RiemannianRobustMestimator Usage Guide

Last updated: 2026-02-22

This guide explains how to use **RiemannianRobustMestimator** to fit robust location (mean/center) estimators for manifold-valued data using gradient-based optimization.

Goal. Clear usage for both non-autograd and autograd workflows, including built-in M-estimators, optimizer options, initialization, and custom losses.

1) What it does

Given samples on a manifold with a metric that supports *log* / *exp*, the estimator minimizes a robust objective of the form below (notation shown for reference):

```
    L(μ) = (1/∑_i ω_i) ∑_{i=1}^n ω_i ρ(r_i(μ)),   r_i(μ)=||log_μ(x_i)||
    ∇_μ L = -(1/∑_i ω_i) ∑_i ω_i w(r_i) log_μ(x_i),   w(r)=ρ'(r)/r
```

2) Backends and prerequisites

Supported methods.

• **default** and **adaptive** work on the NumPy backend and autodiff backends.
• **autograd** requires an autodiff backend (geomstats *autograd* or *pytorch*) and is not available on pure NumPy.

Minimum metric operations. Your space.metric should provide log, exp, and a compatible norm/distance.

3) Inputs and outputs

Constructor inputs

Argument	Meaning
space	Equipped manifold (geomstats)
m_estimator	Built-in loss name or 'custom'
critical_value	Cutoff/scale parameter(s) for the loss (may be auto-filled)
method	Optimization method: 'default', 'adaptive', or 'autograd'
init_point_method	Initialization: 'first', 'mean-projection', 'midpoint'

fit inputs

• **X**: array-like, shape [n_samples, *metric.shape]
• **weights** (optional): array-like, shape [n_samples] (defaults to uniform)

Output: estimate_

Field	Meaning
x	Final estimate (fitted center)
losses	Loss values over iterations
bases	Base points over iterations
n_iter	Number of accepted iterations
time	Wall-clock time (seconds)

4) Quick start

Non-autograd (NumPy-compatible): method='default' or 'adaptive'

```
from geomstats.geometry.hypersphere import Hypersphere
from geomstats.learning.riemannian_robust_m_estimator import RiemannianRobustMestimator

space = Hypersphere(3)
X = space.random_point(200)

est = RiemannianRobustMestimator(
    space=space,
    method="default",               # or "adaptive"
    m_estimator="huber",
    critical_value=None,            # auto default for ~95% ARE (if available)
    init_point_method="mean-projection",
)

est.set(init_step_size=0.5, max_iter=1024, epsilon=1e-7, verbose=False)
est.fit(X)

mu_hat = est.estimate_.x
```

Autodiff: method='autograd' (requires autodiff backend)

```
est = RiemannianRobustMestimator(
    space=space,
    method="autograd",
    m_estimator="huber",
    critical_value=None,
    init_point_method="mean-projection",
)

est.set(init_step_size=0.2, max_iter=512, epsilon=1e-7)
est.fit(X)

mu_hat = est.estimate_.x
```

5) Built-in M-estimators and default critical_value

Accepted names: default, huber, pseudo-huber, cauchy, biweight, fair, hampel, welsch, logistic, lorentzian, correntropy, custom. ('default' maps to 'huber'.)

If critical_value is None and a built-in estimator is used, the code fills a default value (intended to target ~95% ARE under a reference model).

Loss	Default critical_value
huber / default	1.345
pseudo-huber	1.345
cauchy	2.3849
biweight	4.6851
fair	1.3998
hampel	1.35 (base scale)
welsch	2.9846
logistic	1.205
lorentzian	2.678
correntropy	2.1105

6) Optimization methods

default: standard Riemannian gradient descent. Current non-autograd path expects (loss, gradient).

adaptive: step size adaptation (Levenberg–Marquardt-style). Current path expects (loss, gradient).

autograd: gradients via autodiff; loss only needs to return a scalar.

7) Initialization (init_point_method)

Value	Meaning
first	Use X[0]
mean-projection	Project the Euclidean mean onto the manifold
midpoint	Choose a 'midpoint' sample by sorting along the first coordinate

8) Custom loss functions

A) Function-style custom loss (simple)

For method='default'/'adaptive', your function should be able to return (loss, grad) when requested.

```
def my_loss(space, points, base, critical_value, weights=None, loss_and_grad=True):
    # return loss only if loss_and_grad is False
    # return (loss, grad) if loss_and_grad is True
    ...

est = RiemannianRobustMestimator(space=space, method="default", m_estimator="custom")
est.set_loss(my_loss)
est.set(init_step_size=0.5, max_iter=1024, epsilon=1e-7)
est.fit(X)
```

B) Bound loss object (recommended for unified base-only calls)

A cleaner pattern is a loss object with bind(...) and __call__(base, return_grad=...). After bind, you can make a base-only callable loss_with_base(base). To fully unify, update the non-autograd optimizers to call this base-only function as well.

9) Optimizer hyperparameters (set)

Parameter	Meaning
init_step_size	Step size / learning rate
max_iter	Maximum iterations
epsilon	Stopping tolerance
verbose	Print progress periodically

```
est.set(init_step_size=0.2, max_iter=512, epsilon=1e-7, verbose=True)
```

10) Common pitfalls

• autograd method on NumPy backend is unsupported.

• non-autograd custom losses must match the expected signature and provide gradients.

• add a tiny epsilon in terms like c/r to avoid division by zero at r=0.

• some spaces/metrics (e.g., SPD) may need numerical safeguards in exp/log.