# Transport-Type Models

A CRPropa model in the DEMOS format: generator → transport → post

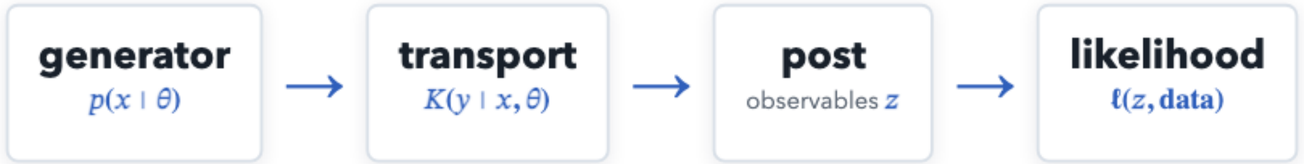
Mikhail Mikhasenko

2026-06-11

# The Use Case

A researcher has performed a **parameter-constraint analysis** with CRPropa (or Geant4). The physics model is buried in scripts, containers, and cluster jobs.

**DEMOS FORMAT – THE PROBABILITY MODEL**




**EXECUTION – STAYS WITH THE ENGINE (CRPROPA, GEANT4)**

sampling algorithm | number of events | seeds | batching | containers | CPU / GPU | cluster jobs

Present the **model building** in the DEMOS format – a transparent probability model, published with the paper, reusable as a pattern. The engine keeps taking care of the execution.

# Example-04 — A Minimal Transport Model

Pluto notebook (U. Hernandez Acosta, MM) – the smallest model with the full structure:

```
σ = 0.2
gun(; pars) = Normal(pars.μ, σ)   # source

function transport(x; pars)  # stochastic, heavy
    Δα = (2rand() + 1) * pars.a
    (x + Δα)^3 * exp(x - pars.b)
end

post(x) = x / 2                   # projection
```

```
function analysis(pars)
    gun_dist = gun(; pars)
    x = rand(gun_dist, 10_000)
    y = transport.(x; pars)
    z = post.(y)
end

glob_pars = (a = 0.1, b = 0.3, μ = 1.1)
```

```
x ~ gun(; pars)
y ~ transport(x; pars)
z ~ post(y)
```

Three stages — source distribution, stochastic transport kernel, observable projection — compared with data via a likelihood.

# The Format Separates Model from Execution

Example-04 shows the **typical workflow** – in a script, model and execution naturally interleave. In the format, each statement splits:

| Workflow (script) | Model – what is computed | Execution – how it is computed |
|---|---|---|
| `gun_dist = gun(; pars)` | defines $p(x \mid \theta)$ | – |
| `x = rand(gun_dist, 10_000)` | $x \sim p(x \mid \theta)$ | drawing algorithm, $N = 10\,000$, RNG seed |
| `y = transport.(x; pars)` | kernel $y \sim K(\cdot \mid x, \theta_T)$ | element-wise loop, batching, CPU/GPU |
| `z = post.(y)` | projection $z = \mathrm{post}(y)$ | vectorization, histogram filling |

The **model column** is the DEMOS description – the declarative computation graph. The **execution column** goes to the engine configuration – modular, exchangeable, reproducible.

# The Generator Describes a Distribution

In the model format we describe **distributions** – there is no operation to sample:

$$x \sim \text{generator}(\theta) \qquad \text{— a declaration, not a function call}$$

**Model** – the generator defines $p(x \mid \theta)$.
For CRPropa, a factorized source density:

$$p(x \mid \theta) = p(A, Z \mid \mathbf{f})\ p(E_0 \mid Z, \gamma, R_{\text{cut}})\ p(d_0 \mid m)$$

**Execution** – sampling happens here:

`rand(gen, N)` drawing algorithm number of events seed

The same distribution can also be reweighted or evaluated as a density – the engine chooses.

Already in example-04, `gun(; pars) = Normal(pars.μ, σ)` returns a distribution object – nothing is drawn until the execution stage.

# CRPropa in One Slide

Public framework for **UHECR transport**: nuclei injected at extragalactic sources, propagated through photon backgrounds (CMB, IRB) to an observer.


- Output contains **both** source and final particle IDs, energies, positions, directions, redshift, weight.
- Documented 1D example: sources at 1–1000 Mpc, $E_{\mathrm{min}} = 1$ EeV, mixed H / He / N / Fe injection.

CRPropa's own architecture **already matches** the generator → transport → post decomposition.

