DAHSI Running Guide

July 18, 2022

This running guide is a replication of the Jupyter Notebook that can be found in the git repo in `Example_Notebook/Lorenz_Walkthrough.ipynb`.

1 Simple running example of DAHSI

This document is for illustration of the method purposes only and so the toy problem chosen contains no hidden variables to be able to go through the code in a few minutes. This notebook will give you the tools and understanding necessary to be able to build your own problem and solve it using DAHSI.

To check an example of hidden variables, go to the `Example_LorenzSynth` folder in the github repo and explore the differences between `File1.txt` and `File2.txt` shown in this tutorial and the ones found in that folder. It is the same problem but y is hidden.

1.1 Mathematical background

The algorithm data assimilation for hidden sparse inference (DAHSI) boils down to minimising the following cost function:

$$A(\mathbf{X},\mathbf{p}) = \frac{1}{N}\sum_{i=1}^{N}\|\mathbf{X}(t_i)-\mathbf{Y}(t_i)\|^2 + \frac{1}{N}\sum_{i=1}^{N-1} R_f\left\{\|\mathbf{X}(t_{i+1})-\mathbf{f}(\mathbf{X}(t_i),\mathbf{p},\hat{\mathbf{F}})\|^2\right\} + \lambda\|\mathbf{p}\|_1. \tag{1}$$

his function is composed of three terms: the experimental error, $A_E(\mathbf{X},\mathbf{Y}) = \frac{1}{N}\sum_{i=1}^{N}\|\mathbf{X}(t_i)-\mathbf{Y}(t_i)\|^2$, the model error term, $A_M(\mathbf{X},\mathbf{p},\hat{\mathbf{F}}) = \frac{1}{N}\sum_{i=1}^{N-1}\left\{\|\mathbf{X}(t_{i+1})-\mathbf{f}(\mathbf{X}(t_i),\mathbf{p},\hat{\mathbf{F}})\|^2\right\}$, and a sparse penalty term $\lambda\|\mathbf{p}\|_1$. Here, $\mathbf{f}(\mathbf{X}(t_i),\mathbf{p},\hat{\mathbf{F}}) = \mathbf{X}(t_{i+1})$ defines the discrete time model dynamics and is obtained by discretizing the governing equations using a Hermite-Simpson collocation.

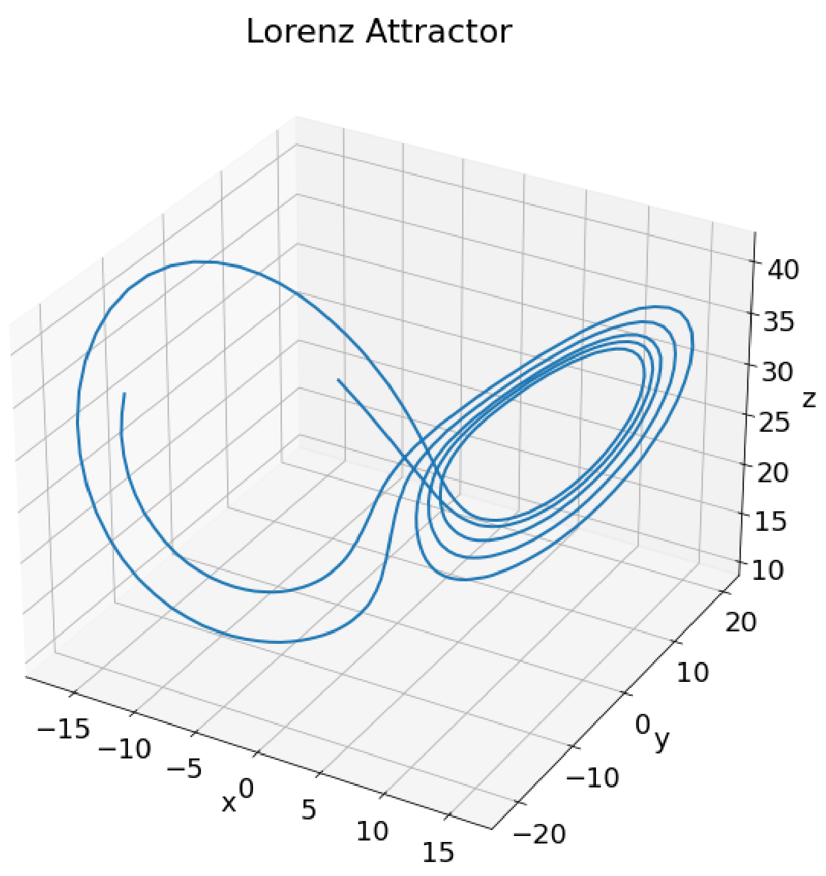
For full details on our method, check out our paper here.

