Current Status of BAT development in Dortmund

Projects assigned to Dortmund Group

1. Plotting
2. Standard Output
3. Combination Tool
4. Examples & Tutorials
5. Benchmarks and performance tests
6. Default models

1. Plotting - Developments since last BAT meeting

- previously presented plotting recipes are available in BAT.jl releases
- most parts of documentation & examples are written → need to be “polished” and uploaded
- further plot recipes to come when priors are available (e.g. knowledge-update-plots)

1. Plotting - Developments since last BAT meeting

- previously presented plotting recipes are available in BAT.jl releases
- most parts of documentation & examples are written → need to be “polished” and uploaded
- further plot recipes to come ~~when priors are available~~ (e.g. knowledge-update-plots)

what’s new ?

- during the last meeting we had discussions on the convergence of Markov chains
- implemented plots for “MCMC diagnostics”
- now possible plot for each individual chain:
  - Trace (time evolution of chain states)
  - Auto Correlation Function (ACF)
  - Kernel Density Estimator (KDE) [using Julia package KernelDensity.jl]
  - Histograms

1. Plotting - Developments since last BAT meeting

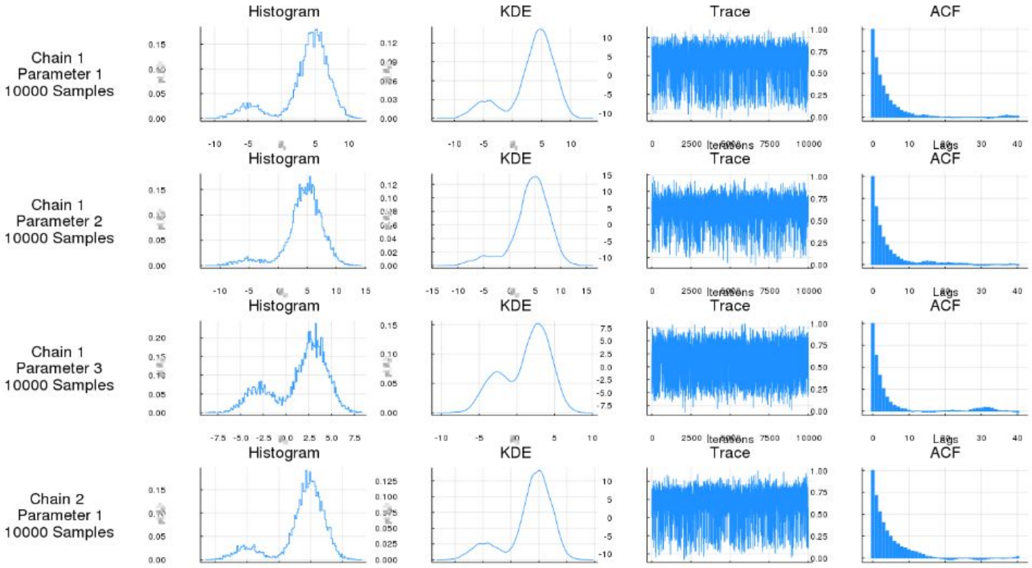
create a `MCMCDiagnostics` object:

In [3]:
```
mcmc = MCMCDiagnostics(samples, chain_results);
```

plot all MCMC diagnostics for all chains and all parameters:

In [5]:
```
plot(mcmc)
```

Out[5]:



code currently only on BAT.jl fork:
https://github.com/Cornelius-G/BAT.jl/blob/output/examples/mcmc_diagnostics_example.ipynb

1. Plotting - Developments since last BAT meeting

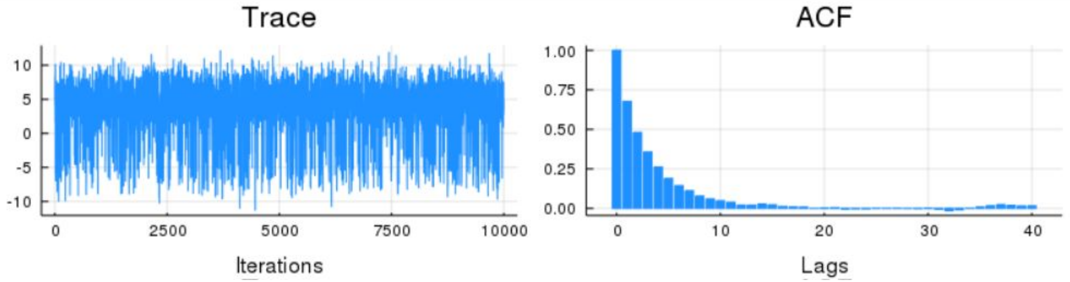
available keyword arguments:

