

SIMCEO

Simulink Client

CEO Server

R. Conan

GMTO Corporation

July 18, 2019

Contents

1	Introduction	4
2	Installation	4
2.1	AWS command line interface	4
2.2	Matlab-ZMQ	4
2.3	UBJSON	5
3	Implementation	5
4	The simulink python module	5
4.1	The broker class	8
4.2	The S classes	12
4.2.1	The SGMT class	12
	Start	13
	Update	14
	InitializeConditions	15
	Outputs	16
4.2.2	The SATmosphere class	16
4.2.3	The SOpticalPath class	17
	Start	17
	Terminate	19
	Update	19
	Outputs	19
	InitializeConditions	22
4.3	The CalibrationMatrix class	27
4.4	The Sensor abstract class	29
5	DOS	29
5.1	DOS driver	32
6	The python client	34
7	The ceo Matlab package	36
7.1	The broker class	36
7.1.1	run_instance	39
7.1.2	terminate_instance	41
7.1.3	start_instance	42
7.2	The dealer class	47
7.2.1	Public methods	50
7.2.2	Private methods	52
7.3	The loadprm function	55
7.4	The SCEO S-function	55
7.4.1	setup	56
7.4.2	Start	57
7.4.3	Outputs	57

7.4.4	Terminate	58
7.5	The block masks	59
7.5.1	Optical Path	59
7.5.2	GMT Mirror	63
8	The CEO server	64
9	Index	66
10	List of code chunks	66

1 Introduction

This document describes SIMCEO, an interface between CEO and Simulink. SIMCEO allows to seamlessly integrate 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 on 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** **windows7** or **windows10**.

¹<https://github.com/fagg/matlab-zmq>

²<http://zeromq.org/>

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 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/>

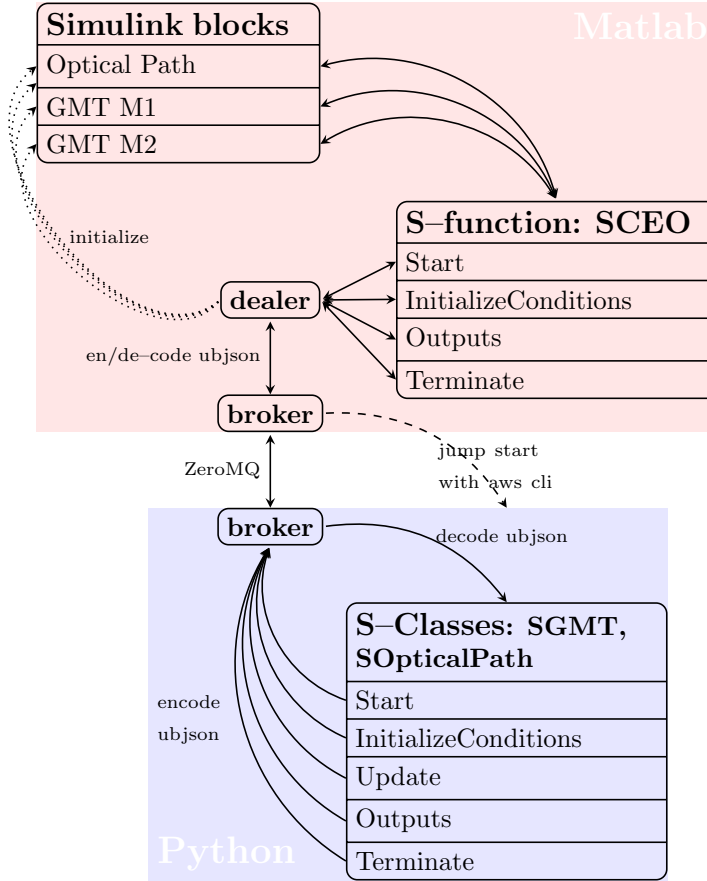


Figure 1: SIMCEO flowchart.