1.2 Generating the data

We will work with the Lorenz system as it is a classical example of chaotic systems.

We numerically simulate the system using Runge-Kutta 4th order with $\Delta t = 0.01$ to obtain the data we will use to show a simple running example. We will consider $N = 501$ time points, and we will add to the data some normally distributed noise with $\omega = 0.01$.

We will save the time-series for each variable in `.dat` files.



2 A quick tour to the problem setup files

First we are going to look into how we define the generic equations, parameters, bounds etc. To do so, we will load `File1.txt` and go over each line.

```
# Number of state variables, number of parameters, number of measured state variables, time step, number of time points.
3,30,3,0.01,501
# State variable names.
x
y
z
# Parameter names.
B101
B102
B103
B104
B105
B106
B107
```

```
B108
B109
B110
B201
B202
B203
B204
B205
B206
B207
B208
B209
B210
B301
B302
B303
B304
B305
B306
B307
B308
B309
B310
# Equations, in the same order as variables.
B101+B102*x+B103*y+B104*z+B105*x*x+B106*x*y+B107*x*z+B108*y*y+B109*y*z+B110*z*z
B201+B202*x+B203*y+B204*z+B205*x*x+B206*x*y+B207*x*z+B208*y*y+B209*y*z+B210*z*z
B301+B302*x+B303*y+B304*z+B305*x*x+B306*x*y+B307*x*z+B308*y*y+B309*y*z+B310*z*z
# Upper and lower bounds for the the state variables search (min,max).
-20,20
-30,30
0,50
# Upper and lower bounds for the parameter search (min,max).
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
```

```
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
-50,50
```

The first line contains the number of state variables (`num_vars`), parameters (`num_params`) and measured state variables (`num_meas`), the time step Δt and the total number of time points (`num_tpoints`), in that order and separated by commas. In the example file above, we can see that our problem has 3 state variables, 30 parameter to estimate and 3 measured state variables. Our data consists of 501 time steps, with a step size of 0.01.

The next lines assign names to our state variables, each name in one different line. The observed variables are written first, and the hidden variables after. In this case we do not consider any hidden variables. In the example file above, the variables are named x, y and z.

The next lines assign names to the parameters we seek to estimate, each name in one different line. In the example provided, the parameter names are $Bk0j$, for $k = 1, 2, 3$, and $j = 1, \dots, 10$.

The next lines define the equations of our problem. One line for each equation. The equations have to be written in the same order as the state variables. For the three variables, we consider a library of monomials of the three variables up to degree two.

We now set the upper and lower bounds for the state variables, separated by a comma. One line for each state variable.

Finally, we also get the upper and lower bounds for the parameters, separated by a comma. One line for each parameter.

Next we will look into `File2.txt`, were we define what our data files are.

```
# Data names.
datax
datay
dataz
# Data file names.extension
datax_Lorenz.dat
datay_Lorenz.dat
dataz_Lorenz.dat
```

This file contains the information about the measured state variables.

In the first lines we give names to the data of each measured state variable. One line for each measured state variable.

The next lines include the file name (and its extension) in which we can find the data of each measured variable. One line per measured variable, and in the same order as in the first part of the file.

In the example file above we have three measured state variables: we call them `datax` and `datay` and `dataz`; we then indicate that the data of our measured state variables can be found in `datax_Lorenz.dat`, `datay_Lorenz` and `dataz_Lorenz` files.

