

# fn make\_laplace\_threshold

Michael Shoemate

This proof resides in “**contrib**” because it has not completed the vetting process.

Proves soundness of the implementation of `make_laplace_threshold` in `mod.rs` at commit `f5bb719` (outdated<sup>1</sup>).

Thresholded noise mechanisms may be parameterized along many different axes:

- key dtype: i8, i16, i32, i64, u8, u16, u32, u64, f32, f64, UBig, IBig, RBig
- metric dtype: i8, i16, i32, i64, u8, u16, u32, u64, f32, f64, UBig, IBig, RBig
- measure: max divergence, zero concentrated divergence
- distribution: laplace, gaussian

All parameterizations reduce to a single core mechanism that perturbs a signed big integers with noise sampled from the appropriate discrete distribution, and then thresholds and shuffles the result.

The implementation of this function constructs a random variable denoting the noise distribution to add, and then dispatches to the `MakeNoiseThreshold<DI, MI, MO>` trait which constructs the core mechanism and wraps it in pre-processing transformations and post-processors to match the desired parameterization.

## 1 Hoare Triple

### Precondition

#### Compiler-Verified

- generic DI implements trait `NoiseDomain`
- generic MI implements trait `Metric`
- generic MO implements trait `Measure`
- generic `DiscreteLaplace` implements trait `MakeNoiseThreshold`
- type (DI, MI) implements trait `MetricSpace`

#### User-Verified

None

---

<sup>1</sup>See new changes with `git diff f5bb719..7a4d8d3 rust/src/measurements/noise_threshold/distribution/laplace/mod.rs`

## Pseudocode

```
1 def make_laplace_threshold(  
2     input_domain: DI,  
3     input_metric: MI,  
4     scale: f64,  
5     threshold: DI_Atom,  
6     k: Option[i32],  
7 ) -> Measurement[DI, DI_Carrier, MI, M0]:  
8     return DiscreteLaplace(scale, k).make_noise_threshold(  
9         (input_domain, input_metric), threshold  
10 )
```

## Postcondition

**Theorem 1.1.** For every setting of the input parameters (`input_domain`, `input_metric`, `scale`, `threshold`, `k`, `DI`, `MI`, `M0`) to `make_laplace_threshold` such that the given preconditions hold, `make_laplace_threshold` raises an error (at compile time or run time) or returns a valid measurement. A valid measurement has the following properties:

1. (Data-independent runtime errors). For every pair of members  $x$  and  $x'$  in `input_domain`, `invoke(x)` and `invoke(x')` either both return the same error or neither return an error.
2. (Privacy guarantee). For every pair of members  $x$  and  $x'$  in `input_domain` and for every pair  $(d\_in, d\_out)$ , where `d_in` has the associated type for `input_metric` and `d_out` has the associated type for `output_measure`, if  $x, x'$  are `d_in`-close under `input_metric`, `privacy_map(d_in)` does not raise an error, and `privacy_map(d_in) = d_out`, then `function(x), function(x')` are `d_out`-close under `output_measure`.

*Proof.* We first construct a random variable `DiscreteLaplace` representing the desired noise distribution. Since `MakeNoiseThreshold.make_noise_threshold` has no preconditions, the postcondition follows, which matches the postcondition for this function.  $\square$