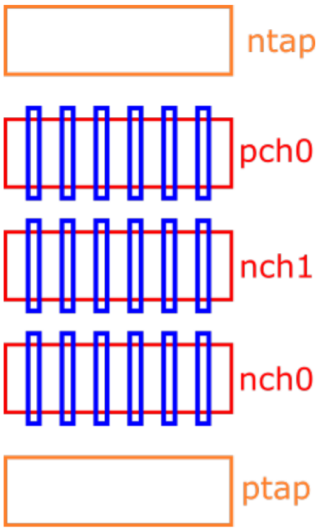
AnalogBase

In this module, you will learn the basics of `AnalogBase`, and how to design a source-follower layout generator using AnalogBase.

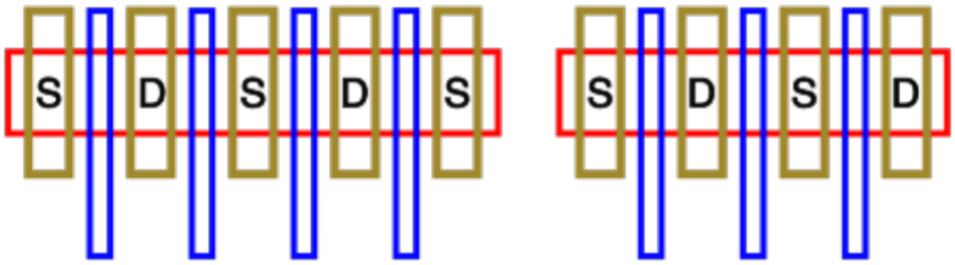
What is AnalogBase?



`AnalogBase` is one of several "layout floorplan" classes that allows designers to easily develop process-portable layout generator for various electromigration-constrained circuits. To do so, `AnalogBase` draws rows of transistors with substrate contacts on the top-most and bottom-most rows, as shown in the figure above. In this floorplan, the number of current-carrying wires scales naturally with number of fingers, which is optimal for circuits with large bias currents.

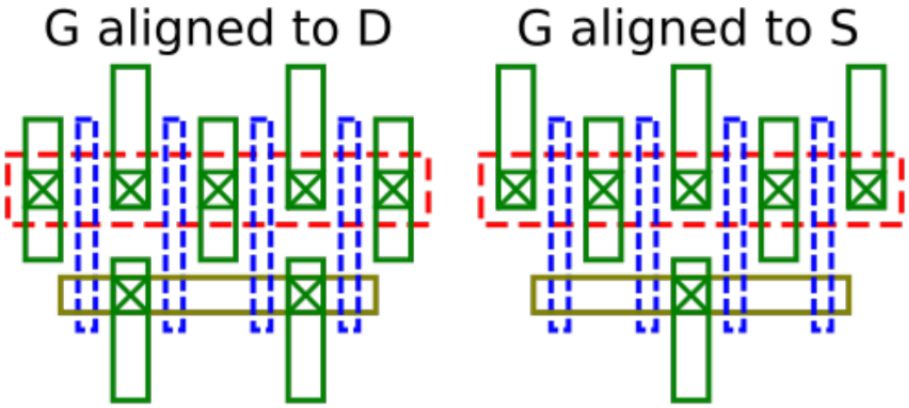
By convention, `AnalogBase` draws N rows of NMOS (labeled as `nch`) and P rows of PMOS (labeled as `pch`), with N and P being nonnegative integers, so you can only draw NMOS rows by setting $P = 0$, and so on. The rows are indexed from bottom to top, so `nch(N-1)` is the top-most NMOS row, and `pch0` is the bottom-most PMOS row.

Transistor Source/Drain Naming Convention



Before we talk about how `AnalogBase` draws transistor connections, we need to establish a naming convention for source/drain junctions of a transistor, since source and drain are often interchangeable. In XBase, the left-most source/drain junction of a transistor is always called "source", and after that source and drain alternates between each other, as shown in the above figure. This implies that for even number of fingers, the right-most junction is always "source", and for odd number of fingers, the left-most junction is always "drain".

