SIMCEO

Simulink Client
CEO Server

R. Conan
GMTO Corporation

August 16, 2019

Contents

1 Introduction

2 Installation

3 Implementation

4 The python server module

The python interface consists in the module *simulink*:

3 ⟨*simceo.py* 3⟩≡

```
import sys
import threading
import time
import zmq
import ceo
import numpy as np
from collections import OrderedDict
import os
import shelve
import traceback
import scipy.linalg as LA
import pickle
import zlib
import logging

logging.basicConfig()

SIMCEOPATH = os.path.abspath(os.path.dirname(__file__))

class testComm:
    def __init__(self):
        pass
    def hello(self,N=1):
        data = np.ones(N)
        return dict(data=data.tolist())

class Timer(object):
    def __init__(self, name=None):
        self.name = name

    def __enter__(self):
        self.tstart = time.time()

    def __exit__(self, type, value, traceback):
        if self.name:
```

```
            print('[%s]' % self.name)
        print('Elapsed time: %s' % (time.time() - self.tstart))
```

⟨*CalibrationMatrix* 22⟩

⟨*S-function* 9⟩

⟨*SGMT* 10a⟩
⟨*SAtmosphere* 12⟩
⟨*SOpticalPath* 13a⟩

⟨*broker* 5⟩

```
if __name__ == "__main__":

    print("********************************")
    print("**   STARTING SIMCEO SERVER    **")
    print("********************************")
    args = sys.argv[1:]
    verbose = int(args[0]) if args else logging.INFO
    agent = broker(verbose=verbose)
    agent.start()
```

4.1 The broker class

The broker class receives requests from the Simulink S–functions, processes the requests and sends a replies to the Simulink client. It inherits from the *threading.Thread* class.

5 ⟨*broker* 5⟩≡ (3)

```
class broker(threading.Thread):

    def __init__(self, verbose=logging.INFO):

        threading.Thread.__init__(self)

        self.logger = logging.getLogger(self.__class__.__name__)
        self.verbose = verbose
        self.logger.setLevel(self.verbose)

        self.context = zmq.Context()
        self.socket = self.context.socket(zmq.REP)
        self.address = "tcp://*:3650"
        self.socket.bind(self.address)
        self.loop = True

        self.ops = []
        self.n_op = 0
        self.currentTime = 0.0
        self.satm = SAtmosphere(self.ops,verbose=self.verbose)
        self.sgmt = SGMT(self.ops, self.satm, verbose=self.verbose)

    def __del__(self):

        self.release()

    def release(self):

        self.socket.close()
        self.context.term()

    def _send_(self,obj,protocol=-1,flags=0):
        pobj = pickle.dumps(obj,protocol)
        zobj = zlib.compress(pobj)
        self.socket.send(zobj, flags=flags)

    def _recv_(self,flags=0):
        zobj = self.socket.recv(flags)
        pobj = zlib.decompress(zobj)
        return pickle.loads(pobj)
```

⟨*broker get item* 8a⟩

⟨*broker run* 6a⟩

The *run* method

6a ⟨*broker run* 6a⟩≡ (5)

```
def run(self):

    while self.loop:

        ⟨broker run details 6b⟩
```

waits for a request from a Simulink S–function:

6b ⟨*broker run details* 6b⟩≡ (6a) 7▷

```
#jmsg = ubjson.loadb(msg)
msg = ''
try:
    self.logger.debug('Waiting for message ...')
    #msg = self.socket.recv()
    #jmsg = ubjson.loadb(msg)
    msg = self._recv_()
    self.logger.debug('Received: %s',msg)
except Exception as E:
    #print("Error raised by ubjson.loadb by that does not stop us!")
    print(msg)
    raise
```

The message received from the S–function contains

- the Simulink simulation time *currentTime*,
- a class identifier, *class_id*: **GMT** for *SGMT*, **ATM** for *SAtmosphere* or **OP** for *SOpticalPath*,
- a method identifier, *method_id*: **Start**, **Terminate**, **Update** or **Outputs**,
- a dictionnary of the arguments to the method, *args*.

The class method is invoked with:

7 ⟨*broker run details* 6b⟩+≡ (6a) ◁6b 8b▷

```
#self.currentTime = float( jmsg["currentTime"][0][0] )
if not 'class_id' in msg:
    self._send_("SIMCEO server received: {}".format(msg))
    continue
class_id  = msg["class_id"]
method_id = msg["method_id"]
self.logger.debug('Calling out: %s.%s',class_id,method_id)
#print "@ %.3fs: %s->%s"%(currentTime,jmsg["tag"],method_id)
#tid = ceo.StopWatch()
try:
    #tid.tic()
    args_out = getattr( self[class_id], method_id )( **msg["args"] )
    #tid.toc()
    #print "%s->%s: %.2f"%(class_id,method_id,tid.elapsedTime)
except Exception as E:
    print("@(broker)> The server has failed!")
    print(msg)
    traceback.print_exc()
    print("@(broker)> Recovering gracefully...")
    class_id = ""
    args_out = "The server has failed!"
```

The dictionary–like call is implemented with

8a ⟨*broker get item* 8a⟩≡ (5)

```
def __getitem__(self,key):
    if key=="GMT":
        return self.sgmt
    elif key=="ATM":
        return self.satm
    elif key[:2]=="OP":
        if key[2:]:
            op_idx = int(key[2:]) - self.n_op + len(self.ops)
            return self.ops[op_idx]
        else:
            self.ops.append( SOpticalPath( len(self.ops) ,
                                           self.sgmt.gmt ,
                                           self.satm.atm ,
                                           verbose=self.verbose) )
            self.n_op = len(self.ops)
            return self.ops[-1]
    elif key=='testComm':
        return testComm()
    else:
        raise KeyError("Available keys are: GMT, ATM or OP")
```

Each optical paths that is defined in the Simulink model is affected an unique ID tag made of the string **OP** followed by the index of the object in the optical path list *ops*. If the ID tag of the optical path is just **OP**, a new *SOpticalPath* object is instanciated and appended to the list of optical path.

When the *Terminate* method of an *SOpticalPath* object is called, the object is removed from the optical path list *ops*.

8b ⟨*broker run details* 6b⟩+≡ (6a) ◁7 8c▷

```
if class_id[:2]=="OP" and method_id=="Terminate":
    self.ops.pop(0)
```

The value return by the method of the invoked object is sent back to the S–function:

8c ⟨*broker run details* 6b⟩+≡ (6a) ◁8b

```
#self.socket.send(ubjson.dumpb(args_out,no_float32=True))
self._send_(args_out)
```

4.2 The S classes

The S classes, *SGMT*, *SAtmosphere* and *SOpticalPath*, are providing the interface with CEO classes. They mirror the *Level–2 Matlab S–functions* by implementing the same method *Start*, *InitializeConditions*, *Terminate*, *Update* and *Outputs*. Each method is triggered by the corresponding function in the Matlab S–function with the exception of the *Update* method that is triggered by the *Outputs* function of the S–function.

