SIMCEO

Simulink Client
CEO Server

R. Conan
GMTO Corporation

August 3, 2019

[git] • Branch: python-client @ 0ef36f7 • Release: 1.1 (2019-08-03)

Contents

1 Introduction

This documents describes SIMCEO, an interface between CEO and Simulink. SIMCEO allows to seamlessly integrates CEO functionalities into a Simulink model. A Simulink library, *CEO*, provides a set of blocks that are used to instantiate CEO objects. The blocks either send data to the CEO objects updating the state of these objects, or query data from the CEO objects. The data received from the CEO objects is then forwarded to the other blocks of the Simulink model.

2 Installation

This section describes the installation of the SIMCEO client i.e. the Matlab and Simulink part of SIMCEO.

To install SIMCEO on your computer, creates a directory `SIMCEO`, downloads the archive `simceo.zip` and extracts it in the `SIMCEO` directory.

In addition to Matlab and Simulink, the client relies of aws cli, ZeroMQ and UBJSON.

2.1 AWS command line interface

The AWS command line interface (`aws cli`) allows to launch/terminate and to start/stop the AWS instances where the SIMCEO server resides. To install it, follows the instructions at
http://docs.aws.amazon.com/cli/latest/userguide/installing.html
Once installed, open a terminal and at the shell prompt enter:
`>> aws configure --profile gmto.control`
and answers the questions using the `gmto.control.credentials` file provided separately.

At Matlab prompt enter: `>> system('aws --version')`. If Matlab cannot find `aws`, replace `aws` in `etc/simceo.json` by the full path to `aws`.

2.2 Matlab–ZMQ

Matlab–ZMQ[1] is a Matlab wrapper for ZeroMQ. ZeroMQ [2] is the messaging library used for the communications between SIMCEO client and server. Both Matlab–ZMQ and ZeroMQ are shipped pre–compiled with SIMCEO. You need however to add, to the Matlab search path, the path to ZeroMQ. To do so, move Matlab current folder to SIMCEO folder and at the Matlab prompt enter:

```
>> addpath([pwd,'/matlab-zmq/your-os/lib/'])
>> savepath
```

where `your-os` is either `unix`, `mac` `windows7` or `windows10`.

[1] https://github.com/fagg/matlab-zmq
[2] http://zeromq.org/

2.3 UBJSON

Universal Binary JSON (UBJSON[3]) is the message format used to exchange data between SIMCEO client and server. The Matlab UBJSON encoder and decoder is JSONLAB. SIMCEO comes with its own version of JSONLAB that fixes a few bugs. To add JSONLAB to the Matlab search path, move Matlab current folder to SIMCEO folder and at the Matlab prompt enter:

```
>> addpath([pwd,'/jsonlab/'])
>> savepath
```

3 Implementation

The interface between CEO and Simulink has two components a Matlab package *ceo* on the user computer, the client, and a python module *simulink* on a CEO AWS instance, the server. A flowchart of SIMCEO is shown in Fig. 3. The Matlab package is written with custom blocks using a *Level–2 Matlab S–function*. A *Level–2 Matlab S–function* consists in a collection of functions that are called by the Simulink engine when a model is running. Inside the *Level–2 Matlab S–function*, the functions *Start*, *Terminate* and *Outputs* are used to exchange information with CEO. The Matlab class *broker* Acknowledging connection to SIMCEO serveris responsible for starting the CEO server in the AWS cloud and for managing the communication with the server.

The requests from the client are managed by the *broker* class of the *simulink* python module on the server. The *simulink* module is providing three python classes to deal with Simulink requests: *SGMT*, *SAtmosphere* and *SOpticalPath*.

The communication between the client and the server uses the Request/Reply messaging pattern of ZeroMQ. The messages exchanged between the client and the server are formatted according to the UBJSON format.

4 The simulink python module

The python interface consists in the module *simulink*:

5 ⟨*simceo.py* 5⟩≡

```
import sys
import threading
import time
import zmq
import ubjson
import ceo
import numpy as np
from collections import OrderedDict
import os
```

[3] http://ubjson.org/

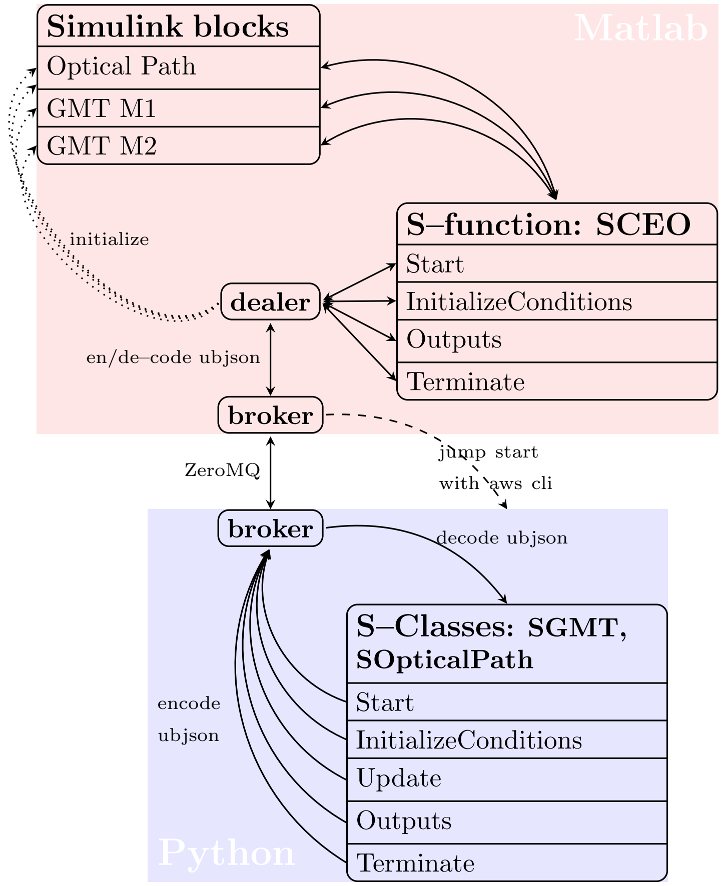

Figure 1: SIMCEO flowchart.

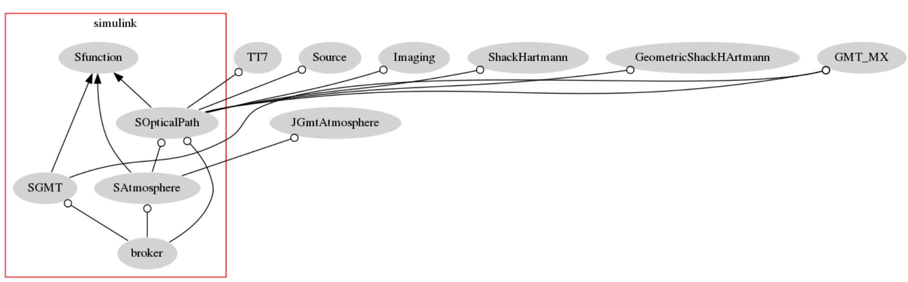

Figure 2: The classes in the simulink python module and their relations with the CEO classes.

```
import shelve
import traceback
import scipy.linalg as LA
import pickle
import zlib
import logging

logging.basicConfig(level=logging.DEBUG)

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 27⟩

⟨S-function 13a⟩

⟨SGMT 13b⟩
⟨SAtmosphere 16a⟩
⟨SOpticalPath 16b⟩

⟨broker 9⟩

if __name__ == "__main__":

    print("*********************************")
    print("**   STARTING SIMCEO SERVER    **")
    print("*********************************")
    agent = broker()
```

```
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.

9 ⟨*broker* 9⟩≡ (5)

```
class broker(threading.Thread):

    def __init__(self):

        threading.Thread.__init__(self)

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

        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)
        self.sgmt = SGMT(self.ops, self.satm)

    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 12a⟩
```

⟨*broker run* 10a⟩

The *run* method

10a ⟨*broker run* 10a⟩≡ (9)

```
def run(self):

    while self.loop:

        ⟨broker run details 10b⟩
```

waits for a request from a Simulink S–function:

10b ⟨*broker run details* 10b⟩≡ (10a) 11 ▷

```
#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:

11 ⟨*broker run details* 10b⟩+≡ (10a) ◁10b 12b▷

```
#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 = b"The server has failed!"
```

The dictionary–like call is implemented with

12a ⟨*broker get item* 12a⟩≡ (9)

```
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 ) )
            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*.

12b ⟨*broker run details* 10b⟩+≡ (10a) ◁11 12c▷

```
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:

12c ⟨*broker run details* 10b⟩+≡ (10a) ◁12b

```
#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:

13a ⟨*S-function* 13a⟩≡ (5)

```
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.

