$$\begin{bmatrix} a_t \\ a_w \\ v \end{bmatrix}_k = \begin{bmatrix} I & 0 & -A_t M_t C_{wt}^T \\ 0 & I & 0 \\ 0 & 0 & \alpha \end{bmatrix} \begin{bmatrix} a_t \\ a_w \\ v \end{bmatrix}_{k-1} - \begin{bmatrix} R \\ A_w M_w C_{wt} H^\dagger \\ (1-\alpha) M_w C_{wt} H^\dagger \end{bmatrix} s_k$$

# …and, with more matrix form manipulation…

which can be written in the form

$$\begin{bmatrix} a_t \\ a_w \\ v \end{bmatrix}_k = \begin{bmatrix} a_t \\ a_w \\ v \end{bmatrix}_{k-1} + \underbrace{\begin{bmatrix} -R & -A_t M_t C_{wt}^T \\ -A_w M_w C_{wt} H^\dagger & 0 \\ -(1-\alpha) M_w C_{wt} H^\dagger & -(1-\alpha) \end{bmatrix}}_{R'} \underbrace{\begin{bmatrix} s_k \\ v_{k-1} \end{bmatrix}}_{s'_k}$$

or

$$\boxed{a_k = a_{k-1} + R' s'_k}$$

This boxed equation is what gets implemented in the c-extension module **rtc2**. We just have to provide $R'$, which would be calculated in the support processing scripts, then loaded using the supervisor module **rtc**.

This is all there is to it folks!

# Stability Analysis

How the WFS responds to DM changes:

$$s_k = H_t A_t^{\dagger} a_{t_{k-1}} + H_w A_w^{\dagger} \bar{a}_{w_{k-1}}$$

How the Woofer responds slower:

$$\bar{a}_{w_k} = \beta \bar{a}_{w_{k-1}} + (1-\beta) a_{w_{k-1}}$$

This allows us to write the closed-loop matrix equation

$$\begin{bmatrix} a_t \\ a_w \\ v \\ \bar{a}_w \end{bmatrix}_k = \begin{bmatrix} I & 0 & -A_t M_t C_{wt}^T & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & \alpha & 0 \\ 0 & (1-\beta) & 0 & \beta \end{bmatrix} \begin{bmatrix} a_t \\ a_w \\ v \\ \bar{a}_w \end{bmatrix}_{k-1} - \begin{bmatrix} A_t H_t^{\dagger} \\ A_w M_w C_{wt} H_t^{\dagger} \\ (1-\alpha) M_w C_{wt} H_t^{\dagger} \\ 0 \end{bmatrix} \begin{bmatrix} H_t A_t^{\dagger} & 0 & 0 & H_w A_w^{\dagger} \end{bmatrix} \begin{bmatrix} a_t \\ a_w \\ v \\ \bar{a}_w \end{bmatrix}_{k-1}$$

or

$$\begin{bmatrix} a_t \\ a_w \\ v \\ \bar{a}_w \end{bmatrix}_k = \begin{bmatrix} I - A_t H_t^{\dagger} H_t A_t^{\dagger} & 0 & -A_t M_t C_{wt}^T & -A_t H_t^{\dagger} H_w A_w^{\dagger} \\ -A_w M_w C_{wt} H_t^{\dagger} H_t A_t^{\dagger} & I & 0 & -A_w M_w C_{wt} H_t^{\dagger} H_w A_w^{\dagger} \\ -(1-\alpha) M_w C_{wt} H_t^{\dagger} H_t A_t^{\dagger} & 0 & \alpha & -(1-\alpha) M_w C_{wt} H_t^{\dagger} H_w A_w^{\dagger} \\ 0 & (1-\beta) & 0 & \beta \end{bmatrix} \begin{bmatrix} a_t \\ a_w \\ v \\ \bar{a}_w \end{bmatrix}_{k-1}$$

More compactly:

$$a'_k = T a'_{k-1}$$

Stability is assured if

$$|\lambda(T)| < 1$$

that is the eigenvalues of T are all inside the unit circle.