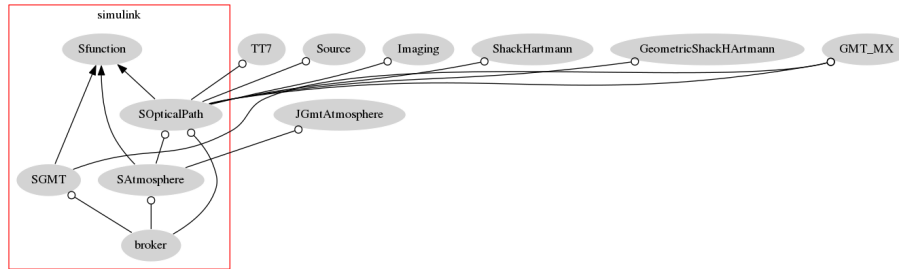


Figure 2: The classes in the simulink python module and their relations with the CEO classes.

```

import shelve
import traceback
import scipy.linalg as LA
import pickle
import zlib
#import M1 as MDL

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 12a>

<SGMT 12b>
<SAtmosphere 16b>
<SOpticalPath 17a>

<broker 8>

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.

```
8  <broker 8>≡ (5)
    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)
            print(self._recv_())
            self._send_("Acknowledging connection to SIMCEO server!")

            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 11a>
```


⟨broker run 9a⟩

The *run* method

9a *⟨broker run 9a⟩*≡ (8)

```

def run(self):

    while True:

        ⟨broker run details 9b⟩
        waits for a request from a Simulink S-function:
9b    ⟨broker run details 9b⟩≡ (9a) 10▷
        #jmsg = ubjson.loadb(msg)
        msg = ''
        try:
            msg = self.socket.recv()
            jmsg = ubjson.loadb(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 dictionary of the arguments to the method, *args*.

The class method is invoked with:

```
10  <broker run details 9b>+≡ (9a) <9b 11b>
    self.currentTime = float( jmsg["currentTime"][0][0] )
    class_id = jmsg["class_id"]
    method_id = jmsg["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 )( **jmsg["args"] )
        #tid.toc()
        #print "%s->%s: %.2f"%(class_id,method_id,tid.elapsedTime)
    except Exception as E:
        print("@(broker)> The server has failed!")
        print(jmsg)
        traceback.print_exc()
        print("@(broker)> Recovering gracefully...")
        class_id = ""
        args_out = b"The server has failed!"
```

The dictionary-like call is implemented with

```

11a  <broker get item 11a>≡ (8)
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 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*.

```

11b  <broker run details 9b>+≡ (9a) <10 11c>
    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:

```

11c  <broker run details 9b>+≡ (9a) <11b>
    self.socket.send(ubjson.dumps(args_out,no_float32=True))

```

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:

```
12a  <S-function 12a>≡ (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.

```
12b  <SGMT 12b>≡ (5) 13b>
      class SGMT(Sfunction):

          def __init__(self, ops, satm):
              self.gmt = ceo.GMT_MX()
              self.M1 = None

          def Terminate(self, args=None):
              print("\n@ (SGMT:Terminate)>")
              self.gmt = ceo.GMT_MX()
              return b"GMT deleted!"
```

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

13a $\langle SGMT \text{ Start message } 13a \rangle \equiv$

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

13b $\langle SGMT \text{ } 12b \rangle + \equiv$ (5) $\langle 12b \text{ } 15a \rangle$

```
def Start(self,mirror=None,mirror_args=None):
    print("\n@(SGMT:Start)>")
    print(mirror_args)
    if mirror_args['mirror_model_args'] is None:
        self.gmt[mirror] = getattr(ceo,"GMT_"+mirror)( **mirror_args )
    else:
        self.M1 = MDL.Mirror(**mirror_args['mirror_model_args'])
        (self.outer_xy,datatri) = MDL.FEM.nodes_coordinates(1)
        (self.center_xy,datatri) = MDL.FEM.nodes_coordinates(7)
        M1_S = self.M1.deltaSplitS
        S = ceo.mapping.cat(ceo.Mapping(self.outer_xy,np.hstack(M1_S[:-1]))(201,8.5),
                           ceo.Mapping(self.center_xy,M1_S[-1])(201,8.5),list(range(7)))
        self.gmt[mirror] = ceo.GMT_M1(mirror_modes=u'truptiBM',N_MODE=1,mirror_modes_c
        self.gmt[mirror].modes.a[:] = np.ones((7,1))
        self.gmt[mirror].modes.update()
    return b"GMT"
```

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

```
14  <SOpticalPath Update message 14>≡ 19c>
    {
      "class_id": "GMT",
      "method_id": "Update",
      "args":
        {
          "mirror": "M1"|"M2",
          "inputs_args":
            {
              "TxyzRxyz": null,
              "mode_coefs": null
            }
        }
    }
```

