

SIMCEO

Simulink Client

CEO Server

R. Conan

GMTO Corporation

August 2, 2018

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 | 11 |
| 4.2.1 | The SGMT class | 11 |
| | Start | 12 |
| | Update | 12 |
| | InitializeConditions | 13 |
| | Outputs | 13 |
| 4.2.2 | The SAtmosphere class | 14 |
| 4.2.3 | The SOpticalPath class | 14 |
| | Start | 15 |
| | Terminate | 17 |
| | Update | 17 |
| | Outputs | 17 |
| | InitializeConditions | 20 |
| 4.3 | The CalibrationMatrix class | 25 |
| 4.4 | The Sensor abstract class | 27 |
| 5 | The ceo Matlab package | 28 |
| 5.1 | The broker class | 28 |
| 5.1.1 | run_instance | 31 |
| 5.1.2 | terminate_instance | 33 |
| 5.1.3 | start_instance | 34 |
| 5.2 | The dealer class | 39 |
| 5.2.1 | Public methods | 42 |
| 5.2.2 | Private methods | 44 |
| 5.3 | The loadprm function | 47 |
| 5.4 | The SCEO S-function | 47 |
| 5.4.1 | setup | 48 |
| 5.4.2 | Start | 49 |
| 5.4.3 | Outputs | 49 |
| 5.4.4 | Terminate | 50 |
| 5.5 | The block masks | 51 |
| 5.5.1 | Optical Path | 51 |
| 5.5.2 | GMT Mirror | 55 |

| | | |
|----------|----------------------------|-----------|
| 6 | The CEO server | 56 |
| 7 | Index | 58 |
| 8 | List of code chunks | 58 |

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/>

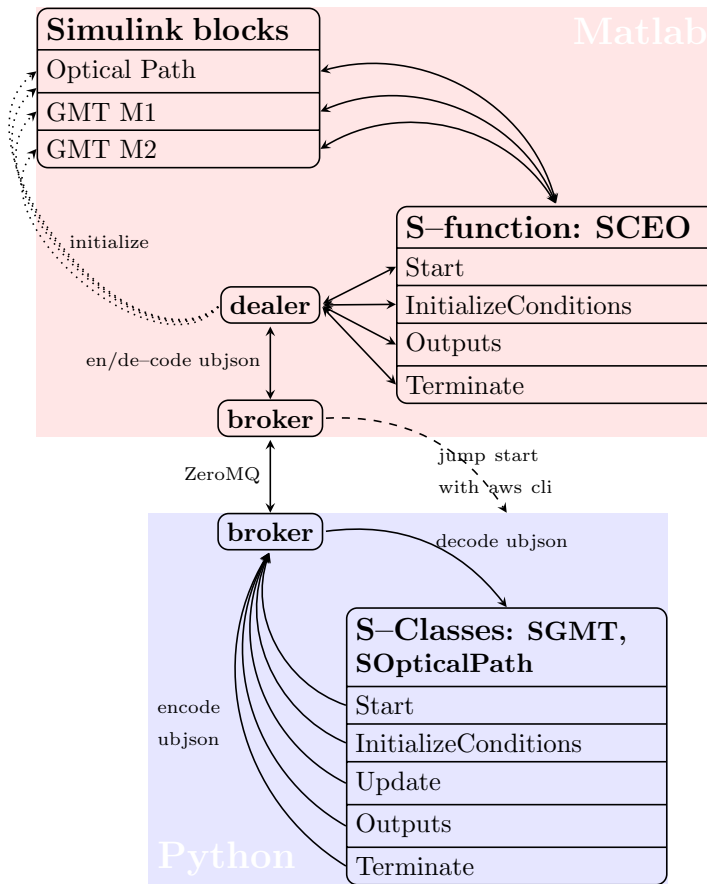


Figure 1: SIMCEO flowchart.

