Introduction to Juobs

Alessandro Conigli

IFT UAM-CSIC

December 13th -17th Tor Vergata

Motivation

- We saw how ADerrors.jl simplifies enormously the task of data analysis

- Error propagation form MC history happens under the carpet

- Sometimes though when dealing with several ensembles it is useful to have some sort of automatised routines:

  - ★ Read several dat files with the same structure
  - ★ Perform advanced linear algebra decompositions, i.e. generalised eigenvalue problem
  - ★ Measure same set of observables several times
  - ★ Advanced user-friendly fitting routines

Reading the data

- Reads 2 and 3-pt correlation function with a given Dirac structure

Hard-coded
Easy to extend

```
read_mesons(path::String, g1::Union{String, Nothing}=nothing, g2::Union{String, Nothing}=n

read_mesons(path::Vector{String}, g1::Union{String, Nothing}=nothing, g2::Union{String, No
```

This function read a mesons dat file at a given path and returns a vector of `CData` structures for different masses and Dirac structures. Dirac structures `g1` and/or `g2` can be passed as string arguments in order to filter correaltors. ADerrors id can be specified as argument. If is not specified, the `id` is fixed according to the ensemble name (example: "H400"-> id = "H400")

- Reads openQCD ms dat files with flow time values

openQCD
compatible

```
read_ms(path::String; id::Union{String, Nothing}=nothing, dtr::Int64=1, obs::String="Y")
```

Reads openQCD ms dat files at a given path. This method return YData:

- `t(t)`: flow time values
- `obs(icfg, x0, t)`: the time-slice sums of the densities of the observable (Wsl, Ysl or Qsl)
- `vtr`: vector that contains trajectory number
- `id`: ensmble id

`dtr` = `dtr_cnfg` / `dtr_ms`, where `dtr_cnfg` is the number of trajectories computed before saving the configuration. `dtr_ms` is the same but applied to the ms.dat file.

Reading the data

- Reads openQCD ms1 dat files containing reweighting factors

openQCD compatible

```
read_ms1(path::String; v::String="1.2")
```

Reads openQCD ms1 dat files at a given path. This method returns a matrix `W[irw, icfg]` that contains the reweighting factors, where `irw` is the `rwf` index and icfg the configuration number. The function is compatible with the output files of openQCD v=1.2, 1.4 and 1.6. Version can be specified as argument.

- Reads openQCD pbp.dat files containing mass derivative of the action

openQCD compatible

`juobs.read_md` — Function

```
read_md(path::String)
```

Reads openQCD pbp.dat files at a given path. This method returns a matrix `md[irw, icfg]` that contains the derivatives dS/dm, where $md[irw = 1] = dS/dm_l$ and $md[irw = 2] = dS/dm_s$

$$Seff = -tr(log(D + m))$$

$$dSeff/dm = -tr((D + m)^{-}1)$$

Tools

- Creates the primary observable as `Vector{uwreal}` with associated error

```
corr_obs(cdata::CData; real::Bool=true, rw::Union{Array{Float64, 2}, Nothing}=nothing, L::

corr_obs(cdata::Array{CData, 1}; real::Bool=true, rw::Union{Array{Array{Float64, 2}, 1}, N
```

Creates a `Corr` struct with the given `CData` struct `cdata` (`read_mesons`) for a single replica. An array of `CData` can be passed as argument for multiple replicas.

The flag `real` select the real or imaginary part of the correlator. If `rw` is specified, the method applies reweighting. `rw` is passed as a matrix of Float64 (`read_ms1`) The correlator can be normalized with the volume factor if `L` is fixed.

- Computes the derivative of an observable with respect to sea quark masses

```
md_sea(a::uwreal, md::Vector{Matrix{Float64}}, ws::ADerrors.wspace=ADerrors.wsg)
```

Computes the derivative of an observable A with respect to the sea quark masses.

$$\frac{d<A>}{dm(sea)} = \sum_i \frac{\partial<A>}{\partial<O_i>}\frac{d<O_i>}{dm(sea)}$$

$$\frac{d<O_i>}{dm(sea)} =< O_i >< \frac{\partial S}{\partial m} > - < O_i \frac{\partial S}{\partial m} >= - < (O_i - < O_i >)(\frac{\partial S}{\partial m} - < \frac{\partial S}{\partial m} >) >$$

where O_i are primary observables

`md` is a vector that contains the derivative of the action S with respect to the sea quark masses for each replica. `md[irep][irw, icfg]`

`md_sea` returns a tuple of uwreal observables $(dA/dm_l, dA/dm_s)|_{sea}$, where m_l and m_s are the light and strange quark masses.