- `params` - list of parameters to be plotted
- `chains` - list of chains to be plotted
- `diagnostics` - list of MCMC diagnostics to be plotted
    - `:histogram` - 1D histograms of samples
    - `:kde` - Kernel density estimate (using *KernelDensity.jl*)
    - `:trace` - Trace plot
    - `:acf` - Autocorrelation function (using *StatsBase.autocor*)
- `description::Bool = true` - show description (current chain, parameter, number of samples) as first column of plots
- `histogram::Dict` - options for histogram plots (supports all arguments for 1D plots for samples)
- `kde::Dict` - options for kde plots
- `trace::Dict` - options for trace plots
- `acf::Dict` - options for acf plots

plot only selected chains, parameters and diagnostics, hide description:

In [6]:
```
plot(mcmc, params=[1, 3], chains=[1, 3], diagnostics=[:trace, :acf], description=false)
```

Out[6]:



code currently only on BAT.jl fork: https://github.com/Cornelius-G/BAT.jl/blob/output/examples/mcmc_diagnostics_example.ipynb

2. Standard Output - Developments since last BAT meeting

- first version for BAT.jl output of result shown in last video meeting
- code & examples in BAT.jl fork
  https://github.com/Cornelius-G/BAT.jl/tree/output/examples
- todo:
  - implement interactive features (links to documentation or code, ...)
  - include further information like priors / parameter names etc.?
  - discuss technical details of *Summary* object

Reminder: BAT.jl output as plain text and HTML

```
summary = Summary(stats, chain_results)
display(summary)
```

```
BAT.jl - Summary
=================


Model
======
  likelihood:  MultiModalDensity([5.0, 5.0, 3.0], [2.0, 2.4, 1.5])
  prior:       HyperRectBounds
                     1.reflective_bounds [-30.0, 30.0]
                     2.reflective_bounds [-30.0, 30.0]
                     3.reflective_bounds [-30.0, 30.0]


Sampling
=========
  algorithm:                 MetropolisHastings
  number of chains:          8
  total number of samples:   8000


  Results
  ========
    parameter 1:
       mean ± std.dev. = 3.895 ± 3.777
       global mode     = 4.990

    parameter 2:
       mean ± std.dev. = 4.310 ± 3.573
       global mode     = 4.687

    parameter 3:
       mean ± std.dev. = 1.298 ± 3.091
       global mode     = 2.975


    covariance matrix:
          14.264  -0.164  0.064
          -0.164  12.765  -0.202
          0.064  -0.202  9.555
```

BAT.jl - Summary

Model

likelihood:	MultiModalDensity([5.0, 5.0, 3.0], [2.0, 2.4, 1.5])
prior:	HyperRectBounds 1. reflective_bounds [-30.0, 30.0] 2. reflective_bounds [-30.0, 30.0] 3. reflective_bounds [-30.0, 30.0]

show line/code of definition

Sampling

algorithm:	MetropolisHastings
number of chains:	8
total number of samples:	8000

hyperlink to documentation

Results

parameter 1	
mean ± std.dev.	3.895 ± 3.777
global mode	4.990

parameter 2	
mean ± std.dev.	4.310 ± 3.573
global mode	4.687

parameter 3	
mean ± std.dev.	1.298 ± 3.091
global mode	2.975

covariance matrix		
14.264	-0.164	0.064
-0.164	12.765	-0.202
0.064	-0.202	9.555

2. Standard Output - Developments since last BAT meeting

- first version for BAT.jl output of result shown in last video meeting

- code & examples in BAT.jl fork
  https://github.com/Cornelius-G/BAT.jl/tree/output/examples

- todo:
  - implement interactive features (links to documentation or code, ...)
  - include further information like priors / parameter names etc.?
  - **discuss technical details of *Summary* object**

```
summary = Summary(stats, chain_results)
```

ideas during last video meeting:

- create a *Summary* object with all information about the chains
- use *Summary* for printing results
- make it possible to re-run sampling from this *Summary*
- (use *Summary* for plotting MCMC diagnostics ?)

