

# MakeNoise<DI, MI, MO> for DiscreteLaplace

Michael Shoemate

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

Proves soundness of the implementation of `MakeNoise` for `DiscreteLaplace` in `mod.rs` at commit `f5bb719` (outdated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

This is an intermediary compile-time layer whose purpose is to dispatch to either the integer or floating-point variations of the mechanism, depending on the type of data in the input domain.

It does this through the use of the `Nature` trait, which has concrete implementations for each possible input type. This layer makes interior layers simpler to work with, and does not have privacy implications. It also makes `make_laplace` easier to call, by simplifying the type signature.

## 1 Hoare Triple

### Precondition

#### Compiler-Verified

`MakeNoise` is parameterized as follows:

- DI implements trait `Domain`
- MI implements trait `Metric`
- MO implements trait `Measure`

The following trait bounds are also required:

- (DI, MI) implements trait `MetricSpace`
- DI\_Atom implements trait `Nature`. This trait encodes the relationship between the atomic data type and the type of the noise distribution that is compatible with it: `DI_Atom_RV1`. In Rust, this corresponds to the (ugly) `<DI::Atom as Nature>::RV<1>` type.
- DI\_Atom\_RV1 implements trait `MakeNoise`. That is, it must be possible to build the mechanism from this new equivalent distribution.

#### User-Verified

None

---

<sup>1</sup>See new changes with `git diff f5bb719..8b6d431 rust/src/measurements/noise/distribution/laplace/mod.rs`

## Pseudocode

```
1 # analogous to impl MakeNoise<DI, MI, MO> for DiscreteLaplace in Rust
2 class DiscreteLaplace:
3     def make_noise(self, input_space: tuple[DI, MI]) -> Measurement[DI, DI_Carrier, MI, MO]:
4         # an equivalent random variable specific to the atom dtype
5         rv_nature = DI_Atom.new_distribution(self.scale, self.k) #
6         # build a measurement sampling from this equivalent distribution
7         return rv_nature.make_noise(input_space) #
```

## Postcondition

**Theorem 1.1.** For every setting of the input parameters (`self`, `input_space`, `DI`, `MI`, `MO`) to `make_noise` such that the given preconditions hold, `make_noise` 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.* On line 7, `make_noise` has no preconditions, so irregardless of any prior logic, the postcondition of `make_noise` follows that the output is a valid measurement.  $\square$

The complexity in the type system here is designed to be free of privacy implications, to help simplify the core, privacy-sensitive implementation.