Operations Simulator Control Flow

Opsim

1. Read LSST configuration
2. Establish DB connection
3. Get session ID
    1. Session ID creation (insert into DB increments Session ID)
    2. Track session (writes to opsimcvs DB)
4. Print configuration information
5. Create simulator object (Simulator)
6. Simulator start (looks like run)
7. Close proposals
    1. Opportunity for proposals to do close out activities like writing unfinished sequences to DB

Simulator

constructor

1. Read scheduled downtime
2. Read unscheduled downtime
3. Setup simulation (::setupSimulation)
    1. Create telescope (Instrument)
        - Park it!
    2. Create filters (Filters)
    3. Create astronomical sky model (AstronomicalSky)
    4. Create scheduling data (SchedulingData)
    5. Create weather model (Weather)
    6. Create observation scheduler (ObsScheduler)
    7. Create TAC (proposal marshalling)
4. Set lunation count to 0
5. Set night count to -1
6. Create TimeHistory

start()

1. Run TAC::start
2. Set lastEvent to 0
3. Get next unscheduled and scheduled down times
4. Set t to lastEvent (t is in seconds)
5. Get AstronomicalSky::getTwilightSunriseSunset for t

6. If between sunrise and sunset
    1. Set t to sunset
    2. Set tonight's twilight to sunset twilight
    3. Call AstronomicalSky::getTwilightSunriseSunset for t plus DAY in seconds
7. If t greater than or equal to sunset
    1. Set tonight's twilight to sunset twilight
    2. Call AstronomicalSky::getTwilightSunriseSunset for t plus DAY in seconds
8. If t less than or equal to sunrise
    1. Call AstronomicalSky::getTwilightSunriseSunset for t minus DAY in seconds
    2. Set tonight's twilight to yesterday's sunset twilight
    3. If t is less than yesterday's sunset
        1. Set t to yesterday's sunset
9. Set midnight to half of tonight's sunset twilight plus sunrise twilight
10. Call ::initMoonPhase with midnight
    1. Calculate MJD
    2. Get previous and current phase from AstronomicalSky::getMoonPhase
    3. If previous phase is less than or equal to current phase
        1. Moon is trending full
        2. Else Moon is trending new (full equal False)
        3. Set latest night phase to previous phase
11. While t is less than nRun times YEAR in seconds
    1. If night count is unscheduled or scheduled downtime
        1. Set startDownTime to t
        2. Set startDownNight to night count
        3. Get cloudiness from Weather::getTransparency
        4. While daysDown greater than 0
            1. ::startNight
            2. Set t to sunrise
            3. ::startDay
            4. Set t to sunset
            5. Do stuff I don't quite understand
            6. Decrement daysDown by 1
        5. Get next unscheduled or scheduled downtime
        6. Check for overlapping downtimes
    2. If not downtime then we have regular observing
        1. ::startNight
        2. Weather::getNightClouds
        3. Weather::getNightSeeing
        4. AstronomicalSky::flushCache
        5. While t less than sunrise
            1. Weather::getTransparency

2. If too cloudy
    1. Instrument::Park
    2. Increment t with idleDelay
3. Ok to observe
    1. AstronomicalSky::computeDateProfile
    2. ObsScheduler::suggestObservation
    3. If above returns observation (winner)
        1. Observation time is exposure time plus slew time
        2. Rank defined by observation
    4. Else rank is 0
    5. If rank is not 0
        1. Increment t by observation time
        2. Idle is t minus lastEvent minus observation time
        3. eventTime equals observation time
        4. ObsScheduler::closeObservation with winner
    6. Else
        1. Increment t by idleDelay
        2. eventTime equals idleDelay
    7. Check for end of sunrise/sunset or dawn
    8. Set lastEvent to t
6. Park telescope at end of night
7. ::startDay at sunrise
8. Set t to sunset
9. Set tonight's sunset twilight to sunset twilight
10. Calculate midnight

startNight(date, midnight, sunRise)