Tools

- Computes the derivative of an observable with respect to valence quark masses

```
md_val(a::uwreal, obs::Corr, derm::Vector{Corr})
```

Computes the derivative of an observable A with respect to the valence quark masses.

$$\frac{d<A>}{dm(val)} = \sum_i \frac{\partial<A>}{\partial<O_i>} \frac{d<O_i>}{dm(val)}$$

$$\frac{d<O_i>}{dm(val)} =< \frac{\partial O_i}{\partial m(val)} >$$

where O_i are primary observables

`md` is a vector that contains the derivative of the action S with respect to the sea quark masses for each replica.
`md[irep][irw, icfg]`

- Perform a linear fit of uwreal variables

```
lin_fit(x::Vector{<:Real}, y::Vector{uwreal})
```

Computes a linear fit of uwreal data points y. This method return uwreal fit parameters and chisqexpected.

```
fitp, csqexp = lin_fit(phi2, m2)
m2_phys = fitp[1] + fitp[2] * phi2_phys
```

source

Tools

- Advanced fitting routines with error propagation

```
fit_routine(model::Function, xdata::Array{<:Real}, ydata::Array{uwreal}, param::Int64=3; w

fit_routine(model::Function, xdata::Array{uwreal}, ydata::Array{uwreal}, param::Int64=3; w
```

Given a model function with a number param of parameters and an array of `uwreal`, this function fit ydata with the given `model` and print fit information The method return an array `upar` with the best fit parameters with their errors. The flag `wpm` is an optional array of Float64 of lenght 4. The first three paramenters specify the criteria to determine the summation windows:

- `vp[1]`: The autocorrelation function is summed up to $t = round(vp[1])$.
- `vp[2]`: The sumation window is determined using U. Wolff poposal with $S_\tau = wpm[2]$
- `vp[3]`: The autocorrelation function $\Gamma(t)$ is summed up a point where its error $\delta\Gamma(t)$ is a factor `vp[3]` times larger than the signal.

An additional fourth parameter `vp[4]`, tells ADerrors to add a tail to the error with $\tau_{exp} = wpm[4]$. Negative values of `wpm[1:4]` are ignored and only one component of `wpm[1:3]` needs to be positive. If the flag `covar` is set to true, `fit_routine` takes into account covariances between x and y for each data point.

```
@. model(x,p) = p[1] + p[2] * exp(-(p[3]-p[1])*x)
@. model2(x,p) = p[1] + p[2] * x[:, 1] + (p[3] + p[4] * x[:, 1]) * x[:, 2]
fit_routine(model, xdata, ydata, param=3)
fit_routine(model, xdata, ydata, param=3, covar=true)
```

source

Observables

- Computes effective masses for a given correlation

```
meff(corr::Vector{uwreal}, plat::Vector{Int64}; pl::Bool=true, data::Bool=false, wpm::Unio

meff(corr::Corr, plat::Vector{Int64}; pl::Bool=true, data::Bool=false, wpm::Union{Dict{Int
```

Computes effective mass for a given correlator corr at a given plateau `plat`. Correlator can be passed as an `Corr` struct or `Vector{uwreal}`.

The flags `pl` and `data` allow to show the plots and return data as an extra result.

- Computes PCAC masses given the correlators `A0P` and `PP`

```
mpcac(a0p::Vector{uwreal}, pp::Vector{uwreal}, plat::Vector{Int64}; ca::Float64=0.0, pl::B

mpcac(a0p::Corr, pp::Corr, plat::Vector{Int64}; ca::Float64=0.0, pl::Bool=true, data::Bool
```

Computes the bare PCAC mass for a given correlator `a0p` and `pp` at a given plateau `plat`. Correlator can be passed as an `Corr` struct or `Vector{uwreal}`.

The flags `pl` and `data` allow to show the plots and return data as an extra result. The `ca` variable allows to compute `mpcac` using the improved axial current.

Observables

- Computes the bare decay constant given the correlators `A0P` and `PP`

```
dec_const(a0p::Vector{uwreal}, pp::Vector{uwreal}, plat::Vector{Int64}, m::uwreal, y0::Int

dec_const(a0p::Corr, pp::Corr, plat::Vector{Int64}, m::uwreal; ca::Float64=0.0, pl::Bool=t
```

Computes the bare decay constant using A_0P and PP correlators . The decay constant is computed in the plateau `plat`. Correlator can be passed as an `Corr` struct or `Vector{uwreal}`. If it is passed as a uwreal vector, effective mass `m` and source position `y0` must be specified.

The flags `pl` and `data` allow to show the plots and return data as an extra result. The `ca` variable allows to compute `dec_const` using the improved axial current.