Stable if all eigenvalues are inside unit circle

# Stability can be enforced

Stability can be enforced if we do two things:

1) use a **leaky integrator** for the actuators, i.e. replace the $I$'s in the first matrix by $\gamma I$, where $0 < \gamma < 1$.

2) multiply the reconstructor matrix by a **feedback gain**:

$$H^{\dagger} \to g H^{\dagger}$$

where $g$ is made sufficiently small. As $g \to 0$ the eigenvalues of $T$ converge to three degenerate eigenvalues, $\gamma$, $\alpha$, and $\beta$ which are all less than 1 in magnitude. Therefore there is a range of gains $g > 0$ where the system is stable. The response time of the system to input disturbance is

$$\tau_r = -T / \ln |\lambda_{\max}|$$

where $T$ is the sample period.

# For Insight: Let's Look at the Mode Coefficients

For further analysis it is instructive to note that only the mode sets selected by $A_t$ and $A_w$ are dynamically affected by feedback. The orthogonal parts of the Hilbert space are in the null space of the reconstructor, so they are neither excited by the disurbance nor fed back but are simply left to decay at a rate set by $\gamma$ without any affect on long-term stability. If we carry just the selected mode coefficients in our analysis state-vector, the stability equation is:

$$\begin{bmatrix} c_t \\ c_w \\ v \\ \bar{c}_w \end{bmatrix}_k = \begin{bmatrix} \gamma - gH_t^\dagger H_t & 0 & M_t C_{wt}^T & -gH_t^\dagger H_w \\ -gM_w C_{wt} H_t^\dagger H_t & \gamma & 0 & -gM_w C_{wt} H_t^\dagger H_w \\ -(1-\alpha) g M_w C_{wt} H_t^\dagger H_t & 0 & \alpha & -(1-\alpha) g M_w C_{wt} H^\dagger H_w \\ 0 & (1-\beta) & 0 & \beta \end{bmatrix} \begin{bmatrix} c_t \\ c_w \\ v \\ \bar{c}_w \end{bmatrix}_{k-1}$$

Feedback Dynamics of the mode coefficients

# Mode Spaces Decouple…

We now make some reasonable approximations to help further simplify the analysis. First, assume that the reconstructor obeys

$$H_t^\dagger H_t \approx I$$

Also, assume that the modes of the woofer match exacly a subset of modes of the tweeter, and furthermore, that the modes in this set are orthonormal. Then

$$C_{wt} = \begin{bmatrix} I_{n_w} & 0 \end{bmatrix} \qquad H_t^\dagger H_w \approx \begin{bmatrix} I_{n_w} \\ 0 \end{bmatrix}$$

and

$$M_w = I_{n_w} \qquad M_t = I_{n_t}$$

where $n_w$ is the number of controlled modes on the woofer and $n_t$ is the number of controlled modes of the tweeter. Then the stability equation is

$$\begin{bmatrix} c_t \\ c_w \\ v \\ \bar{c}_w \end{bmatrix}_k = \begin{bmatrix} \gamma - g & 0 & \begin{bmatrix} I_{n_w} \\ 0 \end{bmatrix} & -g \begin{bmatrix} I_{n_w} \\ 0 \end{bmatrix} \\ -g \begin{bmatrix} I_{n_w} & 0 \end{bmatrix} & \gamma & 0 & -g \\ -(1-\alpha) g \begin{bmatrix} I_{n_w} & 0 \end{bmatrix} & 0 & \alpha & -(1-\alpha) g \\ 0 & (1-\beta) & 0 & \beta \end{bmatrix} \begin{bmatrix} c_t \\ c_w \\ v \\ \bar{c}_w \end{bmatrix}_{k-1}$$