Finally, the variational annealing and model selection tuning parameters are defined in `File3.txt`.

```
# alpha, beta_max.
2.0
30
# For lambda sweep, define lambda_min and lambda_max.
0.000001
68.00
```

The first two lines define values for α and $\beta_{\max}$ for variational annealing.

The next two lines indicate the initial λ and the maximum value it can attain. This λ value controls the sparsity of the models recovered. We should always start with one that includes all possible functions and set a maximum λ for which all the terms drop to zero.

In the example file above we have set $\alpha = 2$ and a maximum β value to 30. Initial λ is set to 10^{-6} and the maximum to 68.

3 Compiling DAHSI

It is **absolutely** necessary to run the following line of code.

```
[6]: !python compile.py
```

`compile.py` runs three crucial scripts that enable DAHSI to run:

- It first calculates the objective function, its Jacobian and Hessian for a general time point and saves them as strings. This is done via `Build_ObjJacHess.py`. The general expression for the objective function can be found in `Obj_Funks.py`;
- Then it runs `WriteStrings.py` which takes the strings generated in the first script and writes them in the "blank template" `OneLoopInC_Blank.pyx`, which will create the functions for the objective function, its Jacobian and Hessian in *cython* format. The output is the file `OneLoopInC.pyx`;
- Finally, it runs the file `setup.py` to create a `build` directory, a C file (`.c`), and a Shared Object file (`.so`). With this, we will be able to import our C-extension functions into our code.

Next, we load the variables needed from `ObjNeed.obj` created in `Build_ObjJacHess.py`. We only need the variables `row_final` and `col_final`, but by the nature of the `pickle` library, we need to load all the objects that were pickled into the file before the ones we need.

[7]:
```
file_ObjJacHess = open('ObjJacHess.obj', 'rb')

ObjFunk_Meas_eval = pickle.load(file_ObjJacHess)
ObjFunk_Model_eval = pickle.load(file_ObjJacHess)
Jacobian_Meas = pickle.load(file_ObjJacHess)
Jacobian_Model = pickle.load(file_ObjJacHess)
Hessian_Meas = pickle.load(file_ObjJacHess)
Hessian_Model = pickle.load(file_ObjJacHess)
row_final = pickle.load(file_ObjJacHess)
col_final = pickle.load(file_ObjJacHess)
nnzh = pickle.load(file_ObjJacHess)
```

We now can import all the modules and functions needed to run our code.

[8]:
```
# Read_Files.py reads the three input text files (File1.txt, File2.txt and File3.
 ↪txt).
from Read_Files import *

# Obj_Funks defines the measured and model part of the action we are minimising.
from Obj_Funks import Meas_Funk
from Obj_Funks import Model_Funk

# We import the cost function, Jacobian and Hessian functions.
import OneLoopInC
from OneLoopInC import eval_f_tricky
from OneLoopInC import eval_grad_f_tricky
from OneLoopInC import eval_h_tricky
```

```
warning: OneLoopInC.pyx:5025:8: Unreachable code
```

Finally we define a class (called DAHSI) that contains the objective function, the Jacobian and the Hessian, and define the constrains of the problem as empty.

[9]:
```
class DAHSI():
    def objective(self, x):
        """Returns the scalar value of the objective given x."""
        return eval_f_tricky(x,Rf)

    def gradient(self, x):
        """Returns the gradient of the objective with respect to x."""
        return np.transpose(eval_grad_f_tricky(x,Rf))

    def constraints(self, x):
        """Returns the constraints."""
        return array([ ], float_)
    def jacobian(self, x):
        """Returns the Jacobian of the constraints with respect to x."""
        return np.array([])

    # Location of the non-zero elements of the Hessian.
    def hessianstructure(self):
        """Returns the row and column indices for non-zero values of the
        Hessian."""

        return (np.array(col_final),np.array(row_final))

    def hessian(self, x, lagrange, obj_factor):
        """Returns the non-zero values of the Hessian."""
        H = eval_h_tricky(x,lagrange,obj_factor,0,Rf)

        row, col = self.hessianstructure()

        return H
```

4 Setting up the initial conditions

[10]:
```
# Choose seed for random number generation.
# We call this IC variable the taskID number.
IC = 0
np.random.seed(IC)
```

Use appropriate initial conditions: for observed state variables, the data provided; for hidden state variables, random; for parameters, set them all to 0.