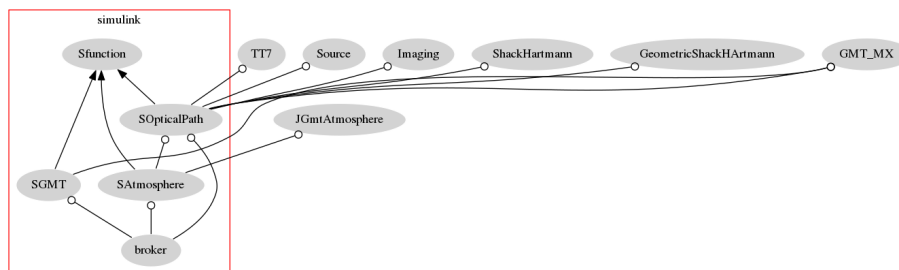


Figure 2: The classes in the simulink python module and their relations with the CEO classes.

```

import shelve
import traceback
import scipy.linalg as LA

SIMCEOPATH = os.path.abspath(os.path.dirname(__file__))
currentTime = 0.0;

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 25>

<S-function 11a>

<SGMT 11b>
<SAtmosphere 14a>
<SOpticalPath 14b>

<broker 8a>

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.

```
8a  <broker 8a>≡ (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)

            self.ops = []
            self.n_op = 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()

        <broker get item 10a>

        <broker run 8b>
        The run method
8b  <broker run 8b>≡ (8a)
        def run(self):

            while True:

                <broker run details 9a>
```


waits for a request from a Simulink S-function:

```
9a  <broker run details 9a>≡ (8b) 9b>
    #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:

```
9b  <broker run details 9a>+≡ (8b) <9a 10b>
    global currentTime
    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

10a $\langle broker\ get\ item\ 10a \rangle \equiv$ (8a)

```
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*.

10b $\langle broker\ run\ details\ 9a \rangle + \equiv$ (8b) $\langle 9b\ 10c \rangle$

```
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:

10c $\langle broker\ run\ details\ 9a \rangle + \equiv$ (8b) $\langle 10b$

```
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:

```
11a  <S-function 11a>≡ (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.

```
11b  <SGMT 11b>≡ (5) 12b>
      class SGMT(Sfunction):

          def __init__(self, ops, satm):
              self.gmt = ceo.GMT_MX()

          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

```
12a  <SGMT Start message 12a>≡
      {
        "class_id": "GMT",
        "method_id": "Start",
        "args":
          {
            "mirror": "M1"|"M2",
            "mirror_args":
              {
                "mirror_modes": u"bending modes"|u"zernike",
                "N_MODE": 162,
                "radial_order": ...
              }
          }
      }

12b  <SGMT 11b>+≡ (5) <11b 13a>
      def Start(self,mirror=None,mirror_args=None):
        print("\n@<SGMT:Start>")
        self.gmt[mirror] = getattr(ceo,"GMT_"+mirror)( **mirror_args )
        return b"GMT"
```

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

```
12c  <SOpticalPath Update message 12c>≡ 17c>
      {
        "class_id": "GMT",
        "method_id": "Update",
        "args":
          {
            "mirror": "M1"|"M2",
            "inputs_args":
              {
                "TxyzRxyz": null,
                "mode_coefs": null
              }
          }
      }
```

13a $\langle SGMT\ 11b \rangle + \equiv$ (5) $\langle 12b\ 13b \rangle$

```

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 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

13b $\langle SGMT\ 11b \rangle + \equiv$ (5) $\langle 13a\ 13c \rangle$

```

def InitializeConditions(self, args=None):
    pass

```

Outputs

13c $\langle SGMT\ 11b \rangle + \equiv$ (5) $\langle 13b \rangle$

```

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.

```
14a  <SATmosphere 14a>≡ (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*.

```
14b  <SOpticalPath 14b>≡ (5) 16>
      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 16.
sensor, used in chunks 16–19, 23, and 51.