```

15a    <SGMT 12b>+= (5) <13b 15b>
def Update(self, mirror=None, inputs_args=None):
    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 mirror=='M1' and self.M1 is not None :
            if key=="TxyzRxyz":
                delta_Txyz = data[:, :3] - self.gmt[mirror].motion_CS.origin[:]
                delta_Rxyz = data[:, 3:] - self.gmt[mirror].motion_CS.euler_angles[:]
                self.gmt[mirror].motion_CS.origin[:] = data[:, :3]
                self.gmt[mirror].motion_CS.euler_angles[:] = data[:, 3:]
                self.gmt[mirror].motion_CS.update()

                self.M1.motion(Txyz=delta_Txyz,Rxyz=delta_Rxyz)
                M1_S = self.M1.deltaSplitS
                S = ceo.mapping.cat(ceo.Mapping(self.outer_xy,np.hstack(M1_S[:-1]))(201,8.5),list
                                ceo.Mapping(self.center_xy,M1_S[-1])(201,8.5),list)
                self.gmt[mirror].modes.modes = S.data['M']
            if key=="mode_coefs":
                self.gmt[mirror].modes.a[:] = np.ones((7,1))
                self.gmt[mirror].modes.update()
                self.M1.reset(force_only=True,stray_weight=False,stray_friction=False)
                self.M1.bmCoefs = data
                M1_S = self.M1.deltaSplitS
                S = ceo.mapping.cat(ceo.Mapping(self.outer_xy,np.hstack(M1_S[:-1])),met
                                ceo.Mapping(self.center_xy,M1_S[-1]),method)
                self.gmt[mirror].modes.modes = S.data['M']
        else:
            print('NOT HERE')
            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 12b>+= (5) <15a 16a>
def InitializeConditions(self, args=None):
    pass

```

Outputs

```
16a  <SGMT 12b>+= (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.

```
16b  <SAtmosphere 16b>= (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*.

17a $\langle SOpticalPath \text{ 17a} \rangle \equiv$ (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, 31a, and 59.

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

17b $\langle SOpticalPath \text{ Start message 17b} \rangle \equiv$
{
 "class_id": "OP",
 "method_id": "Start",
 "args":
 {
 "source_args": { ... } ,
 "sensor_class": null|"Imaging"|"ShackHartmann",
 "sensor_args": null|{ ... },
 "calibration_source": null|{ ... },...
 "miscellaneous_args": null|{...}
 }
}

18 $\langle S_{\text{OpticalPath 17a}} \rangle + \equiv$ (5) $\langle 17a \ 19b \rangle$

```

def Start(self,source_args=None, sensor_class=None, sensor_args=None,
          calibration_source_args=None, miscellaneous_args=None):
    print("\n@($OpticalPath: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 17a and sensor 17a.

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 17a>+≡ (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 14>+≡ <14>
      {
        "class_id": "OP",
        "method_id": "Update",
        "args":
          {
            "inputs": null
          }
      }

19d  <SOpticalPath 17a>+≡ (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 17a 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 $\langle S_{OpticalPath} 17a \rangle + \equiv$ (5) $\langle 19d \ 21 \rangle$

```

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 17a.

and the dictionary implementation is

```
21  <SOpticalPath 17a>+≡ (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=="tip tilt":
            buf = self.src.wavefront.gradientAverage(1, self.src.rays.L)
            buf *= ceo.constants.RAD2ARCSEC
            return buf.tolist()
        elif key=="segment_tip tilt":
            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.ee80(from_ghost=True))
            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 17a 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 $\langle S_{\text{OpticalPath InitializeConditions message 22}} \rangle + \equiv$ $\langle 22 \ 24 \rangle$

```

{
  "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 $\langle S_{\text{OpticalPath}} \text{ InitializeConditions message } 22 \rangle + \equiv$ $\triangleleft 23$

```

{
  "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 17a>+≡ (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 17a 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 , $M = VAU^T$ 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 range(7)]
                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) for k in range(7) )
            # 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 opposite 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 range(self.nSeg) ])

```

```

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

```

4.4 The Sensor abstract class

29a \langle *Sensor abstract class 29a* $\rangle \equiv$

```

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 \langle *dos.yaml 29b* $\rangle \equiv$