# Direct Correspondence

| DEMOS model concept | CRPropa interpretation |
|---|---|
| `pars` | source, transport, nuisance, and normalization parameters |
| `generator(; pars)` | initial cosmic-ray source distribution |
| `x ~ generator` | source species, energy, distance, redshift |
| `transport(x; pars)` | `Candidate` propagated through a `ModuleList` |
| `post(y)` | spectrum, composition, $X_{\max}$, … |
| `analysis(pars)` | generate / propagate events, return observables |
| `likelihood` | compare postprocessed observables with measurements |

One narrow adapter (`run_crpropa_*`) is the **only** framework-specific code – the rest of the model is engine-independent and reusable. Event records, distributions, kernel sketch → appendix.

# Two Execution Plans, One Model

## PLAN A – DIRECT: RERUN AT EVERY PARAMETER POINT

**generator(θ)** sample N events → **transport** CRPropa run → **post** histograms → **ℓ(θ)** likelihood

Transparent – but a CRPropa run for every $\gamma$, $R_{\mathrm{cut}}$, composition change.

## PLAN B – REFERENCE SAMPLE + IMPORTANCE REWEIGHTING

**proposal q(x)** broad, fixed → **transport** **once**, on cluster → **w = p(x|θ)/q(x)** per θ, cheap → **post + ℓ(θ)** weighted histograms

Same model, same observables – the expensive kernel leaves the inference loop.

The **model is unchanged** – the engine swaps execution plans. That is only possible because the generator is a *density*, not a sampling routine.

# Design Summary

**generator** – defines $p(x \mid \theta)$: distributions live in the **model**

**transport** – stochastic kernel $K(y \mid x, \theta_T)$: a narrow adapter to the CRPropa container

**post + likelihood** – observable projection and comparison with data: the **analysis model**

**The DEMOS format captures the probability model – the CRPropa engine takes care of the execution layer.**

# Appendix

# From Scalars to Event Records

In example-04, $x$, $y$, $z$ are scalars. For a real framework they become **structured event records**:

$$x = (A,\, Z,\, E_0,\, d_0,\, z_0), \qquad y = (A_f,\, Z_f,\, E_f,\, \ldots), \qquad z = \text{histograms}$$

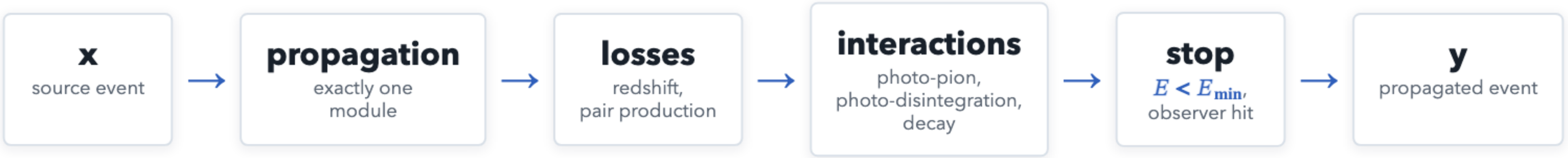
**x – source event**
species, injection energy, distance, redshift, weight

**y – propagated event**
final species, final energy, detection flag, trajectory length

**z – prediction**
spectrum, composition vs energy, $\langle \ln A \rangle$, $\mathrm{Var}(\ln A)$ – the pipeline shape stays **identical**, only the event type gets richer

# Transport Kernel — A Sketch, Not Code

$y \sim K_{\mathrm{CRPropa}}(\cdot \mid x, \theta_T)$ – each `Candidate` walks through the module stack until a stopping condition:




- Stochastic interactions change $(A, Z, E)$; continuous losses change $E$; the loop repeats per step.
- The **Observer** records the final state – detected or lost.

$\theta_T$ (photon field, step limits, $E_{\mathrm{min}}$) is **fixed per simulation** – changing interaction models or backgrounds means a new CRPropa run.

# Projection and Likelihood

**Projection** $z = \mathrm{post}(y)$ – weighted histograms over detected events:

**spectrum**
expected counts $\mu_i(\theta)$ per energy bin

**composition**
$\langle \ln A \rangle_i$, $\mathrm{Var}(\ln A)_i$ vs energy

**later**
$X_{\max}$ distributions

**Likelihood** – compare with measurements:

$$\ell(\theta) = \sum_i \log \mathrm{Pois}(n_i \mid \mu_i(\theta))$$

$$+ \sum_i \log \square\left(\langle \ln A \rangle_i^{\mathrm{obs}} \mid m_i(\theta),\, \sigma_i\right)$$

Poisson counts for the spectrum, Gaussian constraints for composition moments.