13b ⟨*SGMT* 13b⟩≡ (5) 14b ▷

```
class SGMT(Sfunction):

    def __init__(self, ops, satm):
        self.logger = logging.getLogger(self.__class__.__name__)
        self.gmt  = ceo.GMT_MX()

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

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

14a ⟨*SGMT Start message* 14a⟩≡
```
{
"class_id": "GMT",
"method_id": "Start",
"args":
  {
    "mirror": "M1"|"M2",
    "mirror_args":
      {
        "mirror_modes": u"bending modes"|u"zernike",
        "N_MODE": 162,
        "radial_order": ...
      }
  }
}
```

14b ⟨*SGMT* 13b⟩+≡ (5) ◁13b 15a▷
```
    def Start(self,mirror=None,mirror_args=None):
        self.logger.info('Start')
        self.gmt[mirror] = getattr(ceo,"GMT_"+mirror)( **mirror_args )
        return b"GMT"
```

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

14c ⟨*SOpticalPath Update message* 14c⟩≡ 19c▷
```
{
"class_id": "GMT",
"method_id": "Update",
"args":
  {
    "mirror": "M1"|"M2",
    "inputs":
      {
        "TxyzRxyz": null,
        "mode_coefs": null
      }
  }
}
```

15a ⟨*SGMT* 13b⟩+≡ (5) ◁14b 15b▷

```
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=="mode_coefs":
                self.gmt[mirror].modes.a[:] = data
                self.gmt[mirror].modes.update()
```

InitializeConditions

15b ⟨*SGMT* 13b⟩+≡ (5) ◁15a 15c▷

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

Outputs

15c ⟨*SGMT* 13b⟩+≡ (5) ◁15b

```
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.

16a ⟨*SAtmosphere* 16a⟩≡ (5)

```
class SAtmosphere(Sfunction):

    def __init__(self, ops):
        self.atm = None

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

    def Terminate(self, args=None):
        print("\n@(SAtmosphere:Terminate)>")
        self.atm = None
        return b"Atmosphere deleted!"

    def InitializeConditions(self, args=None):
        pass

    def Outputs(self, args=None):
        pass

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

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*.

16b ⟨*SOpticalPath* 16b⟩≡ (5) 18▷

```
class SOpticalPath(Sfunction):

    def __init__(self, idx, gmt, satm):
        self.idx = idx
        self.gmt = gmt
        self.satm = satm
        self.sensor = None
```

Defines:
`idx`, used in chunk 18.
`sensor`, used in chunks 18–21, 25, 31b, and 64.

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

17 ⟨*SOpticalPath Start message* 17⟩≡

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

18 ⟨*SOpticalPath* 16b⟩+≡ (5) ◁16b 19b▷

```
def Start(self,source_args=None, sensor_class=None, sensor_args=None,
          calibration_source_args=None, miscellaneous_args=None):
    print("\n@(SOpticalPath:Start)>")
    self.pssn_data = None
    #self.propagateThroughAtm = miscellaneous_args['propagate_through_atmosphere']
    self.src = ceo.Source( **source_args )
    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.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()
        self.sensor.calibrate(self.calib_src, sensor_args['intensityThreshold'])
        #print "intensity_threshold: %f"%sensor_args['intensityThreshold']

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

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

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

Defines:
`exposure_start`, never used.
`exposure_time`, never used.
`propagateThroughAtm`, never used.
`src`, used in chunks 19d, 21, and 25.

Uses `idx` 16b and `sensor` 16b.

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

19a ⟨*SOpticalPath Terminate message* 19a⟩≡


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


19b ⟨*SOpticalPath* 16b⟩+≡ (5) ◁18 19d▷

```
      def Terminate(self, args=None):
          print("\n@(SOpticalPath:Terminate)>")
          return b"OpticalPath deleted!"
```

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

19c ⟨*SOpticalPath Update message* 14c⟩+≡ ◁14c


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


19d ⟨*SOpticalPath* 16b⟩+≡ (5) ◁19b 20▷

```
      def Update(self, inputs=None):
          +self.src
          #self.src.reset()
          #self.gmt.propagate(self.src)
          #self.sensor.propagate(self.src)
```

Uses `sensor` 16b and `src` 18.

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

19e ⟨*SOpticalPath Outputs message* 19e⟩≡


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

20 ⟨*SOpticalPath* 16b⟩+≡ (5) ◁19d 21▷

```
    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` 16b.

and the dictionnary implementation is

21 ⟨*SOpticalPath* 16b⟩+≡ (5) ◁20 25▷

```
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
    else:
        c = self.comm_matrix[key].dot( self.sensor.Data ).reshape(1,-1)
        return c.tolist()
```

Uses `sensor` 16b and `src` 18.

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

22 ⟨*SOpticalPath InitializeConditions message* 22⟩≡ 23▷

```
{
"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
  }
}
```

23 ⟨*SOpticalPath InitializeConditions message* 22⟩+≡ ◁22 24▷

```
{
"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
  }
}
```

24 ⟨*SOpticalPath InitializeConditions message* 22⟩+≡ ◁23

```
{
"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
  }
}
```

25 ⟨*SOpticalPath* 16b⟩+≡ (5) ◁21

```
def InitializeConditions(self, calibrations=None, filename=None,
                         pseudo_inverse=None):
    print("@(SOpticalPath:InitializeConditions)>")
    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"):
                print(" . Loading command matrix from existing database %s!"%filename)
                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():
            if len(calibrations)>1:
                for key in calibrations: # Through calibrations
                    calibs = calibrations[key]
                    if not isinstance(calibs,list):
                        calibs = [calibs]
                    D = []
                    for c in calibs: # Through calib
                        self.gmt.reset()
                        self.src.reset()
                        self.sensor.reset()
                        D.append( getattr( self.gmt, c["method_id"] )( self.sensor,
                                                                       self.src,
                                                                       **c["args"] ) )
                    self.gmt.reset()
                    self.src.reset()
                    self.sensor.reset()
                    C = ceo.CalibrationVault(D, **pseudo_inverse )
                    self.comm_matrix[key] = C
            else:
                for key in calibrations: # Through calibrations
                    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()
                    C = getattr( self.gmt, calibs["method_id"] )( self.sensor,
                                                                  self.src,
                                                                  calibrationVaultKwargs=ps
                                                                  **calibs["args"])
                    self.gmt.reset()
                    self.src.reset()
                    self.sensor.reset()
                    self.comm_matrix[key] = C


            if filename is not None:
                print(" . Saving command matrix to database %s!"%filename)
                db[str(key)] = C
                db.close()
```
Uses `sensor` 16b and `src` 18.

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

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

27 ⟨*CalibrationMatrix* 27⟩≡ (5)

```
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

29a ⟨*Sensor abstract class* 29a⟩≡

```
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.

29b ⟨*dos.yaml* 29b⟩≡
 ⟨*dos simulation section* 29c⟩
 ⟨*dos drivers section* 34⟩

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

29c ⟨*dos simulation section* 29c⟩≡ (29b)

```
simulation:
  sampling frequency: 100 # [Hertz]
  duration: 1 # [seconds]
  simceo 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.

