

SIMCEO

Simulink Client

CEO Server

R. Conan

GMTO Corporation

September 8, 2016

Contents

1	Introduction	2
2	Installation	2
2.1	AWS command line interface	2
2.2	Matlab-ZMQ	3
2.3	UBJSON	3
3	Implementation	3
4	The simulink python module	4
4.1	The broker class	6
4.2	The S classes	8
4.2.1	The SGMT class	9
	Start	9
	Update	10
	InitializeConditions	10
	Outputs	10
4.2.2	The SAtmosphere class	11
4.2.3	The SOpticalPath class	11
	Start	12
	Terminate	13
	Update	13
	Outputs	14
	InitializeConditions	15
4.3	The CalibrationMatrix class	17

5	The ceo Matlab package	18
5.1	The broker class	18
5.1.1	run_instance	20
5.1.2	terminate_instance	21
5.1.3	start_instance	21
5.2	The messages class	27
5.2.1	IO_setup	29
5.3	The SCEO S-function	32
5.3.1	setup	33
5.3.2	Start	34
5.3.3	Outputs	34
5.3.4	Terminate	35
5.4	The block masks	36
5.4.1	Optical Path	36
5.4.2	GMT Mirror	39
6	The CEO server	40
7	Index	42
8	List of code chunks	42

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-ZMQ¹ is a Matlab wrapper for ZeroMQ. ZeroMQ² 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` or `windows`.

2.3 UBJSON

Universal Binary JSON (UBJSON³) 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* is responsible for starting the

¹<https://github.com/fagg/matlab-zmq>

²<http://zeromq.org/>

³<http://ubjson.org/>

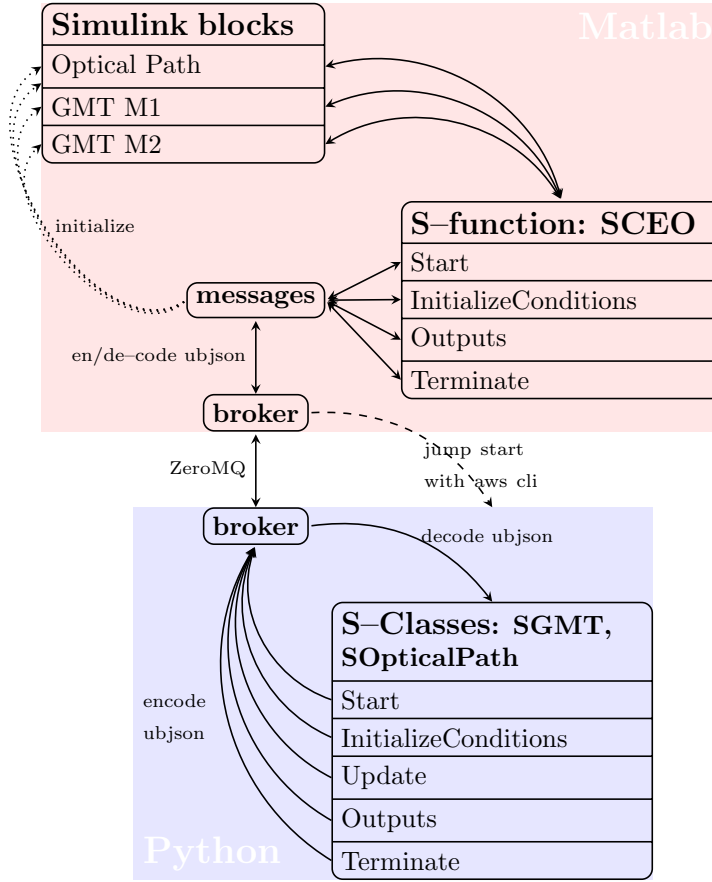


Figure 1: SIMCEO flowchart.

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*:

```
4 <simceo.py 4>≡
  import threading
```

```

import time
import zmq
import ubjson
import ceo
import numpy as np
from collections import OrderedDict
import os
import shelve

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

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 17>

<S-function 8c>

<SGMT 9a>
<SAtmosphere 11a>
<SOpticalPath 11b>

<broker 6a>

if __name__ == "__main__":

    agent = broker()
    agent.start()

```

Uses `start` 27.