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

```
15  <SOpticalPath Start message 15>≡
    {
      "class_id": "OP",
      "method_id": "Start",
      "args":
        {
          "source_args": { ... } ,
          "sensor_class": null|"Imaging"|"ShackHartmann",
          "sensor_args": null|{ ... },
          "calibration_source": null|{ ... },...
          "miscellaneous_args": null|{...}
        }
    }
```

16 $\langle S_{\text{OpticalPath 14b}} \rangle + \equiv$ (5) $\langle 14b \ 17b \rangle$

```

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)

    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.src,0)
        #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 17d, 19, and 23.

Uses idx 14b and sensor 14b.

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

```
17a <SOpticalPath Terminate message 17a>≡
    {
      "class_id": "OP",
      "method_id": "Terminate",
      "args":
        {
          "args": null
        }
    }

17b <SOpticalPath 14b>+≡ (5) <16 17d>
    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

```
17c <SOpticalPath Update message 12c>+≡ <12c>
    {
      "class_id": "OP",
      "method_id": "Update",
      "args":
        {
          "inputs": null
        }
    }

17d <SOpticalPath 14b>+≡ (5) <17b 18>
    def Update(self, inputs=None):
        +self.src
        #self.src.reset()
        #self.gmt.propagate(self.src)
        #self.sensor.propagate(self.src)
```

Uses sensor 14b and src 16.

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

```
17e <SOpticalPath Outputs message 17e>≡
    {
      "class_id": "OP",
      "method_id": "Outputs",
      "args":
        {
          "outputs": ["wfe_rms"|"segment_wfe_rms"|"piston"|"segment_piston"|"ee80"]
        }
    }
```

```

18  <SOpticalPath 14b>+≡ (5) <17d 19>
    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 14b.

and the dictionary implementation is

```
19  <SOpticalPath 14b>+≡ (5) <18 23>
    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 14b and src 16.

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

```
20  <SOpticalPath InitializeConditions message 20>≡                                     21>
    {
      "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
        }
    }
```

21 $\langle S_{\text{OpticalPath InitializeConditions message 20}} \rangle + \equiv$ $\langle 20 \ 22 \rangle$

```

{
  "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
  }
}

```

22 $\langle S_{\text{OpticalPath}} \text{ InitializeConditions message } 20 \rangle + \equiv$ 21

```

{
  "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
  }
}

```

```

23      <SOpticalPath 14b>+≡ (5) <19
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 14b and src 16.

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

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

```

25  <CalibrationMatrix 25>≡ (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)
                # The lenslet flux filter is applied to the lenslet segment filter:
                Q = [ np.logical_and(X,flux_filter2) for X in Qxy ]
                # A filter made of the lenslet used more than once is created:
                Q3 = np.dstack(Q).reshape(flux_filter2.shape + (self.nSeg,))
                Q3clps = np.sum(Q3,axis=2)
                Q3clps = Q3clps>1
                # The oposite filter is applied to the lenslet segment filter leading to 7 val

```

```

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

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

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

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

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

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

```

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

4.4 The Sensor abstract class

```
27  <Sensor abstract class 27>≡
    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
```

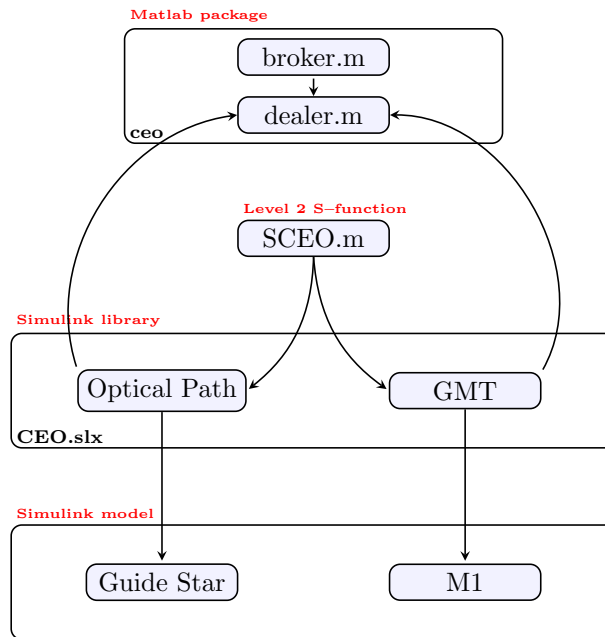


Figure 3: SIMCEO Matlab client flowchart.

5 The ceo Matlab package

5.1 The broker class