30a ⟨*dos imports* 30a⟩≡ (30b) 37b ▷

```
import os
import yaml
import logging
import threading
import numpy as np
```

30b ⟨*dos.py* 30b⟩≡ 37a ▷

```
⟨dos imports 30a⟩

logging.basicConfig(level=logging.DEBUG)

class DOS:
    def __init__(self,path_to_config_dir):
        self.logger = logging.getLogger(self.__class__.__name__)

        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']['simceo server']['IP'])

        self.N_SAMPLE = int(self.cfg['simulation']['sampling frequency']*
                            self.cfg['simulation']['duration'])
        ⟨check parameter file existence 31a⟩
        ⟨linking the drivers IO 32a⟩
        ⟨device to driver association 32e⟩
        ⟨starting the drivers 33b⟩
        ⟨initializing the drivers 33c⟩
        ⟨running the loop 33d⟩
        ⟨terminating 33f⟩

    ⟨stepping through 33e⟩

    ⟨timing diagram 39⟩
```

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

31a ⟨*check parameter file existence* 31a⟩≡ (30b)

```
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)
        self.drivers[d] = Driver(d,self.agent,**v)
    else:
        self.logger.warning('%s is missing!',prm_file)
```

For an optical sensor, the `device name.yaml` file has 3 sections: source, sensor and calibrations. Each section list the arguments of CEO methods.

31b ⟨*device name.yaml* 31b⟩≡

```
source:
  photometric_band: R+I
  zenith:
    value: 8
    units: arcmin
  azimuth:
    value: 66
    units: degree
  magnitude: 0
  rays_box_size: 25.5000
  rays_box_sampling: 769
  rays_origin: [0,0,25]
sensor:
  class: GeometricShackHartmann
  args:
    N_SIDE_LENSLET: 20
  calibrate args: null
calibrations:
  M2TT:
    method_id: calibrate
    args:
      mirror: M2
      mode: segment tip-tilt
      stroke: 1e-6
```

Uses `sensor` 16b.

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

32a ⟨*linking the drivers IO* 32a⟩≡ (30b)
```
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)
```

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

32b ⟨*IO linking* 32b⟩≡ (32)
```
def tie(self,drivers):
    if self.lien is not None:
        d,io = self.lien
        self.logger.info('Linked to %s from %s',io,d)
```

32c ⟨*input linking* 32c⟩≡ (36b)
```
⟨IO linking 32b⟩
        self.data = drivers[d].outputs[io].data
```

32d ⟨*output linking* 32d⟩≡ (36c)
```
⟨IO linking 32b⟩
        self.data = drivers[d].inputs[io].data
```

The device parameters are loaded from the device parameter file and formatted into a message sent to CEO server.

32e ⟨*device to driver association* 32e⟩≡ (30b)
```
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)
```

33a ⟨*device parameter loading and formatting* 33a⟩≡ (37a)

```
def associate(self,prm_file):
    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.
    else:
        self.msg['class_id'] = 'OP'
        self.msg_args['Start'] = {'source_args':{},
                                  'sensor_class':None,
                                  'sensor_args':None,
                                  'calibration_source':None,
                                  'miscellaneous_args':None}
        self.msg_args['Outputs']['outputs'] += [k_o for k_o in self.outputs]
```

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

33b ⟨*starting the drivers* 33b⟩≡ (30b)

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

and `init` methods:

33c ⟨*initializing the drivers* 33c⟩≡ (30b)

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

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

33d ⟨*running the loop* 33d⟩≡ (30b)

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

33e ⟨*stepping through* 33e⟩≡ (30b)

```
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]
```

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

33f ⟨*terminating* 33f⟩≡ (30b)

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

5.1 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.

34 ⟨*dos drivers section* 34⟩≡ (29b)

```
drivers:
  device name:
    delay: 7 # [sample]
    sampling rate: 5 # [sample]
    inputs:
      input name:
        size: 0
        origin: [device, device output name]
    outputs:
      output name:
        sampling rate: 10 # [sample]
        size: 0
        destination: [device, device input name]
```

The `drivers` method are defined in the `Driver` class.

35a ⟨*Driver methods* 35a⟩≡ (37a)

```
def start(self):
    self.logger.debug('Starting!')
    m = 'Start'
    ⟨client-server exchange 35b⟩
def init(self):
    self.logger.debug('Initializing!')
    m = 'Init'
    ⟨client-server exchange 35b⟩
def update(self,step):
    if step>=self.delay and step%self.sampling_rate==0:
        self.logger.debug('Updating!')
        m = 'Update'
        ⟨client-server exchange 35b⟩
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)
                m = 'Outputs'
                ⟨client-server exchange 35b⟩
def terminate(self):
    self.logger.debug('Terminating!')
    m = 'Terminate'
    ⟨client-server exchange 35b⟩
```

Each method communicates with the server using the same protocol

35b ⟨*client-server exchange* 35b⟩≡ (35a)

```
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()
return self.server._recv_()
```

5.1.1 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.

36a ⟨*IO* 36a⟩≡ (37a)

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

36b ⟨*Inputs* 36b⟩≡ (37a)

```
class Input(IO):
    def __init__(self,tag,size=0,origin=None):
        IO.__init__(self,tag,size=size,lien=origin)
    ⟨input linking 32c⟩
```

and `Outputs` classes.

36c ⟨*Outputs* 36c⟩≡ (37a)

```
class Output(IO):
    def __init__(self,tag,size=0,sampling_rate=1,destination=None):
        IO.__init__(self,tag,size=size,lien=destination)
        self.sampling_rate = sampling_rate
    ⟨output linking 32d⟩
```

37a ⟨*dos.py* 30b⟩+≡ ◁30b 38a▷

```
⟨IO 36a⟩
⟨Inputs 36b⟩
⟨Outputs 36c⟩
class Driver:
    def __init__(self,tag,server,delay=0,sampling_rate=1,**kwargs):
        self.logger = logging.getLogger(tag)
        self.tag           = tag
        self.server        = server
        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)
                self.outputs[k] = Output(k,**v)
        self.msg = {'class_id':'',
                    'method_id':'',
                    'args':{}}
        self.msg_args = {'Start':{},
                    'Init':{},
                    'Update':{'inputs':{}},
                    'Outputs':{'outputs':[]},
                    'Terminate':{'args':None}}

    ⟨Driver methods 35a⟩

    ⟨device parameter loading and formatting 33a⟩
```

5.2 The broker

37b ⟨*dos imports* 30a⟩+≡ (30b) ◁30a 38b▷

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

38a ⟨*dos.py* 30b⟩+≡ ◁37a

```
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)
```

5.3 Timing diagram

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

38b ⟨*dos imports* 30a⟩+≡ (30b) ◁37b

```
from graphviz import Digraph
```

39 ⟨*timing diagram* 39⟩≡ (30b)

```
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.origin is not None:
                        main.edge(search_method(data.origin[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.destination is not None:
                        main.edge('{0}_{1}:{1}_output'.format(str(s),d),
                                  search_method(data.destination[0],'update'))

    return sampling,main
```