# Mode Spaces Decouple Into Shared and Tweeter-Only Modes

The dynamics separate into two independent subspaces, one associated with the modes shared by woofer and tweeter, and ones associated with tweeter modes not being sent to the woofer. That is

$$\begin{bmatrix} c_{t\in w} \\ c_w \\ v \\ \bar{c}_w \end{bmatrix}_k = \begin{bmatrix} \gamma - g & 0 & -1 & -g \\ -g & \gamma & 0 & -g \\ -(1-\alpha)g & 0 & \alpha & -(1-\alpha)g \\ 0 & (1-\beta) & 0 & \beta \end{bmatrix} \begin{bmatrix} c_{t\in w} \\ c_w \\ v \\ \bar{c}_w \end{bmatrix}_{k-1}$$

for the shared modes, and

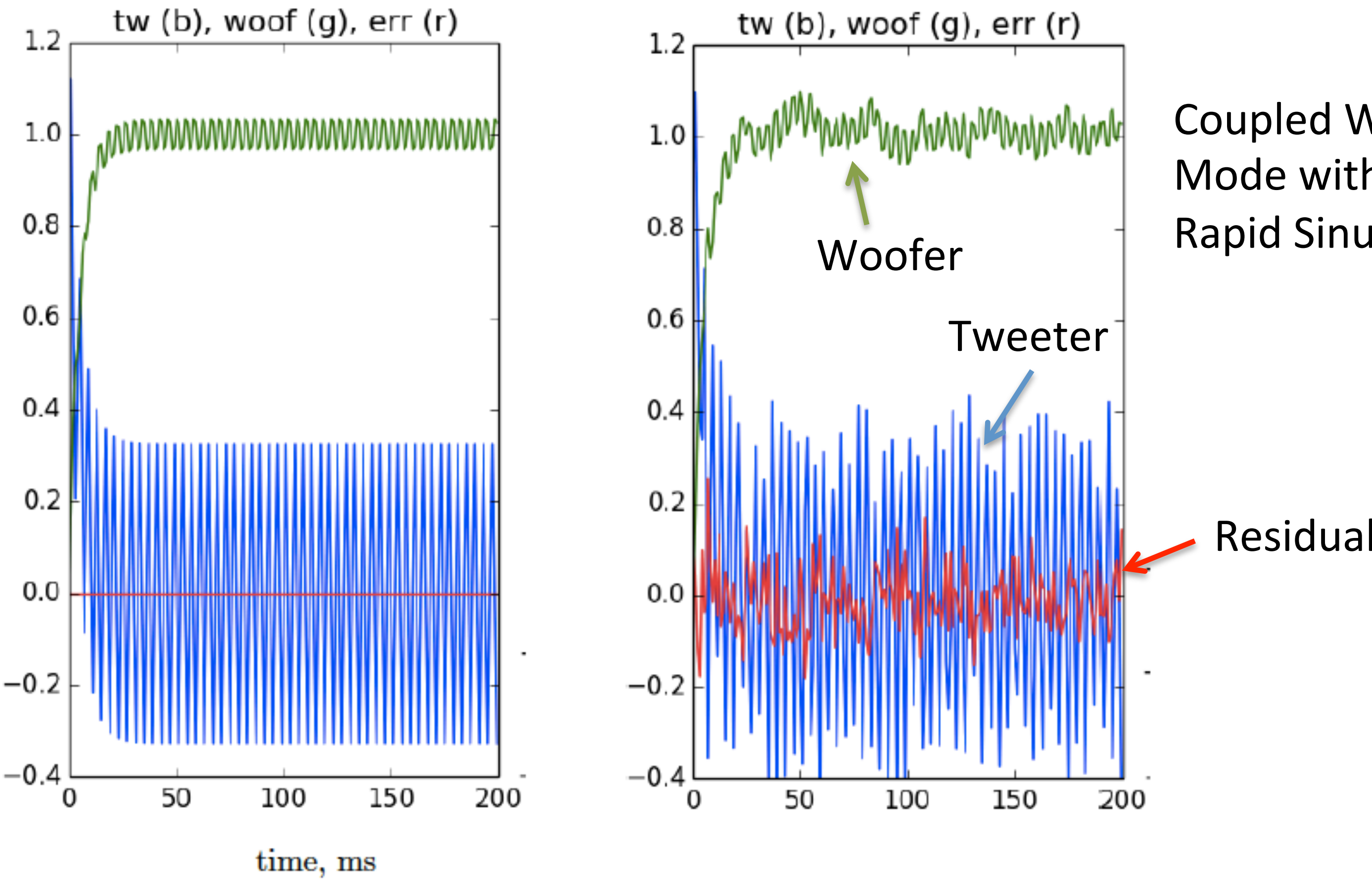
Shared modes: 4-state

$$\left[c_{t\notin w}\right]_k = (\gamma - g)\left[c_{t\notin w}\right]_{k-1}$$

for modes isolated to the tweeter.