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.

6a $\langle \text{broker 6a} \rangle \equiv$ (4)

```
class broker(threading.Thread):

    def __init__(self):

        threading.Thread.__init__(self)

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

        self.sgmt = SGMT()
        self.satm = SATmosphere()
        self.ops = []
        self.n_op = 0

    def __del__(self):

        self.release()

    def release(self):

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

$\langle \text{broker get item 7c} \rangle$

$\langle \text{broker run 6b} \rangle$

Uses `socket 18`.

The *run* method

6b $\langle \text{broker run 6b} \rangle \equiv$ (6a)

```
def run(self):
```

```
    while True:
```

$\langle \text{broker run details 7a} \rangle$

waits for a request from a Simulink S-function:

```
7a  <broker run details 7a>≡ (6b) 7b>
    msg = self.socket.recv()
    jmsg = ubjson.loadb(msg)
    #print jmsg
Uses socket 18.
```

The message received from the S-function contains

- 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 dictionary of the arguments to the method, *args*.

The class method is invoked with:

```
7b  <broker run details 7a>+≡ (6b) <7a 8a>
    class_id = jmsg["class_id"]
    method_id = jmsg["method_id"]
    try:
        args_out = getattr( self[class_id], method_id )( **jmsg["args"] )
    except Exception as E:
        print "The server as failed:"
        print E
        print "Recovering gracefully..."
        class_id = ""
        args_out = "The server has failed!"
```

The dictionary-like call is implemented with

```
7c  <broker get item 7c>≡ (6a)
    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 ) )

                self.n_op = len(self.ops)
                return self.ops[-1]
        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 instantiated 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*.

```
8a <broker run details 7a>+= (6b) <7b 8b>
    if class_id[:2]=="OP" and method_id=="Terminate":
        # del(self.ops[int(class_id[2:])])
        self.ops.pop(0)
```

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

```
8b <broker run details 7a>+= (6b) <8a>
    self.socket.send(ubjson.dumps(args_out))
Uses socket 18.
```

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:

```
8c <S-function 8c>= (4)
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.

9a $\langle SGMT \text{ 9a} \rangle \equiv$ (4) 9c
 class SGMT(Sfunction):

```
        def __init__(self):
            self.gmt = ceo.GMT_MX()

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

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

9b $\langle SGMT \text{ Start message 9b} \rangle \equiv$

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

9c $\langle SGMT \text{ 9a} \rangle + \equiv$ (4) <9a 10b>
 def Start(self, mirror=None, mirror_args=None):
 #print mirror_args
 self.gmt[mirror] = getattr(ceo, "GMT_"+mirror)(**mirror_args)
 #print self.gmt[mirror].modes
 return "GMT"

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

```

10a  <SOpticalPath Update message 10a>≡                                     13c>
    {
      "class_id": "GMT",
      "method_id": "Update",
      "args":
        {
          "mirror": "M1"|"M2",
          "inputs_args":
            {
              "TxyzRxyz": null,
              "mode_coefs": null
            }
        }
    }

10b  <SGMT 9a>+≡                                                         (4) <9c 10c>
    def Update(self, mirror=None, inputs_args=None):
        #print "Updating GMT!"
        for key in inputs_args:
            data = np.array( inputs_args[key], order='C', dtype=np.float64 )
            data = np.transpose( np.reshape( data , (-1,7) ) )
            if key=="TxyzRxyz":
                #print np.array_str(data[:,3], suppress_small=True)
                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 = np.copy( data, order='C')
                self.gmt[mirror].modes.update()

```

Uses [update 27](#).

InitializeConditions

```

10c  <SGMT 9a>+≡                                                         (4) <10b 10d>
    def InitializeConditions(self, args=None):
        pass

```

Outputs

```

10d  <SGMT 9a>+≡                                                         (4) <10c>
    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.