[11]:
```
# Bounds for both state variables and parameters.
x_L = np.ones((num_total))
x_U = np.ones((num_total))

for i in range(num_vars):
    x_L[i:-num_params:num_vars] = float(Input1[1+2*num_vars+num_params+i].
 ↪split(",")[0])
    x_U[i:-num_params:num_vars] = float(Input1[1+2*num_vars+num_params+i].
 ↪split(",")[1])
for i in range(num_params):
    x_L[-num_params+i] = float(Input1[1+3*num_vars+num_params+i].split(",")[0])
    x_U[-num_params+i] = float(Input1[1+3*num_vars+num_params+i].split(",")[1])

# Initial vector.
x0 = (x_U-x_L)*np.random.rand(num_total)+x_L
for i in range(num_meas):
    for k in range(0,num_vars*num_tpoints,num_vars):
        x0[k+i] = data[int(k/num_vars),i]
for i in range(num_params):
    x0[i-num_params] = 0

x_jp = np.zeros((num_total))
for i in range(num_total):
    x_jp[i] = x0[i]
```

Define the problem using cyipopt (the Python wrapper around Ipopt). We also can adjust some parameters for Ipopt iself.

[12]:
```
nlp = cyipopt.Problem(n = num_total,
                      m = 0,
                      problem_obj = DAHSI(),
                      lb = x_L,
                      ub = x_U,
                      cl = np.array([]),
                      cu = np.array([]))

# Change some options of the Ipopt solver.
nlp.add_option('linear_solver', 'ma97')
nlp.add_option('mu_strategy', 'adaptive')
nlp.add_option('adaptive_mu_globalization', 'never-monotone-mode')
nlp.add_option('bound_relax_factor', float(0))
nlp.add_option('print_level',0)
```

5 Running the λ sweep

The core of the DAHSI algorithm is a nonlinear optimization step using VA, which is randomly initialized. At each VA step, we minimize $A_E + R_f A_M$ over state variable trajectories $\mathbf{X}(t)$ and parameters $\mathbf{p}$ given R_f using IPOPT interfaced here via cyipopt.

Now we start the loop on λ. We start with a very small R_f value and increase it as the VA step. We do this for every λ we want to study.