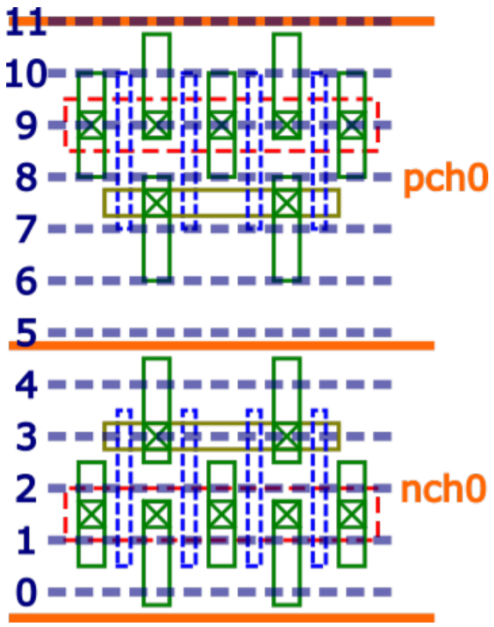
AnalogMosConn Overview



To connect transistors to the routing grid, `AnalogBase` "drops" `AnalogMosConn`, a layout cell consisting only of wires and vias, on top of desired transistors to connect gates, sources, and drains to a vertical routing layer. For most technologies, `AnalogMosConn` draws gate, drain, and source wires on every other source/drain junction, with drain and source wires interleaving with each other. By default, the gate wires are drawn below the transistor row, to draw gate wires above the transistor row, flip the row upside down by changing the row orientation from `R0` to `MX` (we will see an example of this later).

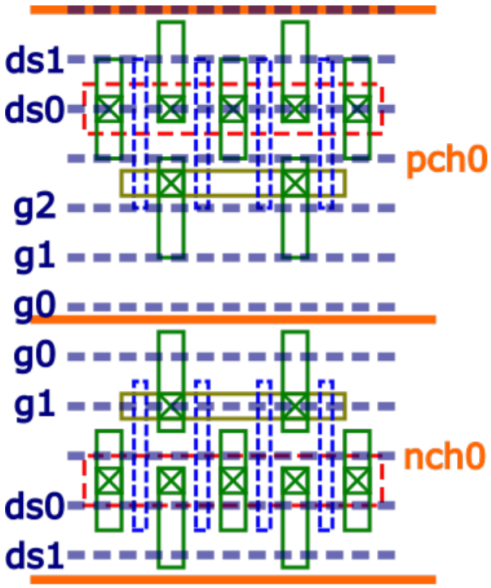
With this layout style, the gate wires can either be drawn in the same tracks as source wires ("G aligned to S"), or they can be drawn in the same tracks as drain wires ("G aligned to D"). The gate wire location is usually determined by source/drain wire direction. For example, in the figure above, if the source of a transistor needs to be connected to the row below it, then gate wires cannot be aligned to source, as this will cause a short between gate and source wires when the source wires is extended downwards. Because of this, when creating a `AnalogMosConn`, designer needs to specify the drain and source wire directions (whether they go "up" or "down"), and the gate wire locations will be determined automatically to avoid shorts.

Connecting to Horizontal Tracks



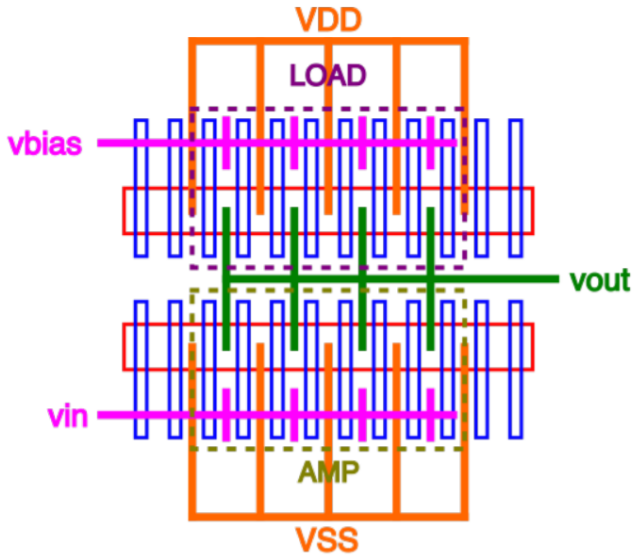
In the previous section, we see that `AnalogMosConn` connects the transistor to vertical tracks. How do we connect those vertical wires to the horizontal tracks above it? If you recall from the previous module, you would need to use the `connect_to_tracks()` method with the horizontal track index. The question now becomes: how do I know which track index can be used for gate/drain/source connections?

