User's notes on 2D Langevin Regression

August 26, 2021

1 Basic use

Introduction This program is made to automatically fit Langevin models to experimental data. This is made possible thanks to Machine Learning methods, especially Structural Risk Minimisation and model selection through validation [1]. This program is based on the one written by Callaham et al. [2] which provided basic methods to do Langevin regression in 1D.

We improved this version by creating a new module, 'functions_regression.py' designed to generalize methods to carry Kramers-Moyal coefficients for the 2D or 3D cases. We also added cross terms in Kramers-Moyal Diffusion terms, which were avoided by Callaham et al. Finally, we introduced a parallelisation of computations to reduce running time which can become huge in the 2D case.

This program is using common scientific python librairies such as Numpy, Scipy, Sympy but also other more specific librairies such as kramersmoyal or netCDF4 (for reading files).

Program steps This program can be summarised in 8 different steps :

- Step 1 : Read experimental data (1D or 2D) and store in a specific variable KMc. We use netCDF4 files for this task (done by default).
- Step 2 : Compute Kramers-Moyal coefficients associated with experimental data. Store this additional information inside KMc.
- Step 3 : Create polynomial models that will be used to fit drift and diffusion Kramers-Moyal coefficient obtained in experiments.
- Step 4 : Compute a first wild guess of polynomial coefficients using least-square method.
- Step 5 : Refine this guess by using Nelder-Mead method. This is helped with a validation procedure introduced by Callaham et al. which adds a correction to computed Kramers-Moyal coefficients. This correction is based on both solving Fokker-Planck equation and adjoint of Fokker-Planck equation. Optimisation method can be modified within the code (l.217 utils_reg.py) [1].
- Step 6 : Do again Step 5 but without one of the polynomial coefficient. This is repeated so that we can find the coefficient that contributes the most to optimisation cost.
- Step 7 : Keep the most efficient terms and do again Step 6 trying to isolate another less efficient coefficient. This is done until we reach polynomes with only one coefficient remaining.
- Step 8 : Save all data in files.

Parameters One just has to change the file path for the experimental data and useful parameters to do Langevin Regression. Here is an explicit list of parameters that can be set. Some of them will be detailed in further sections.

- Nbr_proc (int) : Specifies the number of processes created by the program. More information about parallelisation in the associated section.
- N_bins (int) : Number of bins used to compute Kramers-Moyal coefficients. See methods to compute those in kramersmoyal documentation. Careful, $15O(N^3)$ program in case of use of exp method [3].
- f_sample (float) (Hz): Frequency used in experimental data.

[1] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html`

- stride (int) : Number of snapshots to skip. This parameter is designed to artifically reduce frequency sampling when reading experimental data. See Callaham et al. methodology for more information.
- bw (float) : bandwidth used in computing Kramers-Moyal coefficients. See kramersmoyal documentation for more information.
- powers ((numpy)array[int]) : An array that is composed of tuple [x,y,z,k,...] which represents the power of asked Kramers-Moyal coefficients. Ex : [[0,1],[1,0],[2,0],[0,2],[1,1]]. [2,1] : compute coefficient x^2y. No coefficient higher than order 2 are supported in this program (but it can be modified to support it).
- Coeff_x (list[int]) : A list of orders for polynomes regarding the variable x. Different orders not supported when having more than one variables.
- cross_terms (list[bool]) : A list that specifies if we want to add cross terms for generated polynomes (only for dimensions strictly higher than 1). True = add cross terms.
- type_solver (str) : Can be either 'diff' or 'exp'. Method type to use to compute corrections to Kramers-Moyal coefficients. See associated section for further details.
- type_multi (str) : Can be either 'diff' or 'step'. Specifies parallelisation method to use. This is further discussed in the associated section.
- checkpoint_load (bool) : When set to True, allows the program to restart from a previous save. Each time the program reaches a step (understand find the most efficient polynome for a given number of non zero terms), all information are saved inside a file. This save can be used to reload this reached step in case the program was stopped. Warning : do not change any parameters between the new and the previous sessions (not supported).
- Nbr_iter_max (int) : Set the maximum number of iteration that can be used by the optimisation solver. The method is stopped when it reaches this limit.
- kl_reg (int) : Set how much importance we give to the regularisation of optimisation cost made when solving Fokker-Planck equation. For more details, see papers of Callaham et al.