The method assumes that the source is close to the boundary. It takes the following ratio to cancel boundary effects. $R = \frac{f_A(x_0,y_0)}{\sqrt{f_P(T-y_0,y_0)}} * e^{m(x_0-T/2)}$

- Computes the bare decay constant using WI and `PP` correlator (tm fermion only)

```
dec_const_pcvc(corr::Vector{uwreal}, plat::Vector{Int64}, m::uwreal, mu::Vector{Float64},

dec_const_pcvc(corr::Corr, plat::Vector{Int64}, m::uwreal; pl::Bool=true, data::Bool=false
```

Computes decay constant using the PCVC relation for twisted mass fermions. The decay constant is computed in the plateau `plat`. Correlator can be passed as an `Corr` struct or `Vector{uwreal}`. If it is passed as a uwreal vector, vector of twisted masses `mu` and source position `y0` must be specified.

The flags `pl` and `data` allow to show the plots and return data as an extra result.

The method assumes that the source is in the bulk.

Observables

- Computes the flow t0 using the energy density of the YM action

```
comp_t0(Y::YData, plat::Vector{Int64}; L::Int64, pl::Bool=false, rw::Union{Matrix{Float64}

comp_t0(Y::Vector{YData}, plat::Vector{Int64}; L::Int64, pl::Bool=false, rw::Union{Vector{
```

Computes `t0` using the energy density of the action `Ysl`(Yang-Mills action). `t0` is computed in the plateau `plat`. A polynomial interpolation in `t` is performed to find `t0`, where `npol` is the degree of the polynomial (linear fit by default)

The flag `pl` allows to show the plot.

```
#Single replica
Y = read_ms(path)
rw = read_ms(path_rw)

t0 = comp_t0(Y, [38, 58], L=32)
t0_r = comp_t0(Y, [38, 58], L=32, rw=rw)

#Two replicas
Y1 = read_ms(path1)
Y2 = read_ms(path2)
rw1 = read_ms(path_rw1)
rw2 = read_ms(path_rw2)

t0 = comp_t0([Y1, Y2], [38, 58], L=32, pl=true)
t0_r = comp_t0(Y, [38, 58], L=32, rw=[rw1, rw2], pl=true)
```

source

Linear Algebra

- Sometimes it might be useful to solve problems like

$$C(t)v(t,t_0) = C(t_0)v(t,t_0)\lambda(t,t_0)$$

where $$C_{ij}(t) = \langle O_i(t)O_j(0)\rangle = \sum_{n=1}^{\infty} e^{-E_n t}\psi_{ni}\psi_{nj} \qquad i,j = 1,\ldots,N$$

- Computes eigenvalues and generalised eigenvalues of `uwreal` data type

`juobs.uweigvals` — Function

```
uweigvals(a::Matrix{uwreal}; iter = 30)

uweigvals(a::Matrix{uwreal}, b::Matrix{uwreal};  iter = 30)
```

This function computes the eigenvalues of a matrix of uwreal objects. If a second matrix b is given as input, it returns the generalised eigenvalues instead. It takes as input:

- `a::Matrix{uwreal}` : a matrix of uwreal
- `b::Matrix{uwreal}` : a matrix of uwreal, optional
- `iter=30`: optional flag to set the iterations of the qr algorithm used to solve the eigenvalue problem

Linear Algebra

- Computes the (generalised) eigenvectors of matrices

```
uweigvecs(a::Matrix{uwreal}; iter = 30)

uweigvecs(a::Matrix{uwreal}, b::Matrix{uwreal};  iter = 30)
```

This function computes the eigenvectors of a matrix of uwreal objects. If a second matrix b is given as input, it returns the generalised eigenvectors instead. It takes as input:

- `a::Matrix{uwreal}` : a matrix of uwreal
- `b::Matrix{uwreal}` : a matrix of uwreal, optional
- `iter=30` : the number of iterations of the qr algorithm used to extract the eigenvalues

It returns:

- `res = Matrix{uwreal}`: a matrix where each column is an eigenvector

- Computes the (generalised) eigenvalues and eigenvectors of matrices

```
uweigen(a::Matrix{uwreal}; iter = 30)

uweigen(a::Matrix{uwreal}, b::Matrix{uwreal};  iter = 30)
```

This function computes the eigenvalues and the eigenvectors of a matrix of uwreal objects. If a second matrix b is given as input, it returns the generalised eigenvalues and eigenvectors instead. It takes as input:

Linear Algebra

- Computes the effective energies from the eigenvalues

```
energies(evals::Vector{Array}; wpm::Union{Dict{Int64,Vector{Float64}},Dict{String,Vector{F
```

This method computes the energy level from the eigenvalues according to:

$$E_i(t) = \log(\lambda(t)/\lambda(t+1))$$

where `i=1,..,n` with `n=length(evals[1])` and `t=1,..,T` total time slices. It returns a vector array en where each entry en[i][t] contains the i-th states energy at time t

- Other relevant features for `uwreal` data type
  - ★ qr decomposition
  - ★ Cholesky decomposition
  - ★ Invert a matrix
  - ★ Hessemberg reduction
  - ★ Tridiagonal reduction
  - ★ Matrix-Vector operations with `uwdot`

THANK YOU!