6 The python client

The simulation

40 ⟨*simceoclient.py* 40⟩≡ 41a ▷

```
import zmq
import yaml
import os
import pickle
import zlib

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

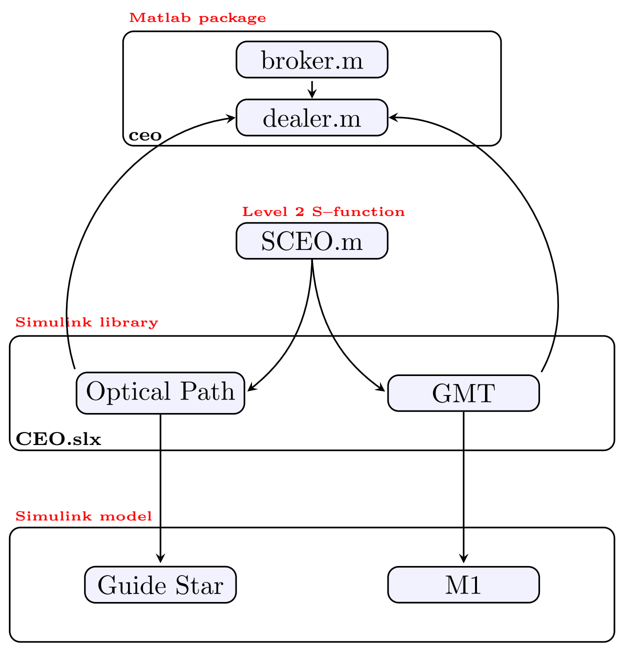

Figure 3: SIMCEO Matlab client flowchart.

41a ⟨*simceoclient.py* 40⟩+≡ ◁40 41b▷

```
class SIM:

    def __init__(self):
        with open(os.path.join(SIMCEOPATH,'etc','sim_prm.yaml')) as f:
            cfg = yaml.load(f)

        self.tau = 1/cfg['simulation']['sampling frequency']
        self.T = cfg['simulation']['duration']

        self.simceo = broker(cfg['simceo server']['IP'])
```

41b ⟨*simceoclient.py* 40⟩+≡ ◁41a

```
if __name__ == "__main__":
    #agent = broker()
    sim = SIM()
```

7 The ceo Matlab package

7.1 The broker class

42 ⟨*broker.m* 42⟩≡

```
classdef (Sealed=true) broker < handle
    % broker An interface to a CEO server
    %  The broker class launches an AWS instance and sets up the connection
    %  to the CEO server

    properties
      ami_id % The AWS AMI ID number
      instance_id % The AWS instance ID number
      public_ip % The AWS instance public IP
      zmqReset % ZMQ connection reset flag
            elapsedTime
    end

    properties (Access=private)
        etc
        instance_end_state
        ctx
        socket
        urlbase
    end

    methods

        ⟨broker client 43⟩

        ⟨release ressources 49a⟩

        ⟨launch AWS AMI 44b⟩

        ⟨start AWS instance 47a⟩

    end

    methods(Static)

        ⟨instanciation and retrieval 49b⟩

        ⟨request and reply 50a⟩
  g
        ⟨reset ZMQ socket 50b⟩
```

```
        ⟨time spent 51⟩

    end

end
```

Uses `etc` 43, `instance_end_state` 44a, `instance_id` 43, `public_ip` 48a, and `zmqReset` 43.

The Matlab broker class starts an AWS machine and sets–up ZeroMQ context and socket.

43 ⟨*broker client* 43⟩≡ (42)

```
function self = broker(varargin)

    self.ctx    = zmq.core.ctx_new();
    self.socket = zmq.core.socket(self.ctx, 'ZMQ_REQ');
    self.zmqReset = true;

    self.elapsedTime = 0;

    currentpath = mfilename('fullpath');
    k = strfind(currentpath,filesep);
    self.etc = fullfile(currentpath(1:k(end)),'..','etc');
    cfg = jsondecode(fileread(fullfile(self.etc,'simceo.json')));
    self.urlbase         = 'http://gmto.modeling.s3-website-us-west-2.amazonaws.com';
    self.ami_id          = cfg.aws_ami_id;
    self.instance_id     = cfg.aws_instance_id;
    self.public_ip       = cfg.public_ip;
    ⟨broker client: AWS instance launch 44a⟩
end
```

Defines:
- `awspath`, used in chunks 45 and 46.
- `etc`, used in chunks 42, 44–46, and 69.
- `instance_id`, used in chunks 42 and 44–49.
- `self.elapsedTime`, used in chunks 50a and 51.
- `zmqReset`, used in chunks 42 and 50b.

Uses `public_ip` 48a.

If no instance ID is given, a new machine is launched based on a given AWS AMI.

44a ⟨*broker client: AWS instance launch* 44a⟩≡ (43)
```
if isempty(self.public_ip)
    if isempty(self.instance_id)
      run_instance(self)
      self.instance_end_state = 'terminate';
    else
      start_instance(self)
      self.instance_end_state = 'stop';
    end
end
```
Defines:
instance_end_state, used in chunks 42, 46c, and 49a.
Uses instance_id 43, public_ip 48a, run_instance 45a, and start_instance 47a.

7.1.1 run_instance

If no instance ID is set in the `simceo.json` configuration file, a new instance is created from the AMI whose ID is given in `etc/ec2runinst.json` file.

44b ⟨*launch AWS AMI* 44b⟩≡ (42)
```
function run_instance(self)
  url = sprintf("%s/simceo_aws_server.html?action=create",self.urlbase);
  fprintf('%s\n',url)
  [status,h] = web(url,'-browser');
  if status~=0
    error('Creating machine failed:\n')
  end
  pause(20)
  url = sprintf('%s/%s.json',self.urlbase,self.ami_id);
  fprintf('%s\n',url)
  instance=jsondecode(char(webread(url))');
  self.instance_id = instance.ID;
  file = fullfile(self.etc,'simceo.json');
  cfg = jsondecode(fileread(file));
  cfg.aws_instance_id = instance.ID;
  savejson('',cfg,file);
  ⟨getting the public IP 48a⟩
end
```
Uses etc 43, instance_id 43, and run_instance 45a.

45a ⟨*launch AWS AMI (old)* 45a⟩≡

```
function run_instance(self)
   ⟨launching an instance 45b⟩
   ⟨waiting for initialization 45c⟩
   ⟨branding instance 46a⟩
   ⟨setting up cloudwatch 46b⟩
   ⟨getting the public IP 48a⟩
end
```

Defines:
`run_instance`, used in chunk 44.

The sequence of operations is:

1. launching the instance,

45b ⟨*launching an instance* 45b⟩≡ (45a)

```
cmd = sprintf(['%s ec2 run-instances --profile gmto.control ',...
               '--cli-input-json file://%s'],...
              self.awspath, fullfile(self.etc,'ec2runinst.json'));
[status,instance_json] = system(cmd);
if status~=0
    error('Launching AWS AMI failed:\n%s',instance_json)
end
instance = loadjson(instance_json);
self.instance_id = instance.Instances{1}(1).InstanceId;
```

Uses `awspath` 43, `etc` 43, and `instance_id` 43.

2. waiting for the confirmation that the instance is running (See page **??**),

3. waiting for the confirmation that the instance has finished to initialize,

45c ⟨*waiting for initialization* 45c⟩≡ (45a)

```
fprintf('>>>> WAITING FOR AWS INSTANCE %s TO INITIALIZE ... \n',self.instance_id)
fprintf('(This usually takes a few minutes!)\n')
tic
cmd = sprintf(['%s ec2 wait instance-status-ok --instance-ids %s ',...
               '--profile gmto.control'],...
              self.awspath,self.instance_id);
[status,~] = system(cmd);
toc
if status~=0
    error('Starting AWS machine %s failed!',self.instance_id')
end
```

Uses `awspath` 43 and `instance_id` 43.

4. setting up the instance name

46a ⟨*branding instance* 46a⟩≡ (45a)