```
11a  ⟨SATmosphere 11a⟩≡ (4)
      class SATmosphere(Sfunction):

          def __init__(self):
              self.atm = None

          def Start(self,atmosphere_args=None):
              self.atm = ceo.GmtAtmosphere( **atmosphere_args )
              return "ATM"

          def Terminate(self, args=None):
              self.atm = None
              return "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*.

```
11b  ⟨SOpticalPath 11b⟩≡ (4) 12b▷
      class SOpticalPath(Sfunction):

          def __init__(self, idx, gmt, atm):
              self.idx = idx
              self.gmt = gmt
              self.atm = atm
              self.sensor = None
              self.D = {}
```

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

12a $\langle SOpticalPath \text{ Start message 12a} \rangle \equiv$

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

12b $\langle SOpticalPath \text{ 11b} \rangle + \equiv$

(4) $\langle 11b \text{ 13b} \rangle$

```
def Start(self,source_args=None, sensor_class=None, sensor_args=None,
          calibration_source_args=None, miscellaneous_args=None):
    print miscellaneous_args
    self.src = ceo.Source( **source_args )
    if sensor_class is not 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.src.reset()
    self.gmt.reset()
    self.gmt.propagate(self.src)
    self.sensor.calibrate(self.calib_src,miscellaneous_args['intensity_threshold'])
    if isinstance(self.sensor,(ceo.ShackHartmann,ceo.GeometricShackHartmann)):
        print "# of valid slopes: %d"%self.sensor.n_valid_slopes

    self.poke_matrix = {}
    self.comm_matrix = {}

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

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

13a $\langle SOpticalPath \text{ Terminate message } 13a \rangle \equiv$

```

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

```

13b $\langle SOpticalPath \text{ 11b} \rangle + \equiv$ (4) $\langle 12b \text{ } 13d \rangle$

```

def Terminate(self, args=None):
    return "OpticalPath deleted!"

```

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

13c $\langle SOpticalPath \text{ Update message } 10a \rangle + \equiv$ $\langle 10a \rangle$

```

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

```

13d $\langle SOpticalPath \text{ 11b} \rangle + \equiv$ (4) $\langle 13b \text{ } 14b \rangle$

```

def Update(self, inputs=None):
    #print "Updating OP!"
    self.src.reset()
    self.gmt.propagate(self.src)
    #print "WFE RMS: %fnm"%self.src.wavefront.rms(-9)
    if self.sensor is not None:
        self.sensor.reset()
        self.sensor.propagate(self.src)
        #self.sensor.readOut(T,0)
        self.sensor.process()

```

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

```
14a  <SOpticalPath Outputs message 14a>≡
      {
        "class_id": "OP",
        "method_id": "Outputs",
        "args":
          {
            "outputs": ["wfe_rms"|"segment_wfe_rms"|"piston"|"segment_piston"|"ee80"]
          }
      }
```

Uses outputs 27.

```
14b  <SOpticalPath 11b>+≡ (4) <13d 14c>
      def Outputs(self, outputs=None):
          doutputs = OrderedDict()
          for element in outputs:
              #print self.src.wavefront.rms()
              doutputs[element] = self[element]
          return doutputs
```

Uses outputs 27.

and the dictionary implementation is

```
14c  <SOpticalPath 11b>+≡ (4) <14b 16>
      def __getitem__(self, key):
          if key=="wfe_rms":
              return self.src.wavefront.rms().tolist()
          elif key=="segment_piston":
              return self.src.piston(where="segments").tolist()
          elif key=="ee80":
              return self.sensor.ee80().tolist()
          else:
              c = np.dot(self.comm_matrix[key],
                          self.sensor.valid_slopes.host(
                              shape=(self.sensor.n_valid_slopes,1) ) ).reshape(1,-1)
              c[c==0] = 1e-100
              return c.tolist()
```

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