To get a better understanding of this problem, consider the layout shown in the figure above. The PMOS drain wires can be easily connected to track 10 with no issues, but the PMOS gate wires cannot be connected to track 10 without shorting with drain wires. In fact, the PMOS gate wires can only be connected to tracks 5, 6, and 7 without running into minimum line-end spacing rules with other wires. How can we determine what the legal track indices are? Furthermore, if our particular circuit requires more than 3 horizontal tracks for PMOS gate connections, how can we tell `AnalogBase` to space the rows further apart?



To address these issues, `AnalogBase` introduces the concept of relative track indices, as shown in the

CS Amplifier Layout Example



Now that you have a general idea of how `AnalogBase` works, lets walk through a common-source amplifier example. The figure above shows a rough sketch of the layout floorplan (**NOTE: ALWAYS DRAW FLOORPLAN BEFORE CODING!**). We have one NMOS row on the bottom, one PMOS row on the top, and we put extra dummy transistors on both sides to reduce edge layout effects. The input connects to NMOS gates from below the NMOS row, the PMOS bias connects to PMOS gates from above the PMOS row, and the output drain/source of NMOS/PMOS are connected to a horizontal track between the two rows. Finally, the supply drain/source wires are extended and shorted on top of the substrate contacts on both ends.

The entire common-source amplifier layout generator code is reproduced below. We will walk through important sections of the code and describe what they do.

```
class AmpCS(AnalogBase):
    """A common source amplifier."""
    def __init__(self, temp_db, lib_name, params, used_names, **kwargs
):
        AnalogBase.__init__(self, temp_db, lib_name, params, used_name
s, **kwargs)
        self._sch_params = None

    @property
    def sch_params(self):
        return self._sch_params

    @classmethod
    def get_params_info(cls):
        """Returns a dictionary containing parameter descriptions.

        Override this method to return a dictionary from parameter nam
es to descriptions.

        Returns
        -------
```

Class Definition

```
class AmpCS(AnalogBase):
    """A common source amplifier."""
    def __init__(self, temp_db, lib_name, params, used_names, **kwargs
):
        AnalogBase.__init__(self, temp_db, lib_name, params, used_name
s, **kwargs)
        self._sch_params = None

        @property
        def sch_params(self):
            return self._sch_params
```

The layout generator code starts with the Python class definition. We subclass the `AnalogBase` class to inherit various functions described earlier. The constructor doesn't do much besides calling the super constructor and initializing a private attribute. Finally, we declare a read-only property, `sch_params`, which we will compute later. It contains the schematic parameters for the schematic generator we will see in the next module.

Parameter Specifications

```
@classmethod
def get_params_info(cls):
        """Returns a dictionary containing parameter descriptions.
        Override this method to return a dictionary from parameter nam
es to descriptions.
        Returns
        -------
        param_info : dict[str, str]
            dictionary from parameter name to description.
        """
        return dict(
            lch='channel length, in meters.',
            w_dict='width dictionary.',
            intent_dict='intent dictionary.',
            fg_dict='number of fingers dictionary.',
            ndum='number of dummies on each side.',
            ptap_w='NMOS substrate width, in meters/number of fins.',
            ntap_w='PMOS substrate width, in meters/number of fins.',
            show_pins='True to draw pin geometries.',
        )
```

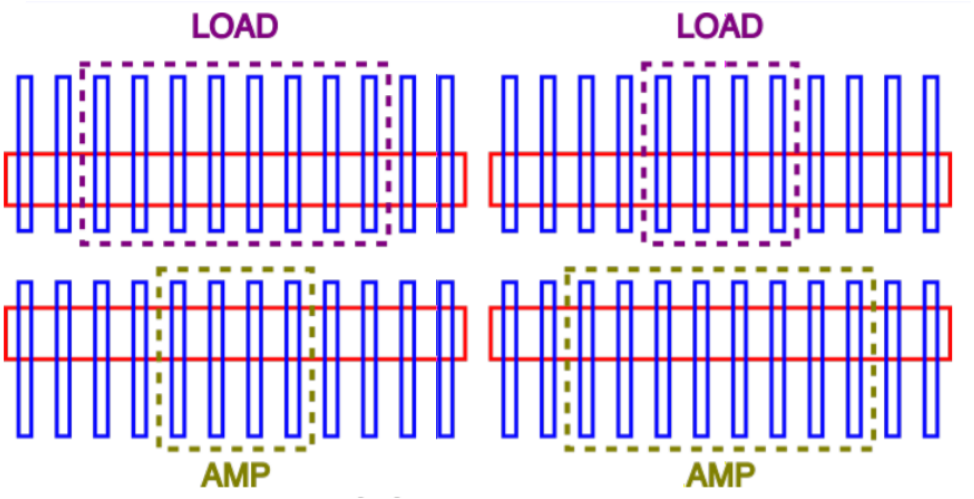
Next we have a class method, `get_params_info()`, that simply returns a Python dictionary from layout parameter names to a brief description of the corresponding parameter. This method should list all layout parameters, and it is used to determine to compute a unique ID to represent the generated instance. This allows XBase to avoid re-generating existing layouts when constructing a complex layout hierarchy with many duplicate layout instances.