```
[~,username] = system('whoami');
[~,hostname] = system('hostname');
cmd = sprintf('%s ec2 create-tags --resources %s --tags Key=Name,Value=%s',...
              self.awspath,self.instance_id,...
              ['SIMCEO(',strtrim(username),...
               '@',strtrim(hostname),')']);
system(cmd);
```

Uses `awspath` 43 and `instance_id` 43.

5. setting up an alarm that terminates an instance idle for man than 4hours,

46b ⟨*setting up cloudwatch* 46b⟩≡ (45a)

```
cmd = sprintf(['%s cloudwatch put-metric-alarm ',...
               '--profile gmto.control ',...
               '--dimensions Name=InstanceId,Value=%s ',...
               '--cli-input-json file://%s'],...
              self.awspath,...
              self.instance_id,...
              fullfile(self.etc,'cloudwatch.json'));
[status,~] = system(cmd);
if status~=0
    error('Setting alarm for AWS machine %s failed!',self.instance_id')
end
```

Uses `awspath` 43, `etc` 43, and `instance_id` 43.

6. getting the public IP of the instance (See page 48).

7.1.2 terminate_instance

46c ⟨*terminate AWS instance* 46c⟩≡

```
function terminate_instance(self)
    if strcmp(self.instance_end_state,'terminate')
        fprintf('@(broker)> Terminating instance %s!\n',self.instance_id)
        [status,~] = system(sprintf(['%s ec2 %s-instances',...
                            ' --instance-ids %s --profile gmto.control'],...
                                    self.awspath, self.instance_end_state,...
                                    self.instance_id));
        if status~=0
            error('Terminating AWS instance %s failed!',self.instance_id')
        end
    end
end
```

Defines:
`terminate_instance`, never used.
Uses `awspath` 43, `instance_end_state` 44a, and `instance_id` 43.

7.1.3 start_instance

If an instance ID has been set in the simceo.json configuration file, this instance is started.

47a ⟨*start AWS instance* 47a⟩≡ (42)

```
function start_instance(self)
      ⟨starting an instance 47b⟩
      ⟨getting the public IP 48a⟩
end
```

Defines:
start_instance, used in chunk 44a.

The sequence of operations is:

1. starting the instance:

47b ⟨*starting an instance* 47b⟩≡ (47a)

```
fprintf('@(broker)> Starting AWS machine %s...\n',self.instance_id)

url = sprintf('%s/simceo_aws_server.html?action=start&instance_ID=%s',self.urlbase,se
fprintf('%s\n',url)
[status,h] = web(url,'-browser');
if status~=0
  error('Starting AWS machine %s failed:\n',self.instance_id)
end
pause(3)
```

Uses instance_id 43.

2. getting the public IP of the instance.

48a ⟨*getting the public IP* 48a⟩≡ (44b 45a 47a)

```
url = sprintf('%s/%s.json',self.urlbase,self.instance_id);
fprintf('%s\n',url)
instance=jsondecode(char(webread(url))');
fprintf('STATE: %s\n',instance.STATE)
n=1;
while (~strcmp(instance.STATE,'running')) && (n<=3)
  fprintf('Probing instance state (20s wait time) ...\n')
  pause(20)
  instance=jsondecode(char(webread(url))');
  n = n + 1;
end
if (~strcmp(instance.STATE,'running')) && (n>3)
  error('Failed to start server!')
end
self.public_ip = instance.IP;
fprintf('\n ==>> machine is up and running @%s\n',self.public_ip)
%pause(2)
%close(h)
```

Defines:
`public_ip`, used in chunks 42–44 and 48b.
Uses `instance_id` 43.

Once the instance is running, ZeroMQ connects the client to the server port of ZeroMQ on the AWS instance:

48b ⟨*broker client: setup ZMQ connection* 48b⟩≡ (50b)

```
self.socket = zmq.core.socket(self.ctx, 'ZMQ_REQ');
status = zmq.core.setsockopt(self.socket,'ZMQ_RCVTIMEO',60e3);
if status<0
    error('broker:zmqRcvTimeOut','Setting ZMQ_RCVTIMEO failed!')
end
status = zmq.core.setsockopt(self.socket,'ZMQ_SNDTIMEO',60e3);
if status<0
    error('broker:zmqSndTimeOut','Setting ZMQ_SNDTIMEO failed!')
end
address     = sprintf('tcp://%s:3650',self.public_ip);
zmq.core.connect(self.socket, address);
fprintf('@(broker)> %s connected at %s\n',class(self),address)
```

Uses `public_ip` 48a.

The allocated ZeroMQ ressources are released with:

49a ⟨*release ressources* 49a⟩≡ (42)
```
  function delete(self)
      fprintf('@(broker)> Deleting %s\n',class(self))
      zmq.core.close(self.socket);
      zmq.core.ctx_shutdown(self.ctx);
      zmq.core.ctx_term(self.ctx);
      if ~isempty(self.instance_end_state)
        url = sprintf('%s/simceo_aws_server.html?action=%s&instance_ID=%s',...
                         self.urlbase,self.instance_end_state,self.instance_id);
        fprintf('%s\n',url)
        [status,h] = web(url,'-browser');
        if status~=0
           error('Shutting down AWS machine %s failed:\n',self.instance_id)
        end
      end
  end
```
Uses `instance_end_state` 44a and `instance_id` 43.

Two static methods are defined. *getBroker* instanciates and retrieves the broker object. There can be only one broker object per Matlab session.

49b ⟨*instanciation and retrieval* 49b⟩≡ (42)
```
  function self = getBroker(varargin)
  % getBroker Get a pointer to the broker object
  %
  % agent = ceo.broker.getBroker() % Launch an AWS instance and returns
  % a pointer to the broker object
  % agent = ceo.broker.getBroker('awspath','path_to_aws_cli') % Launch
  % an AWS instance using the given AWS CLI path and returns a pointer to
  % the broker object
  % agent =
  % ceo.broker.getBroker('instance_id','the_id_of_AWS_instance_to_start')
  % Launch the AWS instance 'instance_id' and returns a pointer to the broker object

      persistent this
      if isempty(this) || ~isvalid(this)
          fprintf('~~~~~~~~~~~~~~~~~~~')
          fprintf('\n SIMCEO CLIENT!\n')
          fprintf('~~~~~~~~~~~~~~~~~~~\n')
          this = ceo.broker(varargin{:});
      end
      self = this;
  end
```
Defines:
`getBroker`, used in chunks 50 and 51.

sendrecv sends a request to the server and returns the server reply:

50a ⟨*request and reply* 50a⟩≡ (42)

```
function jmsg = sendrecv(send_msg)
    tid = tic;
    self = ceo.broker.getBroker();
    jsend_msg = saveubjson('',send_msg);
    zmq.core.send( self.socket, uint8(jsend_msg) );
    rcev_msg = -1;
    count = 0;
    while all(rcev_msg<0) && (count<15)
        rcev_msg = zmq.core.recv( self.socket , 2^24);
        if count>0
            fprintf('@(broker)> sendrecv: Server busy (call #%d)!\n',15-count)
        end
        count = count + 1;
    end
    if count==15
        set_param(gcs,'SimulationCommand','stop')
    end
    jmsg = loadubjson(char(rcev_msg),'SimplifyCell',1);
    if ~isstruct(jmsg) && strcmp(char(jmsg),'The server has failed!')
        disp('Server issue!')
        set_param(gcs,'SimulationCommand','stop')
    end
    self.elapsedTime = self.elapsedTime + toc(tid);
end
```

Defines:
`sendrecv`, used in chunks 57–59.
Uses `getBroker` 49b and `self.elapsedTime` 43.

resetZMQ resets the ZeroMQ socket

50b ⟨*reset ZMQ socket* 50b⟩≡ (42)