3. CombinationTool - Developments since last BAT meeting

- module for performing user friendly combinations of measurements using a BLUE-like likelihood

$$\ln L\big(\vec{x}|\vec{y}(\vec{\lambda})\big) = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\left[\vec{x} - U\,\vec{y}(\vec{\lambda})\right]_i \mathcal{M}_{ij}^{-1}\left[\vec{x} - U\,\vec{y}(\vec{\lambda})\right]_j$$

- basic implementation and main functionality is given
- tests are implemented, checking for correct implementation of likelihood & correct handling of inputs
- currently using it for a physics project (combination of Top & B measurements for EFT interpretations)

- what needs to be done:
    - handling of user errors (wrong/insufficient inputs etc.)
    - ranking of measurements & uncertainties

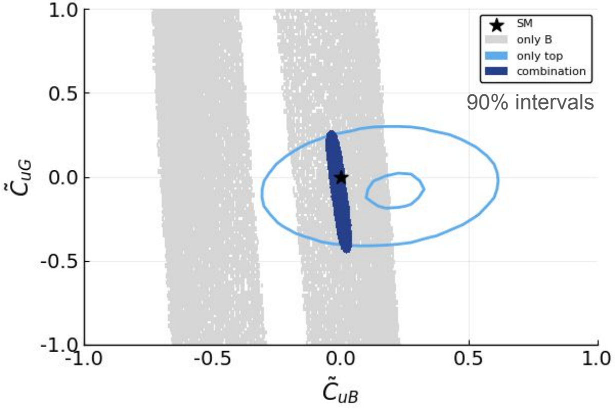
3. CombinationTool - Ranking of measurements

- how much impact has each individual measurement on the result of the fit ?
- idea:
  - take out one measurement from the fit at a time & redo the fit
  - calculate the change in the “area” of the posterior distribution

with BAT 1.0: calculate size of p% intervals in 1d & 2d marginal posterior distributions (i.e. summing the histograms)

with BAT.jl: want to calculate size of p% interval in the n-dim. posterior distribution

Example:



- question: how to determine the n-dim integral of the p% interval ?
  - is it possible to use Harmonic Mean Integration ?

4. Examples

- Implementation of examples from BAT 1.0
- So far implementation of simple Binomial, Poisson and Gaussian examples

⇨ Basically the same as in 1.0, more interesting Error Propagation example

- BAT 1.0 added new Observable

```cpp
RatioModel::RatioModel(const std::string& name)
    : BCModel(name)
{
    // define the parameters x and y
    AddParameter("x", 0., 8.); // index 0
    AddParameter("y", 0., 16.); // index 1

    GetParameters().SetPriorConstantAll();

    AddObservable("r", 0, 2, "#frac{x}{y}");
}
```

- and calculation

```cpp
void RatioModel::CalculateObservables(const std::vector<double>& parameters)
{
    // store ratio, if denominator is not zero
    if (parameters[1] != 0)
        GetObservable(0).Value(parameters[0] / parameters[1]);
    // else store zero
    else if (parameters[0] == 0)
        GetObservable(0).Value(0);
}
```