How many fingers in a row?

Next, in the `draw_layout()` method is where all the layout generation happens. The beginning is rather straight-forward, then we get to the following section:

```
    # compute total number of fingers in each row
    fg_half_pmos = fg_load // 2
    fg_half_nmos = fg_amp // 2
    fg_half = max(fg_half_pmos, fg_half_nmos)
    fg_tot = (fg_half + ndum) * 2
```

This section computes how many fingers we need to draw in each transistor row. To get a better understanding, consider the two scenarios below:



Since `AnalogBase` must draw the same number of fingers for each row, we see that total number of fingers in each row depends on whether the AMP transistor or the LOAD transistor has more fingers. We resolve this by using the `max()` function to get the larger of the two.

Drawing Transistor Rows

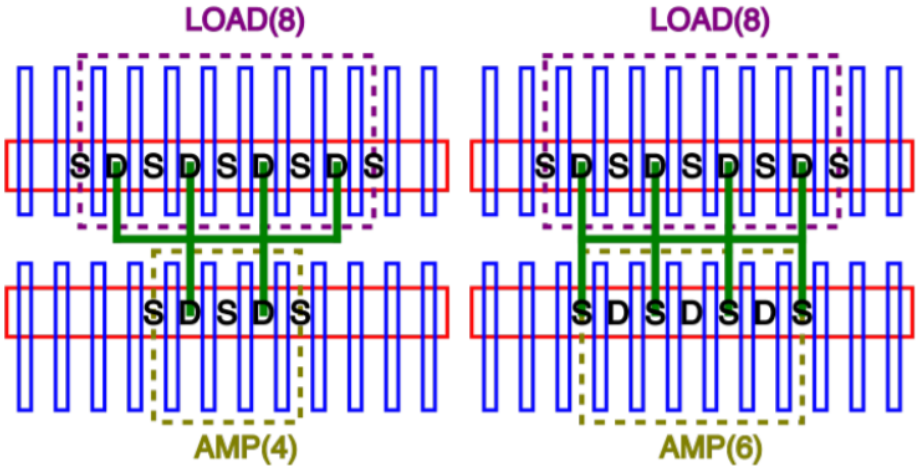
```
# specify width/threshold of each row
nw_list = [w_dict['amp']]
pw_list = [w_dict['load']]
nth_list = [intent_dict['amp']]
pth_list = [intent_dict['load']]

# specify number of horizontal tracks for each row
ng_tracks = [1]  # input track
nds_tracks = [1]  # one track for space
pds_tracks = [1]  # output track
pg_tracks = [1]  # bias track

# specify row orientations
n_orient = ['R0'] # gate connection on bottom
p_orient = ['MX'] # gate connection on top

self.draw_base(lch, fg_tot, ptap_w, ntap_w, nw_list,
               nth_list, pw_list, pth_list,
               ng_tracks=ng_tracks, nds_tracks=nds_tracks,
               pg_tracks=pg_tracks, pds_tracks=pds_tracks,
               n_orientations=n_orient, p_orientations=p_orient,
               )
```

This section specifies the layout parameters for each row, then calls the `draw_base()` method in `AnalogBase` to draw the transistor and substrate contact rows. Note that the PMOS row orientation is set to `MX` so that `AnalogMosConn` will draw gate wires on the top of PMOS row.