An abstract class, *Sfunction*, implements the four S–function method:

9 ⟨*S-function* 9⟩≡ (3)

```
from abc import ABCMeta, abstractmethod

class Sfunction:
    __metaclass__ = ABCMeta
    @abstractmethod
    def Start(self):
        pass
    @abstractmethod
    def Terminate(self):
        pass
    @abstractmethod
    def Update(self):
        pass
    @abstractmethod
    def Outputs(self):
        pass
    @abstractmethod
    def InitializeConditions(self):
        pass
```

4.2.1 The SGMT class

The *SGMT* class is the interface class between a CEO *GMT_MX* object and a *GMT Mirror* Simulink block.

10a ⟨*SGMT* 10a⟩≡ (3) 10c▷

```
class SGMT(Sfunction):

    def __init__(self, ops, satm, verbose=logging.INFO):
        self.logger = logging.getLogger(self.__class__.__name__)
        self.logger.setLevel(verbose)
        self.logger.info('Instantiate')
        self.gmt  = ceo.GMT_MX()

    def Terminate(self, args=None):
        self.logger.info('Terminate')
        self.gmt = ceo.GMT_MX()
        return "GMT deleted!"
```

Start The message that triggers the call to the *Start* method is

10b ⟨*SGMT Start message* 10b⟩≡

```
{
"class_id": "GMT",
"method_id": "Start",
"args":
  {
    "mirror": "M1"|"M2",
    "mirror_args":
      {
        "mirror_modes": u"bending modes"|u"zernike",
        "N_MODE": 162,
        "radial_order": ...
      }
  }
}
```

10c ⟨*SGMT* 10a⟩+≡ (3) ◁10a 11b▷

```
    def Start(self,mirror=None,mirror_args={}):
        self.logger.info('Start')
        if mirror_args:
            self.gmt[mirror] = getattr(ceo,"GMT_"+mirror)( **mirror_args )
        return "GMT"
```

Update The message that triggers the call to the *Update* method is

11a ⟨*SOpticalPath Update message* 11a⟩≡ 15c▷

```
{
"class_id": "GMT",
"method_id": "Update",
"args":
  {
    "mirror": "M1"|"M2",
    "inputs":
      {
        "TxyzRxyz": null,
        "mode_coefs": null
      }
  }
}
```

11b ⟨*SGMT* 10a⟩+≡ (3) ◁10c 11c▷

```
    def Update(self, mirror=None, inputs=None):
        for key in inputs:
            data = np.array( inputs[key], order='C', dtype=np.float64 )
            #data = np.transpose( np.reshape( data , (-1,7) ) )
            if key=="TxyzRxyz":
                self.gmt[mirror].motion_CS.origin[:]       = data[:,:3]
                self.gmt[mirror].motion_CS.euler_angles[:] = data[:,3:]
                self.gmt[mirror].motion_CS.update()
            elif key=="Rxy":
                self.gmt[mirror].motion_CS.euler_angles[:,:2] = data
                self.gmt[mirror].motion_CS.update()
            elif key=="mode_coefs":
                    self.gmt[mirror].modes.a[:] = data
                    self.gmt[mirror].modes.update()
```

InitializeConditions

11c ⟨*SGMT* 10a⟩+≡ (3) ◁11b 11d▷

```
    def Init(self, args=None):
        pass
```

Outputs

11d ⟨*SGMT* 10a⟩+≡ (3) ◁11c

```
    def Outputs(self, args=None):
        pass
```

4.2.2 The SAtmosphere class

The *SAtmosphere* class is the interface class between a CEO *GmtAtmosphere* object and a *Atmosphere* Simulink block.

12 ⟨*SAtmosphere* 12⟩≡ (3)

```
class _Atmosphere_():
    def __init__(self,**kwargs):
        print(kwargs)
        self.__atm = ceo.GmtAtmosphere(**kwargs)
        self.N = kwargs['NXY_PUPIL']
        self.L = kwargs['L']
        self.delta = self.L/(self.N-1)
    def propagate(self,src):
        self.__atm.ray_tracing(src,self.delta,self.N,self.delta,self.N,src.timeStamp)
class SAtmosphere(Sfunction):

    def __init__(self, ops, verbose=logging.INFO):
        self.logger = logging.getLogger(self.__class__.__name__)
        self.logger.setLevel(verbose)
        self.atm = None

    def Start(self, **kwargs):
        print("\n@(SAtmosphere:Start)>")
        #self.atm = _Atmosphere_( **kwargs )
        self.atm = ceo.GmtAtmosphere(**kwargs)
        return "ATM"

    def Terminate(self, args=None):
        self.logger.info("Atmosphere deleted")
        self.atm = None
        return "Atmosphere deleted!"

    def InitializeConditions(self, args=None):
        pass

    def Outputs(self, args=None):
        pass

    def Update(self, args=None):
        pass
```

Uses `Atmosphere` 39 and `src` 14.

4.2.3 The SOpticalPath class

The *SOpticalClass* gathers a source object *src*, the GMT model object *gmt*, an atmosphere object *atm*, a sensor object *sensor* and a calibration source *calib_src*.

13a ⟨*SOpticalPath* 13a⟩≡ (3) 14▷

```
class SOpticalPath(Sfunction):

    def __init__(self, idx, gmt, atm, verbose=logging.INFO):
        self.logger = logging.getLogger(self.__class__.__name__)
        self.logger.setLevel(verbose)
        self.logger.info('Instantiate')
        self.idx = idx
        self.gmt = gmt
        self.atm = atm
        self.sensor = None
```

Defines:
`idx`, used in chunk 14.
`sensor`, used in chunks 14–17 and 21.