```
15  <SOpticalPath InitializeConditions message 15>≡
    {
      "class_id": "OP",
      "method_id": "InitializeConditions",
      "args":
        {
          "calibrations":
            {
              "M2_TT":
                {
                  "mirror": "M2",
                  "mode": "segment tip-tilt",
                  "stroke": 1e-6
                }
              "M12_Rxyz": [
                {
                  "mirror": "M1",
                  "mode": "Rxyz",
                  "stroke": 1e-6
                },
                {
                  "mirror": "M2",
                  "mode": "Rxyz",
                  "stroke": 1e-6
                }
              ]
            }
          "calibration_file": null,
          "SVD_truncation": 0
        }
    }
```

```

16    <SOpticalPath 11b>+≡ (4) <14c
    def InitializeConditions(self, calibrations = None,
                             calibration_file = None,
                             SVD_truncation = 0):
    print "@(SOpticalPath:InitializeConditions)>"
    if calibrations is not None:
        #import scipy.io.matlab as matlab
        #print kwargs
        k = 0
        if not isinstance(SVD_truncation,list):
            SVD_truncation = [SVD_truncation]
        if calibration_file is not None:
            filepath = os.path.join(SIMCEOPATH,"calibration_dbs",calibration_file)
            db = shelve.open(filepath)
            if os.path.isfile(filepath+".dir"):
                print " . Loading command matrix from existing database %s!"%calibration_file
                for key in db:
                    C = db[key]
                    C.nThreshold = SVD_truncation[k]
                    k+=1
                    self.comm_matrix[key] = C.M
                    db[key] = C
                db.close()
                return
            #db = shelve.open(SIMCEOPATH+"/calibration_dbs/"+calibration_file)
        with Timer():
            for key in calibrations: # Through calibrations
                calibs = calibrations[key]
                if not isinstance(calibs,list):
                    calibs = [calibs]
                D = []
                #print calibs
                for c in calibs: # Through calib
                    #print c
                    D.append( self.gmt.calibrate(self.sensor,self.src,**c) )
                self.gmt.reset()

                Dc = np.concatenate( D, axis=1 )
                C = CalibrationMatrix(Dc, SVD_truncation[k])
                k+=1
                self.poke_matrix[key] = C
                #matlab.savemat('poke_matrix.mat',{'D':Dc})
                self.comm_matrix[key] = self.poke_matrix[key].M

        if calibration_file is not None:
            print " . Saving command matrix to database %s!"%calibration_file

```



```

db[str(key)] = C

if calibration_file is not None:
    db.close()

```

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

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

```

17  <CalibrationMatrix 17>≡ (4)
    class CalibrationMatrix(object):

        def __init__(self, D, n):
            print "@(CalibrationMatrix)> Computing the SVD and the pseudo-inverse..."
            self.D = D
            self._n = n
            with Timer():
                self.U, self.s, self.V = np.linalg.svd(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))

        @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
            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))

```

5 The ceo Matlab package

5.1 The broker class

```
18  <broker.m 18>≡
    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
            awspath % full path to the AWS CLI
            instance_id % The AWS instance ID number
            public_ip % The AWS instance public IP
            zmqReset % ZMQ connection reset flag
        end

        properties (Access=private)
            etc
            instance_end_state
            ctx
            socket
        end

        methods

            <broker client 19a>

            <release ressources 23b>

            <launch AWS AMI 20a>

            <start AWS instance 21c>

            <terminate AWS instance 21b>
        end

        methods(Static)

            <instanciation and retrieval 24>

            <request and reply 25>

            <reset ZMQ socket 26>
        end
    end
```

end

Defines:

awspath, used in chunks 19–22 and 26.
ctx, used in chunks 19a and 23.
etc, used in chunks 19–21 and 40.
instance_end_state, used in chunks 19b and 21b.
instance_id, used in chunks 19–22 and 26.
public_ip, used in chunks 22c and 23a.
socket, used in chunks 6–8, 19a, 23, 25, and 26.
zmqReset, used in chunks 19a and 26.

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

```
19a  <broker client 19a>≡ (18)
      function self = broker(varargin)

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

          currentpath = mfilename('fullpath');
          k = strfind(currentpath,filesep);
          self.etc = fullfile(currentpath(1:k(end)),'..','etc');
          cfg = loadjson(fullfile(self.etc,'simceo.json'));
          self.awspath      = cfg.awsclipath;
          self.instance_id  = cfg.aws_instance_id;
          <broker client: AWS instance launch 19b>
      end
```

Uses awspath 18, ctx 18, etc 18, instance_id 18, socket 18, and zmqReset 18.

If not instance ID is given, a new machine is launched based on a given AWS AMI.