1. Increment night count by one
2. AstronomicalSky::computeDateProfile for date
3. AstronomicalSky::computeMoonProfile at midnight
4. If moon trending to full
    1. If phase is decreasing
        1. startNewLunation is True
        2. Add DB record in TimeHistory
        3. If lunationCount greater than 0 and lunationCount modulus 12 is 0
            1. startNewYear is true
            2. Add DB record to TimeHistory
        4. Increment lunationCount by one
        5. moonTrendToFull is false
5. Else

        1. If phase is increasing
            1. moonTrendToFull is true
            2. Add DB record to TimeHistory
    6. Set lastNightPhase to moon percent
    7. Add DB record to TimeHistory
    8. ObsScheduler::startNight

startDay(date)

1. AstronomicalSky::computeDateProfile for date
2. Add DB record to TimeHistory
3. AstronomicalSky::computeMoonProfile for date
4. ObsScheduler::startDay with moon profile

ObsScheduler

constructor(lsstDB, schedulingData, obsProfile, dbTableDict, telescope, weather, sky, filters, sessionID, runSeeingFudge, schedulerConf)

1. Bunch of configuration statements

startNight(dateProfile, moonProfile, startNewLunation, startNewYear, fov, nRun, nightCnt)

1. Instrument::GetMountedFiltersList
2. Loop through all proposals
    1. If proposal.nextNight less than night count
        1. Set NextNight to night count plus proposal.hiatusNights
    2. If proposal is not active move to next proposal
    3. Proposal::updateTargetList
    4. targets.update with fields from above line
    5. Proposal::startNight
    6. If startNewYear is true
        1. Proposals::startNewYear
3. SchedulingData::startNight with date profile

startDay(moonProfile)

1. If NewMoonPeriod
    1. If current phase is greater than NewMoonPhaseThreshold
        1. ::SwapExtraFilterOut
        2. Set NewMoonPeriod true

2. Else
    1. If current phase is less than NewMoonPhaseThreshold
        1. ::SwapExtraFilterIn
        2. Set NewMoonPeriod false
3. Instrument::GetMountedFiltersList
4. Instrument::GetUnmountedFiltersList

closeObservation (winning observation)

1. Instrument::Observe
2. Set information on winning observation
    1. AstronomicalSky::getSunAltAz
    2. AstronomicalSky::computeDateProfile
    3. AstronomicalSky::computeMoonProfileAltAz
    4. AstronomicalSky::getSkyBrightness
    5. Filters::computeSkyBrightnessForFilter
    6. Filters::computeFilterSeeing
    7. AstronomicalSky::getPlanetDistance
3. Add DB record for Observation
4. Add DB record for SlewHistory
5. Add DB record for SlewState (initial and final)
6. Add DB record for SlewMaxSpeeds
7. Add DB records for SlewActivities
8. For each proposal
    1. Proposal::closeObservation
9. Set targetRank(fieldID, filter) to 0

suggestObservation

1. Get date, moon and twilight profiles
2. Get transparency
3. SchedulingData::findNightAndTime
4. If reuseRanking less than zero
    1. Empty dict of targetRank and targetXblk
    2. If recalcSky less than zero
        1. RawSeeing equals Weather::getSeeing
        2. If seeing less than tooGoodSeeingLimit
            1. Seeing equals tooGoodSeeingLimit
        3. Seeing equals seeing times runSeeingFudge
        4. racalcSky equals recalcSkyCount
    3. Set totPotentialTargets to zero