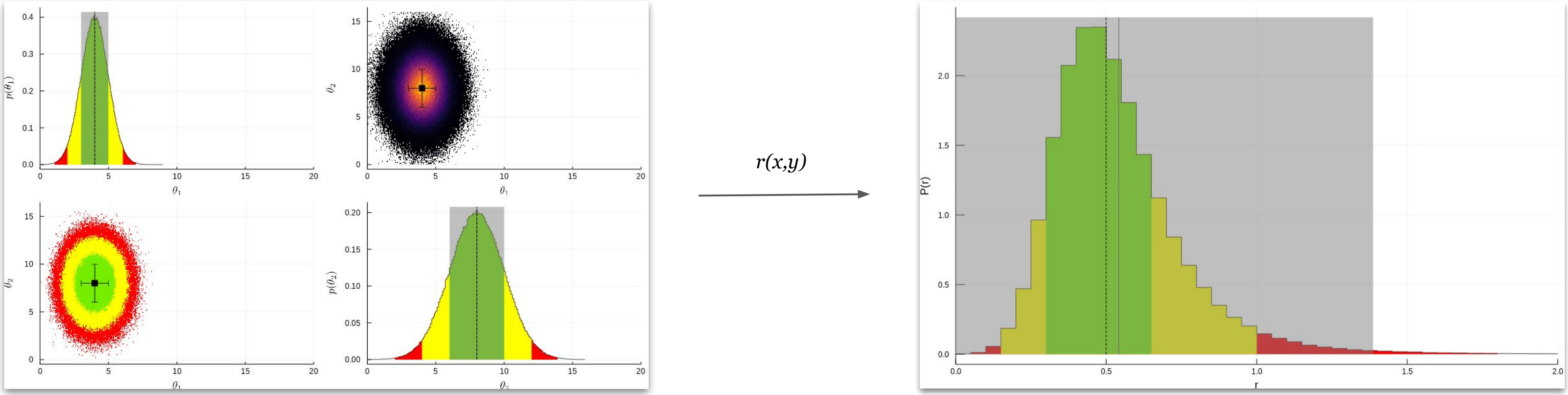
- BAT.jl possible calculation from samples, but "better" in Likelihood as parameter

```julia
In [3]: struct GaussianDensity<:AbstractDensity
            μ::Vector{Float64}
            σ::Vector{Float64}
            any_func::Function
        end
```

```julia
In [6]: function func(params)
            return params[1]/params[2]
        end
        model = GaussianDensity([μ_x,μ_y],[σ_x,σ_y],func)
```

4. Examples

- Treating the value as parameter takes advantage of Cornelius plot recipes
- Same results as BAT 1.0 (also for other examples)



- Further talking points:
  - Implementation of examples in BAT.jl (so far literate.jl format)
  - Possible Binder implementation?

5. Benchmarks and performance tests - First Ideas

want to benchmark the performance of BAT.jl:

a) compare performance of BAT.jl on different systems
- with minimal setup (e.g. on laptop in single-core mode)
- multi-core on laptop & workstations
- multi-core on cluster

b) compare BAT.jl with BAT 1.0
- run same models with both versions

c) compare different algorithms

⇨ need to create a **set of test models** to be used for:
- comparisons of different versions & setups
- testing performance of (new) sampling algorithms

5. Benchmarks and performance tests - Reminder: BAT 1.0

Results of performance testing for BAT version 0.9.4

Overview

Number of tests	26
Number of successful tests	25
Number of acceptable tests	1
Number of bad tests	0
Number of fatal tests	0
Number of tests unkown status	0

Function1D

Test	Status	Subtests	Good	Acceptable	Bad	Fatal	Unknown
1d_slope	good	14	14	0	0	0	0
1d_squared	good	14	14	0	0	0	0
1d_gaus	good	14	14	0	0	0	0
1d_poisson_0	good	14	14	0	0	0	0
1d_poisson_1	good	14	14	0	0	0	0
1d_poisson_2	good	14	14	0	0	0	0
1d_poisson_3	good	14	14	0	0	0	0
1d_poisson_4	acceptable	14	13	1	0	0	0
1d_poisson_5	good	14	14	0	0	0	0
1d_poisson_6	good	14	14	0	0	0	0
1d_poisson_7	good	14	14	0	0	0	0
1d_poisson_8	good	14	14	0	0	0	0
1d_poisson_9	good	14	14	0	0	0	0
1d_poisson_10	good	14	14	0	0	0	0
1d_exponential	good	14	14	0	0	0	0
1d_cauchy	good	14	14	0	0	0	0
1d_lognormal	good	14	14	0	0	0	0
1d_sin2	good	14	14	0	0	0	0
1d_2gaus	good	13	13	0	0	0	0

Function2D

Test	Status	Subtests	Good	Acceptable	Bad	Fatal	Unknown
2d_flat	good	1	1	0	0	0	0
2d_gaus	good	1	1	0	0	0	0
2d_2gaus	good	1	1	0	0	0	0

Varying parameters

Test	Status	Subtests	Good	Acceptable	Bad	Fatal	Unknown
1d_gaus_lag	good	0	0	0	0	0	0
2d_gaus_lag	good	0	0	0	0	0	0
1d_gaus_iter	good	0	0	0	0	0	0
2d_gaus_iter	good	0	0	0	0	0	0

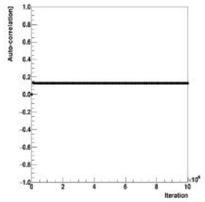
Results of performance testing for BAT version 0.9.4

Test "1d_poisson_6"