```

28  <broker.m 28>≡
    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 30a>

        <release ressources 36a>

        <launch AWS AMI 31a>

        <start AWS instance 34a>

    end

    methods(Static)

        <instanciation and retrieval 36b>

        <request and reply 37a>
g    <reset ZMQ socket 37b>

        <time spent 38>

    end

end

Uses etc 30a, instance_end_state 30b, instance_id 30a, public_ip 35a, and zmqReset 30a.

```

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

```

30a  <broker client 30a>≡ (28)
      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 30b>
      end

```

Defines:

awspath, used in chunks 32 and 33.
 etc, used in chunks 28, 31–33, and 56.
 instance_id, used in chunks 28 and 30–36.
 self.elapsedTime, used in chunks 37a and 38.
 zmqReset, used in chunks 28 and 37b.

Uses public_ip 35a.

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

```

30b  <broker client: AWS instance launch 30b>≡ (30a)
      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 28, 33b, and 36a.

Uses instance_id 30a, public_ip 35a, run_instance 31b, and start_instance 34a.

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.

```
31a  <launch AWS AMI 31a>≡ (28)
      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 35a>
      end
```

Uses `etc 30a`, `instance_id 30a`, and `run_instance 31b`.

```
31b  <launch AWS AMI (old) 31b>≡
      function run_instance(self)
          <launching an instance 32a>
          <waiting for initialization 32b>
          <branding instance 32c>
          <setting up cloudwatch 33a>
          <getting the public IP 35a>
      end
```

Defines:

`run_instance`, used in chunks `30b` and `31a`.

The sequence of operations is:

1. launching the instance,

```
32a  <launching an instance 32a>≡ (31b)
      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 30a, etc 30a, and instance_id 30a.
```

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

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

```
32b  <waiting for initialization 32b>≡ (31b)
      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 30a and instance_id 30a.
```

4. setting up the instance name

```
32c  <branding instance 32c>≡ (31b)
      [~,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 30a and instance_id 30a.
```


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

```
33a  <setting up cloudwatch 33a>≡ (31b)
      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 30a, etc 30a, and instance_id 30a.
```

6. getting the public IP of the instance (See page 35).

5.1.2 terminate_instance

```
33b  <terminate AWS instance 33b>≡
      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 30a, instance_end_state 30b, and instance_id 30a.

5.1.3 start_instance

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

34a $\langle \text{start AWS instance 34a} \rangle \equiv$ (28)

```
function start_instance(self)
     $\langle \text{starting an instance 34b} \rangle$ 
     $\langle \text{getting the public IP 35a} \rangle$ 
end
```

Defines:

`start_instance`, used in chunk 30b.

The sequence of operations is:

1. starting the instance:

34b $\langle \text{starting an instance 34b} \rangle \equiv$ (34a)

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

url = sprintf('%s/simceo_aws_server.html?action=start&instance_ID=%s',self.urlbase,self.instance_id)
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` 30a.

2. getting the public IP of the instance.

```
35a  <getting the public IP 35a>≡ (31 34a)
      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 28, 30, and 35b.
Uses instance_id 30a.

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

```
35b  <broker client: setup ZMQ connection 35b>≡ (37b)
      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 35a.

The allocated ZeroMQ resources are released with:

```
36a  <release resources 36a>≡ (28)
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` 30b and `instance_id` 30a.

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

```
36b  <instantiation and retrieval 36b>≡ (28)
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 37 and 38.

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

```

37a  <request and reply 37a>≡ (28)
      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 44–46.

Uses *getBroker* 36b and *self.elapsedTime* 30a.

resetZMQ resets the ZeroMQ socket

```

37b  <reset ZMQ socket 37b>≡ (28)
      function resetZMQ()
          self = ceo.broker.getBroker();
          if self.zmqReset
              zmq.core.close(self.socket);
              <broker client: setup ZMQ connection 35b>
          end
          self.zmqReset = false;
      end
      function setZmqResetFlag(val)
          self = ceo.broker.getBroker();
          self.zmqReset = val;
      end

```

Uses *getBroker* 36b and *zmqReset* 30a.