Tweeter-only modes: scalar-state

# Simulation Results:
# Everything is Stable and Behaves as Expected




Coupled Woofer-Tweeter Mode with a Step Function + Rapid Sinusoid Disturbance

**Figure 1** A simulation of the $c_{t\in w}$ and $\bar{c}_w$ states in response to disturbance of a unit step plus sinusoid of magnitude 0.3 at 250 Hz. Left: with zero measurement noise, right, with 0.07 rms measurement noise. The simulation parameters are $\alpha = 0.82$, $\beta = 0.82$, $\gamma = 1$, $g = 1$.

# RTC module

# RTC processing steps

- Done in serial by the RTC engine (`RTC2` module), written in C-language:

1. Map pixels to subaps (indirect map)
2. Centroid
3. Matrix-multiply
4. Accumulate/Limit
5. Push to DM through indirect map

$$\mathbf{s}_i = \mathbf{W}\mathbf{p}_i; i \in subaps$$

$$\partial\mathbf{a} = -\alpha\mathbf{a} + \beta\mathbf{R}\mathbf{s}$$

- Coding takes advantage of BLAS routines (cblas_dgemv) to optimize/ parallelize linear algebra steps.
- Timing tests show no need to overlap operations of multiple frame steps. Gets done in under 660 us, even in 30x mode.

# RTC processing code

17 lines of code…

```
# This simulates one step of the real-time control loop, given the current parameters
def oneStep(self):

    # wfs camera (one would use i_map instead of u_map with the real interlaced camera data)
    pix = (self.wfs[self.u_map] - self.wfs_background) * self.wfs_flat

    # centroider
    wx = self.centWts[0,:]
    wy = self.centWts[1,:]
    wi = self.centWts[2,:]
    for k in range(self.ns):
        p = pix[k*25:(k+1)*25]
        x = dot(p,wx)
        y = dot(p,wy)
        i = dot(p,wi) + 1.
        self.s[k] = x/i
        self.s[k+self.ns] = y/i
    self.s[0:2*self.ns] -= self.s_ref

    # reconstructor
    self.da = dot(self.cm,self.s)

    # integrator
    a = (self.a - self.a0)*self.integrator_bleeds + self.a0 + self.da
    self.a = clip(a,self.a_limit[0,:],self.a_limit[1,:])

    self.buf[self.tweeter_map] = self.a[0:1024]
    self.woof = self.a[1024:1024+self.na_woof]
```

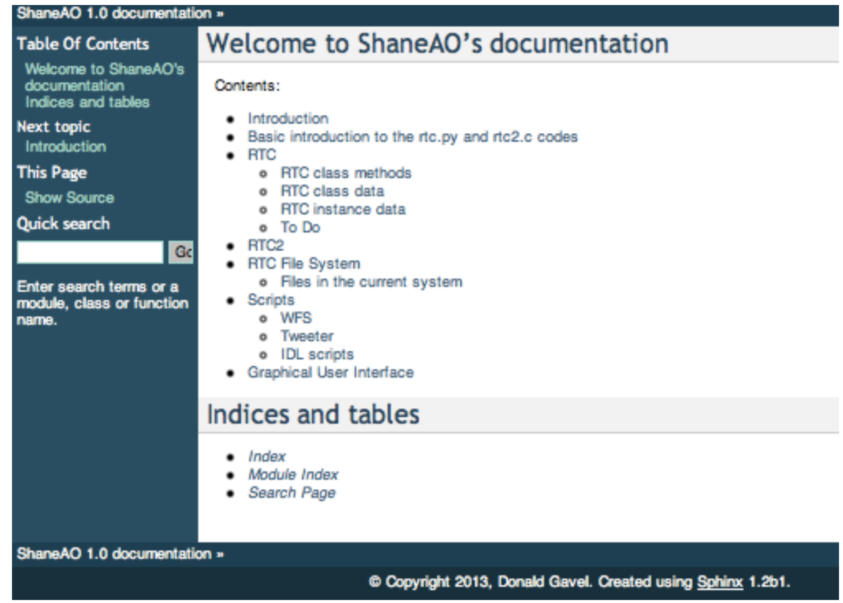
…just kidding! This is the simulator in rtc.py.