\langle *dos simulation section 29c* \rangle

\langle *dos drivers section 32b* \rangle

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

29c \langle *dos simulation section 29c* $\rangle \equiv$ (29b)

```

simulation:
    sampling frequency: 100 # [Hertz]
    duration: 1 # [seconds]
    sinceo 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)
      import os
      import yaml
      import logging
      import threading

30b  <dos.py 30b>≡ 34a▷
      <dos imports 30a>

      logging.basicConfig(level=logging.DEBUG)

      class DOS:
          def __init__(self,path_to_config_dir):
              threading.Thread.__init__(self)
              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.N_SAMPLE = int(self.cfg['simulation']['sampling frequency']*
                                  self.cfg['simulation']['duration'])
              <check parameter file existence 30c>
              <starting the drivers 31b>
              <initializing the drivers 31c>
              <running the loop 31d>
              <terminating 32a>

              <stepping through 31e>

```

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

```

30c  <check parameter file existence 30c>≡ (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,**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.

```
31a  <device name.yaml 31a>≡
      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 17a.

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

```
31b  <starting the drivers 31b>≡ (30b)
      self.start = map(lambda x: x.start(), self.drivers.values())
      and init methods:
```

```
31c  <initializing the drivers 31c>≡ (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.

```
31d  <running the loop 31d>≡ (30b)
      self.step = self.stepping()
```

```
31e  <stepping through 31e>≡ (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.

32a $\langle \text{terminating 32a} \rangle \equiv$ (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.

32b $\langle \text{dos drivers section 32b} \rangle \equiv$ (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.

33a $\langle \text{Driver methods 33a} \rangle \equiv$ (34a)

```
def start(self):
    self.logger.debug('Starting!')
def init(self):
    self.logger.debug('Initializing!')
def update(self,step):
    if step>self.delay and step%self.sampling_rate==0:
        self.logger.debug('Updating!')
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)
def terminate(self):
    self.logger.debug('Terminating!')
```

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

33b $\langle \text{IO 33b} \rangle \equiv$ (34a)

```
class IO:
    def __init__(self,size=0,**kwargs):
        self.size = size
```

33c $\langle \text{Inputs 33c} \rangle \equiv$ (34a)

```
class Input(IO):
    def __init__(self,size=0,origin=None):
        IO.__init__(self,size)
        self.origin = origin
```

and `Outputs` classes.

33d $\langle \text{Outputs 33d} \rangle \equiv$ (34a)

```
class Output(IO):
    def __init__(self,**kwargs):
        IO.__init__(self,**kwargs)
        self.sampling_rate = kwargs['sampling rate']
        self.destination = kwargs['destination']
```

```

34a  <dos.py 30b>+=                                     <30b>
      <IO 33b>
      <Inputs 33c>
      <Outputs 33d>
      class Driver:
          def __init__(self,tag,**kwargs):
              self.logger = logging.getLogger(tag)
              self.delay    = kwargs['delay']
              self.sampling_rate = kwargs['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(**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(**v)

      <Driver methods 33a>

```

6 The python client

The simulation

```

34b  <simceochient.py 34b>≡                             35a>
      import zmq
      import yaml
      import os
      import pickle
      import zlib

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

```

```

35a  <simceoclient.py 34b>+=                                     <34b 35b>
      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'])

35b  <simceoclient.py 34b>+=                                     <35a 35c>
      class broker:

          def __init__(self,IP):
              with open(os.path.join(SIMCEOPATH,'etc','simceo.yaml')) as f:
                  cfg = yaml.load(f)
              self.context = zmq.Context()
              print("@(broker)> 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):
              print('@(broker)> 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)

35c  <simceoclient.py 34b>+=                                     <35b>
      if __name__ == "__main__":
          #agent = broker()
          sim = SIM()

```

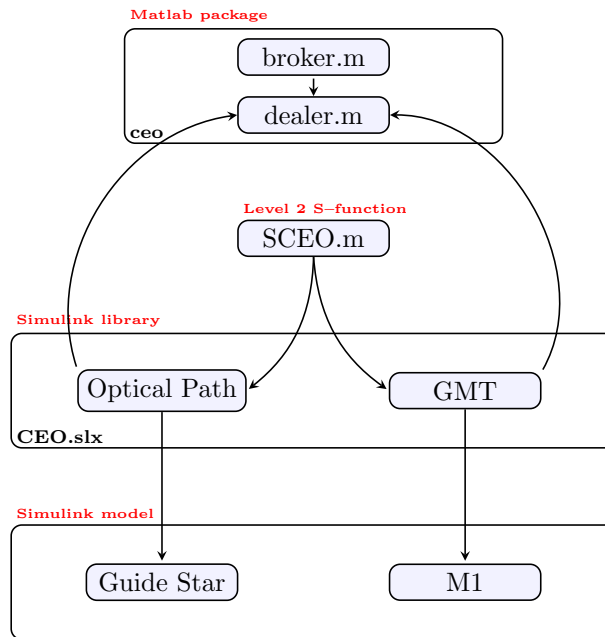


Figure 3: SIMCEO Matlab client flowchart.