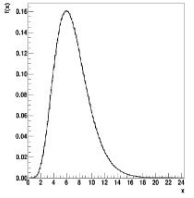
Results	
Status	good
CPU time	15.15 s
Real time	15.16 s
Plots	1d_poisson_6.pdf
Log	1d_poisson_6.log

Settings	
N chains	10
N lag	10
Convergence	true
N iterations (pre-run)	1000
N iterations (run)	10000000

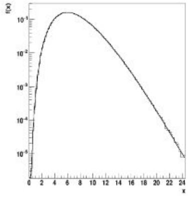
Plots



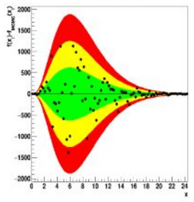
Auto-correlation for the parameter.



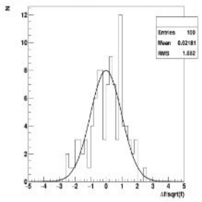
Distribution from MCMC and analytic function.



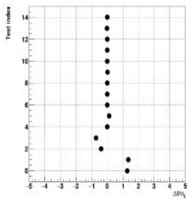
Distribution from MCMC and analytic function in log-scale.



Difference between the distribution from MCMC and the analytic function. The one, two and three sigma uncertainty bands are colored green, yellow and red, respectively.



Pull between the distribution from MCMC and the analytic function. The Gaussian has a mean value of 0 and a standard deviation of 1 (not fitted).



Summary of subtest values.

Subtest	Status	Target	Test	Uncertainty	Deviation [%]	Deviation [sigma]	Tol. (Good)	Tol. (Acceptable)	Tol. (Bad)
correlation par 0	off	0	0.129	0.0128	-	-10.08	0.3	0.5	0.7
chi2	good	98	116.9	14	19.33	-1.353	42	70	98
KS	good	1	0.8749	0.95	-12.51	0.1317	0.95	0.99	0.9999
mean	good	7	6.999	0.0008434	-0.008693	0.7215	0.00253	0.004217	0.005904
mode	good	6	6.001	0.165	0.02083	-0.007576	0.4949	0.8249	1.155
variance	good	6.997	7.137	1.183	2.008	-0.1188	3.548	5.914	8.279
quantile10	good	3.893	3.893	0.165	-0.01747	0.004123	0.4949	0.8249	1.155
quantile20	good	4.732	4.731	0.165	-0.0174	0.00499	0.4949	0.8249	1.155
quantile30	good	5.41	5.41	0.165	-0.00844	0.002768	0.4949	0.8249	1.155
quantile40	good	6.039	6.041	0.165	0.0238	-0.008714	0.4949	0.8249	1.155
quantile50	good	6.67	6.671	0.165	0.01419	-0.005737	0.4949	0.8249	1.155
quantile60	good	7.343	7.343	0.165	-0.00251	0.001117	0.4949	0.8249	1.155
quantile70	good	8.112	8.11	0.165	-0.01674	0.008232	0.4949	0.8249	1.155
quantile80	good	9.076	9.074	0.165	-0.01755	0.009657	0.4949	0.8249	1.155

5. Benchmarks and performance tests - First Ideas

need to find a set of test models:

- sets of test functions are available for optimizers

  https://www.sfu.ca/~ssurjano/optimization.html

  http://benchmarkfcns.xyz/fcns

- no special test functions for MCMC
- use (some of) the optimizer test functions as models for benchmarking
- define own (maybe physics related) test models

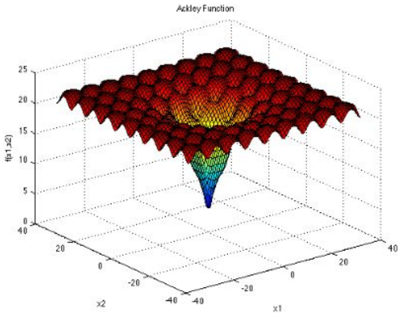
Many Local Minima

1. Ackley Function
2. Bukin Function N. 6
3. Cross-in-Tray Function
4. Drop-Wave Function
5. Eggholder Function
6. Gramacy & Lee (2012) Functic
7. Griewank Function
8. Holder Table Function
9. Langermann Function
10. Levy Function
11. Levy Function N. 13
12. Rastrigin Function
13. Schaffer Function N. 2
14. Schaffer Function N. 4
15. Schwefel Function
16. Shubert Function