But even it runs at ~ hundred hz!

# Documentation

# Documented Modules…

- RTC2 – the rtc engine
- RTC – the supervisor
- File system
- WFS – definition of mode sets, creation of reconstructor matrix
- IDL scripts - paramgen.pro
- GUI – (not yet..)



# On-line Docs

## (example of HTML *Sphinx* auto-docs)





Quick search

Go

Enter search terms or a module, class or function name.

# RTC

## RTC class methods

*class* `rtc.rtc`(*mode*)
RTC means "Real-Time Controller."

This class implements the python interface to the real-time engine.

The work flow logic:

- initialize the supervisor. - it reads in the parameters and puts the rtc in go state, open loop
- a call to open_loop saves the closed loop gain and opens the loop by setting the gain to zero
- another call to open_loop while in the open state does not destroy the saved closed loop gain
- a call to close_loop sets the loop gain to the saved gain. if this gain is not zero, the loop is closed
- a call to set_gain closes the loop if the gain is non-zero
- a call to set_gain with zero gain opens the loop, but it does not save the last gain
- there is a default gain. restore it with a call to set_gain('default').
- the default gain is viewable as instance variable defaultGain
- if you want to change the gain without closing the loop, modify savedGain
- you can also modify the defaultGain

Example start up and run code:

```
u = rtc('16x')
u.open_loop()
u.set_gain(10.) # this also closes the loop
u.set_gain(5.)  # this can be done on the fly
```

Example system modification cycle:

```
# execute codes to generate and store new matrices (using module wfs, etc.)
w = wfs('16x')
w.Hmatrix()
w.save()
# load the new parameters into rtc and go
u.load()
u.close_loop()
```

`close_loop`()
Close the AO loop

`go`()
Start or resume the controller engine.

`load`()
Load tells the interface to read the controller definition files, associated with self.mode, into the real-time controller c-extension's memory.

As a convenience, the definitions are also assigned to instance variables within the rtc object as well.

`manyStep`(*nSteps*)
Run the rtc simulator many steps.

`oneStep`()
The interface has its own RTC simulator. This method runs one step of it. This is handy for diagnostics as the rtc engine should produce results identical to the simulator.

`open_loop`()
Open the AO loop

`set_gain`(*gain*)
Set the gain of the real-time controller.

`manyStep`(*nSteps*)
Run the rtc simulator many steps.

`oneStep`()
The interface has its own RTC simulator. This method runs one step of it. This is handy for diagnostics as the rtc engine should produce results identical to the simulator.

`open_loop`()
Open the AO loop

`set_gain`(*gain*)
Set the gain of the real-time controller.

`status`()
Report the current AO control system state, including running state of the c-exension module, and the loop status and gain

`stop`()
Stop the controller engine (computations halted)

## RTC class data

`rtc.pdict8x`

`rtc.pdict16x`

`rtc.pdict30x`
These are the dictionaries that map rtc variable names to their FITS files.

## RTC instance data

An rtc contains instance variables for every parameter that is loaded from FITS files, plus a few internal ones of its own. Here are some of the important ones:

`self.gain`
The gain of the control loop. It multiplies controlMatrix.

`self.controlMatrix`
The control matrix, as loaded from the FITS file.

`self.cm`
The control matrix after it is multiplied by the gain. This is loaded into rtc2.

`self.mode`
The string '8x', '16x', or '30x' depending on the wavefront sensing mode.