```
function resetZMQ()
    self = ceo.broker.getBroker();
    if self.zmqReset
        zmq.core.close(self.socket);
        ⟨broker client: setup ZMQ connection 48b⟩
    end
    self.zmqReset = false;
end
function setZmqResetFlag(val)
    self = ceo.broker.getBroker();
    self.zmqReset = val;
end
```

Uses `getBroker` 49b and `zmqReset` 43.

Time spent communicating:

51 ⟨*time spent* 51⟩≡ (42)

```
function timeSpent()
    self = ceo.broker.getBroker();
    fprintf('@(broker)> Time spent communicating with the server: %.3fs\n',...
                        self.elapsedTime)
    self.elapsedTime = 0;
end
```

Uses `getBroker` 49b and `self.elapsedTime` 43.

7.2 The dealer class

The *dealer* class contains the messages that are sent by the different functions of the S–function. Each CEO block instantiates a *dealer* class and tailors the messages in the initialization of the block mask. It also holds the number of inputs and outputs of the block as well as the dimensions of the inputs and outputs.

52 ⟨*dealer.m* 52⟩≡

```
classdef dealer < handle

    properties
        n_in
        n_in_ceo
        dims_in
        n_out
        n_out_ceo
        dims_out
        start
        update
        outputs
        terminate
        init
        sampleTime
        enabled
        triggered
        tag
    end

    properties (Dependent)
        currentTime
        class_id
    end

    properties (Access=private)
        p_currentTime
        p_class_id
        tid
    end

    methods

        ⟨dealer public methods 53⟩

    end

    methods (Access=private)
```

⟨*dealer private methods* 57⟩

```
    end
end
```

There are five messages that corresponds to 4 four S–function routines:

53 ⟨*dealer public methods* 53⟩≡ (52) 54 ▷

```
function self = dealer(class_id,tag)

    self.p_class_id = class_id;
    self.tag = strrep(tag,char(10),' ');
    proto_msg = struct('currentTime',[],...
                       'class_id',self.p_class_id,...
                       'method_id','',...
                       'tag',self.tag,...
                       'args',struct('args',[]));
    % Start
    self.start     = proto_msg;
    self.start.method_id = 'Start';
    % InitializeConditions
    self.init      = proto_msg;
    self.init .method_id = 'InitializeConditions';
    % Outputs
    self.update    = proto_msg;
    self.update.method_id  = 'Update';
    self.outputs   = proto_msg;
    self.outputs.method_id = 'Outputs';
    % Terminate
    self.terminate = proto_msg;
    self.terminate.method_id = 'Terminate';

    self.enabled = true;
    self.triggered = true;
end
```

Both, the *currentTime* and the *class_id* properties trigger an update of all the messages:

54 ⟨*dealer public methods* 53⟩+≡ (52) ◁53 55a▷

```
function val = get.class_id(self)
    val = self.p_class_id;
end
function set.class_id(self,val)
    self.p_class_id = val;
    self.start.class_id     = val;
    self.init.class_id      = val;
    self.update.class_id    = val;
    self.outputs.class_id   = val;
    self.terminate.class_id = val;
end
function val = get.currentTime(self)
    val = self.p_currentTime;
end
function set.currentTime(self,val)
    self.p_currentTime = val;
    self.start.currentTime     = val;
    self.init.currentTime      = val;
    self.update.currentTime    = val;
    self.outputs.currentTime   = val;
    self.terminate.currentTime = val;
end
```

7.2.1 Public methods

The properties of the blocks inputs and outputs are set with:

55a ⟨*dealer public methods* 53⟩+≡ (52) ◁54 55b▷

```
function IO_setup(self,block)
    block.NumInputPorts  = self.n_in;
    for k_in=1:self.n_in
        block.InputPort(k_in).Dimensions  = self.dims_in{k_in};
        block.InputPort(k_in).DatatypeID  = 0;  % double
        block.InputPort(k_in).Complexity  = 'Real';
        block.InputPort(k_in).SamplingMode = 'sample';
        block.InputPort(k_in).DirectFeedthrough = true;
    end
    block.NumOutputPorts = self.n_out;
    for k_out=1:self.n_out
        block.OutputPort(k_out).Dimensions   = self.dims_out{k_out};
        block.OutputPort(k_out).DatatypeID   = 0; % double
        block.OutputPort(k_out).Complexity   = 'Real';
        block.OutputPort(k_out).SamplingMode = 'sample';
    end
    block.SampleTimes = self.sampleTime;
end
```

Defines:
`IO_setup`, used in chunk 61.

The names of the output ports are set with:

55b ⟨*dealer public methods* 53⟩+≡ (52) ◁55a 56a▷

```
function output_names(self,port_handle)
    for k_out=1:self.n_out
        set(port_handle.Outport(k_out), ...
            'SignalNameFromLabel', self.outputs.args.outputs{k_out})
    end
end
```

Defines:
`output_names`, used in chunk 61.

The *deal* method sends the message to the CEO server, waits for the server replies and process the reply.

56a ⟨*dealer public methods* 53⟩+≡ (52) ◁55b 56b▷

```
function deal(self,block,tag)
    self.currentTime = {block.currentTime};
    switch tag
      case 'start'
        deal_start(self);
      case 'init'
        deal_init(self);
      case 'inputs'
        deal_inputs(self, block);
      case 'outputs'
        deal_outputs(self, block);
      case 'IO'
        deal_inputs(self, block);
        deal_outputs(self, block);
      case 'terminate'
        deal_terminate(self);
      otherwise
        fprintf(['@(dealer)> deal: Unknown tag;',...
                 ' valid tags are: start, init, IO and terminate!'])
    end
end
```

Defines:
`deal`, used in chunks 61–63.

Uses `deal_init` 57, `deal_inputs` 58, `deal_outputs` 59, `deal_start` 57, and `deal_terminate` 57.

The messages are concatenated into a single json file with:

56b ⟨*dealer public methods* 53⟩+≡ (52) ◁56a

```
function dump(self)
    s = struct('start',     self.start,...
               'init',      self.init,...
               'update',    self.update,...
               'outputs',   self.outputs,...
               'terminate', self.terminate);
    [status,message,messageid] = mkdir('JSON',gcs);
    if status<1
        error(messageid,message)
    end
    dirpath = fullfile('JSON',gcs);
    filename = [strrep(get_param(gcb,'Name'),char(10),' '),'.json'];
    savejson('',s,fullfile(dirpath,filename));
end
```

Defines:
`dump`, used in chunk 57.

7.2.2 Private methods

57 ⟨*dealer private methods* 57⟩≡ (52) 58▷

```
function deal_start(self)
    ceo.broker.resetZMQ()
    jmsg = ceo.broker.sendrecv(self.start);
    self.class_id = char(jmsg);
    fprintf('@(%s)> Object created!\n',self.tag)
    self.tid = tic;
end

function deal_init(self)
    ceo.broker.sendrecv(self.init);
    fprintf('@(%s)> Object calibrated!\n',self.tag)
    self.tid = tic;
end

function deal_terminate(self)
    toc(self.tid)
    jmsg = ceo.broker.sendrecv(self.terminate);
    dump(self)
    fprintf('@(%s)> %s\n',self.tag,jmsg)
    ceo.broker.setZmqResetFlag(true)
    ceo.broker.timeSpent()
end
```

Defines:
`deal_init`, used in chunk 56a.
`deal_start`, used in chunk 56a.
`deal_terminate`, used in chunk 56a.
Uses `dump` 56b and `sendrecv` 50a.

deal_inputs reads the block inputs and affects the input data to the corresponding field in the update message:

58 ⟨*dealer private methods* 57⟩+≡ (52) ◁57 59▷