Bowl-Shaped

17. Bohachevsky Functions
18. Perm Function 0, d, β
19. Rotated Hyper-Ellipsoid Function
20. Sphere Function
21. Sum of Different Powers Function
22. Sum Squares Function
23. Trid Function
24. …
25. …

ACKLEY FUNCTION



$$f(\mathbf{x}) = -a \exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

Description:

Dimensions: d

The Ackley function is widely used for testing optimization algorithms. In its two-dimensional form, as shown in the plot above, it is characterized by a nearly flat outer region, and a large hole at the centre. The function poses a risk for optimization algorithms, particularly hillclimbing algorithms, to be trapped in one of its many local minima.

Recommended variable values are: a = 20, b = 0.2 and c = 2π.

Input Domain:

The function is usually evaluated on the hypercube $x_i \in [-32.768, 32.768]$, for all i = 1, …, d, although it may also be restricted to a smaller domain.

Global Minimum:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (0, \dots, 0)$$

Code:

MATLAB Implementation
R Implementation

Appendix

Appendix - 1. Plotting

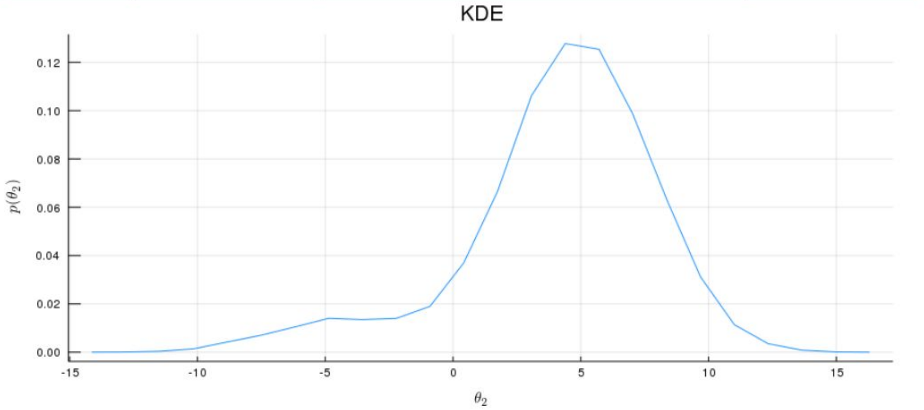
- special arguments for kde and acf

special options arguments for `:kde` (see *KernelDensity.jl*)

- `npoints`: number of interpolation points to use (default: npoints = 2048)
- `boundary`: lower and upper limits of the kde as a tuple
- `kernel`: the distributional family from *Distributions.jl* to use as the kernel (default = Distributions.Normal)
- `bandwidth`: bandwidth of the kernel

```
using Distributions
plot(mcmc, chains=[1], params=[2], diagnostics=[:kde],
    kde=Dict("npoints" =>24, "kernel" => Distributions.Logistic),
    description = false, size=(900, 400)
)
```

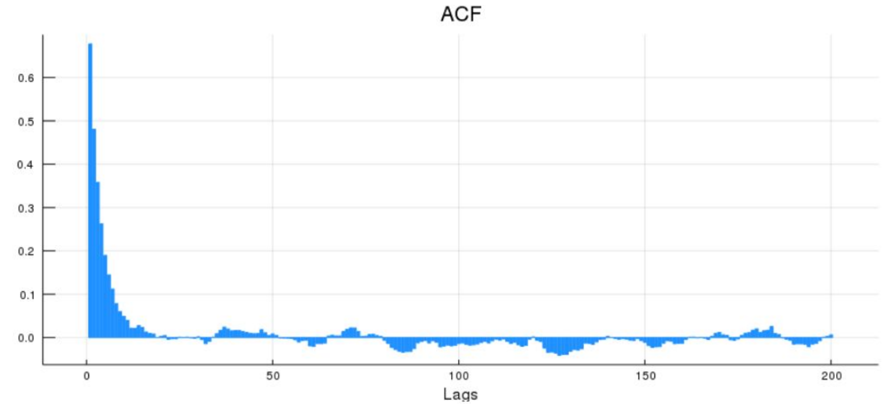
```
WARNING: using Distributions.nsamples in module Main conflicts with an existing identifier.
```



special keyword arguments for `:acf` (see *StatsBase.autocor*)