Start The message that triggers the call to the *Start* method is

13b ⟨*SOpticalPath Start message* 13b⟩≡

```
{
"class_id": "OP",
"method_id": "Start",
"args":
  {
    "source_args": { ... } ,
    "sensor_class": null|"Imaging"|"ShackHartmann",
    "sensor_args": null|{ ... },
    "calibration_source": null|{ ... },...
    "miscellaneous_args": null|{...}
  }
}
```

14 ⟨*SOpticalPath* 13a⟩+≡ (3) ◁13a 15b▷

```
def Start(self,source_args=None, source_attributes={},
          sensor_class=None, sensor_args=None,
          calibration_source_args=None, calibrate_args=None):
    self.pssn_data = None
    #self.propagateThroughAtm = miscellaneous_args['propagate_through_atmosphere']
    self.logger.info('Instantiating source')
    self.src = ceo.Source( **source_args )
    for key in source_attributes:
        attr = source_attributes[key]
        if isinstance(attr,dict):
            for kkey in attr:
                setattr(getattr(src,key),kkey,atrr[kkey])
        else:
            setattr(src,key,attr)
    self.src.reset()
    self.gmt.reset()
    self.gmt.propagate(self.src)
    self.sensor_class = sensor_class

    if not (sensor_class is None or sensor_class=='None'):

        self.logger.info('Instantiating sensor')
        self.logger.debug(sensor_class)
        self.logger.debug(sensor_args)
        self.sensor = getattr(ceo,sensor_class)( **sensor_args )
        if calibration_source_args is None:
            self.calib_src = self.src
        else:
            self.calib_src = ceo.Source( **calibration_source_args )

        self.sensor.reset()
        if calibrate_args is not None:
            self.sensor.calibrate(self.calib_src, **calibrate_args)
        #print "intensity_threshold: %f"%sensor_args['intensityThreshold']

        self.sensor.reset()
        self.comm_matrix = {}

    self.src>>tuple(filter(None,(self.atm,self.gmt,self.sensor)))

    return "OP"+str(self.idx)
```

Defines:
`exposure_start`, never used.
`exposure_time`, never used.
`propagateThroughAtm`, never used.

src, used in chunks 12, 15d, 17, and 21.
Uses idx 13a and sensor 13a.

Terminate The message that triggers the call to the *Terminate* method is

15a ⟨*SOpticalPath Terminate message* 15a⟩≡

```
{
"class_id": "OP",
"method_id": "Terminate",
"args":
  {
    "args": null
  }
}
```

15b ⟨*SOpticalPath* 13a⟩+≡ (3) ◁14 15d▷

```
    def Terminate(self, args=None):
        self.logger.info("OpticalPath deleted")
        return "OpticalPath deleted!"
```

Update The message that triggers the call to the *Update* method is

15c ⟨*SOpticalPath Update message* 11a⟩+≡ ◁11a

```
{
"class_id": "OP",
"method_id": "Update",
"args":
  {
    "inputs": null
  }
}
```

15d ⟨*SOpticalPath* 13a⟩+≡ (3) ◁15b 16b▷

```
    def Update(self, inputs=None):
        self.logger.debug('src time stamp: %f',self.src.timeStamp)
        +self.src
        #self.src.reset()
        #self.gmt.propagate(self.src)
        #self.sensor.propagate(self.src)
```

Uses sensor 13a and src 14.

Outputs The message that triggers the call to the *Outputs* method is

16a ⟨*SOpticalPath Outputs message* 16a⟩≡

```
{
"class_id": "OP",
"method_id": "Outputs",
"args":
  {
      "outputs": ["wfe_rms"|"segment_wfe_rms"|"piston"|"segment_piston"|"ee80"]
  }
}
```

16b ⟨*SOpticalPath* 13a⟩+≡ (3) ◁15d 17▷

```
    def Outputs(self, outputs=None):
        if self.sensor is None:
            doutputs = OrderedDict()
            for element in outputs:
                doutputs[element] = self[element]
        else:
            #+self.sensor
            self.sensor.process()
            doutputs = OrderedDict()
            for element in outputs:
                doutputs[element] = self[element]
            self.sensor.reset()
        return doutputs
```

Uses `sensor` 13a.

and the dictionnary implementation is

17 ⟨*SOpticalPath* 13a⟩+≡ (3) ◁16b 21▷

```
def __getitem__(self,key):
    if key=="wfe_rms":
        return self.src.wavefront.rms(units_exponent=-6).tolist()
    elif key=="segment_wfe_rms":
        return self.src.phaseRms(where="segments",
                                 units_exponent=-6).tolist()
    elif key=="piston":
        return self.src.piston(where="pupil",
                               units_exponent=-6).tolist()
    elif key=="segment_piston":
        return self.src.piston(where="segments",
                               units_exponent=-6).tolist()
    elif key=="tiptilt":
        buf = self.src.wavefront.gradientAverage(1,self.src.rays.L)
        buf *= ceo.constants.RAD2ARCSEC
        return buf.tolist()
    elif key=="segment_tiptilt":
        buf = self.src.segmentsWavefrontGradient().T
        buf *= ceo.constants.RAD2ARCSEC
        return buf.tolist()
    elif key=="ee80":
        #print "EE80=%.3f or %.3f"%(self.sensor.ee80(from_ghost=False),self.sensor.ee8
        return self.sensor.ee80(from_ghost=False).tolist()
    elif key=="PSSn":
        if self.pssn_data is None:
            pssn , self.pssn_data = self.gmt.PSSn(self.src,save=True)
        else:
            pssn = self.gmt.PSSn(self.src,**self.pssn_data)
        return pssn
    elif hasattr(self.src,key):
        return getattr(self.src,key)
    elif hasattr(self.sensor,key):
        return getattr(self.sensor,key)
    else:
        c = self.comm_matrix[key].dot( self.sensor.Data ).ravel()
        return c.tolist()
```

Uses `sensor` 13a and `src` 14.

InitializeConditions The message that triggers a call to the *InitializeConditions* method is

18 ⟨*SOpticalPath InitializeConditions message* 18⟩≡ 19 ▷

```
{
"class_id": "OP",
"method_id": "InitializeConditions",
"args":
  {
      "calibrations":
      {
          "M2_TT":
          {
              "method_id": "calibrate",
              "args":
              {
                  "mirror": "M2",
                  "mode": "segment tip-tilt",
                  "stroke": 1e-6
              }
          },
      },
      "pseudo_inverse":
      {
          "nThreshold": null
      },
      "filename": null
  }
}
```

19 ⟨*SOpticalPath InitializeConditions message* 18⟩+≡ ◁18 20▷

```
{
"class_id": "OP",
"method_id": "InitializeConditions",
"args":
  {
      "calibrations":
      {
          "M12_Rxyz": [
              {
                  "method_id": "calibrate",
                  "args":
                  {
                      "mirror": "M1",
                      "mode": "Rxyz",
                      "stroke": 1e-6
                  }
              },
              {
                  "method_id": "calibrate",
                  "args":
                  {
                      "mirror": "M2",
                      "mode": "Rxyz",
                      "stroke": 1e-6
                  }
              }]
      },
      "pseudo-inverse":
      {
          "nThreshold": [0],
          "concatenate": true
      },
      "filename": null
  }
}
```

20 ⟨*SOpticalPath InitializeConditions message* 18⟩+≡ ◁19

```
{
"class_id": "OP",
"method_id": "InitializeConditions",
"args":
  {
      "calibrations":
      {
          "AGWS":
          {
              "method_id": "AGWS_calibrate",
              "args":
              {
                  "decoupled": true,
                  "stroke": [1e-6,1e-6,1e-6,1e-6,1e-6],
                  "fluxThreshold": 0.5
              }
          }
      },
      "pseudo-inverse":
      {
          "nThreshold": [2,2,2,2,2,2,0],
          "insertZeros": [null,null,null,null,null,null,[2,4,6]]
      },
      "filename": null
  }
}
```

