Observation Simulation

name ? models required for ... ? with same link though

Having defined the Available Model Types, you can now simulate actual observations to use in your analysis, or load real data into your analysis. Below, we first describe Defining observation simulation settings, and how to analyze the resulting data structures. Finally, we provide a (preliminary) introduction to Loading external observations.

a bit unnecessary in my opinion .. the link

maybe reformulate a bit ? sounds slightly off

Defining observation simulation settings

In addition to the definition of the observation model, simulating the observations themselves requires a definition of the time(s) at which the observation is to be simulated, as well as a definition of which observation model these are to be simulated from (in addition to optional additional settings, see below). Settings for simulating observations are defined by the creation of a so-called `ObservationSimulationSettings` class ~~(or derived class)~~. The basic manner in which to define an observation simulation settings object uses the `tabulated_settings()`, specifying the observation times explicitly as follows:

grammarly odd

would for now only mention below.. just talk about the "basic" functionalities

```
one_way_nno_mex_link_ends = dict( );
one_way_nno_mex_link_ends[ transmitter ] =
estimation_setup.observation.body_reference_point_link_end_id( "Earth", "NNO"
);
one_way_nno_mex_link_ends[ receiver ] =
estimation_setup.observation.body_origin_link_end_id( "MeX" );
one_way_nno_mex_link_definition =
estimation_setup.observation.link_definition( one_way_nno_mex_link_ends )

observation_times = list( )
observation_times = [10.0, 20.0, 30.0]

observation_simulation_settings = observation_setup.tabulated_settings(
   one_way_range_type
   one_way_nno_mex_link_definition,
   observation_times )
```

where a list of times ($t = 10, 20, 30$ s) is explicitly specified, and an observation simulation settings object is created, which specifies that a one-way range observation is to be simulated at these times, with the link ends specified by `one_way_nno_mex_link_definition`.

would put this before the actual example ...

By default, the reference time for the one-way range observable is the receiver (see `get_default_reference_link_end`). This means that, for the above, these settings will simulate observations which are *received* by MeX at t=10, t=20, and t=30, respectively. To override this behaviour, (...) we can specify a reference link end manually:

```
observation_simulation_settings = observation_setup.tabulated_settings(
    one_way_range_type
    one_way_nno_mex_link_ends,
    observation_times,
    reference_link_end = observation_setup.transmitter )
```

which will yield observations *transmitted* at t=10, t=20, and t=30 by NNO.

As an extension of the above, you can also use `tabulated_settings_list()`:

```
observation_simulation_settings_list =
observation_setup.tabulated_settings_list(
    link_definitions_per_observable,
    observation_times )
```

Brilliant example of why I'd change the general structure w.r.t. examples: I have no clue what this does, but I first need the text below to understand that, before the code helps in any way

Instead of creating a single object to simulate observations, it contains a list of objects, for any number of observable types and link ends.

The `tabulated_settings()` is the simplest manner in which to define the times (and other settings) at which to simulate observations. By adding observation constraints (see below), this list of times may be filtered during the observation simulation process to only retain those times at which specific conditions are met (e.g. target above the horizon). For many practical cases, it is desirable to have continuous tracking passes of a given length that are not interrupted by such constraints. The `continuous_arc_simulation_settings()` can be used to achieve such behaviour.

not sure if this is the best place here... sounds more like intro to constraints

Defining additional settings

In addition to defining the observable type, link ends, observation times and (optionally) reference link ends for simulating an observation, you can define a number of additional settings to be taken into account:

- **Ancilliary settings**: Some observables may or must get additional quantitative data that influences the ideal value of the observable. Examples are the integration time for averaged Doppler observables, and retransmission times for n-way observables. nice ☺
- **Constraints**: You can define settings such that an observation is only simulated if certain conditions (elevation angle, no occultation, *etc.*) are (not) met
- **Noise levels**: You can define a functions which adds (random) noise to the simulated observations. This noise is typically, but not necesarilly, Gaussian
- **Additional output**: Similarly to the state propagation framework, you can define a wide range of *dependent variables* to be calculating during the simulation of observations. Note that the *type* of variables you can choose from is distinct from those available during state proagation.

Typically, these settings are defined and added to the observation simulation settings *after* the nominal settings have been defined ~~(in the process outlined above)~~.

To efficiently achieve this, there are several functions available in Tudat, which take a list of `ObservationSimulationSettings` objects (such as those returned by the `tabulated_settings_list()` function), and add specifics for ~~any~~ one of the above options to any number of observation simulation settings. For each of the above three options, three separate functions are provided to modify the list of observation simulation settings (see Ancilliary settings, Defining noise levels and Defining additional output for API links, and examples): four?

