Annealing code User Guide: Python/sympy interface

Jingxin Ye & Paul Rozdeba

September 11, 2014

1 Introduction

This document describes the annealing code used for calculating action levels of dynamical systems. It explains how to use the code I have developed, which employs the BFGS unconstrained optimization package provided by ALGLIB(http://www.alglib.net/).

2 Problem Statement

Given a dynamical system modeled by discrete map

$$x(n+1) = f(x(n)),$$

the probability distribution of its states can be expressed as $P(X|Y) = \exp(-A_0)$, when obersations Y are present. If one assumes both measurement noises and model error are independent and gaussian, the action A_0 has the format of

$$A_0(X) = \sum_{n=0}^{m} \frac{R_m(n)}{2} \sum_{l=1}^{L} [x_l(n) - y_l(n)]^2 + \frac{R_f}{2} \sum_{n=0}^{m-1} \sum_{a=1}^{D} [x_a(n+1) - f_a(\mathbf{x}(n))]^2. \tag{1}$$

where R_m and R_f are the inverse of variances. The prior distribution of $x(0)$ is assumed to be uniformly distributed, so the term $\log[P(\mathbf{x}(0))]$ is ignored. In our study, we found, under fixed observation noise level, action exhibits discrete levels for different model error levels. The behavior of action levels has significant application in statistical data assimilation.

3 Installation

The code can be run successfully on ubuntu Linux and MacOS. All the tests below are run successfully on Ubuntu 12.04. The needed softwares are:

gcc version 4.6.3 The codes and library are written in C++.

ALGLIB version 3.8.2 ALGLIB provides the optimization routine. (http://www.alglib.net/translator/re/alglib-3.8.2.cpp.zip) In the given examples, I put all the ALGLIB source codes in `lib` folder.

python version 2.7

sympy version 0.6.3 The sympy module is used to perform symbolic differentiation to obtained the Jacobian matrix of a dynamical system.

The installation of gcc, python and sympy should be straightforward using `apt-get install` under ubuntu. No installation is needed for ALGLIB. Extract all the source codes into a folder and it is done.

4 Usage

Setting up your particular problem requires the creation of 2 files. One is a Python script which contains the dynamical system itself (called `equations.py` by default), requiring declarations of sympy symbols for the dynamical states, parameters, and external forces in the model. The second is a C++ header file which contains the details of the measurements & the search procedure (`minAzero.h`).

Another script called `pyMakeFunc.py` takes `equations.txt` and generates a C++ source file (`func.cpp` by default), which contains C++ functions for the dynamical system and its Jacobian. Finally, `func.cpp` and `minAzero.h` are compiled with a "static" C++ source file called `minAzero.cpp` into an executable file which runs the L-BFGS minimization & annealing. This also requires linking to ALGLIB, which will be discussed shortly.

4.1 `equations.py`

In this file, several variables must be declared:

problem_name - a Python string, used for naming files to identify the annealing "problem".

syms - a list of sympy symbols for all of the states & parameters to be estimated during the procedure.

stims - a list of sympy symbols for all external stimuli.

VF - a sympy matrix (of size $(1, D)$ for a $D = N_x + N_p$ dimensional system) containing sympy-symbolic definitions of the ODE's.

How you choose to "build up" VF is up to you, as long as it is a list which, when evaluated, unambiguously returns symbolic definitions of the ODE's. Note that you are restricted to the use of sympy (or Python built-in) math functions which are recognized by the `ccode` module in sympy; see BLANKURL for a full list of allowed functions.

4.2 `makeFunc.py`

Given a properly created `equations.py`, the script `pyMakeFunc.py` employs sympy rountes performing symbolic differentiation to generate an unique cpp file `func.cpp`. If you are in the folder that contains `equations.py`, run the script with the command `python path/to/pyMakeFunc.py`

If your equations file has a different name, the insert it with the `-e` option, e.g.

```
python pyMakeFunc.py -e hhNet.py
```