21 ⟨*SOpticalPath* 13a⟩+≡ (3) ◁17

```
def Init(self, calibrations=None, filename=None,
     pseudo_inverse={}):
    self.logger.info('INIT')
    if calibrations is not None:
        if filename is not None:
            filepath = os.path.join(SIMCEOPATH,"calibration_dbs",filename)
            db = shelve.open(filepath)

            if os.path.isfile(filepath+".dir"):
                self.logger.info("Loading command matrix from existing database %s!",f
                for key in db:
                    C = db[key]
                    #C.nThreshold = [SVD_truncation[k]]
                    self.comm_matrix[key] = C
                    db[key] = C
                db.close()
                return

        with Timer():
            for key in calibrations: # Through calibrations
                self.logger.info('Calibrating: %s',key)
                calibs = calibrations[key]
                #Gif not isinstance(calibs,list):
                #    calibs = [calibs]
                #GD = []
                #for c in calibs: # Through calib
                self.gmt.reset()
                self.src.reset()
                self.sensor.reset()
                D = getattr( self.gmt, calibs["method_id"] )( \
                                    self.sensor,
                                    self.src,
                                    **calibs["args"])
                C = ceo.CalibrationVault([D],**pseudo_inverse)
                self.gmt.reset()
                self.src.reset()
                self.sensor.reset()
                self.comm_matrix[key] = C


        if filename is not None:
            self.logger.info("Saving command matrix to database %s!",filename)
            db[str(key)] = C
            db.close()
```

Uses sensor 13a and src 14.

4.3 The CalibrationMatrix class

The *CalibrationMatrix* class is a container for several matrices:

- the poke matrix D,
- the eigen modes U, V and eigen values S of the singular value decomposition of $D = USV^T$
- the truncated inverse M of D, $M = V\Lambda U^T$ where

$$\begin{aligned} \Lambda_i &= 1/S_i, \quad \forall i < n \\ \Lambda_i &= 0, \quad \forall i \geq n \end{aligned}$$

22 ⟨*CalibrationMatrix* 22⟩≡ (3)

```
class CalibrationMatrix(object):

    def __init__(self, D, n,
                 decoupled=True, flux_filter2=None,
                 n_mode = None):
        print("@(CalibrationMatrix)> Computing the SVD and the pseudo-inverse...")
        self._n = n
        self.decoupled = decoupled
        if self.decoupled:
            self.nSeg = 7
            self.D = D
            D_s = [ np.concatenate([D[0][:,k*3:k*3+3],
                                    D[1][:,k*3:k*3+3],
                                    D[2][:,k*3:k*3+3],
                                    D[3][:,k*3:k*3+3],
                                    D[4][:,k*n_mode:k*n_mode+n_mode]],axis=1) for k in ran
            for k in range(7):
                D_s[k][np.isnan(D_s[k])] = 0
            lenslet_array_shape = flux_filter2.shape

            ### Identification process
            # The non-zeros entries of the calibration matrix are identified by filtering
            # which are a 1000 less than the maximum of the absolute values of the matrix
            # collapsing (summing) the matrix along the mirror modes axis.
            Qxy = [ np.reshape( np.sum(np.abs(D_s[k])>1e-2*np.max(np.abs(D_s[k])),axis=1)!
            # The lenslet flux filter is applied to the lenslet segment filter:
            Q = [ np.logical_and(X,flux_filter2) for X in Qxy ]
            # A filter made of the lenslet used more than once is created:
            Q3 = np.dstack(Q).reshape(flux_filter2.shape + (self.nSeg,))
            Q3clps = np.sum(Q3,axis=2)
            Q3clps = Q3clps>1
            # The oposite filter is applied to the lenslet segment filter leading to 7 val
```

```
            # one filter per segment and no lenslet used twice:
            self.VLs = [ np.logical_and(X,~Q3clps) for X in Q]

            # Each calibration matrix is reduced to the valid lenslet:
            D_sr = [ D_s[k][self.VLs[k].ravel(),:] for k in range(self.nSeg) ]
            print([ D_sr[k].shape for k in range(self.nSeg)])
            # Computing the SVD for each segment:
            self.UsVT = [LA.svd(X,full_matrices=False) for X in D_sr]

            # and the command matrix of each segment
            self.M = [ self.__recon__(k) for k in range(self.nSeg) ]
        else:
            self.D = np.concatenate( D, axis=1 )
            with Timer():
                self.U,self.s,self.V = LA.svd(self.D,full_matrices=False)
                self.V = self.V.T
                iS = 1./self.s
                if self._n>0:
                    iS[-self._n:] = 0
                self.M = np.dot(self.V,np.dot(np.diag(iS),self.U.T))

    def __recon__(self,k):
        iS = 1./self.UsVT[k][1]
        if self._n>0:
            iS[-self._n:] = 0
        return np.dot(self.UsVT[k][2].T,np.dot(np.diag(iS),self.UsVT[k][0].T))

    @property
    def nThreshold(self):
        "# of discarded eigen values"
        return self._n
    @nThreshold.setter
    def nThreshold(self, value):
        print("@(CalibrationMatrix)> Updating the pseudo-inverse...")
        self._n = value
        if self.decoupled:
            self.M = [ self.__recon__(k) for k in range(self.nSeg) ]
        else:
            iS = 1./self.s
            if self._n>0:
                iS[-self._n:] = 0
            self.M = np.dot(self.V,np.dot(np.diag(iS),self.U.T))

    def dot( self, s ):
        if self.decoupled:
            return np.concatenate([ np.dot(self.M[k],s[self.VLs[k].ravel()]) for k in rang
```

```
        else:
            return np.dot(self.M,s)
```

4.4 The Sensor abstract class

24a ⟨*Sensor abstract class* 24a⟩≡

```
class Sensor:
    __metaclass__ = ABCMeta
    @abstractmethod
    def calibrate(self):
        pass
    @abstractmethod
    def reset(self):
        pass
    @abstractmethod
    def analyze(self):
        pass
    @abstractmethod
    def propagate(self):
        pass
    @abstractmethod
    def process(self):
        pass
```

5 DOS

`dos` is the interface to the dynamic optical simulation. A `dos` simulation is defined with a parameter file `dos.yaml`. `dos.yaml` is divided into several sections.

24b ⟨*dos.yaml* 24b⟩≡

⟨*dos simulation section* 25a⟩
⟨*dos drivers section* 32a⟩

24c ⟨*init.py* 24c⟩≡

```
from .dos import DOS
from . import driver
```

5.1 Simulation

The first section is `simulation` where the simulation sampling frequency and duration is given as well as the address of the SIMCEO server.