```
19b  <broker client: AWS instance launch 19b>≡ (19a)
      if isempty(self.instance_id)
          run_instance(self)
          self.instance_end_state = 'terminate';
      else
          start_instance(self)
          self.instance_end_state = 'stop';
      end
```

Uses instance_end_state 18 and instance_id 18.

5.1.1 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.

20a $\langle \text{launch AWS AMI 20a} \rangle \equiv$ (18)

```
function run_instance(self)
     $\langle \text{launching an instance 20b} \rangle$ 
     $\langle \text{waiting for the running state 22b} \rangle$ 
     $\langle \text{waiting for initialization 20c} \rangle$ 
     $\langle \text{setting up cloudwatch 21a} \rangle$ 
     $\langle \text{getting the public IP 22c} \rangle$ 
end
```

The sequence of operations is:

1. launching the instance,

20b $\langle \text{launching an instance 20b} \rangle \equiv$ (20a)

```
cmd = sprintf(['%s ec2 run-instances --profile gmtto.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 18, etc 18, and instance_id 18.
```

2. waiting for the confirmation that the instance is running,

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

20c $\langle \text{waiting for initialization 20c} \rangle \equiv$ (20a)

```
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 gmtto.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 18 and instance_id 18.
```

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

21a \langle setting up cloudwatch 21a $\rangle \equiv$ (20a)

```

    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 18, etc 18, and instance_id 18.

```

5. getting the public IP of the instance.

5.1.2 terminate_instance

21b \langle terminate AWS instance 21b $\rangle \equiv$ (18)

```

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
end

```

Uses awspath 18, instance_end_state 18, and instance_id 18.

5.1.3 start_instance

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

21c \langle start AWS instance 21c $\rangle \equiv$ (18)

```

function start_instance(self)
     $\langle$ starting an instance 22a $\rangle$ 
     $\langle$ waiting for the running state 22b $\rangle$ 
     $\langle$ getting the public IP 22c $\rangle$ 
end

```

The sequence of operations is:

1. starting the instance:

```
22a  <starting an instance 22a>≡ (21c)
      cmd = sprintf(['%s ec2 start-instances --instance-ids %s',...
                    ' --profile gmtto.control'],...
                    self.awspath,self.instance_id);
      fprintf('%s\n',cmd)
      fprintf('@(broker)> Starting AWS machine %s...\n',self.instance_id)
      [status,cmdout] = system(cmd);
      if status~=0
          error('Starting AWS machine %s failed:\n%s',self.instance_id,cmdout)
      end
```

Uses awspath 18, instance_id 18, and start 27.

2. waiting for the confirmation that the instance is running

```
22b  <waiting for the running state 22b>≡ (20a 21c)
      fprintf('>>>> WAITING FOR AWS INSTANCE %s TO START ... \n',self.instance_id)
      tic
      [status,~] = system(sprintf(['%s ec2 wait instance-running --instance-ids %s',...
                                  ' --profile gmtto.control'],...
                                  self.awspath,self.instance_id));

      toc
      if status~=0
          error('Starting AWS machine %s failed!',self.instance_id')
      end
```

Uses awspath 18 and instance_id 18.

3. getting the public IP of the instance.