```
function deal_inputs(self, block)
    n = self.n_in - self.n_in_ceo;
    if n>0
        self.enabled   = block.InputPort(1).Data;
        self.triggered = block.InputPort(2).Data;
    end
    if self.enabled
        if self.n_in_ceo>0
            fields = fieldnames(self.update.args.inputs_args);
            for k_in=1:self.n_in_ceo
                self.update.args.inputs_args.(fields{k_in+n}) = ...
                    reshape(block.InputPort(k_in).Data,1,[]);
            end
        end
        ceo.broker.sendrecv(self.update);
    end
end
```

Defines:
`deal_inputs`, used in chunk 56a.
Uses `sendrecv` 50a.

deal_outputs affects the inputs from the CEO server to the corresponding data field of the block outputs:

59 ⟨*dealer private methods* 57⟩+≡ (52) ◁58

```
function deal_outputs(self, block)
    if self.n_out>0
        if self.enabled && self.triggered
            outputs_msg = ceo.broker.sendrecv(self.outputs);
            try
                fields = fieldnames(outputs_msg);
            catch ME
                disp('ERROR in output_msg:')
                disp(outputs_msg)
                rethrow(ME)
            end
            for k_out=1:self.n_out
                data = outputs_msg.(fields{k_out});
                if isempty(data)
                    data = NaN(size(block.OutputPort(k_out).Data));
                end
                if iscell(data)
                    data = cellfun(@(x) double(x), data{1});
                else
                    data = double(data);
                end
                block.OutputPort(k_out).Data = data;
            end
        else
            for k_out=1:self.n_out
                block.OutputPort(k_out).Data = zeros(1,block.OutputPort(k_out).Dimensions)
            end
        end
    end
end
```

Defines:
`deal_outputs`, used in chunk 56a.

Uses `sendrecv` 50a.

7.3 The loadprm function

60a ⟨*liftprm.m* 60a⟩≡

```
function args = liftprm(prm_src)
if isstruct(prm_src)
    args = prm_src;
elseif ischar(prm_src)
    [~,~,ext] = fileparts(prm_src);
    switch ext
        case '.ubj'
            args = loadubjson(prm_src,'simplifyCell',1);
        case '.json'
            args = loadjson(prm_src,'simplifyCell',1);
        otherwise
            error('simceo:loadprm:file_error','Unrecognized file type! Valid file extensio
    end
else
    error('simceo:loadprm:type_error','Input must be either a structure or a filename!')
end
```

7.4 The SCEO S–function

60b ⟨*SCEO.m* 60b⟩≡

```
function SCEO(block)

setup(block);

⟨SCEO setup 61⟩

⟨SCEO Start 62a⟩

⟨SCEO Outputs 62b⟩

⟨SCEO Terminate 63⟩
```

7.4.1 setup

61 ⟨*SCEO setup* 61⟩≡ (60b)

```
function setup(block)

msg_box   = get(gcbh,'UserData');
fprintf('__ %s: SETUP __\n',msg_box.tag)
% Register number of ports
%block.NumInputPorts  = 0;

% Setup port properties to be inherited or dynamic
%block.SetPreCompInpPortInfoToDynamic;
%block.SetPreCompOutPortInfoToDynamic;

IO_setup(msg_box, block)

% Register sample times
%  [0 offset]            : Continuous sample time
%  [positive_num offset] : Discrete sample time
%
%  [-1, 0]               : Inherited sample time
%  [-2, 0]               : Variable sample time
%block.SampleTimes = [1 0];

% Specify the block simStateCompliance. The allowed values are:
%    'UnknownSimState', < The default setting; warn and assume DefaultSimState
%    'DefaultSimState', < Same sim state as a built-in block
%    'HasNoSimState',   < No sim state
%    'CustomSimState',  < Has GetSimState and SetSimState methods
%    'DisallowSimState' < Error out when saving or restoring the model sim state
block.SimStateCompliance = 'DefaultSimState';

%% -----------------------------------------------------------------
%% The MATLAB S-function uses an internal registry for all
%% block methods. You should register all relevant methods
%% (optional and required) as illustrated below. You may choose
%% any suitable name for the methods and implement these methods
%% as local functions within the same file. See comments
%% provided for each function for more information.
%% -----------------------------------------------------------------

block.RegBlockMethod('Start', @Start);
block.RegBlockMethod('Outputs', @Outputs);     % Required
block.RegBlockMethod('Update', @Update);
block.RegBlockMethod('Terminate', @Terminate); %
block.RegBlockMethod('PostPropagationSetup', @PostPropagationSetup);
```

```
block.RegBlockMethod('InitializeConditions', @InitializeConditions);
%end setup

function PostPropagationSetup(block)
msg_box    = get(gcbh,'UserData');
fprintf('__ %s: PostPropagationSetup __\n',msg_box.tag)
output_names(msg_box,get(gcbh, 'PortHandles'))

function InitializeConditions(block)
msg_box    = get(gcbh,'UserData');
fprintf('__ %s: InitializeConditions __\n',msg_box.tag)
deal(msg_box,block,'init')
```

Uses `deal` 56a, `IO_setup` 55a, and `output_names` 55b.

7.4.2 Start

62a ⟨*SCEO Start* 62a⟩≡ (60b)

```
function Start(block)

msg_box    = get(gcbh,'UserData');
fprintf('__ %s: START  __\n',msg_box.tag)
deal(msg_box,block,'start')
%set(gcbh,'UserData',msg_box)
%end Start
```

Uses `deal` 56a.

7.4.3 Outputs

62b ⟨*SCEO Outputs* 62b⟩≡ (60b)

```
function Outputs(block)

msg_box    = get(gcbh,'UserData');
%fprintf('__ %s: OUTPUTS __\n',msg_box.class_id)
deal(msg_box,block,'IO')

%end Outputs
```

Uses `deal` 56a.

7.4.4 Terminate

63 ⟨*SCEO Terminate* 63⟩≡ (60b)

```
function Update(block)

%msg_box   = get(gcbh,'UserData');
%deal(msg_box,block,'inputs')

%end Update

function Terminate(block)

msg_box = get(gcbh,'UserData');
deal(msg_box,block,'terminate')
%set(gcbh,'UserData',[])
%end Terminate
```

Uses `deal` 56a.

7.5 The block masks

7.5.1 Optical Path

64 ⟨*OpticalPath.md* 64⟩≡

```
# Optical Path

## Guide Star Tab

#### Zenith angle

The guide star zenith angle, in arcsecond, given with respect to
the telescope optical axis.

#### Azimuth angle

The guide star azimuth angle in degree.

#### Photometry

The guide star photometry to choose from.
This will set the wavelength, the spectral bandwidth and the magnitude zero
point.

The table below gives the values of those:

                             V       R       I       J       H       K       Ks
--------------              -------- ------- ------- ------- ------- ------- -------
$\lambda$[$\mu$m]             0.550   0.640   0.790   1.215   1.654   2.179   2.157
$\Delta\lambda$[$\mu$m]       0.090   0.150   0.150   0.260   0.290   0.410   0.320
Zero point[m$^{-2}.s^{-1}$]   8.97E9  10.87E9 7.34E9  5.16E9  2.99E9  1.90E9 1.49E9
--------------              -------- ------- ------- ------- ------- ------- --------

#### Magnitude

The guide star magnitude used to derive the number of photon taking
into account the guide star photometry.