7 The ceo Matlab package

7.1 The broker class

```

36  <broker.m 36>≡
    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 38a>

        <release ressources 44a>

        <launch AWS AMI 39a>

        <start AWS instance 42a>

    end

    methods(Static)

        <instanciation and retrieval 44b>

        <request and reply 45a>
g    <reset ZMQ socket 45b>

        <time spent 46>

    end

end

```

Uses etc 38a, instance_end_state 38b, instance_id 38a, public_ip 43a, and zmqReset 38a.

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

```

38a  <broker client 38a>≡ (36)
      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 38b>
      end

```

Defines:

- awspath, used in chunks 40 and 41.
- etc, used in chunks 36, 39–41, and 64.
- instance_id, used in chunks 36 and 38–44.
- self.elapsedTime, used in chunks 45a and 46.
- zmqReset, used in chunks 36 and 45b.

Uses public_ip 43a.

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

```

38b  <broker client: AWS instance launch 38b>≡ (38a)
      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
      end

```

Defines:

- instance_end_state, used in chunks 36, 41b, and 44a.

Uses instance_id 38a, public_ip 43a, run_instance 39b, and start_instance 42a.

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.

```
39a  <launch AWS AMI 39a>≡ (36)
      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 43a>
      end
```

Uses `etc 38a`, `instance_id 38a`, and `run_instance 39b`.

```
39b  <launch AWS AMI (old) 39b>≡
      function run_instance(self)
          <launching an instance 40a>
          <waiting for initialization 40b>
          <branding instance 40c>
          <setting up cloudwatch 41a>
          <getting the public IP 43a>
      end
```

Defines:

`run_instance`, used in chunks `38b` and `39a`.

The sequence of operations is:

1. launching the instance,

```
40a  <launching an instance 40a>≡ (39b)
      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 38a, etc 38a, and instance_id 38a.
```

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

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

```
40b  <waiting for initialization 40b>≡ (39b)
      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 38a and instance_id 38a.
```

4. setting up the instance name

```
40c  <branding instance 40c>≡ (39b)
      [~,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 38a and instance_id 38a.
```


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

```
41a  <setting up cloudwatch 41a>≡ (39b)
      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 38a, etc 38a, and instance_id 38a.
```

6. getting the public IP of the instance (See page 43).

7.1.2 terminate_instance

```
41b  <terminate AWS instance 41b>≡
      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` 38a, `instance_end_state` 38b, and `instance_id` 38a.

7.1.3 start_instance

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

42a $\langle \textit{start AWS instance 42a} \rangle \equiv$ (36)
 function `start_instance`(self)
 $\langle \textit{starting an instance 42b} \rangle$
 $\langle \textit{getting the public IP 43a} \rangle$
 end

Defines:

`start_instance`, used in chunk 38b.

The sequence of operations is:

1. starting the instance:

42b $\langle \textit{starting an instance 42b} \rangle \equiv$ (42a)
 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` 38a.

2. getting the public IP of the instance.

```
43a  <getting the public IP 43a>≡ (39 42a)
      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 36, 38, and 43b.
Uses instance_id 38a.

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

```
43b  <broker client: setup ZMQ connection 43b>≡ (45b)
      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 43a.

The allocated ZeroMQ ressources are released with:

```
44a <release ressources 44a>≡ (36)
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` 38b and `instance_id` 38a.

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

```
44b <instanciation and retrieval 44b>≡ (36)
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 45 and 46.

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

```

45a  <request and reply 45a>≡ (36)
      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 52–54.