4. For each proposal in proposal list
    1. If proposal is not active, continue
    2. targetObs = Proposal::suggestObs
    3. If no targetObs, continue
    4. Set self.expTime to proposal.exposureTime
    5. propID equals proposal.propID
    6. For obs in targetObs
        1. Get some fields
        2. Check for obs.exclusiveBlock
            1. Set propIDforXblk to propID if True else set to None
        3. If fieldID not in targetRank
            1. targetRank[fieldID] = {filter: rank}
            2. targetXblk[fieldID] = {filter: propIDforXblk}
            3. Increment totPotentialTargets by one
        4. Else
            1. If filter not in tagetRank[fieldID]
                1. targetRank[fieldID][filter] = rank
                2. targetXblk[fieldID][filter] = propIDforXblk
                3. Increment totPotentialTargets by one
            2. Else
                1. targetRank[fieldID][filter] += rank
                2. If propIDforXblk not None
                3. targetXblk[fieldID][filter] = propIDforXblk
5. Notify if totPotentialTargets is zero
6. If totPotentialTargets less than reuseRankingCount
    1. reuseRanking equals totPotentialTargets
7. Else
    1. reuseRanking equals reuseRankingCount
8. Loop through all fields in targetRank
    1. Loop through all filters in field
        1. Skip combo if rank less than zero
        2. Set expTime form self.ExpTime???
        3. slewTime = Instrument::GetDelayForTarget
        4. If slewTime greater than or equal to zero
            1. Calculate slewRank
            2. If slewRank greater than max rank
                1. Set winning information
9. If maxrank greater than zero
    1. winExposureTime *= Filters::ExposureFactor(winFilter)
    2. Create Observation instance
    3. Set extra fields on winning observation

4. If winner.exclusiveBlockRequired
    1. Deep copy winner to exclusiveObs
    2. Set recalcSky and reuseRanking to zero
5. Else
    1. Decrement by one recalcSky and reuseRanking

10. Else
    1. Set recalcSky and reuseRanking to zero
11. Return winner

SchedulingData

constructor(configFile, surveyStartTime, surveyEndTime, astroSky, lsstDB, sessionID)

1. Set config parameters
2. ::initSurvey(surveyStartTime, surveyEndTime)

initSurvey(surveyStartTime, surveyEndTime)

1. t equals surveyStartTime
2. AstronomicalSky::getIntTwilightSunriseSunset(t)
3. If t less than sunrise
    1. AstronomicalSky::getIntTwilightSunriseSunset(t - DAY)
    2. If t less than yesterday's sunset
        1. t equals yesterday's sunset
        2. Set tonight info from yesterday
4. If sunrise greater than or equal to t and t less than sunset
    1. AstronomicalSky::getIntTwilightSunriseSunset(t + DAY)
    2. t equals sunset
    3. Set tonight info from tomorrow
5. If t greater than or equal to sunset
    1. AstronomicalSky::getIntTwilightSunriseSunset(t + DAY)
    2. Set tonight info from tomorrow
6. Calculate midnight
7. Set many empty lists and dictionaries
8. Then set values into lists and dictionaries based on night equal zero
    1. This includes moon, date and twilight profiles
9. ::updateLookAheadWindow

updateLookAheadWindow()

1. Find last night and nights to add

2. Get midnight for last night (before adding)
3. AstronomicalSky::getIntTwilightSunriseSunset(midnight)
4. While last sunset less than surveyEndTime and night less than lookAheadLastNight
    1. Increment night by one
    2. Increment midnight by DAY
    3. AstronomicalSky::getIntTwilightSunriseSunset(midnight)
    4. Calculate new midnight
    5. Add parameters to lists and dictionaries
    6. AstronomicalSky::computeMoonProfile(midnight)
    7. Create lookAheadTimes for night by range(sunset, sunrise, lookAheadInterval)
    8. For date in lookAheadTimes
        1. AstronomicalSky::computeDateProfile(date)
        2. Set empty lists and dictionaries for date
5. Clean out lookahead data from start to current night

findNightAndTime(time)

1. n equals first look ahead night
2. foundNight is False
3. while n less than or equals to last look ahead night and not foundNight
    1. If time less than sunset[n]
        1. t equals sunset[n]
        2. foundNight = True
    2. Elsif sunset[n] less than or equal to time and time greater than or equal to sunrise[n]
        1. t equals time
        2. foundNight = True
    3. Else
        1. Increment n by 1
4. If foundNight
    1. ix equals zero
    2. foundTime is False
    3. while ix less than length of lookAheadTimes and not foundTime
        1. If t greater than lookAheadTimes[n][ix]
            1. Increment ix
        2. Elif ix equal zero
            1. next_time equals lookAheadTimes[n][ix]
            2. foundTime is True
        3. Elif t-lookAheadTimes[n][ix-1] < lookAheadTimes[n][ix]-t
            1. next_time = lookAheadTimes[n][ix-1]
            2. foundTime is True
        4. Else