Is output on source or drain?

```
# figure out if output connects to drain or source of nmos
if (fg_amp - fg_load) % 4 == 0:
    aout, aoutb, nsdir, nddir = 'd', 's', 0, 2
else:
    aout, aoutb, nsdir, nddir = 's', 'd', 2, 0
```

This section determines if the output should connect to drain or source of the nmos transistor, and as the result what should the nmos source/drain wire directions be. To see why this is necessary, consider the two cases shown below:



In both cases, we have 8 PMOS fingers, and 4 or 6 NMOS fingers, respectively. To make life simpler, we decide to always connect the output wires to PMOS drain (if you expect PMOS to always be larger, this gives you less parasitic capacitance). Furthermore, to have better symmetric, we align the center of the PMOS and NMOS transistors. Then, to minimize interconnect resistance, we should connect output to the NMOS junction that is aligned to PMOS drain. If we check the above figure, we see that the corresponding NMOS junction is drain when NMOS has 4 fingers, but it is source when NMOS has 6 fingers! This means that the correct NMOS junction to connect to actually depends on both `fg_amp` and `fg_load`. By sketching a few example, you should be able to figure out that we need to connect output to NMOS drain if the difference in number of fingers is a multiple of 4, and connect output to NMOS drain otherwise. This is exactly what this section of code does.

Drawing Transistor Connections

```python
# create transistor connections
load_col = ndum + fg_half - fg_half_pmos
amp_col = ndum + fg_half - fg_half_nmos
amp_ports = self.draw_mos_conn('nch', 0, amp_col, fg_amp, nsdir, nddir,
                               s_net=s_net, d_net=d_net)
load_ports = self.draw_mos_conn('pch', 0, load_col, fg_load, 2, 0,
                                s_net='', d_net='vout')
# amp_ports/load_ports are dictionaries of WireArrays representing
# transistor ports.
print(amp_ports)
print(amp_ports['g'])
```

Now we are ready to draw the actual transistor connections. To do so, we use the `draw_mos_conn()` function. As an example, `self.draw_mos_conn('pch', 0, load_col, fg_load, 2, 0)` creates an `AnalogMosConn` object on top of PMOS row 0, starting at transistor index `load_col` (with index 0 being left-most transistor), using `fg_load` fingers to the right, with source going up (code 2) and drain going down (code 0). Remember that the source/drain directions are used to determine gate wires location.

The optional parameters `s_net` and `d_net` specify the net names of the source and drain of the transistor drawn, respectively. By default, if these are not specified (or set to empty strings), AnalogBase assume they connect to VDD for PMOS or VSS for NMOS. These parameters are used to infer dummy transistor schematic to simplify the process of generating LVS-clean schematics.

the `draw_mos_conn()` method will return a dictionary from the strings `'g'`, `'d'`, and `'s'` to the `WireArray` objects for the corresponding vertical wires.

Connecting Wires

```
# create TrackID from relative track index
vin_tid = self.make_track_id('nch', 0, 'g', 0)
vout_tid = self.make_track_id('pch', 0, 'ds', 0)
vbias_tid = self.make_track_id('pch', 0, 'g', 0)
# can also convert from relative to absolute track index
print(self.get_track_index('nch', 0, 'g', 0))

vin_warr = self.connect_to_tracks(amp_ports['g'], vin_tid)
vout_warr = self.connect_to_tracks([amp_ports[aout], load_ports['d']],
vout_tid)
vbias_warr = self.connect_to_tracks(load_ports['g'], vbias_tid)
self.connect_to_substrate('ptap', amp_ports[aoutb])
self.connect_to_substrate('ntap', load_ports['s'])
```

This section used the `make_track_id()` and `get_track_index()` methods described before to get horizontal track indices from relative index. We then use `connect_to_tracks()` to connect wires to the desired tracks. `connect_to_substrate()` method is used to connect transistor junctions to the specified substrate contacts.

Dummies and Pins

```
vss_warrs, vdd_warrs = self.fill_dummy()

self.add_pin('VSS', vss_warrs, show=show_pins)
self.add_pin('VDD', vdd_warrs, show=show_pins)
self.add_pin('vin', vin_warr, show=show_pins)
self.add_pin('vout', vout_warr, show=show_pins)
self.add_pin('vbias', vbias_warr, show=show_pins)
```

After all connections are made, the `fill_dummy()` method can be used to automatically connect all unconnected transistors to corresponding substrate contacts as dummy transistors. `add_pin()` function is used to add layout pins, as seem from the routing demo module.