Uses *getBroker* 44b and *self.elapsedTime* 38a.

resetZMQ resets the ZeroMQ socket

```

45b  <reset ZMQ socket 45b>≡ (36)
      function resetZMQ()
          self = ceo.broker.getBroker();
          if self.zmqReset
              zmq.core.close(self.socket);
              <broker client: setup ZMQ connection 43b>
          end
          self.zmqReset = false;
      end
      function setZmqResetFlag(val)
          self = ceo.broker.getBroker();
          self.zmqReset = val;
      end

```

Uses *getBroker* 44b and *zmqReset* 38a.

Time spent communicating:

```
46  <time spent 46>≡ (36)
    function timeSpent()
        self = ceo.broker.getBroker();
        fprintf('@(broker)> Time spent communicating with the server: %.3fs\n',...
                self.elapsedTime)
        self.elapsedTime = 0;
    end
```

Uses `getBroker` 44b and `self.elapsedTime` 38a.

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.

```
47  <dealer.m 47>≡
    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 48>

        end

        methods (Access=private)
```

<dealer private methods 52>

```
end
end
```

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

```
48 <dealer public methods 48>≡ (47) 49▷
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:

```
49  <dealer public methods 48>+≡ (47) <48 50a>
    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
end
```

7.2.1 Public methods

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

```
50a <dealer public methods 48>+≡ (47) <49 50b>
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 56.

The names of the output ports are set with:

```
50b <dealer public methods 48>+≡ (47) <50a 51a>
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 56.

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

```
51a  <dealer public methods 48>+≡ (47) <50b 51b>
      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 56-58.

Uses *deal_init* 52, *deal_inputs* 53, *deal_outputs* 54, *deal_start* 52, and *deal_terminate* 52.

The messages are concatenated into a single json file with:

```
51b  <dealer public methods 48>+≡ (47) <51a
      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 52.

7.2.2 Private methods

```
52  <dealer private methods 52>≡ (47) 53>
    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 51a.
    deal_start, used in chunk 51a.
    deal_terminate, used in chunk 51a.
Uses dump 51b and sendrecv 45a.
```

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

```
53  <dealer private methods 52>+≡ (47) <52 54>
    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 51a.

Uses `sendrecv` 45a.

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

```

54  <dealer private methods 52>+≡ (47) <53
    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
end

```

Defines:

`deal_outputs`, used in chunk 51a.

Uses `sendrecv` 45a.

7.3 The loadprm function

```
55a <liftprm.m 55a>≡
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 extensions are .ubj or .json')
    end
else
    error('simceo:loadprm:type_error','Input must be either a structure or a filename!')
end
```

7.4 The SCEO S-function

```
55b <SCEO.m 55b>≡
function SCEO(block)

setup(block);

<SCEO setup 56>

<SCEO Start 57a>

<SCEO Outputs 57b>

<SCEO Terminate 58>
```

7.4.1 setup

```
56  <SCEO setup 56>= (55b)
    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 51a, IO_setup 50a, and output_names 50b.

7.4.2 Start

57a $\langle SCEO \text{ Start } 57a \rangle \equiv$ (55b)

```

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 51a.

7.4.3 Outputs

57b $\langle SCEO \text{ Outputs } 57b \rangle \equiv$ (55b)

```

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 51a.

7.4.4 Terminate

58 $\langle SCEO \text{ Terminate } 58 \rangle \equiv$ (55b)

```
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 51a.

7.5 The block masks

7.5.1 Optical Path

```
59  <OpticalPath.md 59>≡
    # 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
-----	-----	-----	-----	-----	-----	-----	-----
λ [μm]	0.550	0.640	0.790	1.215	1.654	2.179	2.157
$\Delta\lambda$ [μm]	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 intensity profile.
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 the configuration file.

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

$$\frac{\lambda}{d} \left(\frac{b}{a} \right)$$

where both a and b are integers.

b is set by 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 the 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: $[R_{xy}^1, R_{xy}^2, R_{xy}^3, R_{xy}^4]$
 For M1_RTBM, the output vector is: $[R_{xyz}, T_{xyz}, BM]$ with $(R_{xyz} \equiv X, T_{xyz} \equiv Y)$

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

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
 Otherwise M1 and M2 mirrors are controlled in the same way that non segmented mirrors.

Uses sensor [17a](#).

7.5.2 GMT Mirror

63 $\langle GMTMirror.md \ 63 \rangle \equiv$
 # 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

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.

```
64 <CEO.sh 64>≡
    #!/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` [38a](#).