```
22c  <getting the public IP 22c>≡ (20a 21c)
      cmd = sprintf(['%s ec2 describe-instances --instance-ids %s',...
                    ' --output text',...
                    ' --query Reservations[*].Instances[*].PublicIpAddress',...
                    ' --profile gmtto.control'],...
                    self.awspath,self.instance_id);
      [status,public_ip_] = system(cmd);
      if status~=0
          error('Getting AWS machine public IP failed!')
      end
      self.public_ip = strtrim(public_ip_);
      fprintf('\n ==>> machine is up and running @%s\n',self.public_ip)
```

Uses awspath 18, instance_id 18, and public_ip 18.

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

```
23a  <broker client: setup ZMQ connection 23a>≡ (26)
    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 ctx 18, public_ip 18, and socket 18.

The allocated ZeroMQ resources are released with:

```
23b  <release resources 23b>≡ (18)
    function delete(self)
        fprintf('@(broker)> Deleting %s\n', class(self))
        terminate_instance(self)
        zmq.core.close(self.socket);
        zmq.core.ctx_shutdown(self.ctx);
        zmq.core.ctx_term(self.ctx);
    end
```

Uses ctx 18 and socket 18.

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

24 \langle instanciation and retrieval 24 $\rangle \equiv$ (18)

```

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)
    fprintf('~~~~~')
    fprintf('\n SIMCEO CLIENT!\n')
    fprintf('~~~~~\n')
    this = ceo.broker(varargin{:});
end
self = this;
end

```


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

```
25  <request and reply 25>≡ (18)
    function jmsg = sendrecv(send_msg)
        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
    end
end
```

Uses `socket` 18.

resetZMQ resets the ZeroMQ socket

```
26  <reset ZMQ socket 26>≡ (18)
    function resetZMQ()
        self = ceo.broker.getBroker();
        if self.zmqReset
            [~,aws_instance_state] = system(...
                sprintf(['%s ec2 describe-instances --instance-ids %s',...
                    ' --output text',...
                    ' --query Reservations[*].Instances[*].State.Name ',...
                    '--profile gmtto.control'],...
                self.awspath, self.instance_id));
            if any(strcmp(strtrim(aws_instance_state),{'shutting-down','terminated'}))
                run_instance(self)
            end
            zmq.core.close(self.socket);
            <broker client: setup ZMQ connection 23a>
        end
        self.zmqReset = false;
    end
    function setZmqResetFlag(val)
        self = ceo.broker.getBroker();
        self.zmqReset = val;
    end
```

Uses awspath 18, instance_id 18, socket 18, and zmqReset 18.

5.2 The messages class

The *messages* class contains the messages that are sent by the different functions of the S-function. Each CEO block instantiates a *messages* 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.

```
27  <messages.m 27>≡
    classdef messages < handle

        properties
            n_in
            dims_in
            n_out
            dims_out
            start
            update
            outputs
            terminate
            init
            sampleTime
            tag
        end

        properties (Dependent)
            class_id
        end

        properties (Access=private)
            p_class_id
        end

        methods

            <messages public methods 28>

        end

        methods (Access=private)

            <messages private methods 30b>

        end
    end
end
```

Defines:

`dims_in`, used in chunk [29b](#).

dims_out, used in chunk 29b.
 init, used in chunks 28–30 and 40.
 n_in, used in chunks 29b and 31a.
 n_out, used in chunks 29b and 31b.
 outputs, used in chunks 14, 28, 29a, and 31b.
 start, used in chunks 4, 22a, 28–30, and 40.
 terminate, used in chunks 28–30.
 update, used in chunks 10b, 28, 29a, and 31a.

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

```

28  <messages public methods 28>≡ (27) 29a>
      function self = messages(class_id, tag)

          self.p_class_id = class_id;
          self.tag = tag;
          proto_msg = struct('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';
      end
  
```

Uses init 27, outputs 27, start 27, terminate 27, and update 27.

The *class_id* property triggers an update of all the messages:

```
29a  <messages public methods 28>+≡ (27) <28 29b>
      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
```

Uses init 27, outputs 27, start 27, terminate 27, and update 27.

5.2.1 IO_setup

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

```
29b  <messages public methods 28>+≡ (27) <29a 30a>
      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).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
```

Uses dims_in 27, dims_out 27, n_in 27, and n_out 27.