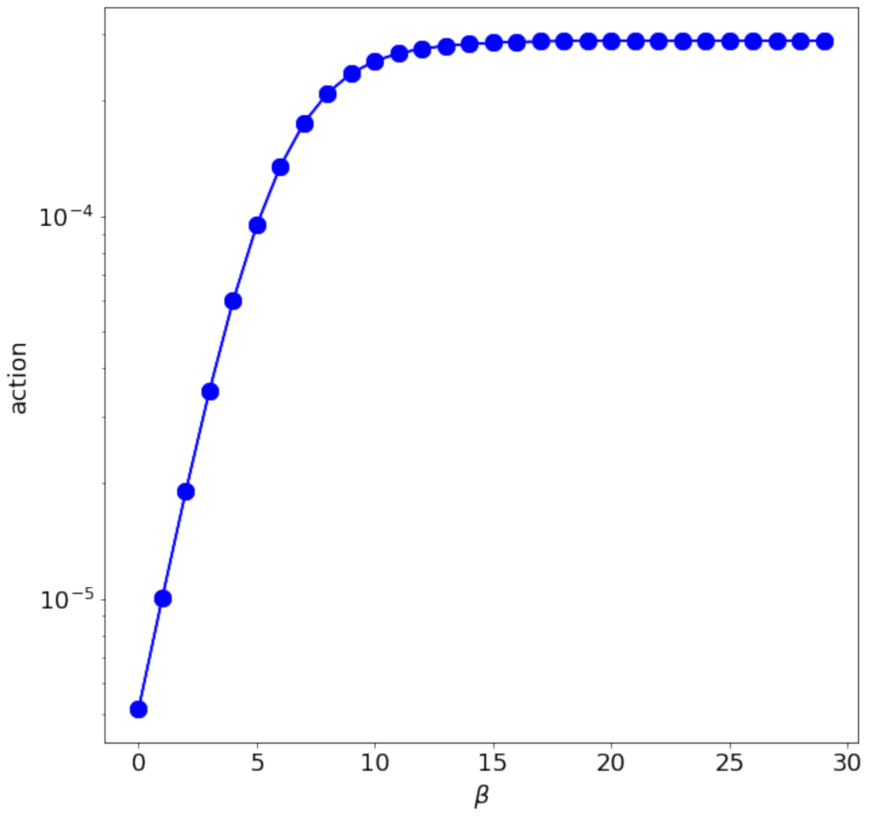
[13]:
```
lambd = lambd_0

file_name = "D%s_M%s_IC%s_LorenzNotebook.dat" % (num_vars, num_meas, IC)
file_results = os.path.join("outputfiles",file_name)
f = open(file_results,"w+")

print(colored('Variatonal annealing for different \lambda.', attrs=['bold']))
iter_count = 1
# Here starts the main loop
while lambd < lambd_max:
    f = open(file_results,"a+")

    Rf0 = 1e-2
    for i in range(num_total):
        x_jp[i] = x0[i]

    print("Iteration #%d: \lambda = %f"%(iter_count,lambd))
    for beta in tqdm_notebook(range(beta_max+1)):
        f = open(file_results,"a+")
        # Make note in results file which \lambda and \beta we are at.
        f.write("%f %f " % (lambd, beta))

        # Controlling how much the model is enforced.
        Rf = Rf0*(alpha**beta)

        # Solve it via IPOPT (solution is x_jn).
        x_jn, info = nlp.solve(x_jp)

        obj = info['obj_val']

        # We hard threshold the parameter part of the solution (the last␣
 ↪num_params elements).
        for i in range(num_params):
            if abs(x_jn[i-num_params]) < lambd:
                x_jn[i-num_params] = 0

        # We set this solution as the initial condition for the next iteration␣
 ↪of IPOPT.
        x_jp = x_jn

        # Write cost function value in file.
        f.write("%e " % obj)

        for k in range(num_params):
            f.write("%f " % x_jp[k-num_params])
        f.write("\n")

        f.close()

    # Increase \lambda value.
    lambd = 2*lambd
    iter_count = iter_count+1
f.close()

num_lambda = iter_count-1
```

6 Basic analysis of the results

We first read the output file generated by the λ sweep. This file is namedd `Di_Mj_ICk_LorenzNotebook.dat`, where `i` is the number of state variables, `j` is the number of measured state variables and `k` is the `taskID` we have chosen.

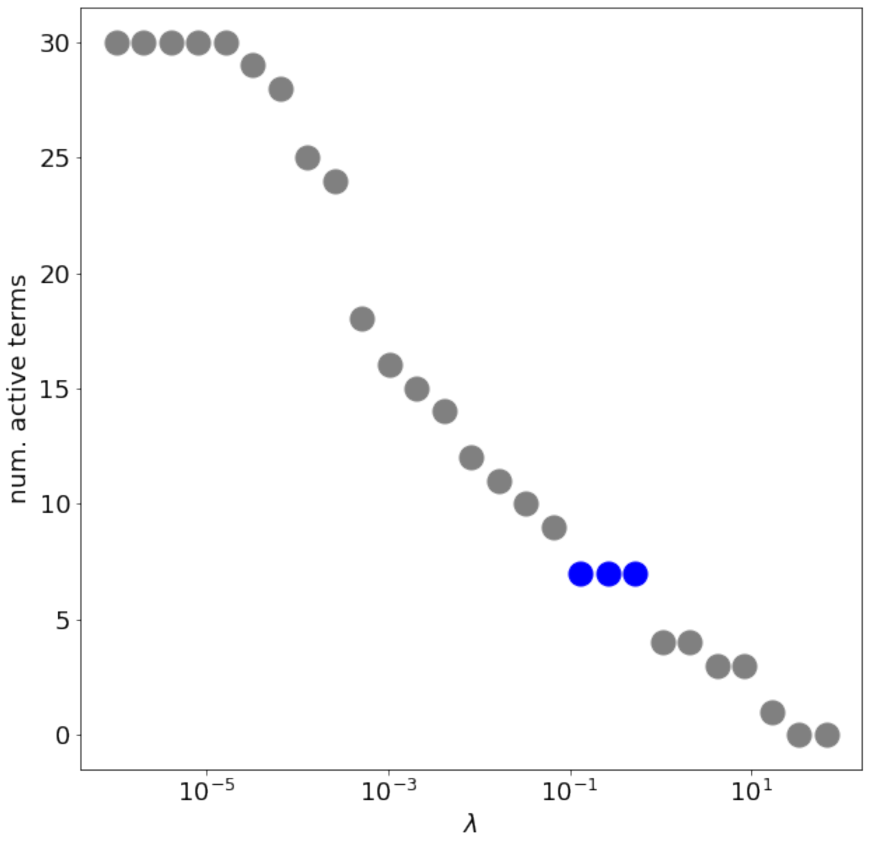
Each line in this file is the solution of the problem given a λ and a β value. For each λ, we will have $\beta_{\max} + 1$ lines. So, if we do a sweep for N_λ different λ values, the file will have $N_\lambda(\beta_{\max} + 1)$ lines.