                1. next_time = lookAheadTimes[n][ix]
                2. foundTime is True
        4. If not foundTime
            1. next_time = lookAheadTimes[n][-1]
        5. Return (n, next_time)
    5. Else
        1. Return None

startNight(dateProfile)

1. nextNight, nextTime = ::findNightAndTime(dateProfile.date)
2. currentNight = nextNight
3. currentTime = nextTime
4. if lookAheadnights[-1] - currentNight less than lookAheadNights
    1. ::updateLookAheadWindow
5. For propID in list_propID
    1. Set dictionaries for propID
    2. ::computeTargetData(nextNight, propID, dictionaries)
    3. Set empty list and dictionaries

updateTargets(propID, dictOfNewFields, maxAirmass, dictFilterMinBrig, dictFilterMaxBrig)

1. Append to list and add key to dictionaries of objects based on propID

computeTargetData(initNight, dictOfNewFields, propID, maxAirmass, dictFilterMinBrig, dictFilterMaxBrig)

1. Sort keys from dictFilterMinBrig (list of Filters)
2. If propID not in list of proposals append to list
3. Make sorted lists from dictOfNewFields, self.dictOfAllFields and self.dictOfActiveFields
4. newfields and new props set to zero
5. For each field in list of NewFields
    1. If field not in AllFields
        1. self.dictOfAllFields[field] = dictOfNewField[field]
        2. self.proposals[field] = propID
        3. Add ProposalField to DB
        4. Increment newfields by one
    2. Else
        1. If propID not in self.proposals[field]
            1. self.proposals[field].append(propID)
            2. Add ProposalField to DB

            3. Increment new props by one
        3. If field not in ActiveFields
            1. self.dictActiveFields[field] = dictOfNewFields[field]
6. Make sorted lists from self.dictOfAllFields and self.dictOfActiveFields
7. For each field in list of AllFields
    1. If field not in self.visibleTime
        1. Create empty dictionary of field
    2. For filter in list of Filters
        1. If filter not in self.visibleTime[field]
            1. Create empty dictionary of field, filter
    3. If propID not in self.visibleTime[field][filter]
        1. Set self.visibleTime[field][filter][propID] to zero
8. For n in range from initNight to lookAheadnights[-1]+1
    1. Set computed and vis to zero
    2. If propId not in self.computedVisible[n]
        1. self.computedVisible[n][propID] = []
    3. For field in list of ActiveFields
        1. If field not in self.computedFields[n]
            1. Get ra, dec from dictOfAllFields[field]
            2. for t in self.lookAheadTimes[n]
                1. Get am, alt, az, pa from AstronomicalSky::airmass(t, ra, dec)
                2. Get br, dist2moon, moonAlt, brprofile from AstronomicalSky::getSkyBrightness(0, ra, dec, alt, dateProfile[t], moonProfile[n], twilightProfile[n])
                3. Set values in dictionaries for t, field
                4. Create empty dictionary self.visible[t][field] = {}
                5. Append field to self.computedFields[n]
                6. Increment computed by one
        2. If propID in self.proposals[field]
            1. If field not in self.computedVisible[n][propID]
                1. For t in self.lookAheadTimes[n]
                    1. self.visible[t][field][propID] = []
                2. For filter in list of Filters
                    1. If self.airmass[t][field] < maxAirmass
                        1. If filter equals u and moonProfile[n][2] > self.NewMoonThreshold
                            1. delta equals zero
                        2. Elif dictFilterMinBrig[filter] < self.brightness[t][field] < dictFilterMaxBrig[filter]
                            1. Append filter to self.visible[t][field][propID]

                2. Add lookAheadInterval to self.visibleTime[field][filter][propID]
            3. Else
                1. delta equals 0
        2. Else
            1. delta equals 0
        3. Increment vis by one
    2. Append field to self.computedVisible[n][propID]?
9. Calculate memory footprint

WeakLensingProp

start

1. Nothing to see here

startNight(dateProfile, moonProfile, startNewLunation, randomizeSequencesSelection, nRun, mountedFiltersList)