25a ⟨*dos simulation section* 25a⟩≡ (24b)

```
simulation:
  sampling frequency: # [Hertz]
  duration: # [seconds]
  server:
    IP: # 127.0.0.1
```

The `DOS` class acts as the simulation conductor. It is initialized with the path to the directory where the configuration and parameter files reside.

25b ⟨*dos imports* 25b⟩≡ (26) 41a▷

```
import os
import yaml
import logging
import threading
import numpy as np
from . import driver
from simceo import Timer
```

26 ⟨*dos.py* 26⟩≡ 40 ▷

```
⟨dos imports 25b⟩

logging.basicConfig()

class DOS(threading.Thread):
    def __init__(self,path_to_config_dir,verbose=logging.INFO):

        threading.Thread.__init__(self)

        self.logger = logging.getLogger(self.__class__.__name__)
        self.logger.setLevel(verbose)

        cfg_file = os.path.join(path_to_config_dir,'dos.yaml')
        self.logger.info('Reading config from %s',cfg_file)
        with open(cfg_file) as f:
            self.cfg = yaml.load(f)

        self.agent = broker(self.cfg['simulation']['server']['IP'])

        self.N_SAMPLE = int(self.cfg['simulation']['sampling frequency']*
                            self.cfg['simulation']['duration'])
        self.__k_step = 0
        self.pushed = False
        self.initialized = False
        ⟨check parameter file existence 27a⟩
        ⟨linking the drivers IO 27b⟩
        ⟨device to driver association 28d⟩
        ⟨starting the drivers 30c⟩
        ⟨initializing the drivers 30e⟩
        ⟨running the loop 30g⟩
        ⟨terminating the drivers 31c⟩
        self.logger.info('Simulation setup for a duration of {0}s @ {1}Hz ({2} steps)!'.fo
            self.cfg['simulation']['duration'],
            self.cfg['simulation']['sampling frequency'],
            self.N_SAMPLE))

    ⟨starting the simulation 30d⟩

    ⟨initializing the simulation 30f⟩

    ⟨stepping through 31a⟩

    ⟨running the simulation 31b⟩

    ⟨terminating the simulation 31d⟩
```

⟨*timing diagram* 42⟩

```
@property
def pctComplete(self):
    return round(100*self.__k_step/(self.N_SAMPLE-1))
```

Each device must have a corresponding parameter file in the same directory than the configuration file.

27a ⟨*check parameter file existence* 27a⟩≡ (26)

```
tau = 1/self.cfg['simulation']['sampling frequency']
self.logs = Logs(tau)
self.drivers = {}
for d,v in self.cfg['drivers'].items():
    prm_file = os.path.join(path_to_config_dir,d+'.yaml')
    if os.path.isfile(prm_file):
        self.logger.info('New driver: %s',d)
        if 'server' in v and v['server'] is False:
            self.drivers[d] = driver.Client(tau,d,
                                            self.logs,
                                            verbose=verbose,**v)
        elif d=='atmosphere':
            self.drivers[d] = driver.Atmosphere(tau,d,self.agent,
                                                verbose=verbose)
        else:
            self.drivers[d] = driver.Server(tau,d,
                                            self.logs,
                                            self.agent,
                                            verbose=verbose,**v)
    else:
        self.logger.warning('%s is missing!',prm_file)
```

Uses `Atmosphere` 39, `Client` 37b, and `Server` 36.

Once each driver is instantiated, their inputs and outputs are tied

27b ⟨*linking the drivers IO* 27b⟩≡ (26)

```
for k_d in self.drivers:
    d = self.drivers[k_d]
    for k_i in d.inputs:
        d.inputs[k_i].tie(self.drivers)
    for k_o in d.outputs:
        d.outputs[k_o].tie(self.drivers)
```

Uses `tie` 28c.

The `Input` and `Output` `tie` methods set the `data` pointer when a `lien` to another `Driver` exists:

28a ⟨*IO linking* 28a⟩≡ (28)

```
def tie(self,drivers):
    if self.lien is not None:
        d,io = self.lien
        self.logger.info('Linked to %s from %s',io,d)
```

Uses `tie` 28c.

28b ⟨*input linking* 28b⟩≡ (33c)

⟨*IO linking* 28a⟩

```
        self.data = drivers[d].outputs[io].data
        self.size = self.data.shape
```

28c ⟨*output linking* 28c⟩≡ (34a)

⟨*IO linking* 28a⟩

```
        self.data = drivers[d].inputs[io].data
        self.size = self.data.shape
```

Defines:
`tie`, used in chunks 27b and 28a.

The device parameters are loaded from the device parameter file and the device is associated to the driver

28d ⟨*device to driver association* 28d⟩≡ (26)

```
for k_d in self.drivers:
    d = self.drivers[k_d]
    device = os.path.join(path_to_config_dir,k_d+'.yaml')
    d.associate(device)
```

The device parameter are formatted into messages that will be used to communicate to the server.

29 ⟨*Server device parameter loading and formatting* 29⟩≡ (34b)

```
def associate(self,prm_file):
    base_units = np.pi/180
    units = {'degree': base_units,
             'arcmin': base_units/60,
             'arcsec': base_units/60/60,
             'mas': base_units/60/60/1e3}
    with open(prm_file) as f:
        prm = yaml.load(f)
    if 'mirror' in prm:
        self.msg['class_id'] = 'GMT'
        self.msg_args['Start'].update(prm)
        self.msg_args['Update']['mirror'] = prm['mirror']
        self.msg_args['Update']['inputs'].update(\
                {k_i:v_i.data for k_i,v_i in self.inputs.items()})
    else:
        self.msg['class_id'] = 'OP'
        if isinstance(prm['source']['zenith'],dict):
            prm['source']['zenith'] = np.asarray(prm['source']['zenith']['value'])*\
                                      units[prm['source']['zenith']['units']]
        if isinstance(prm['source']['azimuth'],dict):
            prm['source']['azimuth'] = np.asarray(prm['source']['azimuth']['value'])*\
                                       units[prm['source']['azimuth']['units']]
        prm['source'].update({'samplingTime':self.tau*self.sampling_rate})
        self.msg_args['Start'].update({'source_args':prm['source'],
                                       'sensor_class':prm['sensor']['class'],
                                       'sensor_args':{},
                                       'calibration_source_args':None,
                                       'calibrate_args':None})

        if 'source_attributes' in prm['source']:
            if 'rays' in prm['source']['source_attributes'] and \
               'rot_angle' in  prm['source']['source_attributes']['rays'] and \
               isinstance(prm['source']['source_attributes']['rays']['rot_angle'],dict):
                prm['source']['source_attributes']['rays']['rot_angle'] = \
                  np.asarray(prm['source']['source_attributes']['rays']['rot_angle']['valu
                   units[prm['source']['source_attributes']['rays']['rot_angle']['units']]
                self.msg_args['Start'].update({'source_attributes':prm['source']['source_a

        if prm['sensor']['class'] is not None:
            self.msg_args['Start']['sensor_args'].update(prm['sensor']['args'])
            self.msg_args['Start']['calibrate_args'] = prm['sensor']['calibrate_args']
        if 'interaction matrices' in prm:
```

```
            self.msg_args['Init'].update(prm['interaction matrices'])
        self.msg_args['Outputs']['outputs'] += [k_o for k_o in self.outputs]
```