4.3 `func.cpp`

`func.cpp` is the file generated by `makeFunc.py`, including two functions: the vector field and Jacobian matrix of the dynamical system. Example `func.cpp` for lorenz96 D=5,

```
#define NX 6 // dim of state variable + number of parameters
#define NS 0 // number of stimulus
using namespace alglib;
void func_origin(real_1d_array &x, int it, real_2d_array &sti, real_1d_array &dxdt);
void func_DF(real_1d_array &x, int it, real_2d_array &sti, real_2d_array &Jac);


void func_origin(real_1d_array &x, int it, real_2d_array &sti, real_1d_array &dxdt){
//lorenz96 vector field
dxdt[0]=x[5] - x[0] + x[4]*(x[1] - x[3]);
dxdt[1]=x[5] + x[0]*(x[2] - x[4]) - x[1];
```

```
dxdt[2]=x[5] + x[1]*(-x[0] + x[3]) - x[2];
dxdt[3]=x[5] + x[2]*(-x[1] + x[4]) - x[3];
dxdt[4]=x[5] + x[3]*(x[0] - x[2]) - x[4];
dxdt[5]=0;
}

void func_DF(real_1d_array &x, int it, real_2d_array &sti, real_2d_array &Jac){
//lorenz96 Jacobian matrix
Jac[0][0]=-1;
  ....
  ......
Jac[5][5]=0;
}
```

4.4 minAzero.cpp

`minAzero.cpp` is the cpp file containing the main function, action and its gradient function and the annealing procedure. Usually users do not need to modify its content for use, all the functions are generalized for any problems. The only things need to change is the `#define` directives listed below.

```
#define NT 1000       // number of time steps
#define DT 0.05     // time step size
#define NMEA 1       // number of measurements
#define NPATH 10    // number of paths
#define NBETA 30     // maximal beta
#define RM 4         // inverse of measurement noise variance
#define RF0 0.01     // initial Rf
```

4.5 Compilation

We need compile all the source files provided by ALGLIB and `minAzero.cpp`. No compilation is needed for `func.cpp`, as it is included in `minAzero.cpp` (see `#include "func.cpp"` in `minAzero.cpp`). The compilation can be implemented with simple command

```
g++ -I /path/to/alglib/ /path/to/alglib/* minAzero.cpp -o min
```

Flag `-I` is followed by the alglib directory, which tells g++ to search for necessary header files in that folder. All the alglib files `/path/to/alglib/*` and `minAzero.cpp` will be compiled and it generates the executable file `min`.

4.6 Input Files

Before we run `./min`, some preparation work is needed. Within the folder where you want to run `min`,

- create a folder named `path` to store output files.
- put `twin_data.dat` of the following format (the first NMEA column(s) are the measured ones)

```
x1(0) x2(0) x3(0) x4(0) x5(0) f(0)
x1(1) x2(1) x3(1) x4(1) x5(1) f(1)
x1(2) x2(2) x3(2) x4(2) x5(2) f(2)
....
```

- If your model contains external stimulus, put `stimulus.dat`, if you have two stimulus, the format will be

```
stimulus1(0) stimulus2(0)
stimulus1(1) stimulus2(1)
stimulus1(2) stimulus2(2)
...
```

5 Output

Each path will be stored in individual file with the name like `D6_M1_PATH0.dat` in the `path` folder. Each line of `D6_M1_PATH0.dat` contains the optimal path at different values of `beta`. The first three numbers are beta exitflag and action value, respectively. Exitflag can be 0 or 1. 1 means BFGS routines find the optimal path and 0 means it fails. The rest numbers represent the optimal path.

```
beta exitflag action_value
optimal_path[x1(0) x2(0) x3(0) x4(0) x5(0) f(0) x1(1) x2(1) x3(1)
x4(1) x5(1) f(1) ..... x1(NT) x2(NT) x3(NT) x4(NT) x5(NT) f(NT)]
```

6 Example

Three examples are provided.

6.1 Lorenz96 D=5

6.2 Colpitts Oscillator

6.3 NaKL

$$\frac{dV}{dt} = CI_{inj}(t) + g_{Na}m^3h(E_{Na} - V) + g_K n^4(E_K - V) + g_L(E_L - V)$$
$$\frac{da}{dt} = \frac{a_\infty - a}{\tau_a}$$
$$a_\infty = \frac{1}{2} + \frac{1}{2}\tanh\left(\frac{V - V_a}{\Delta V_a}\right)$$
$$\tau_a = \tau_{a0} + \tau_{a0}\left(1 - \tanh^2\left(\frac{V - V_a}{\Delta V_a}\right)\right)$$

7 Troubleshooting

I have tested these scripts over a wide range of problems, so I believe that the algorithms are correct. However, there are a few common errors that may crop up.

- Variable and parameter naming is very important. At few common problems can crop up. Never use a variable name that includes the name of another variable. For instance p1 and p11 would be bad, since p11 includes p1. In this case, p01 and p11 would be adequate. Along this vein, all variable names should be at least 2 characters long, just in case.