- One function modifying each `ObservationSimulationSettings` object in the list (for instance: regardless of the type or link end of the observation, always save the light-time as dependent variable)
- One function modifying each `ObservationSimulationSettings` object in the list which contains settings for a given `ObservableType()` (for instance: regardless of link ends, use 1 mm/s random noise for all two-way Doppler observables)
- One function modifying each `ObservationSimulationSettings` object in the list which contains settings for a given `ObservableType()` and a given set of link ends (for instance: for all one-way range observables between New Norcia ground

station and Mars Express, only simulate an observation if Mars Express is at last 15 degrees abov the horizon.

nice ☺

Ancilliary settings

somewhere link to API with list of funct.

or

Some observation models depend on data in addition to that normally contained in either the observation model of the observation simulation settings to fully determine the value of the observable. In some cases, these data *may* be defined, in other cases they *must* be defined. At present, the following ancilliary settings are supported:

- Integration time. This is *required* for each averaged Doppler observable. A value of 60 s is set by default. It is stored as a single floating point value. The integration time defines the time over which the averaged Doppler observable is to be averaged (or, the so-called ‘count interval’).
- Retransmission delays. This is *optional* for each N-way (including each two-way) observable. It is undefined (no retransmission delay) by default. It is stored as a list of floating point values. The retransmission delays quantify how much time elapses between the reception and retransmission of a signal at one of the retransmitter link ends

with one entry per retransmitter

To set a 5 s Doppler integration time for every averaged Doppler observable (after the simulation settings creation),

```
integration_time = 5.0
doppler_ancilliary_settings = doppler_ancilliary_settings( integration_time )
observation.add_ancilliary_settings_to_observable(
    observation_simulation_settings_list,
    doppler_ancilliary_settings,
    observation.n_way_averaged_doppler_type )
```

Defining observation constraints

In many cases, whether an observation at a given time should be realized will depend on a number of constraints that must be satisfied. We have termed such constraints ‘observation viability settings’, and we have currently implemented the following types:

- **Minimum_elevation_angle**: Minimum elevation angle at a ground station: target must be at least a certain elevation above the horizon (see `elevation_angle_viability()`).
- **Body avoidance angle**: the line-of-sight vector from a link end A to a given third body must have an angle w.r.t. the line-of-sight between link end A and any other link ends that it obsereved that is sufficiently large. This constraint is typically used to prevent the Sun from being too close to the field-of-view of the telescope(s), (see `body_avoidance_viability()`)
- **Body occultation**: the link must not be obscured by a given third body. For instance: the Moon occulting a link between Earth and Mars (see `body_occultation_viability()`)

For example, the `observation_simulation_settings_list` list created in the example above can be modified such that only observations above a 15 degree elevation angle at New Norcia ~~(for those observations in which New Norcia is a ground station)~~ are accepted:

```
station_id = [ "Earth", "NNO" ];
viability_settings_list = list()
viability_settings_list.append(
estimation_setup.observation.elevation_angle_viability(
    station_id,
    np.deg2rad( 15.0 ) ) )
observation.add_viability_check_to_all(
  observation_simulation_settings_list,
  viability_settings_list )
```

put here

In this case ~~(the `add_viability_check_to_all()` function)~~, the list of settings in `viability_settings_list` is applied to *all* observation simulation settings in `observation_simulation_settings_list`. To only add the viability settings to observation simulation settings of a given type of observable, or only to those of a given observable **and** a give link definition, use the `add_viability_check_to_observable()` and `add_viability_check_to_observable_for_link_ends()` functions, respectively.

while still similar (for ancillary as well

To add viability settings directy to a single `ObservationSimulationSettings` object, use the `viability_settings_list()` attribute.

Defining noise levels

maybe just put in API? not really important to get acquainted with estimation; after that one only checks the API

If no noise is defined, the observations are simulated according to the determininistic model that has been defined in the Observation Model Setup. We stress that this 'noise-free' observation can contain a simulated bias, if such a bias ~~is included~~ in the observation model settings (see ~~Defining observation settings~~). By adding noise settings, a user can add (typically, but not necesarilly) random noise to the simulation of the observations. We currently have two types of interfaces for adding noise to an observation:

has been included

here

- **Gaussian noise**: By specifying the standard deviation, you can add uncorrelated, zero-mean Gaussian noise to the observations
- **Generic noise**: By specifying an arbitrary function that generates noise (as a function of time), a user can add noise from any type of distribution to the simulated observations

Adding Gaussian noise to all observations of a given type can be done by:

```
noise_level = 0.1
observation.add_gaussian_noise_to_observable(
    observation_simulation_settings_list,
    noise_level,
    observation.one_way_range_type )
```

which will add 10 cm random noise to each one-way range observable in the `observation_simulation_settings_list` list. In this case (the `add_gaussian_noise_to_observable()` function), the noise is applied to all observations of a given type. To add the noise to observation simulation settings of all observables, or only to those of a given observable **and** a give link definition, use the `add_gaussian_noise_to_all()` and `add_gaussian_noise_to_observable_for_link_ends()` functions, respectively.