1. Call base startNight

suggestObs(dateProfile, n, exclusiveObservation, mindistance2moon, rawseeing, seeing, transparency, sdnight, sdtime)

1. If exclusiveObservation is not None
    1. If exclusiveObservation.fieldID in list of targets
        1. Set list of fieldToEvaluate to exclusiveObservation.fieldID
    2. Else
        1. Create empty list of fieldsToEvaluate
    3. Set numberOfObsToPropose to zero
2. Else
    1. Set list of fieldsToEvaluate to list of targets
    2. Set numberOfObsToPropose to n (n is input).
3. Clear suggestList
4. If length of list of fieldsToEvaluate is greater than zero
    1. If useLookAhead
        1. ::rankAreaDistributionWithLookAhead
    2. Else
        1. ::rankAreaDistribution
    3. Return Proposal::suggestList(numberOfObsToPropose)

closeObservation(observation, obsHistID, twilightProfile)

1. Call base class closeObservation and get obs
2. If obs is not None
    1. Increment visits[obs.filter][obs.fieldId] by one
    2. If above fails, set visits[obs.filter][obs.fieldId] to one
    3. Increment VisitsTonight by one
3. progress = visits[obs.filter][obs.fieldId] / GoalVisitsFieldFilter[obs.filter]
4. Return obs

rankAreaDistribution(listOfFieldsToEvaluate, sdnight, sdtime, dateProfile, rawSeeing, seeing, transparency)

1. needTonight equals GoalVisitsTonight - VisitsTonight
2. If needTonight greater than zero
    1. GlobalNeedFactor equals needTonight / GoalVisitsTonight
3. Else
    1. GlobalNeedFactor equals (maxNeedAfterOverflow / (VisitsTonight - GoalVisitsTonight + 1)) / GoalVisitsTonight
4. For fieldID in list of fieldsToEvaluate
    1. If fieldID equals last observed fieldID and last observed was for this proposal and not accept consecutive observations
        1. continue
    2. If airmass greater than maxairmass
        1. Increment fields_invisible by one
        2. Continue
    3. If distance to moon less than distance to moon from SchedulingData
        1. Increment fields_moon by one
        2. Remove fieldID from targets
    4. For filter in filterNames
        1. If nVisits for filter exists, set to visits for filter, fieldID
        2. Otherwise set nVisits for filter to zero
        3. Set progress for filter to nVisits for filter / GoalVisitsFieldFilter for filter
        4. Add to progress_avg min of progress for filter or one and divide by length of filterNames
    5. Set FieldNeedFactor to one minus progress_avg
    6. If progress_avg between ProgressToStartBoost and one
        1. Add to FieldNeedFactor MaxBoostComplete times progress_avg minus ProgressToStartBoost divided by one minus ProgressToStartBoost
    7. Proposal::allowedFiltersForBrightness
    8. Filters::computeFilterSeeing

9. For filter in allowedFilterList
    1. Increment ffilter_allowed by one
    2. If filterSeeingList for filter greater than FilterMaxSeeing for filter
        1. Increment ffilter_badseeing by one
        2. Continue
    3. If GlobalNeedFactor greater than zero
        1. If FieldNeedFactor greater than zero
            1. If progress for filter less than one
                1. Set FilterNeedFactor to one minus progress for filter
                2. Set rank to scale times one half times FieldNeedFactor plus FilterNeedFactor divided by GlobalNeedFactor
            2. Else
                1. Set rank to zero
        2. Else
            1. Set FilterNeedFactor to maxNeedAfterOverflow divided by nVisits for filter minus GoalVisitsFieldFilter for filter plus one divided by GoalVisitFieldFilter for filter
            2. Set rank to scale times FilterNeedFactor divided by GlobalNeedFactor
    4. Else
        1. Set rank to zero
    5. If rank greater than zero
        1. Increment ffilter_proposed by one
        2. Get record from obsPool for fieldID, filter
        3. Set information on record
        4. Proposal::addToSuggestList