`self.loop`
Loop state - either 'open' or 'closed'

# RTC System Status

Present Status:

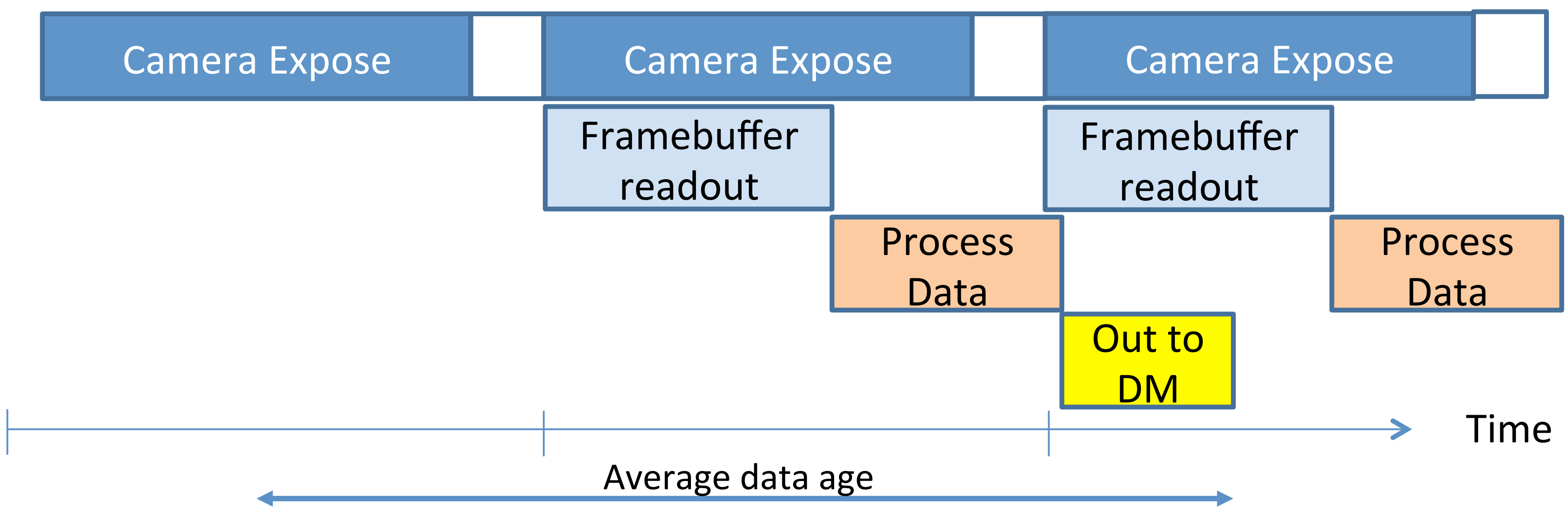
- Low, Mid, and Support: These are mostly done in Python and c-extensions now, but some scripts still in IDL
- No work has been done on the GUI
- RTC2 has all risks retired. Timing tests passed, up to 30x
- Code maintenance and doc systems in place.

Surprises:

- All VMMs test passed; exceed 1.5kHz frame rate
- No kernel modules – no need for “RT” Linux
- The 24 CPU machine is not the fastest we have (!) (i7s doing better than Xeons)
- BLAS doing 2-3x better than “hand coded.” Surprising trade of || processing and pipeline

# “Bare minimum” RTC engine requirement

- WFS Cam readout
  - 1kHz “frame rate”: 1 ms allocated roughly as follows
    - 985 us expose
    - 15 us frame transfer
  - Camera collecting photons the majority of time (98.5% duty cycle)
- DM output
  - Get this out by the time average data age = 1.5 ms



# How to do wind-predictive control?

- 30x mode only?
- Wind measurement algorithm
  - Probably implemented in supervisor, or separate thread
    - not real-time critical
    - uses telemetry data
  - Decimate(?) Anti-alias filter(?) the raw data
- RT Wind-Blown wavefront predictor
  - Load new matrices and proceed with VMM?

  Or

  - Code a Fourier version of the engine and “Fourier-shift”