30a ⟨*driver imports* 30a⟩≡ (32c) 32b▷

```
from scipy import signal
```

30b ⟨*Client device parameter loading and formatting* 30b⟩≡ (37b)

```
def associate(self,prm_file):
    with open(prm_file) as f:
        prm = yaml.load(f)
    if 'transfer function' in prm['system']:
        system = prm['system']
        self.system = signal.dlti(system['transfer function']['num'],
                                  system['transfer function']['denom'])
    elif 'zeros poles gain' in prm['system']:
        system = prm['system']
        self.system = signal.dlti(system['transfer function']['zeros'],
                                  system['transfer function']['poles'],
                                  system['transfer function']['gain'])
    else:
        raise Exception("System should be of the type "+\
                        "'transfer function' or 'zeros poles gains'")
```

Next we check if an atmosphere is define

Once the parameters are loaded and the drivers linked, we call the drivers `start`

30c ⟨*starting the drivers* 30c⟩≡ (26)

```
self.__start = map(lambda x: x.start(), self.drivers.values())
```

30d ⟨*starting the simulation* 30d⟩≡ (26)

```
def push(self):
    self.logger.info('Pushing configuration to server')
    list(self.__start)
    self.pushed = True
```

and `init` methods:

30e ⟨*initializing the drivers* 30e⟩≡ (26)

```
self.__init = map(lambda x: x.init(), self.drivers.values())
```

30f ⟨*initializing the simulation* 30f⟩≡ (26)

```
def init(self):
    self.logger.info('Initializing')
    list(self.__init)
    self.initialized = True
```

Then the `update` and `output` methods are called successively for the total `duration` of the simulation.

30g ⟨*running the loop* 30g⟩≡ (26)

```
self.step = self.stepping()
```

31a ⟨*stepping through* 31a⟩≡ (26)

```
def stepping(self):
    v = self.drivers.values()
    for l in range(self.N_SAMPLE):
        self.logger.debug('Step #%d',l)
        yield [x.update(l) for x in v] + [x.output(l) for x in v]
```

31b ⟨*running the simulation* 31b⟩≡ (26)

```
def run(self):
    if not self.pushed:
        self.push()
    if not self.initialized:
        self.init()
    self.logger.info('Running')
    with Timer():
        for self.__k_step in range(self.N_SAMPLE):
            next(self.step)
    self.terminate()
def _run_(self):
    if not self.pushed:
        self.push()
    if not self.initialized:
        self.init()
    self.logger.info('Running')
    with Timer():
        for self.__k_step in range(self.N_SAMPLE):
            next(self.step)
    self.terminate()
```

The simulation ends-up with calling the `terminate` methods.

31c ⟨*terminating the drivers* 31c⟩≡ (26)

```
self.__terminate = map(lambda x: x.terminate(), self.drivers.values())
```

31d ⟨*terminating the simulation* 31d⟩≡ (26)

```
def terminate(self):
    self.logger.info('Terminating')
    list(self.__terminate)
```

5.2 DOS driver

The next section is the `drivers` section. This section lists all the devices that makes the simulation. There is a many subsections as drivers. A `drivers` has a unique name `device name` that must be matched by a parameter file of the same name `device name`.yaml. An object is associated to each device. The object have the following methods: `start`,`init`,`update`,`output` and `terminate`. Each device execute first the `start` method followed by the `init` method. Then after `delay` samples, the `update` method is called at the given `sampling rate` reading its inputs. Each device inputs is defined by a name and has for properties either a size or a list with the origin device and origin device output name. The `update` method is followed by the `output` method. Each device outputs is defined by a name and has for properties a given sampling frequency and either a size or a list with the input destination device and destination device input name.

32a ⟨*dos drivers section* 32a⟩≡ (24b)

```
drivers:
  device name:
    server: true
    delay: 0 # [sample]
    sampling rate: 1 # [sample]
    inputs:
      input name:
        size: 0
        lien: [device, device output name]
    outputs:
      output name:
        sampling rate: 1 # [sample]
        size: 0
        lien: [device, device input name]
```

The driver module defines classes that are interfaced to devices either of the server or on the client.

32b ⟨*driver imports* 30a⟩+≡ (32c) ◁30a

```
import numpy as np
import yaml
import logging
```

32c ⟨*driver.py* 32c⟩≡ 33a▷

```
⟨driver imports 30a⟩
logging.basicConfig()
⟨IO 33b⟩
⟨Inputs 33c⟩
⟨Outputs 34a⟩
```

All class must have the following methods: `start`, `init`, `update`, `output` and `terminate`.

33a ⟨*driver.py* 32c⟩+≡ ◁32c 34b▷

```
class Driver:
    def __init__(self,tau,tag):
        self.tau = tau
        self.tag = tag
    def start(self):
        pass
    def init(self):
        pass
    def update(self,_):
        pass
    def output(self,_):
        pass
    def terminate(self):
        pass
```

Defines:
`Driver`, used in chunks 34b, 35, 37b, and 39.

5.2.1 Driver inputs/outputs

Inputs and outputs are saved as dictionaries with the input and output names as keys and the values being an instance of the `Inputs` and `Outputs` classes.

33b ⟨*IO* 33b⟩≡ (32c)

```
class IO:
    def __init__(self,tag,size=0, lien=None, logs=None):
        self.logger = logging.getLogger(tag)
        self.size = size
        self.data = np.zeros(size)
        self.lien = lien
        self.logs = logs
```

Defines:
`IO`, used in chunks 33c and 34a.

33c ⟨*Inputs* 33c⟩≡ (32c)

```
class Input(IO):
    def __init__(self,*args,**kwargs):
        IO.__init__(self,*args,**kwargs)
    ⟨input linking 28b⟩
```

Defines:
`Input`, used in chunk 35.
Uses `IO` 33b.

and `Outputs` classes.

34a ⟨*Outputs* 34a⟩≡ (32c)

```
class Output(IO):
    def __init__(self,*args,sampling_rate=1,**kwargs):
        IO.__init__(self,*args,**kwargs)
        self.sampling_rate = sampling_rate
    ⟨output linking 28c⟩
```

Defines:
`Output`, used in chunk 35.
Uses `IO` 33b.

5.2.2 Server

The `Server` class is the interface with the server where the devices are CEO objects.