WLProp

suggestObs

1. Calls TransSubSeqProp::suggestObs

TransSubSeqProp

suggestObs(dateProfile, n, exclusiveObservation, mindistance2moon, rawseeing, seeing, transparency, sdnight, sdtime)

1. Get information from dateProfile and SchedulingData::moonProfile for night
2. If ::CheckObservingCycle for date
    1. Clear suggestList
    2. If exclusive observation is not None

1. If propID for exclusive observation equals propID for self
    1. Set rank to one
    2. Get filter from sequences[fieldID]::GetNextFilter(subseq)
    3. Get exclusiveBlockRequired from sequences[fieldID]::GetExclusiveBlockNeeded(subseq)
    4. Get record from obsPool for fieldID, filter
    5. Add record to Proposal::addToSuggestList
    6. Return Proposal::getSuggestList
2. Else
    1. If exclusiveObservation.fieldID not in tonightTargets
        1. Set list of fieldsToEvaluate to exclusiveObservation.fieldID
    2. Else
        1. Create empty list of fieldsToEvaluate
        2. Set numberOfObsToPropose to zero
3. Else (normal observation)
    1. Set list of fieldsToEvaluate from tonightsTargets
    2. Set numberOfObsToPropose to n
    3. If exclusiveBlockNeeded
        1. DB::addMissedObservation
        2. For fieldID in list of fieldsToEvaluate
            1. If fieldID equals last observed fieldID and last observed was for this proposal and not accept consecutive observations
                1. continue
            2. If airmass greater than maxairmass
                1. Increment fields_invisible by one
                2. Continue
            3. If distance to moon less than distance to moon from SchedulingData
                1. Increment fields_moon by one
                2. Remove fieldID from targets
            4. Get sky brightness from schedulingData.brightness[sdtime][fieldID]
            5. Get allowedFilterList from Proposal::allowedFiltersForBrightness(skyBrightness)
            6. Get filterList from Filters::computeFilterSeeing
            7. For subset from tonightSubseqsForTarget for fieldID
            8. If sequences for fieldID ::IsLost
                1. continue
            9.
4. For record in fieldRecordList
    1. Proposal::addToSuggestList

5. Return Proposal::getSuggestList
3. Else
    1. Return empty list as cycle has ended

Proposal

addToSuggestList(observation)

1. Set rankInternal to observation rank
2. Scale observation rank by relativeProposalPriority
3. Add to queue tuple minus rankInternal and observation

allowedFiltersForBrightness(brightness)

1. For filter in filterNames
    1. If filter in mountedFiltersList and brightness between FilterMinBright for filter and FilterMaxBright for filter
        1. Append filter to filterList
2. Return filterList

getSuggestList(n=1)

1. For i in range of number of requested observations (n)
    1. Append to winners list the observation
2. Put rest of observations into losers list
3. Return winners list

Filters

constructor

1. Set lots of internal variables
2. Sort information from filter list including ranking filters

computeFilterSeeing(seeing, airmass)

1. Take airmass to the 3/5 power
2. For index in range of length of filter names
    1. wvSee equals seeing times basefilterWavelenSorted
    2. adjustSeeing equals square root of wvSee times air3_5 squared plus telSeeing times air3_5 squared opticalDesSeeing squared plus cameraSeeing squared

        3. Set filterList for filter name equal to adjustSeeing
    3. Return filter list

computeFiltersForSky(brightness, seeing, airmass)

1. Same as ::computeFilterSeeing but for filters when brightness between FilterMinBrigSorted and FilterMaxBrigSorted

computeSkyBrightnessForFilter(filter, skyBrightness, date, twilightProfile, moonProfileAltAz)

1. If filter is y return 17.3
2. Else
    1. If moon altitude less than or equal to 6 degrees
        1. Set adjustBright to filterOffset for filter and zero
    2. Elif moon phase percent is not in skyBrightKeys
        1. Loop through keys and linearly interpolate to find adjustBright
    3. Else
        1. Set adjustBright to filterOffset for filter, moon phase percent
    4. Set filterSkyBright to skyBrightness plus adjustBright
3. If filter is z and filterSkyBright less than 17.0 return 17.0
4. If date less than sunsetTwil or date greater than sunriseTwil
    1. If filter is z or y
        1. Return 17.0

