```julia
"""
    coadd(sc, D)

Co-add a sampled up-the-ramp datacube into a SIRSCore.

    Parameters: sc::SIRSCore
                  A SIRSCore struct
                D::Array{Float64,3}
                  An HxRG datacube
"""
function coadd!(sc::SIRSCore, D::Array{Float64,3})

    # Compute residuals by fitting and subtracting a straight line.
    # A bit of testing shows that using MKL to do the matrix multiplications is
    # faster than using mapslices. In other contexts, I have found
    # that using MKL is faster than tensor operations. Therefore,
    # overwrite D so that it will work with MKL.
    D = permutedims(D, (3,1,2)) # This is expensive, on Racy ~14 seconds but
                                # done only once per exposure
    Δ = D - reshape(sc.L_x_Linv * reshape(D, (sc.naxis3,sc.naxis1*sc.naxis2)),
            (sc.naxis3,sc.naxis1,sc.naxis2))

    # Work in frames
    for z in 1:sc.naxis3

        # Pick out one frame
        frm = Δ[z,:,:]

        # Get a copy of the good pixel mask. Make a copy so
        # as not to mess it up.
        gdpx = copy(sc.gdpx)

        # Compute incomplete Fourier transforms of reference pixels.
        # Note added 2/21/22. I tried parallelizing these two lines on Racy.
        # There was no speed improvement. The overheads associated with starting
        # the additional threads were larger than the improvement.
        # From timing the code, these are actually very fast.
        ℓ = inc_rfft(sc.SFT, reshape(frm[1:sc.rb,:], :))
        𝓇 = inc_rfft(sc.SFT, reshape(frm[end-sc.rb+1:end,:], :))

        # Loop over outputs
        Threads.@threads for op in 1:sc.nout

            # Get column range for this output
```

```
c1 = (op-1) * sc.xsize + 1
c2 = c1 + sc.xsize - 1
crng = c1:c2

# Get data for this output.
#   1) We overwrite all elements, so it is faster to leave this
#      uninitialized.
#   2) Include NROH additional columns for the new row overhead
#      to eventually hold new row overhead
#   3) Flip axis 1 of even numbered columns
d = Array{Float64, 2}(undef, (sc.xsize+sc.nroh,sc.ysize))
if mod(op,2) != 0
    d[1:sc.xsize,:] = frm[crng,:]             # Don't flip
else
    d[1:sc.xsize,:] = frm[crng,:][end:-1:1,:] # Flip
end

# Get the good pixel map for this output flipping
# axis 1 as necessary
_gdpx = BitArray{2}(Array{Bool,2}(undef, (sc.xsize,sc.ysize)))
if mod(op,2) != 0
    _gdpx = gdpx[crng,:]
else
    _gdpx = gdpx[crng,:][end:-1:1,:]
end

# Find and flag transients. This only makes sense
# in the regular pixels. Known bad pixels are already
# flagged. Trim thresholds are defined in SIRS.jl.
if (op==1) || (op==32)
    regpix_mask = sc.regpix_edge
else
    regpix_mask = sc.regpix_middle
end
regular_pixels = d[1:sc.xsize,:][regpix_mask]
good_pixels = _gdpx[regpix_mask]
sorted = sort(regular_pixels[good_pixels])
nrej = Int64(round(TRIM_HARD * length(sorted))) # Figure out how
                                                # many to trim
                                                # on either side
min_good = sorted[nrej]
max_good = sorted[end-nrej+1]
good_pixels[regular_pixels .<= min_good] .= 0
good_pixels[regular_pixels .>= max_good] .= 0
```

```julia
    _gdpx[regpix_mask] = good_pixels

    # Output level quality control.
    # Compute the mean and standard deviation of remaining good pixels.
    # Use this to find any anomalous behavior in this output and frame.
    # Also compute the mean since it will be useful later.
    here = d[1:sc.xsize,HXRG_RB+1:end-HXRG_RB]
    μ = mean(here[_gdpx[:,HXRG_RB+1:end-HXRG_RB]])
    # This seems to cause problems...
    # if ! ((GD_OP_MU_MIN .< μ) * (μ .< GD_OP_MU_MAX))
    #     println("Skipping frame ", z, " output ", op, " μ = ", μ)
    #     flush(stdout)
    #     continue
    # end
    σ = std(here[_gdpx[:,HXRG_RB+1:end-HXRG_RB]])
    if ! ((GD_OP_SIG_MIN .< σ) * (σ .< GD_OP_SIG_MAX))
        println("Skipping frame ", z, " output ", op, " σ = ", σ)
        flush(stdout)
        continue
    end

    # Get the (robust) mean reference row value.
    # From looking at a lot of data, the rows selected here tend to be
    # most indicative of the regular pixels in Roman H4RGs.
    μ_refrows = mean(trim(reshape((d[1:sc.xsize,
                    sc.naxis2-2:sc.naxis2-1]), :),
                       prop=TRIM_HARD))

    # Replace known bad pixels and transients with the mean
    # of all good pixels in the same row.
    for y in 1:sc.ysize
        # Only touch rows that contain bad pixels
        if !(0 ∈ _gdpx[:,y])
            continue
        end
        here = d[1:sc.xsize,y]
        here[_gdpx[:,y] .== 0] .= mean(here[_gdpx[:,y] .== 1])
        d[1:sc.xsize,y] = here
    end

    # Fill overhead columns by mirroring
    roi = d[end-2sc.nroh+1:end-sc.nroh,:]
    d[end-sc.nroh+1:end,:] = roi[end:-1:1,:]
```

```
            # Go to vectors. View the reference pixel stream as an xy-plot.
            # The x-axis is pixel index and the y-axis is signal in DN.
            d_vec = reshape(d,:)

            # Compute the FFT
            𝓃 = rfft(d_vec)

            # Keep just frequencies of interest. These are less than Nyquist
            # on the row rate and within the same frequency interval of Nyquist.
            𝓃 = cat(𝓃[1:sc.naxis2÷2+1],𝓃[length(𝓃)-(sc.naxis2÷2-1):end],
                       dims=1)

            # Coadd sums for frequencies > 0 Hz
            sc.ℕ[:,op] .+= real.(𝓃 .* conj(𝓃))
            sc.𝕃[:,op] .+= real.(ℓ .* conj(ℓ))
            sc.ℝ[:,op] .+= real.(𝓇 .* conj(𝓇))
            sc.𝕏[:,op] .+= 𝓃 .* conj(𝓇)
            sc.𝕐[:,op] .+= 𝓃 .* conj(ℓ)
            sc.ℤ[:,op] .+= 𝓇 .* conj(ℓ)

            # Coadd sums for f = 0 Hz
            sc.R[op]    += μ - μ_refrows
            sc.N[op]    += 1          # Serves as frame counter

        end
    end
end
```