34b ⟨*driver.py* 32c⟩+≡ ◁33a 37b▷

```
class Server(Driver):
    def __init__(self,tau,tag,logs,server,delay=0,sampling_rate=1,
                 verbose=logging.INFO,**kwargs):
        ⟨commom server/client driver 35⟩
        self.server        = server
        self.msg = {'class_id':'',
                    'method_id':'',
                    'args':{}}
        self.msg_args = {'Start':{},
                    'Init':{},
                    'Update':{'inputs':{}},
                    'Outputs':{'outputs':[]},
                    'Terminate':{'args':None}}

    ⟨Server methods 36⟩

    ⟨Server device parameter loading and formatting 29⟩
```

Uses `Driver` 33a and `Server` 36.

with

35 ⟨*commom server/client driver* 35⟩≡ (34b 37b)

```
Driver.__init__(self,tau,tag)
self.logger = logging.getLogger(tag)
self.logger.setLevel(verbose)
self.delay         = delay
self.sampling_rate = sampling_rate
self.inputs        = {}
if 'inputs' in kwargs:
    for k,v in kwargs['inputs'].items():
        self.logger.info('New input: %s',k)
        self.inputs[k] = Input(k,**v)
self.outputs       = {}
if 'outputs' in kwargs:
    for k,v in kwargs['outputs'].items():
        self.logger.info('New output: %s',k)
        if 'logs' in v:
            logs.add(tag,k,v['logs']['decimation'])
            v['logs'] = logs.entries[tag][k]
            self.logger.info('Output logged in!')
        self.outputs[k] = Output(k,**v)
```

Uses `Driver` 33a, `Input` 33c, and `Output` 34a.

The inherited `Server` method are:

36 ⟨*Server methods* 36⟩≡ (34b)

```
def start(self):
    self.logger.debug('Starting!')
    m = 'Start'
    ⟨client-server exchange 37a⟩
    self.msg['class_id'] = reply
    self.logger.info('%s',reply)
def init(self):
    self.logger.debug('Initializing!')
    m = 'Init'
    ⟨client-server exchange 37a⟩
    self.logger.info('%s',reply)
def update(self,step):
    if step>=self.delay and step%self.sampling_rate==0:
        self.logger.debug('Updating!')
        m = 'Update'
        ⟨client-server exchange 37a⟩
def output(self,step):
    if step>=self.delay:
            m = 'Outputs'
            if self.msg_args[m]['outputs']:
                ⟨client-server exchange 37a⟩
                self.logger.debug("Reply: %s",reply)
                for k,v in self.outputs.items():
                    if step%v.sampling_rate==0:
                        self.logger.debug('Outputing %s!',k)
                        try:
                            v.data[...] = np.asarray(reply[k]).reshape(v.size)
                        except ValueError:
                            self.logger.warning('Resizing %s!',k)
                            __red = np.asarray(reply[k])
                            v.size = __red.size
                            v.data = np.zeros(__red.shape)
                            v.data[...] = __red
                        if v.logs is not None and step%v.logs.decimation==0:
                            self.logger.debug('LOGGING')
                            v.logs.add(v.data.copy())

def terminate(self):
    self.logger.debug('Terminating!')
    m = 'Terminate'
    ⟨client-server exchange 37a⟩
    self.logger.info(reply)
```

Each method communicates with the server using the same protocol

37a ⟨*client-server exchange* 37a⟩≡ (36)

```
self.msg['method_id'] = m
self.msg['args'].update(self.msg_args[m])
self.server._send_(self.msg)
self.msg['method_id'] = ''
self.msg['args'].clear()
reply = self.server._recv_()
```

5.2.3 Client

The `Client` class is the interface with the client devices such as temporal controllers.

37b ⟨*driver.py* 32c⟩+≡ ◁34b 39▷

```
class Client(Driver):
    def __init__(self,tau,tag,logs,delay=0,sampling_rate=1,
                 verbose=logging.INFO,**kwargs):
        ⟨commom server/client driver 35⟩
        self.system = None
        self.__xout = np.zeros(0)
        self.__yout = np.zeros(0)

    ⟨Client methods 38⟩

    ⟨Client device parameter loading and formatting 30b⟩
```

Defines:
`Client`, used in chunk 27a.
Uses `Driver` 33a.

38 ⟨*Client methods* 38⟩≡ (37b)

```
def start(self):
    self.logger.debug('Starting!')
    pass
def init(self):
    self.logger.debug('Initializing!')
    self.system = self.system._as_ss()
    self.__xout = np.zeros((1,self.system.A.shape[0]))
    self.__yout = np.zeros((1, self.system.C.shape[0]))
def update(self,step):
    if step>=self.delay and step%self.sampling_rate==0:
        self.logger.debug('Updating!')
        u = np.hstack([_.data.reshape(1,-1) for _  in self.inputs.values()])
        self.logger.debug('u: %s',u)
        self.__yout = np.dot(self.system.C, self.__xout) + np.dot(self.system.D, u)
        self.__xout = np.dot(self.system.A, self.__xout) + np.dot(self.system.B, u)

def output(self,step):
    if step>=self.delay:
        for k,v in self.outputs.items():
            if step%v.sampling_rate==0:
                self.logger.debug('Outputing %s!',k)
                a = 0
                for k in self.outputs:
                    b = a + self.outputs[k].data.size
                    self.outputs[k].data[...] = \
                                self.__yout[0,a:b].reshape(self.outputs[k].size)
                    a = b

def terminate(self):
    self.logger.debug('Terminating!')
```

5.2.4 Atmosphere

A special driver is the atmosphere driver that is used to instantiate an atmosphere object on CEO server

39 ⟨*driver.py* 32c⟩+≡ ◁37b

```
class Atmosphere(Driver):
    def __init__(self,tau,tag,server,verbose=logging.INFO,**kwargs):
        Driver.__init__(self,tau,tag)
        self.logger = logging.getLogger(tag)
        self.logger.setLevel(verbose)
        self.server = server
        self.inputs = {}
        self.outputs= {}
        self.msg    = {'class_id':'ATM',
                       'method_id':'Start',
                       'args':{}}
    def start(self):
        self.server._send_(self.msg)
        reply = self.server._recv_()
        self.logger.info('%s',reply)

    def terminate(self):
        self.server._send_({'class_id':'ATM',
                            'method_id':'Terminate',
                            'args':{'args':None}})
        reply = self.server._recv_()
        self.logger.info('%s',reply)

    def associate(self,prm_file):
        with open(prm_file) as f:
            prm = yaml.load(f)
        self.msg['args'].update(prm)
```

Defines:
`Atmosphere`, used in chunks 12 and 27a.
Uses `Driver` 33a.