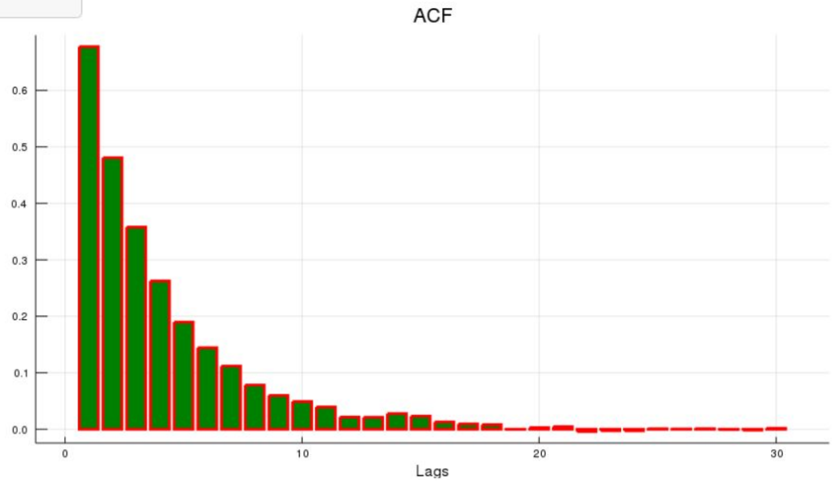
- `lags` - list of lags to be considered for ACF plots
- `demean` - denotes whether the mean should be subtracted before computing the ACF

```
plot(mcmc, chains=[1], params=[1], diagnostics=[:acf],
    acf=Dict("lags"=>collect(1:200), "demean" => true),
    description = false, size=(900, 400)
)
```



Appendix - 1. Plotting


```
plot(mcmc,
    chains=collect(1:2),
    params=[1, 3],
    histogram = Dict("seriestype" => :smallest_intervals, "mean"=>true, "localmode"=>Dict("line
color"=>:blue)),
    acf = Dict("lags"=>collect(1:10), "seriescolor"=>:red),
    kde = Dict("linecolor"=>:red, "title"=> "Kernel Density Estimate"),
    trace = Dict("linecolor"=>:green, "title"=> "Trace plot")
)

plot(mcmc,
    chains = [1],
    params = [1],
    diagnostics = [:acf],
    description = false,
    acf = Dict("lags"=>collect(1:30), "seriescolor"=>:green, "linecolor"=>:red, "linewidth"=> 3
, "linealpha"=>1),
    size = (900, 500)
)
```



Appendix - 3. Combination Tool

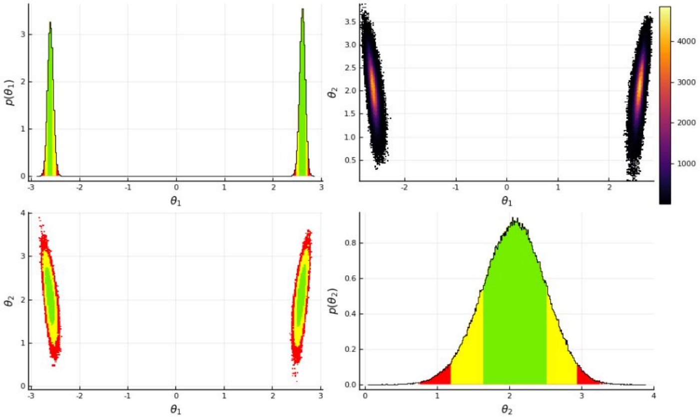

```julia
using CombinationTool
```

```julia
#============= Parameters ==============================================#
parameters = [
    Parameter("Param1", -5.0, 5.0),
    Parameter("Param2", -5.0, 5.0)
]

#============= Observables =============================================#
function obs2(params)
    2*params[1]^2 + 4*params[2]
end

observables = [
    Observable("Obs1", params -> params[1]^4-params[2]^2),
    Observable("Obs2", obs2)
]

#============= Measurements ============================================#
measurements = [
    Measurement("Meas1","Obs1", 42.0, Uncertainties("stat"=>1.1,
                                                     "syst"=>3.2)),

    Measurement("Meas2","Obs2", 22.0, Uncertainties("stat"=>2.1,
                                                     "syst"=>2.2))
]

#============= Correlations ============================================#
correlations = [
    Correlation("stat", [1.0 0.0
                         0.0 1.0], false),

    Correlation("syst", [1.0 0.5
                         0.5 1.0])
]
```

```julia
m = createmodel(parameters, observables, measurements, correlations)
density = EFTfitterDensity(m)

algorithm = MetropolisHastings()
bounds = HyperRectBounds(m.parameter_mins, m.parameter_maxs, reflective_bounds)


chainspec = MCMCSpec(algorithm, BayesianModel(density, bounds))
chains = 8
nsamples = 10^5

#define function to generate samples
samples, sampleids, stats = rand(chainspec, nsamples, chains)
```

```
INFO (1, 1): Trying to generate 8 viable MCMC chain(s).
INFO (1, 1): Selected 8 MCMC chain(s).
INFO (1, 1): Begin tuning of 8 MCMC chain(s).
INFO (1, 1): MCMC tuning of 8 chains successful after 15 cycle(s).
INFO (1, 1): Starting iteration over 8 MCMC chain(s).
```

```julia
plot(samples)
```