Instrument

constructor

- Setup InstrumentParams object
- Setup four InstrumentState objects
    - Current (current_state)
    - Target (target_state)
    - Park (park_state)
    - Next (next_state)
- Setup InstrumentSlewParams object
- Setup InstrumentPosition object (targetPosition)
- Setup some empty dictionaries and fill them by looping through the slew activities

TimeAccelMove(distance_RAD, maxspeed, accel, decel)

- Convert distance to degrees
- Calculate peak speed assuming max speed not reached
- If peak speed is less than max speed
    - Delay equal peak speed * (1/accel + 1/decel) [triangle]
- Else
    - d1 equals 0.5 * max speed^2 / accel

    - d3 equals 0.5 * max speed^2 / decel
    - d2 equals distance – d1 – d3
    - t1 equals max speed / accel
    - t3 equals max speed / decel
    - t2 equals d2 / max speed
    - delay equals sum of ti
    - peak velocity equals max speed
- speed mod equals compare(distance_RAD, 0) [-1 if x<y, 0 if x==y, +1 if x>y]
- Return delay and peak speed * DEG2RAD * speed mod

GetDelayForTarget(ra_RAD, dec_RAD, dateProfile, exposureTime, filter)

- If not current_state::IsFilterMounted(filter) is True, return -1.0
- ha_RAD equals lst_RAD (from dateProfile) minus ra_RAD
- az_RAD, alt_RAD, pa_RAD equals palpy::altaz(ha_RAD, dec_RAD, params.latitude_RAD)
- If slew_params.RotatorFollowSky is True
    - angle_RAD equals 0
- Else
    - angle_RAD equals pa_RAD minus next_state.Rotator_Pos_RAD
- targetPosition::Set
- If targetPosition.alt_RAD less than slew_params.Telescope_AltMinRAD return -1.0
- Elif targetPosition.alt_RAD greater than slew_params.Telescope_AltMaxRAD return -1.0
- Else ::GetSlewDelay
- Return delay

GetSlewDelay(targetPosition, next_state, allSlewData)

- target_state::SetClosestState
- If prev_state.Tracking equals False
    - If allSlewData is True
        - Return (0.0, target_state)
    - Else
        - Return (0.0, None)
- last_activity equals “Exposure”
- slew_delay equals ::GetDelayAfter
- If allSlewData is True
    - For ac in slew_params.activities
        - dt equals self.delay_time_for[ac]
        - If dt greater than 0
            - lastslew.delays[ac] equals dt
        - activity equals last_activity
        - while activity not equals to empty string
            - dt equals delay_time_for[activity]
            - If dt greater than 0
                - lastslew_criticalpath.append(activity)
            - activity equals longest_prereq_for[activity]
    - Return (slew_delay, target_state)
- Else

    - Return (slew_delay, None)

Slew(new_position)
- (delay, new_state) equals ::GetSlewDelay(new_position, current_state, allSlewData=True)
- new_state.Tracking equals True
- ::SetState(new_state)

GetDelayAfter(activity, target_state, prev_state)
- activity_delay equals functionGetDelayFor[activity](target_state, prev_state)
- If activity_delay equals None
    - Return None
- prereq_list equals slew_params.prerequisites[activity]
- For prereq in prereq_list
    - prev_delay equals ::GetDelayAfter(prereq, target_state, prev_state)
    - If prev_delay greater than longest_previous_delay
        - longest_previous_delay equals prev_delay
        - longest_prereq equals prereq
- longest_prereq_for[activity] equals longest_prereq
- delay_time_for[activity] equals activity_delay
- Return activity_delay plus longest_previous_delay