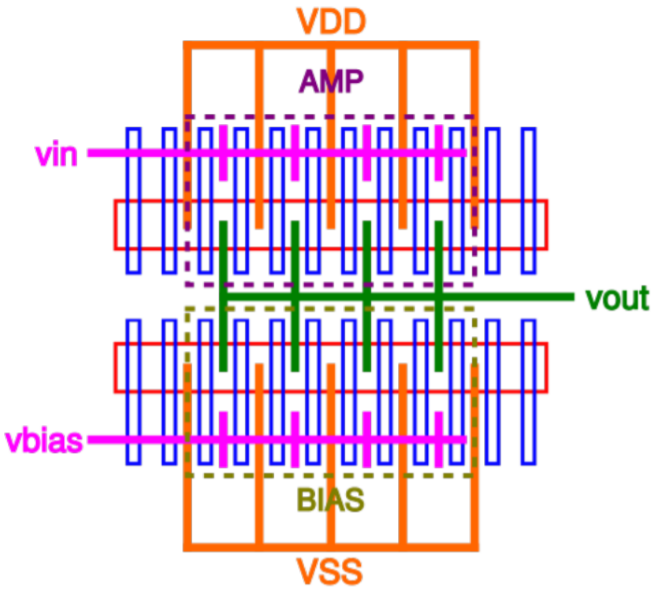
Schematic Parameters

```
# compute schematic parameters
self._sch_params = dict(
    lch=lch,
    w_dict=w_dict,
    intent_dict=intent_dict,
    fg_dict=fg_dict,
    dum_info=self.get_sch_dummy_info(),
)
```

Finally, we compute the schematic parameter dictionary, which will be used with the schematic generator later to produce LVS clean schematic. The `get_sch_dummy_info()` method will return a data structure that describes all the dummy transistors in this AnalogBase. This data structure will be used by the schematic generator to create the corresponding transistors.

SF Amplifier Exercise

Now that you understand how the common-source amplifier layout generator works, try to complete the following source-follower amplifier class by filling in missing codes. The floorplan for the source-follower amplifier is drawn for you below:



Notice that:

- we have two rows of NMOS.
- Gate connection is on the top for second row
- To minimize parasitics, we will use leave 1 horizontal track empty between vin and VDD.

You can evaluate the next cell (Press Ctrl+Enter) to see a preliminary layout of the source follower. It will also run LVS after generating the layout, which will fail if your layout is not correct.

In [1]:
```
from abs_templates_ec.analog_core import AnalogBase


class AmpSF(AnalogBase):
    """A template of a single transistor with dummies.

    This class is mainly used for transistor characterization or
    design exploration with config views.

    Parameters
    ----------
    temp_db : :class:`bag.layout.template.TemplateDB`
            the template database.
    lib_name : str
        the layout library name.
    params : dict[str, any]
        the parameter values.
    used_names : set[str]
        a set of already used cell names.
    kwargs : dict[str, any]
        dictionary of optional parameters.  See documentation of
        :class:`bag.layout.template.TemplateBase` for details.
    """

    def __init__(self, temp_db, lib_name, params, used_names, **kwargs):
        AnalogBase.__init__(self, temp_db, lib_name, params, used_names, **kw
        self._sch_params = None

    @property
    def sch_params(self):
        return self._sch_params

    @classmethod
    def get_params_info(cls):
        """Returns a dictionary containing parameter descriptions.

        Override this method to return a dictionary from parameter names to d

        Returns
        -------
        param_info : dict[str, str]
            dictionary from parameter name to description.
        """
        return dict(
            lch='channel length, in meters.',
            w_dict='width dictionary.',
            intent_dict='intent dictionary.',
            fg_dict='number of fingers dictionary.',
            ndum='number of dummies on each side.',
            ptap_w='NMOS substrate width, in meters/number of fins.',
            ntap_w='PMOS substrate width, in meters/number of fins.',
            show_pins='True to draw pin geometries.',
        )

    def draw_layout(self):
        """Draw the layout of a transistor for characterization.
        """

        lch = self.params['lch']
        w_dict = self.params['w_dict']
        intent_dict = self.params['intent_dict']
        fg_dict = self.params['fg_dict']
        ndum = self.params['ndum']
```