Appendix - 5. Performance tests

Results of performance testing for BAT version 0.9.4

Test "1d_poisson_6"

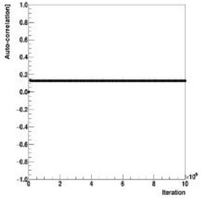
Results

Status	good
CPU time	15.15 s
Real time	15.16 s
Plots	1d_poisson_6.pdf
Log	1d_poisson_6.log

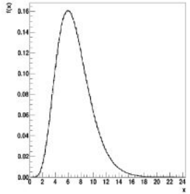
Settings

N chains	10
N lag	10
Convergence	true
N iterations (pre-run)	1000
N iterations (run)	10000000

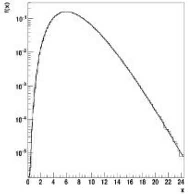
Plots



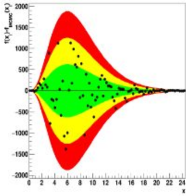
Auto-correlation for the parameter.



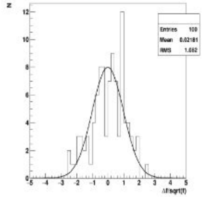
Distribution from MCMC and analytic function.



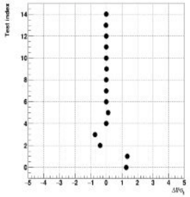
Distribution from MCMC and analytic function in log-scale.



Difference between the distribution from MCMC and the analytic function. The one, two and three sigma uncertainty bands are colored green, yellow and red, respectively.



Pull between the distribution from MCMC and the analytic function. The Gaussian has a mean value of 0 and a standard deviation of 1 (not fitted).



Summary of subtest values.

Subtest	Status	Target	Test	Uncertainty	Deviation [%]	Deviation [sigma]	Tol. (Good)	Tol. (Acceptable)	Tol. (Bad)
correlation par 0	off	0	0.129	0.0128	-	-10.08	0.3	0.5	0.7
chi2	good	98	116.9	14	19.33	-1.353	42	70	98
KS	good	1	0.8749	0.95	-12.51	0.1317	0.95	0.99	0.9999
mean	good	7	6.999	0.0008434	-0.008693	0.7215	0.00253	0.004217	0.005904
mode	good	6	6.001	0.165	0.02083	-0.007576	0.4949	0.8249	1.155
variance	good	6.997	7.137	1.183	2.008	-0.1188	3.548	5.914	8.279
quantile10	good	3.893	3.893	0.165	-0.01747	0.004123	0.4949	0.8249	1.155
quantile20	good	4.732	4.731	0.165	-0.0174	0.00499	0.4949	0.8249	1.155
quantile30	good	5.41	5.41	0.165	-0.00844	0.002768	0.4949	0.8249	1.155
quantile40	good	6.039	6.041	0.165	0.0238	-0.008714	0.4949	0.8249	1.155
quantile50	good	6.67	6.671	0.165	0.01419	-0.005737	0.4949	0.8249	1.155
quantile60	good	7.343	7.343	0.165	-0.00251	0.001117	0.4949	0.8249	1.155
quantile70	good	8.112	8.11	0.165	-0.01674	0.008232	0.4949	0.8249	1.155
quantile80	good	9.076	9.074	0.165	-0.01755	0.009657	0.4949	0.8249	1.155