Modules description We summarise here all bespoke modules used in this program :

- functions_regression : This module is essentially made to do computations over Kramers-Moyal coefficients. Some functions are designed to transform any Kramers-Moyal coefficient into Langevin coefficient and vice-versa. Some less specific functions can generate polynomes given their orders and if we need cross terms or not for instance. The main content is 'KM_list' class that encapsulates all the following information :
  - KM_coeffs (list[KM_coefficient]) : A list containing every Kramers-Moyal coefficient. Each coefficient is related to an experimental value, to model values (which are by default set to 0), to a symbolic form as for the model (written with Sympy).
  - powers (list[list[int]]) : A list containing every coefficient's related order.
  - type_diff (str) : A string that indicates in which mode experimental values are : either in Fokker-Planck formalism or in Langevin formalism
- utils_reg : This is the main library containing cost functions, optimisation methods and essential functions that are at the core of the 8 Steps introduced above. Multiprocessing methods are introduced in this library. Some more basic functions can read/write into files to keep a track of the events when the program is running.
- fpsolve_reg : This a library containing classes, methods and functions designed to compute corrections of Kramers-Moyal coefficients. Most of methods are based on the work of Callaham et al.

2 Solver types

The correction used by Callaham et al. to compute accurate Kramers-Moyal coefficients is based on the computation of an exponential of the adjoint of Fokker-Planck :

$$m_\tau^{(m,n)}(x,y) = \frac{1}{(m+n)!\tau}[e^{\tau L^\dagger(x',y')}(x'-x)^m(y'-y)^n]_{x'=x}, \tag{1}$$

where $L^\dagger$ (dimensions in 2D : $N\_bins^2 \times N\_bins^2$) is the adjoint of Fokker-Planck and $m_\tau^{(m,n)}(x,y)$ Kramers-Moyal coefficient of order (m,n). For N_bins too high, the computation of the exponential becomes really slow. As proposed by Callaham et al., we introduced the possibility to do the same coefficient correction by solving a differential equation instead. This can be done thanks to the following Cauchy problem :

$$\begin{cases} \frac{\partial w^{(m,n)}}{\partial t} = L^\dagger(x,y) w^{(m,n)}(x,y,t) \\ w^{(m,n)}(x,y,t=0) = x^m y^n \end{cases} . \tag{2}$$

Solving this equation can give $w^{(m,n)}$ which can be converted to $m_\tau^{(m,n)}$ using equation 1. As for the 2D case we have :

$$\begin{cases} m_\tau^{(1,0)}(x,y) = \frac{1}{\tau}(w^{(1,0)}(x,y,\tau) - x) \\ m_\tau^{(0,1)}(x,y) = \frac{1}{\tau}(w^{(0,1)}(x,y,\tau) - y) \\ m_\tau^{(2,0)}(x,y) = \frac{1}{2\tau}(w^{(2,0)}(x,y,\tau) - 2x w^{(1,0)}(x,y) + x^2) \\ m_\tau^{(0,2)}(x,y) = \frac{1}{2\tau}(w^{(0,2)}(x,y,\tau) - 2y w^{(0,1)}(x,y) + y^2) \\ m_\tau^{(1,1)}(x,y) = \frac{1}{2\tau}(w^{(1,1)}(x,y,\tau) - x w^{(0,1)}(x,y) - y w^{(1,0)}(x,y) + xy) \end{cases} . \tag{3}$$

In N dimensions, the general formula to get $m_\tau^{(i_1,i_2,\dots)}$ from $w^{(i_1,i_2,\dots)}$ is the following :

$$m_\tau^{(i_1,i_2,\dots)} = \frac{1}{(\sum_k i_k)!\,\tau} \sum_{j_1,j_2,\dots}^{i_1,i_2,\dots} \left[\prod_k^N (-X_k)^{i_k - j_k} \binom{i_k}{j_k} \right] w^{(j_1,j_2,\dots)}, \tag{4}$$

where X_k represents the coordinates (meshing) along k axis. Please note that we also set $w^{(0,0,\dots)} = 1$. This formula has been already implemented in the program.

You can choose in the program if you want to compute those coefficient either by the exponential ('exp') or by solving 5 differential equations ('diff'). It is highly recommended to use ('diff') for the 2D case, especially for high N_bins (> 10). As for the 1D case, ('exp') is faster for low enough N_bins (not determined).