5.3 Logs

Driver output data can be logged in using the `Logs` class:

40 ⟨*dos.py* 26⟩+≡ ◁26 41b▷

```
class Entry:
    def __init__(self,tau,decimation):
        self.tau = tau
        self.decimation = decimation
        self.data = []
    def add(self,value):
        self.data += [value]
    @property
    def timeSeries(self):
        time   = np.arange(len(self.data))*self.decimation*self.tau
        values = np.vstack(self.data) if self.data[0].ndim<2 else np.dstack(self.data)
        return time,values
class Logs:
    def __init__(self,sampling_time):
        self.sampling_time = sampling_time
        self.entries = {}
    def add(self,driver,output,decimation):
        if driver in self.entries:
            self.entries[driver][output] = Entry(self.sampling_time,decimation)
        else:
            self.entries[driver] = {output:Entry(self.sampling_time,decimation)}
    def __repr__(self):
        if self.entries:
            line = ["The 'logs' has {} entries:".format(self.N_entries)]
            for d in self.entries:
                line += [" * {}".format(d)]
                for k,e in enumerate(self.entries[d]):
                    v = self.entries[d][e]
                    if v.data:
                        line += ["   {0}. {1}: {2}x{3}".format(k+1,e,v.data[0].shape,len(v
                    else:
                        line += ["   {0}. {1}".format(k+1,e)]
        else:
            line = ["The 'logs' has no entries!"]
        return "\n".join(line)
    @property
    def N_entries(self):
        return sum([len(_) for _ in self.entries.values()])
```

5.4 The broker

41a ⟨*dos imports* 25b⟩+≡ (26) ◁25b 41c▷

```
import zmq
import pickle
import zlib
```

41b ⟨*dos.py* 26⟩+≡ ◁40

```
class broker:

    def __init__(self,IP):
        self.logger = logging.getLogger(self.__class__.__name__)
        self.context = zmq.Context()
        self.logger.info("Connecting to server...")
        self.socket = self.context.socket(zmq.REQ)
        self.socket.connect("tcp://{}:3650".format(IP))
        self._send_("Acknowledging connection from SIMCEO client!")
        print(self._recv_())

    def __del__(self):
        self.logger.info('Disconnecting from server!')
        self.socket.close()
        self.context.term()

    def _send_(self,obj,protocol=-1,flags=0):
        pobj = pickle.dumps(obj,protocol)
        zobj = zlib.compress(pobj)
        self.socket.send(zobj, flags=flags)

    def _recv_(self,flags=0):
        zobj = self.socket.recv(flags)
        pobj = zlib.decompress(zobj)
        return pickle.loads(pobj)
```

5.5 Timing diagram

A timing diagram can be generated with the `diagram` method. It is produced with the `graphviz` module.

41c ⟨*dos imports* 25b⟩+≡ (26) ◁41a

```
from graphviz import Digraph
```

42 ⟨*timing diagram* 42⟩≡ (26)

```
def diagram(self):
    def add_item(sample_rate,driver_name,method):
        if not sample_rate in sampling:
            sampling[sample_rate] = {}
        if not driver_name in sampling[sample_rate]:
            sampling[sample_rate][driver_name] = [method]
        else:
            sampling[sample_rate][driver_name] += [method]
    def make_nodes(_s_):
        ss = str(_s_)
        c = Digraph(ss)
        c.attr(rank='same')
        c.node(ss,time_label(_s_))
        [c.node(ss+'_'+_,make_label(_,sampling[_s_][_])) for _ in sampling[_s_]]
        return c
    def make_label(d,dv):
        label = "<TR><TD><B>{}</B></TD></TR>".format(d)
        for v in dv:
            label += '''<TR><TD PORT="{0}_{1}">{1}</TD></TR>'''.format(d,v)
        return '''<<TABLE BORDER="0" CELLBORDER="1">{}</TABLE>>'''.format(label)
    def search_method(d,m):
        for s in sampling:
            if d in sampling[s]:
                if m in sampling[s][d]:
                    return '{0}_{1}:{1}_{2}'.format(str(s),d,m)
    def time_label(n):
        nu = self.cfg['simulation']['sampling frequency']
        t = n/nu
        if t<1:
            return '{:.1f}ms'.format(t*1e3)
        else:
            return '{:.1f}s'.format(t)

    main = Digraph(format='png', node_attr={'shape': 'plaintext'})

    sampling = {}
    for dk in self.drivers:
        d = self.drivers[dk]
        if d.delay>0:
            add_item(d.delay,dk,'delay')
        add_item(d.sampling_rate,dk,'update')
        for ok in d.outputs:
            o = d.outputs[ok]
            add_item(o.sampling_rate,dk,'output')
```

```
    s = sorted(sampling)
    [main.subgraph(make_nodes(_)) for _ in s]

    for k in range(1,len(s)):
        main.edge(str(s[k-1]),str(s[k]))

    for s in sampling:
        for d in sampling[s]:
            m = sampling[s][d]
            if not (len(m)==1 and m[0]=='delay'):
                for ik in self.drivers[d].inputs:
                    data = self.drivers[d].inputs[ik]
                    if data.lien is not None:
                        main.edge(search_method(data.lien[0],'output'),
                                  '{0}_{1}:{1}_update'.format(str(s),d))
                for ok in self.drivers[d].outputs:
                    data = self.drivers[d].outputs[ok]
                    if data.lien is not None:
                        main.edge('{0}_{1}:{1}_output'.format(str(s),d),
                                  search_method(data.lien[0],'update'))

    return sampling,main
```

5.6 Main

43 ⟨*sim.py* 43⟩≡

```
import sys
import dos

if __name__=="__main__":

    dospath = sys.argv[1]
    sim = dos.DOS(dospath)
    sim._run_()
```

6 Index

7 List of code chunks

⟨*output linking* 28c⟩
⟨*Outputs* 34a⟩
⟨*running the loop* 30g⟩
⟨*running the simulation* 31b⟩
⟨*S-function* 9⟩
⟨*SAtmosphere* 12⟩
⟨*Sensor abstract class* 24a⟩
⟨*Server device parameter loading and formatting* 29⟩
⟨*Server methods* 36⟩
⟨*SGMT* 10a⟩
⟨*SGMT Start message* 10b⟩
⟨*sim.py* 43⟩
⟨*simceo.py* 3⟩
⟨*SOpticalPath* 13a⟩
⟨*SOpticalPath InitializeConditions message* 18⟩
⟨*SOpticalPath Outputs message* 16a⟩
⟨*SOpticalPath Start message* 13b⟩
⟨*SOpticalPath Terminate message* 15a⟩
⟨*SOpticalPath Update message* 11a⟩
⟨*starting the drivers* 30c⟩
⟨*starting the simulation* 30d⟩
⟨*stepping through* 31a⟩
⟨*terminating the drivers* 31c⟩
⟨*terminating the simulation* 31d⟩
⟨*timing diagram* 42⟩