The action plot shown below shows how the action changes with increased β values. We only show the action paths that correspond to the recovered correct model structure. We can observe that for high β the model is fully enforced (i.e., the action plot plateaus), which is a good indicator that the algorithm has worked.



6.1 Models recovered

The number of active terms decreses as a function of λ, which shows us that the sparse model selection is working.



In the following table we display the models recovered that contain 4, 7, 8 and 10 active terms. No models are found that contain 5, 6 or 9 active terms. This is because the we are doubling the λ at every step and this is too coarse to capture all model complexities.

	Term	Parameter	4 active terms	7 active terms	9 active terms	10 active terms
eq. $\dot{x}$	1	$p_{1,1}$		**–**	–	x
	x	$p_{1,2}$	x	**x**	x	x
	y	$p_{1,3}$	x	**x**	x	x
	z	$p_{1,4}$	–	**–**	–	–
	x^2	$p_{1,5}$	–	**–**	–	–
	xy	$p_{1,6}$	–	**–**	–	–
	xz	$p_{1,7}$	–	**–**	–	–
	y^2	$p_{1,8}$	–	**–**	–	–
	yz	$p_{1,9}$	–	**–**	–	–
	z^2	$p_{1,10}$	–	**–**	–	–
eq. $\dot{y}$	1	$p_{2,1}$	–	**–**	x	x
	x	$p_{2,2}$	x	**x**	x	x
	y	$p_{2,3}$	–	**x**	x	x
	z	$p_{2,4}$	–	**–**	–	–
	x^2	$p_{2,5}$	–	**–**	–	–
	xy	$p_{2,6}$	–	**–**	–	–
	xz	$p_{2,7}$	–	**x**	x	x
	y^2	$p_{2,8}$	–	**–**	–	–
	yz	$p_{2,9}$	–	**–**	–	–
	z^2	$p_{2,10}$	–	**–**	–	–
eq. $\dot{z}$	1	$p_{3,1}$	–	**–**	x	x
	x	$p_{3,2}$	–	**–**	–	–
	y	$p_{3,3}$	–	**–**	–	–
	z	$p_{3,4}$	x	**x**	x	x
	x^2	$p_{3,5}$	–	**–**	–	–
	xy	$p_{3,6}$	–	**x**	x	x
	xz	$p_{3,7}$	–	**–**	–	–
	y^2	$p_{3,8}$	–	**–**	–	–
	yz	$p_{3,9}$	–	**–**	–	–
	z^2	$p_{3,10}$	–	**–**	–	–

The correct model is the model with 7 active terms (highlighted in bold in the table), and its equations are

$$\dot{x} = p_{1,2}x + p_{1,3}y, \tag{2}$$

$$\dot{y} = p_{2,2}x + p_{2,3}y + p_{2,7}xz, \tag{3}$$

$$\dot{z} = p_{3,4}z + p_{3,6}xy. \tag{4}$$

This is the structure of the Lorenz system, which means that we were able to recover the true system.