# Generator — Distributions to Support

| Quantity | Distribution (model) | CRPropa source feature (engine) |
|---|---|---|
| Fixed species | Dirac (conceptually) | `SourceParticleType` |
| Species mixture | `Categorical` | `SourceMultipleParticleTypes` / `SourceComposition` |
| Fixed energy | Dirac (conceptually) | `SourceEnergy` |
| Power-law energy | `Pareto` / truncated power law | `SourcePowerLawSpectrum` |
| Custom cutoff spectrum | custom density | `SourceGenericComposition` |
| Fixed position | deterministic value | `SourcePosition` |
| Uniform 1D distance | `Uniform` | `SourceUniform1D` |
| Fixed redshift | deterministic value | `SourceRedshift` |
| Distance-derived redshift | deterministic transform | `SourceRedshift1D` |
| Isotropic direction | uniform sphere sampler | `SourceIsotropicEmission` |
| Directed emission | von-Mises–Fisher-like | `SourceDirectedEmission` |

Keep the distributions in the **model**; the engine maps them to source features – it never needs to understand `Normal`, `Dirichlet`, mixtures, or hierarchies.

# The Execution Layer — the Engine's Job

**Sampling**
drawing algorithm, RNG streams, seeds

**Statistics**
how many events per parameter point

**Batching**
serialize events, invoke the container **once**

**Parallelism**
candidates are independent → shared-memory / cluster

**Hardware**
CPU now, GPU when the engine supports it

**Caching**
reference samples, reweighting, lookup tables

None of this is physics – but reproducibility dies if it stays unspecified. The format should fix it **modularly, separate from the model**.

# Parameter Blocks

Split by what they control — the source, the expensive transport, or the observation model:

```
source_pars = (
    γ       = 1.5,
    logRcut = 18.8,        # log10(Rcut / V)
    fractions = (
        H  = 0.25, He = 0.25,
        N  = 0.25, Fe = 0.25,
    ),
    source_evolution = 0.0,
)
observation_pars = (
    flux_normalization = 1.0,
    energy_scale_shift = 0.0,
)
```

```
transport_pars = (
    dmin = 1.0,            # Mpc
    dmax = 1000.0,         # Mpc
    Emin = 1.0,            # EeV
    photon_field = :Kneiske04,
    minimum_step = 1e-3,   # Mpc = 1 kpc
    maximum_step = 10.0,   # Mpc
)

pars = (;
    source_pars...,
    transport = transport_pars,
    observation = observation_pars,
)
```

# Priors (Inference Specification)

```julia
priors = (
    γ                 = Uniform(0.0, 3.0),
    logRcut           = Uniform(18.0, 20.5),
    source_evolution  = Uniform(-5.0, 8.0),
    # Distributions.jl provides Dirichlet for simplex-valued fractions
    fractions         = Dirichlet(fill(1.0, 4)),
    energy_scale_shift = Normal(0.0, 0.14),
)
```

**Free** – source spectral index, rigidity cutoff, evolution, composition fractions; energy-scale nuisance.

**Fixed** – transport settings: changing interaction models, magnetic fields, or photon backgrounds requires a **new CRPropa simulation**.

# Event-Record Types

```
struct SourceEvent
    A::Int
    Z::Int
    energy::Float64     # EeV
    distance::Float64   # Mpc
    redshift::Float64
    weight::Float64
end
```

```
struct PropagatedEvent
    initial::SourceEvent
    final_A::Int
    final_Z::Int
    final_energy::Float64
    detected::Bool
    trajectory_length::Float64
    final_redshift::Float64
    weight::Float64
end
```

In the official example `SourceUniform1D` sets the distance and `SourceRedshift1D` then derives the redshift – the order matters.

# Sampling $p(x \mid \theta)$ — Execution Layer

```julia
const SPECIES = ((:H, 1, 1), (:He, 4, 2), (:N, 14, 7), (:Fe, 56, 26))  # name, A, Z

function sample_powerlaw_cutoff(rng, γ, Emin, Emax, Z, Rcut)
    proposal = truncated(Pareto(Emin, max(γ - 1, 1e-3)), Emin, Emax)
    while true  # rejection sampler for p(E) ∝ E^(-γ) exp[-E / (Z Rcut)]
        E = rand(rng, proposal)
        rand(rng) < exp(-E / (Z * Rcut)) && return E
    end
end

function sample_source(rng = Random.default_rng(); pars)
    name, A, Z = SPECIES[rand(rng, Categorical(collect(pars.fractions)))]
    Rcut = 10.0^(pars.logRcut - 18)            # EV ≡ EeV per unit charge
    E = sample_powerlaw_cutoff(rng, pars.γ, pars.transport.Emin, 1000.0, Z, Rcut)
    d = rand(rng, Uniform(pars.transport.dmin, pars.transport.dmax))
    SourceEvent(A, Z, E, d, distance_to_redshift(d), 1.0)
end
```

One possible drawing algorithm for the generator – the model itself only fixes the density.