[Similar interfaces exist to add a generic noise function to the observation:

```
def custom_noise_function( current_time ):
    return np.ndarray([np.random.lognormal(0.0,1.0)])

observation.add_noise_function_to_observable(
    observation_simulation_settings_list,
    custom_noise_function,
    observation.one_way_range_type )
```

where it is important to realize that the noise function *must* have a single float representing time as input, and returns a vector (of the size of a single observation) as output. For many observables (range, Doppler), this size will be 1. For angular position observables, for instance, the size will be 2. The `add_noise_function_to_all()`, `add_noise_function_to_observable()` and `add_noise_function_to_observable_for_link_ends()` functions can be used to add a noise function to a subset of all observation simulation settings.

To add a generic noise function directy to a single `ObservationSimulationSettings` object, use the `noise_function()` attribute.

maybe also for API or for the ObsSimSett page

Defining additional output

As is the case with the state propagation (see here), you can define any number of dependent variable to be saved along with the observations. These include distances between link ends, angles between link ends, and a variety of other options. Note that this functionality is relatively new, and the list of implemented dependent variables is currently limited. A full list of options can be found in TODO

☺

Creating observations

some text here?

Simulating the observations

Having fully defined the list of observation simulation settings `observation_simulation_settings`, as well as the `observation_simulators` (see `create_observation_simulators()`), the actual observations can be simulated as follows:

I'd use what is referenced in the text

```
simulated_observations = estimation.simulate_observations(
    observation_simulation_settings,
    estimator.observation_simulators,
    bodies)
```

where ~~the~~ `bodies` is the usual `SystemOfBodies` object that defines the physical environment (see Environment Setup for details on creation and usage). The `simulate_observations()` function returns an object of `ObservationCollection` type, which stores all observations and dependent variables.

the

associated

Accessing and analyzing the observations

Literally the last sentence repeated

The full set of observations is stored in an object of type `ObservationCollection`, both when they are simulated, or loaded from a real data source. From this object, the full vector of observations $\mathbf{h}$ can be obtained, with length n_{obs}. Internally, this observation collection stores the observations (and any associated data), as a nested dictionary sorted by:

- Firstly, per observable type
- Secondly, (for each observable type) per link definition
- For each combination of observable type and link definition, a list of `SingleObservationSet` objects is stored (see below)

vague, would chose diff. ex.

Consequently, the vector $\mathbf{h}$ provides the observations stored in this manner. A vector of observable types, link definitions and times (each with length n_{obs}) can be extracted from the `ObservationCollection` using various properties. This allows a user to keep track of which entry of $\mathbf{h}$ represents what. For observable that have a size > 1 (for instance, angular position is size 2; Cartesian position is size 3), the associated entries in the vector of times (and link defintion, etc.) are copied. For instance, for an observable vector $\mathbf{h}$ consisting of three angular position observables, we will have $\mathbf{h} = [\alpha(t_1); \delta(t_1); \alpha(t_2); \delta(t_2); \alpha(t_3); \delta(t_3)]$, and the associated vector of times will be $\mathbf{t} = [t_1; t_1; t_2; t_2; t_3; t_3]$.

what is n_{obs} ?

since at first glance one might confuse this with one time setting

Something's wrong with the parenthesis here

When simulating the observations using a set of `ObservationSimulationSettings` objects (see here, each of these will result in an object of type `SingleObservationSet` (a set of which in turn constitutes the `ObservationCollection`; see above). For a given observable type and link definition, there will typically but not necesarilly be a single one of these `SingleObservationSet` objects inside a `ObservationCollection`. Observables, and their associated properties can be extracted from these objects ```SingleObservationSet``s, instead of the ``ObservationCollection```, for a more fine-grained analysis of the results. A list of all `SingleObservationSet` objects for a given observable type and link end can be extracted using the `ObservationCollection` function.

Grammar of sentence: verb?

rewrite? not very clear paragraph

Since the dependent variables that are saved in the `ObservationCollection` will typically differ per constituent `SingleObservationSet`, it is not possible to extract a single list of these from the full collection. Instead, they can only be extracted from the single observation set.

Loading external observations

Tudat contains a number of functions for loading typical tracking data types (TODO) into a list of `SingleObservationSet` objects. A user may also load any external data source into Tudat-compatible observations. This can be done using the `create_single_observation_set()` function, which allows a user to load all the required raw data for an observabtion. A list of these observation sets can then be put into an observation collection using the `observation_collection` function.

expand ☺