#### \# of rays per lenslet

The \# of rays per lenslet corresponds to the number of rays used
for ray tracing through the telescope.
It has different meanings depending on the value of Sensor (See below).

### Sensor
```

The type of sensor:

* ‘None‘: No sensor is used;
the \# of rays per lenslet corresponds to the number of rays
across the telescope diameter,
* ‘Imaging‘: The sensor creates an image at the focal plane of the telescope;
the \# of rays per lenslet corresponds to the number of rays
across the diameter of the imaging lens,
* ‘ShackHartmann‘: A shack-Hartmann model where the wavefront of the guide star is
propagated from the telescope exit pupil to the focal plane of the lenslet array
using Fourier optics propagation;
the \# of rays per lenslet corresponds to the number of rays across one lenslet,
* ‘GeometricShackHartmann‘: A shack-Hartmann model where the centroids are derived
from the finite difference of the wavefront averaged on the lenslets;
the \# of rays per lenslet corresponds to the number of rays across one lenslet.
* ‘TT7‘: A shack-Hartmann model where the centroids are derived
from the finite difference of the wavefront averaged on each segment of the GMT;
the \# of rays per lenslet corresponds to the number of rays across the telescope diameter

Source FWHM

The full width at half maximum of the source intensity profile assuming a Gaussian intensi
The FWHM is given in units of pixel before binning.

Propagate through the atmosphere

If checked, the guide star is propagated through the atmosphere using the model defined in

Sample Time

The sampling time of the block outputs.

Sensor Tab

\# of lenslet

The linear size of the lenslet array.

lenslet size

The physical length of one lenslet project on M1 in meter.

camera resolution

The detector resolution of the optical sensor in pixel.

Intensity threshold

The threshold on the lenslet integrated flux. Any lenslet, whose fraction of integrated in

Pixel scale

The angular size of a pixel of the detector in arcsec.
It is given by
$(\lambda/d)(b/a)$
where both a and b
are integers.
b ia set by the adjusting the binning factor and a is set by adjusting the sampling fa

Field-of-view

The field-of-view of the wavefront sensor in arcsec.

Exposure time

The detector exposure time. A value of -1 will set it to the same value that the exposure

Exposure start

Start of the exposure delay time.

Outputs Tab

Star

Each output is derived on the telescope full pupil and/or on each segment.

Wavefront error rms

The RMS of the guide star wavefront in micron.

Piston

The piston component of the guide star wavefront in micron.

Tip-tilt

The tip-tilt component of the guide star wavefront in arcsec.

Sensor

EE80

The 80% encircled energy diameter in pixel.

Commands: Load calibration from file

The name of the file where the calibration matrices are saved to.
If the file already exists on the CEO server, the calibration matrices are loaded from thi

Commands: Calibration inputs

A ShackHartmann or GeometricShackHartmann sensor can return an estimate of
the mirror commands based on its measuremnts.
The mirror commands are given by the matrix multiplication of
the inverse of the poke matrix and the sensor measurements.
To generate the poke matrix, CEO needs to know which modes to calibrate
from which mirror (`M1` or `M2`) and what stroke to apply to these modes.

The available mirror modes are:

* `segment tip-tilt`: to calibrate the tip (Rx) and tilt (Ry) of each segment,
* `Txyz`: to calibrate the translation of each segment along its x, y and z axis,
* `Rxyz`: to calibrate the rotation of each segment along its x, y and z axis,
* `zernike`: to calibrate the Zernike modes of each segment,
* `bending modes`: to calibrate the bending modes of M1.

For example:

* to calibrate M2 segment tip--tilt, the calibration inputs argument is
```matlab
struct('M2_TT',struct('mirror','M2','mode','segment tip-tilt','stroke',1e-6))
```
where `M2_TT` is the name of the output port consisting of the 14 tip and tilts,

* to calibrate all M1 modes and to concatenate all the modes into a single calibration mat
```matlab
struct('M1_RTBM',[struct('mirror','M1','mode','Rxyz','stroke',1e-6),...
                      struct('mirror','M1','mode','Txyz','stroke',1e-6),...
                  struct('mirror','M1','mode','bending modes','stroke',1e-6)])
```

Commands: Command vector length

The length of the different command vector defined with calibration inputs.
For the examples in Calibration inputs, the length of the command vector are 14 for M2_TT
Modes Rz and Tz for segment #1 of M1 are un-observable by the WFS.
Only mode Rz for segment #1 of M2 is un-observable by the WFS.

```
For M2_TT, the output vector has the following structure: $[R_{xy}^1,R_{xy}^2,R_{xy}^3,R_{
For M1_RTBM, the output vector is: $[R_{xyz},T_{xyz},BM]$ with $(R_{xyz} \equiv X,T_{xyz}

#### Commands: SVD truncation

The number of eigen values, from the singular value decomposition of the calibration matri
If the calibration is loaded from a previously saved file, the threshold is re-applied and

#### Commands: Decoupling segments

If checked, eaach segment is controlled independently from the others,
the lenslets that span across two segments are rejected and there are 7 command matrices i
Otherwise M1 and M2 mirrors are controlled in the same way that non segmented mirrors.
```

Uses `sensor` 16b.

7.5.2 GMT Mirror

68 ⟨*GMTMirror.md* 68⟩≡

```
# GMT Mirror

#### Mirror

Either the primary M1 or the secondary M2 mirror.

### Mirror commands

The mirrors accept two types of intputs:

#### Txyz and Rxyz rigid body

A $7\times6$ matrix concatenating row wise the vectors '[Tx,Ty,Tz,Rx,Ry,Rz]' of segments 1

#### Mirror mode coefficients

The coefficients of the segments modal basis that is used to shape the segments.
It is a $7\times$'n_mode' matrix of either bending mode for M1 or Zernike coefficients for
```

8 The CEO server

The CEO daemon is start at boot time with the *CEO.sh* shell script. It must be placed in the `/etc/init.d` directory.

69 ⟨*CEO.sh* 69⟩≡

```
#!/bin/bash -e

DAEMON="/usr/bin/env LD_LIBRARY_PATH=/usr/local/cuda/lib64 PYTHONPATH=/home/ubuntu/CEO/pyt
daemon_OPT=""
DAEMONUSER="root"
daemon_NAME="ceo_server"
PIDFILE=/var/run/$daemon_NAME.pid

PATH="/sbin:/bin:/usr/sbin:/usr/bin" #Ne pas toucher

#test -x $DAEMON || exit 0

. /lib/lsb/init-functions

d_start () {
        log_daemon_msg "Starting system $daemon_NAME Daemon"
        start-stop-daemon --background --name $daemon_NAME --start --quiet --make-pidfile
        log_end_msg $?
}

d_stop () {
        log_daemon_msg "Stopping system $daemon_NAME Daemon"
        start-stop-daemon --name $daemon_NAME --stop --retry 5 --quiet --pidfile "$PIDFILE
        log_end_msg $?
}

case "$1" in

        start|stop)
                d_${1}
                ;;

        restart|reload|force-reload)
                        d_stop
                        d_start
                ;;

        force-stop)
               d_stop
                killall -q $daemon_NAME || true
                sleep 2
```

```
                killall -q -9 $daemon_NAME || true
                ;;

        status)
                status_of_proc "$daemon_NAME" "$DAEMON" "system-wide $daemon_NAME" && exit
                ;;
        *)
                echo "Usage: /etc/init.d/$daemon_NAME {start|stop|force-stop|restart|reloa
                exit 1
                ;;
esac
exit 0
```
Uses etc 43.

9 Index

10 List of code chunks