Time spent communicating:

```
38   $\langle \textit{time spent 38} \rangle \equiv$  (28)  
    function timeSpent()  
        self = ceo.broker.getBroker();  
        fprintf('@(broker)> Time spent communicating with the server: %.3fs\n',...  
                self.elapsedTime)  
        self.elapsedTime = 0;  
    end
```

Uses `getBroker 36b` and `self.elapsedTime 30a`.

5.2 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.

```
39  <dealer.m 39>≡
    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 40>

        end

        methods (Access=private)
```

⟨dealer private methods 44⟩

```
end
end
```

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

```
40  ⟨dealer public methods 40⟩≡ (39) 41▷
    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:

```
41  <dealer public methods 40>+≡ (39) <40 42a>
    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
```

5.2.1 Public methods

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

```
42a <dealer public methods 40>+≡ (39) <41 42b>
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 48.

The names of the output ports are set with:

```
42b <dealer public methods 40>+≡ (39) <42a 43a>
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 48.

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

```
43a  <dealer public methods 40>+≡ (39) <42b 43b>
      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 48-50.

Uses *deal_init* 44, *deal_inputs* 45, *deal_outputs* 46, *deal_start* 44, and *deal_terminate* 44.

The messages are concatenated into a single json file with:

```
43b  <dealer public methods 40>+≡ (39) <43a
      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 44.

5.2.2 Private methods

```
44  <dealer private methods 44>≡ (39) 45▷
    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 43a.
- deal_start, used in chunk 43a.
- deal_terminate, used in chunk 43a.

Uses dump 43b and sendrecv 37a.

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

```

45  <dealer private methods 44>+≡ (39) <44 46>
    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
end

```

Defines:

`deal_inputs`, used in chunk 43a.

Uses `sendrecv` 37a.

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

```

46  <dealer private methods 44>+≡ (39) <45
    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 43a.

Uses `sendrecv` 37a.

5.3 The loadprm function

```
47a <liftprm.m 47a>≡
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
```

5.4 The SCEO S-function

```
47b <SCEO.m 47b>≡
function SCEO(block)

setup(block);

<SCEO setup 48>

<SCEO Start 49a>

<SCEO Outputs 49b>

<SCEO Terminate 50>
```

5.4.1 setup

```
48   $\langle SCEO\ setup\ 48 \rangle \equiv$  (47b)
    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 43a, IO_setup 42a, and output_names 42b.

5.4.2 Start

49a $\langle SCEO \text{ Start } 49a \rangle \equiv$ (47b)

```

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 43a.

5.4.3 Outputs

49b $\langle SCEO \text{ Outputs } 49b \rangle \equiv$ (47b)

```

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 43a.

5.4.4 Terminate

50 $\langle SCEO \text{ Terminate } 50 \rangle \equiv$ (47b)

```
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 43a.

5.5 The block masks

5.5.1 Optical Path

```
51  <OpticalPath.md 51>≡
    # 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.
The FWHM is given in units of pixel before binning.

Propagate through the atmosphere

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

Sample Time

The sampling time of the block outputs.

Sensor Tab

\# of lenslet

The linear size of the lenslet array.

lenslet size

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

camera resolution

The detector resolution of the optical `sensor` in pixel.

Intensity threshold

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

Pixel scale

The angular size of a pixel of the detector in arcsec.

It is given by

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

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 [14b](#).