Observe(ra_RAD, dec_RAD, dateProfile, exposureTime, filter, delay)
- Increment slewCount by one
- current_state::UpdateState(date)
- init_state is deep copy of current_state
- slewInitState equals current_state slew information marked as SLEWINITSTATE
- ha_RAD equals lst_RAD (from dateProfile) minus ra_RAD
- az_RAD, alt_RAD, pa_RAD equals palpy::altaz(ha_RAD, dec_RAD, params.latitude_RAD)
- If slew_params.RotatorFollowSky is True
    - angle_RAD equals 0
- Else
    - angle_RAD equals pa_RAD minus next_state.Rotator_Pos_RAD
- targetPosition::Set
- delay equals ::Slew(targetPosition)
- slewFinalState equals current_state slew information marked as SLEWFINALSTATE
- slewMaxSpeeds is current_state slew speeds information
- rotator_skypos equals current_state::GetRotatorSkyPos
- rotator_telpos equals current_state::GetRotatorTelPos
- altitude equals current_state.ALT_RAD
- azimuth equals current_state.AZ_RAD
- slewDistance equals palpy::dsep(ra_RAD, dec_RAD, init_state.RA_RAD, init_state.DEC_RAD)
- slewdata equals (slewCount, date (from dateProfile), date+delay, delay, slewDistance)
- For activity in lastslew_delays
    - If activity in lastslew_criticalpath
        - cp equals True

        - Else
            - cp equals False
        - slewactivity equals (activity, lastslew_delays[activity], cp)
        - listSlewActivities.append(slewactivity)
- next_state is deep copy of current_state
- next_state::UpdateState(date+delay+exposureTime)
- Return (delay, rotator_skypos, rotator_telpos, altitude, azimuth, slewdata, slewInitState, slewFinalState, slewMaxSpeeds, listSlewActivities)

InstrumentState

constructor
- Call InstrumentPosition constructor
- Set internal variables from configuration parameters and constructor parameters

SetClosestState(targetPos, initstate)
- ::SetPosition(targetPos)
- DomAlt_Pos_RAD equals ALT_RAD
- DomAz_Pos_RAD equals initstate::GetShortestDistanceWithWrap(AZ_RAD, initstate.DomAz_POS_RAD)
- TelAlt_Pos_RAD equals ALT_RAD
- TelAz_Pos_RAD equals initstate::GetTelAzDistanceWithWrap(AZ_RAD)[1]
- If initstate::GetFilter equals ::GetFilter
    - Rotator_Pos_RAD equals initstate::GetRotatorDistanceWithWrap(PA_RAD – ANG_RAD)
- Else
    - Rotator_Pos_RAD equals 0.0

GetTelAzDistanceWithWrap(az_RAD)
- Return ::GetShortestDistanceWithWrap(az_RAD, TelAz_Pos_RAD, TelAz_MinPos_RAD, TelAz_MaxPos_RAD)

GetRotatorDistanceWithWrap(newangle_RAD)
- Return ::GetShortestDistanceWithWrap(newangle_RAD, Rotator_Pos_RAD, Rotator_PosMin_RAD, Rotator_PosMax_RAD)

GetShortestDistanceWithWrap(target_RAD, current_abs_RAD, min_abs_RAD, max_abs_RAD)
- If min_abs_RAD not equal to None
    - norm_target_RAD equals (target_RAD – min_abs_RAD) % 2pi + min_abs_RAD
    - If norm_target_RAD greater than max_abs_RAD
        - norm_target_RAD subtract pi
    - Else
        - norm_target_RAD equals target_RAD
    - distance_RAD equals (norm_target_RAD – current_abs_RAD) % 2pi
    - If distance_RAD greater that pi
        - distance_RAD minus 2pi
- If min_abs_RAD not equal to None and max_abs_RAD not equal to None
    - accum_abs_RAD equals current_abs_RAD plus distance_RAD

        - If accum_abs_RAD greater than max_abs_RAD
            - distance_RAD minus 2pi
        - If accum_abs_RAD less than min_abs_RAD
            - distance_RAD plus 2pi
    - final_abs_RAD equals current_abs_RAD plus distance_RAD
    - Return distance_RAD, final_abs_RAD

UpdateState(date)
- If Tracking is True
    - Set internal variables to calculated values

SetPosition(newpos)
- Call InstrumentState::Copy(newpos)

GetRotatorTelPos
- Return Rotator_Pos_RAD % 2pi

GetRotatorSkyPos
- Return (Rotator_Pos_RAD – PA_RAD) % 2pi