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

```

30a  <messages public methods 28>+≡ (27) <29b
      function deal(self,block,tag)
        switch tag
          case 'start'
            deal_start(self);
          case 'init'
            deal_init(self);
          case 'IO'
            deal_inputs(self, block);
            deal_outputs(self, block);
          case 'terminate'
            deal_terminate(self);
          otherwise
            fprintf(['@(messages)> deal: Unknown tag;',...
                    ' valid tags are: start, init, IO and terminate!'])
        end
      end
    end

```

Uses init 27, start 27, and terminate 27.

```

30b  <messages private methods 30b>≡ (27) 31a>
      function deal_start(self)
        ceo.broker.resetZMQ()
        jmsg = ceo.broker.sendrecv(self.start);
        tag = char(jmsg);
        self.class_id = tag;
        fprintf('@(s)> Object created!\n',self.tag)
      end

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

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

```

Uses init 27, start 27, and terminate 27.

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

31a $\langle \text{messages private methods 30b} \rangle + \equiv$ (27) $\langle \text{30b 31b} \rangle$

```

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

```

Uses *n_in* 27 and *update* 27.

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

31b $\langle \text{messages private methods 30b} \rangle + \equiv$ (27) $\langle \text{31a} \rangle$

```

function deal_outputs(self, block)
    if self.n_out>0
        outputs_msg = ceo.broker.sendrecv(self.outputs);
        fields = fieldnames(outputs_msg);
        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
            block.OutputPort(k_out).Data = data;
        end
    end
end

```

Uses *n_out* 27 and *outputs* 27.

5.3 The SCEO S-function

32 $\langle SCEO.m \text{ 32} \rangle \equiv$
function SCEO(block)

setup(block);

 $\langle SCEO \text{ setup } 33 \rangle$

 $\langle SCEO \text{ Start } 34a \rangle$

 $\langle SCEO \text{ Outputs } 34b \rangle$

 $\langle SCEO \text{ Terminate } 35 \rangle$

5.3.1 setup

```

33   $\langle SCEO \text{ setup } 33 \rangle \equiv$  (32)
    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); % Required
        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)

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

```

5.3.2 Start

34a $\langle \text{SCEO Start 34a} \rangle \equiv$ (32)

```

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)
tic
%end Start

```

5.3.3 Outputs

34b $\langle \text{SCEO Outputs 34b} \rangle \equiv$ (32)

```

function Outputs(block)

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

deal(msg_box,block,'IO')

%end Outputs

```

5.3.4 Terminate

35 $\langle SCEO \text{ Terminate } 35 \rangle \equiv$ (32)

```
function Update(block)

%end Update

function Terminate(block)

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

5.4 The block masks

5.4.1 Optical Path

36 $\langle \text{OpticalPath.md } 36 \rangle \equiv$

```
# 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.

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.

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

Outputs Tab

Star

Wavefront error rms

The RMS, over the telescope pupil, of the guide star wavefront in meter.

Piston

The piston component of the guide star wavefront in meter.

Segment Piston

The piston component of the guide star wavefront, per segment, in meter.

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 this file.

Commands: Calibration inputs

A ShackHartmann or GeometricShackHartmann sensor can return an estimate of the mirror commands based on its measurements.

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 matrix

```
'''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: $[R_{xy}^1, R_{xy}^2, R_{xy}^3, R_{xy}^4, R_{xy}^5, R_{xy}^6, R_{xy}^7, R_{xy}^8, R_{xy}^9, R_{xy}^{10}, R_{xy}^{11}, R_{xy}^{12}, R_{xy}^{13}, R_{xy}^{14}]$

For M1_RTBM, the output vector is: $[R_{xyz}, T_{xyz}, BM]$ with $(R_{xyz} \equiv X, T_{xyz} \equiv Y, BM \equiv Z)$

Commands: SVD truncation

The number of eigen values, from the singular value decomposition of the calibration matrix

If the calibration is loaded from a previously saved file, the threshold is re-applied and

5.4.2 GMT Mirror

39 $\langle GMTMirror.md \ 39 \rangle \equiv$
 # GMT Mirror

Mirror

Either the primary M1 or the secondary M2 mirror.

Mirror commands

The mirrors accept two types of inputs:

Txyz and Rxyz rigid body

A 7×6 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

6 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.