9 Index

awspath: [38a](#), [40a](#), [40b](#), [40c](#), [41a](#), [41b](#)
deal: [51a](#), [56](#), [57a](#), [57b](#), [58](#)
deal_init: [51a](#), [52](#)
deal_inputs: [51a](#), [53](#)
deal_outputs: [51a](#), [54](#)
deal_start: [51a](#), [52](#)
deal_terminate: [51a](#), [52](#)
dump: [51b](#), [52](#)
etc: [36](#), [38a](#), [39a](#), [40a](#), [41a](#), [64](#)
exposure_start: [18](#)
exposure_time: [18](#)
getBroker: [44b](#), [45a](#), [45b](#), [46](#)
idx: [17a](#), [18](#)
instance_end_state: [36](#), [38b](#), [41b](#), [44a](#)
instance_id: [36](#), [38a](#), [38b](#), [39a](#), [40a](#), [40b](#), [40c](#), [41a](#), [41b](#), [42b](#), [43a](#), [44a](#)
IO_setup: [50a](#), [56](#)
output_names: [50b](#), [56](#)
propagateThroughAtm: [18](#)
public_ip: [36](#), [38a](#), [38b](#), [43a](#), [43b](#)
run_instance: [38b](#), [39a](#), [39b](#)
self.elapsedTime: [38a](#), [45a](#), [46](#)
sendrecv: [45a](#), [52](#), [53](#), [54](#)
sensor: [17a](#), [18](#), [19d](#), [20](#), [21](#), [25](#), [31a](#), [59](#)
src: [18](#), [19d](#), [21](#), [25](#)
start_instance: [38b](#), [42a](#)
terminate_instance: [41b](#)
zmqReset: [36](#), [38a](#), [45b](#)

10 List of code chunks

<branding instance 40c>
<broker 8>
<broker client 38a>
<broker client: AWS instance launch 38b>
<broker client: setup ZMQ connection 43b>
<broker get item 11a>
<broker run 9a>
<broker run details 9b>
<broker.m 36>
<CalibrationMatrix 27>
<CEO.sh 64>
<check parameter file existence 30c>
<dealer private methods 52>

- ⟨dealer public methods 48⟩
- ⟨dealer.m 47⟩
- ⟨device name.yaml 31a⟩
- ⟨dos drivers section 32b⟩
- ⟨dos imports 30a⟩
- ⟨dos simulation section 29c⟩
- ⟨dos.py 30b⟩
- ⟨dos.yaml 29b⟩
- ⟨Driver methods 33a⟩
- ⟨getting the public IP 43a⟩
- ⟨GMTMirror.md 63⟩
- ⟨initializing the drivers 31c⟩
- ⟨Inputs 33c⟩
- ⟨instanciation and retrieval 44b⟩
- ⟨IO 33b⟩
- ⟨launch AWS AMI 39a⟩
- ⟨launch AWS AMI (old) 39b⟩
- ⟨launching an instance 40a⟩
- ⟨liftprm.m 55a⟩
- ⟨OpticalPath.md 59⟩
- ⟨Outputs 33d⟩
- ⟨release ressources 44a⟩
- ⟨request and reply 45a⟩
- ⟨reset ZMQ socket 45b⟩
- ⟨running the loop 31d⟩
- ⟨S-function 12a⟩
- ⟨SAtmosphere 16b⟩
- ⟨SCEO Outputs 57b⟩
- ⟨SCEO setup 56⟩
- ⟨SCEO Start 57a⟩
- ⟨SCEO Terminate 58⟩
- ⟨SCEO.m 55b⟩
- ⟨Sensor abstract class 29a⟩
- ⟨setting up cloudwatch 41a⟩
- ⟨SGMT 12b⟩
- ⟨SGMT Start message 13a⟩
- ⟨simceo.py 5⟩
- ⟨simceoclient.py 34b⟩
- ⟨SOpticalPath 17a⟩
- ⟨SOpticalPath InitializeConditions message 22⟩
- ⟨SOpticalPath Outputs message 19e⟩
- ⟨SOpticalPath Start message 17b⟩
- ⟨SOpticalPath Terminate message 19a⟩
- ⟨SOpticalPath Update message 14⟩
- ⟨start AWS instance 42a⟩
- ⟨starting an instance 42b⟩

⟨starting the drivers 31b⟩
⟨stepping through 31e⟩
⟨terminate AWS instance 41b⟩
⟨terminating 32a⟩
⟨time spent 46⟩
⟨waiting for initialization 40b⟩