The solver used for solving those differential equation is 'DOP853', an explicit Runge-Kutta method of order 8 found in Scipy [2]. Other solvers can be used, but at the cost of precision. Those methods determine automatically the most efficient sampling rates to solve those equations.

3 Setting up multi-processing

We introduced some multi-processing methods to accelerate the computations. Two types have been created.

- 'step' can divide the research of the most expensive coefficient in Step 5. This method is allowed for both solvers.
- 'diff' makes the resolution of differential equations on different processes. This method is only supported for 'diff' solver.

'diff' method should not be used as it is slower than 'step' method. Further tests and improvements should be done to make the second method more efficient.

Disclaimer We want to highlight that the line (if __name__ == '__main__':) at the beginning of the script must not be forgotten. This line is necessary for a proper use of multiprocessing module in Python (especially on windows)[3] !

4 Improvements

Model types In this program, we chose to model everything with polynomes. But if data does not seem to be close from a polynomial behaviour, other models can be introduced thanks to Sympy [4]. One must however define manually each model for each Kramers-Moyal coefficient. This can be long to do for large number of parameters, and we advise to create an automatic method to do so (as it has been done for polynomial models).

[2] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html
[3] https://stackoverflow.com/questions/20222534/python-multiprocessing-on-windows-if-name-main
[4] https://www.sympy.org/en/index.html

More dimensions Parts of librairies used can be modified to handle more than the 1D and 2D cases. You can find in the following the main guidelines to do to add more dimensions in this program :

- Compute and define Langevin/Fokker-Planck conversions (l.0 functions_regression.py). In our case, we assumed that both Fokker-Planck and Langevin diffusion terms were symmetric. But other asumptions can be made to solve the system of equations defining the relation between the two set of terms. In the ND case in the symmetric asumption, one can solve the system by solving the associated eigenvalues problem (use Scipy to do so). Instances of 'convert_to_langevin_2D_sym' and of 'convert_to_FP_2D' must be changed accordingly (mostly located in 'cost_reg' in utils_reg.py).
- Increase number of variables of polynomes (l.91 functions_regression.py). Only three symbolic variables (from Sympy) have already been defined for this function : x,y and z. Feel free to add other symbolic variables in case of more than 3 dimensions.
- Add support for SteadyFP and AdjFP classes (l.22 and l.120 fpsolve_reg.py). For higher dimensions, one must add initialisations and computations to do for higher dimensions for those classes, namely :
  - SteadyFP :: __init__ (l.32) // Add Fourier transform in other dimensions.
  - SteadyFP :: precompute_operator (l.72) // Modify Einstein sums to include other dimensions.
  - AdjFP :: (l.120) // Add a static method 'derivsnd' which introduces derivative coefficients along different axis.
  - AdjFP :: (l.120) // Add a method 'operatornd' which can define and compute adjoint of Fokker-Planck using derivatives defined above.
  - AdjFP :: __init__ (l184) // Modify the constructor so that it can include other dimensions (and not only the 2D case as by default).
- Modify the main file accordingly (KM_regression_XX.py). Add other dimensions to powers, compute histogram in Nd instead, change Langevin/Fokker-Planck conversions methods...

Use of weights The original program made by Callaham et al. was taking computational weights into account. We did not as this point was not discussed in their papers and did not work well for us. It should be however an interesting idea to do quantitative studies of the influence of this on the final results obtained by the program.

Use of kernels The library kramersmoyal is computing Kramers-Moyal coefficient using kernels. This method that can parametrised with bw (bandwidth, float) can be useful to obtain better estimations of those coefficients. We however did not check how to set this parameter correctly. A work on this might become handy as it would refine outcomes. See kramersmoyal documentation for more information.

More Kramers-Moyal orders This program is intended to be limited to 2 orders of Kramers-Moyal coefficient, assuming that our stochastic process has a white gaussian noise. But one might expect to go beyond this hypothesis, needing therefore more orders for coefficients. While it is supported by some librairies (kramersmoyal, functions_regression), the program need however to be totally rewritten to fully take into account more orders.

References

[1] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[2] Jared L Callaham, J-C Loiseau, Georgios Rigas, and Steven L Brunton. Nonlinear stochastic modelling with langevin regression. *Proceedings of the Royal Society A*, 477(2250):20210092, 2021.

[3] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM review*, 45(1):3–49, 2003.