5.5.2 GMT Mirror

55 $\langle GMTMirror.md \ 55 \rangle \equiv$
 # 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.

```
56 <CEO.sh 56>≡
    #!/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 30a`.

7 Index

awspath: [30a](#), [32a](#), [32b](#), [32c](#), [33a](#), [33b](#)
deal: [43a](#), [48](#), [49a](#), [49b](#), [50](#)
deal_init: [43a](#), [44](#)
deal_inputs: [43a](#), [45](#)
deal_outputs: [43a](#), [46](#)
deal_start: [43a](#), [44](#)
deal_terminate: [43a](#), [44](#)
dump: [43b](#), [44](#)
etc: [28](#), [30a](#), [31a](#), [32a](#), [33a](#), [56](#)
exposure_start: [16](#)
exposure_time: [16](#)
getBroker: [36b](#), [37a](#), [37b](#), [38](#)
idx: [14b](#), [16](#)
instance_end_state: [28](#), [30b](#), [33b](#), [36a](#)
instance_id: [28](#), [30a](#), [30b](#), [31a](#), [32a](#), [32b](#), [32c](#), [33a](#), [33b](#), [34b](#), [35a](#), [36a](#)
IO_setup: [42a](#), [48](#)
output_names: [42b](#), [48](#)
propagateThroughAtm: [16](#)
public_ip: [28](#), [30a](#), [30b](#), [35a](#), [35b](#)
run_instance: [30b](#), [31a](#), [31b](#)
self.elapsedTime: [30a](#), [37a](#), [38](#)
sendrecv: [37a](#), [44](#), [45](#), [46](#)
sensor: [14b](#), [16](#), [17d](#), [18](#), [19](#), [23](#), [51](#)
src: [16](#), [17d](#), [19](#), [23](#)
start_instance: [30b](#), [34a](#)
terminate_instance: [33b](#)
zmqReset: [28](#), [30a](#), [37b](#)

8 List of code chunks

<branding instance 32c>
<broker 8a>
<broker client 30a>
<broker client: AWS instance launch 30b>
<broker client: setup ZMQ connection 35b>
<broker get item 10a>
<broker run 8b>
<broker run details 9a>
<broker.m 28>
<CalibrationMatrix 25>
<CEO.sh 56>
<dealer private methods 44>
<dealer public methods 40>

- ⟨*dealer.m* [39](#)⟩
- ⟨*getting the public IP* [35a](#)⟩
- ⟨*GMTMirror.md* [55](#)⟩
- ⟨*instanciation and retrieval* [36b](#)⟩
- ⟨*launch AWS AMI* [31a](#)⟩
- ⟨*launch AWS AMI (old)* [31b](#)⟩
- ⟨*launching an instance* [32a](#)⟩
- ⟨*liftprm.m* [47a](#)⟩
- ⟨*OpticalPath.md* [51](#)⟩
- ⟨*release ressources* [36a](#)⟩
- ⟨*request and reply* [37a](#)⟩
- ⟨*reset ZMQ socket* [37b](#)⟩
- ⟨*S-function* [11a](#)⟩
- ⟨*SAtmosphere* [14a](#)⟩
- ⟨*SCEO Outputs* [49b](#)⟩
- ⟨*SCEO setup* [48](#)⟩
- ⟨*SCEO Start* [49a](#)⟩
- ⟨*SCEO Terminate* [50](#)⟩
- ⟨*SCEO.m* [47b](#)⟩
- ⟨*Sensor abstract class* [27](#)⟩
- ⟨*setting up cloudwatch* [33a](#)⟩
- ⟨*SGMT* [11b](#)⟩
- ⟨*SGMT Start message* [12a](#)⟩
- ⟨*simceo.py* [5](#)⟩
- ⟨*SOpticalPath* [14b](#)⟩
- ⟨*SOpticalPath InitializeConditions message* [20](#)⟩
- ⟨*SOpticalPath Outputs message* [17e](#)⟩
- ⟨*SOpticalPath Start message* [15](#)⟩
- ⟨*SOpticalPath Terminate message* [17a](#)⟩
- ⟨*SOpticalPath Update message* [12c](#)⟩
- ⟨*start AWS instance* [34a](#)⟩
- ⟨*starting an instance* [34b](#)⟩
- ⟨*terminate AWS instance* [33b](#)⟩
- ⟨*time spent* [38](#)⟩
- ⟨*waiting for initialization* [32b](#)⟩