# Inside the Adapter

## Essentially the module list of the official 1D example:

```
sim = ModuleList()
sim.add(SimplePropagation(1 * kpc, 10 * Mpc))   # exactly ONE propagation module
sim.add(Redshift())
sim.add(PhotoPionProduction(CMB()))
sim.add(PhotoPionProduction(IRB_Kneiske04()))
sim.add(PhotoDisintegration(CMB()))
sim.add(PhotoDisintegration(IRB_Kneiske04()))
sim.add(NuclearDecay())
sim.add(ElectronPairProduction(CMB()))
sim.add(ElectronPairProduction(IRB_Kneiske04()))
sim.add(MinimumEnergy(1 * EeV))

observer = Observer()
observer.add(ObserverPoint())
sim.add(observer)
```

One propagation module is required; the others implement redshift losses, stochastic interactions, continuous losses, decays, stopping conditions, and observation.

# Batched Transport

## Pedagogically true:

```
y = transport.(x; pars)
```

**But implemented batched** – never call the container once per event:

```
function transport(xs::Vector{SourceEvent}; pars)
    run_crpropa_batch(xs, pars.transport)
end

xs = sample_source_batch(10_000; pars)
ys = transport(xs; pars)
```

**1** serialize source events – Arrow, HDF5, JSONL, binary

**2** invoke the CRPropa container **once**

**3** propagate events in parallel (candidates are independent → shared-memory parallelism)

**4** return the final event table

# `post()` Implementation

```julia
struct Prediction
    energy_edges::Vector{Float64}
    expected_counts::Vector{Float64}
    mean_logA::Vector{Float64}
    variance_logA::Vector{Float64}
end
```

```julia
function analysis(pars; nevents = 10_000,
                  energy_edges)
    x = sample_source_batch(nevents; pars)
    y = transport(x; pars)
    post(y; energy_edges,
         normalization =
             pars.observation.flux_normalization)
end
```

```julia
function post(events; energy_edges,
              normalization = 1.0)
    det = filter(e -> e.detected, events)
    logE = [log10(e.final_energy) + 18 for e in det]
    logA = [log(e.final_A) for e in det]
    nb = length(energy_edges) - 1
    counts, s1, s2 = (zeros(nb) for _ in 1:3)
    for (E, ℓA, e) in zip(logE, logA, det)
        i = searchsortedlast(energy_edges, E)
        1 ≤ i ≤ nb || continue
        w = e.weight
        counts[i] += w; s1[i] += w*ℓA; s2[i] += w*ℓA^2
    end
    m = s1 ./ counts
    Prediction(energy_edges, normalization .* counts,
               m, s2 ./ counts .- m .^ 2)
end
```

# Likelihood Implementation

```
data = (
    counts = [...],
    mean_logA = [...],
    mean_logA_errors = [...],
)

function evaluate(pars, data;
                  nevents, energy_edges)
    pred = analysis(pars;
        nevents, energy_edges)
    loglikelihood(pred, data)
end
```

```
function loglikelihood(pred, data)
    ℓ_spectrum = sum(
        logpdf(Poisson(max(μ, eps())), n)
        for (μ, n) in zip(
            pred.expected_counts,
            data.counts))
    ℓ_composition = sum(
        logpdf(Normal(μ, σ), x)
        for (μ, σ, x) in zip(
            pred.mean_logA,
            data.mean_logA_errors,
            data.mean_logA))
    ℓ_spectrum + ℓ_composition
end
```

# Reference Sample + Reweighting (Code)

```julia
function proposal_source(rng = default_rng())
    name, A, Z = SPECIES[rand(rng, 1:4)]
    logE = rand(rng,                  # q(E) ∝ 1/E
        Uniform(log(1.0), log(1000.0)))
    d = rand(rng, Uniform(1.0, 1000.0))
    SourceEvent(A, Z, exp(logE), d,
                distance_to_redshift(d), 1.0)
end

# propagate ONCE
x_ref = [proposal_source() for _ in 1:1_000_000]
y_ref = transport(x_ref; ref_transport_pars)
```

```julia
function source_weight(x; pars)
    target_density(x; pars) /
        proposal_density(x)
end

function analysis_reweighted(pars, x_ref, y_ref;
                             energy_edges)
    weighted = map(x_ref, y_ref) do x, y
        @set y.weight = source_weight(x; pars)
    end
    norm = pars.observation.flux_normalization
    post(weighted; energy_edges,
         normalization = norm)
end
```

# Recommended Notebook Structure

```
## Parameters
   physical source parameters, fixed transport settings, nuisance parameters

## Source model
   x ~ generator(; pars)

## CRPropa transport kernel
   y ~ transport(x; pars.transport)

## Postprocessing
   z = post(y; pars.observation)

## Likelihood
   data ~ observation_model(z)

## Full analysis
   prediction, log-likelihood
```