```
40 <CEO.sh 40>≡
    #!/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
    esac
```



```

        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|reload}"
    exit 1
    ;;
esac
exit 0

```

Uses [etc 18](#), [init 27](#), and [start 27](#).

7 Index

awspath: [18](#), [19a](#), [20b](#), [20c](#), [21a](#), [21b](#), [22a](#), [22b](#), [22c](#), [26](#)
ctx: [18](#), [19a](#), [23a](#), [23b](#)
dims_in: [27](#), [29b](#)
dims_out: [27](#), [29b](#)
etc: [18](#), [19a](#), [20b](#), [21a](#), [40](#)
init: [27](#), [28](#), [29a](#), [30a](#), [30b](#), [40](#)
instance_end_state: [18](#), [19b](#), [21b](#)
instance_id: [18](#), [19a](#), [19b](#), [20b](#), [20c](#), [21a](#), [21b](#), [22a](#), [22b](#), [22c](#), [26](#)
n_in: [27](#), [29b](#), [31a](#)
n_out: [27](#), [29b](#), [31b](#)
outputs: [14a](#), [14b](#), [27](#), [28](#), [29a](#), [31b](#)
public_ip: [18](#), [22c](#), [23a](#)
socket: [6a](#), [7a](#), [8b](#), [18](#), [19a](#), [23a](#), [23b](#), [25](#), [26](#)
start: [4](#), [22a](#), [27](#), [28](#), [29a](#), [30a](#), [30b](#), [40](#)
terminate: [27](#), [28](#), [29a](#), [30a](#), [30b](#)
update: [10b](#), [27](#), [28](#), [29a](#), [31a](#)
zmqReset: [18](#), [19a](#), [26](#)

8 List of code chunks

<broker 6a>
<broker client 19a>
<broker client: AWS instance launch 19b>
<broker client: setup ZMQ connection 23a>
<broker get item 7c>
<broker run 6b>
<broker run details 7a>
<broker.m 18>
<CalibrationMatrix 17>
<CEO.sh 40>
<getting the public IP 22c>
<GMTMirror.md 39>
<instanciation and retrieval 24>
<launch AWS AMI 20a>
<launching an instance 20b>
<messages private methods 30b>
<messages public methods 28>
<messages.m 27>
<OpticalPath.md 36>
<release ressources 23b>
<request and reply 25>
<reset ZMQ socket 26>
<S-function 8c>

⟨*SAtmosphere* [11a](#)⟩
 ⟨*SCEO Outputs* [34b](#)⟩
 ⟨*SCEO setup* [33](#)⟩
 ⟨*SCEO Start* [34a](#)⟩
 ⟨*SCEO Terminate* [35](#)⟩
 ⟨*SCEO.m* [32](#)⟩
 ⟨*setting up cloudwatch* [21a](#)⟩
 ⟨*SGMT* [9a](#)⟩
 ⟨*SGMT Start message* [9b](#)⟩
 ⟨*simceo.py* [4](#)⟩
 ⟨*SOpticalPath* [11b](#)⟩
 ⟨*SOpticalPath InitializeConditions message* [15](#)⟩
 ⟨*SOpticalPath Outputs message* [14a](#)⟩
 ⟨*SOpticalPath Start message* [12a](#)⟩
 ⟨*SOpticalPath Terminate message* [13a](#)⟩
 ⟨*SOpticalPath Update message* [10a](#)⟩
 ⟨*start AWS instance* [21c](#)⟩
 ⟨*starting an instance* [22a](#)⟩
 ⟨*terminate AWS instance* [21b](#)⟩
 ⟨*waiting for initialization* [20c](#)⟩
 ⟨*waiting for the running state* [22b](#)⟩