Kongming HV

PDF download

Kongming is a hyperdimensional computing library implementing sparse binary hypervectors for cognitive computing applications.

The Python package of `kongming-rs-hv` is released under MIT license, essentially no limitation (and assumes no liability) to any use, personal or commercial.

The core engine is implemented in **Go** / **Rust** for maximum efficiency, while ergonomic APIs are open-sourced in **Python**, for better usability.

We do have an internal Go/Rust APIs that interacts directly with the core engine: overall all APIs maintains minimalistic abstractions and wire-identical serialization.

See Hypervectors for an introduction to hyperdimensional computing and the sparse binary representation.

License

MIT License

Install

```
pip install kongming-rs-hv
```

See Installation for supported platforms and verification steps.

Published notebooks

See Notebook Platforms for all available notebooks and platform details.

Guides

Guide	Description
Python Quick Start	Installation, examples, and walkthrough
Notebook Quick Start	Platform setup, interactive notebooks, cell-by-cell walkthrough

Language Support

This documentation covers code snippets in multiple languages (if available) side by side.

- **Python**: bindings to the underlying Rust implementation (public `kongming-rs-hv` on PyPI);
- **Go**: canonical / reference implementation in proprietary package;
- **Rust**: parallel implementation, carefully maintained in feature parity;

Reference

The work was initially outlined in this arxiv paper, built on top of the work from many others, and here is the citation:

Yang, Zhonghao (2023). Cognitive modeling and learning with sparse binary hypervectors. arXiv:2310.18316v1 [cs.AI]

Feedback

Found a bug, have a question, or want to suggest an improvement? Open an issue on GitHub.

Last change: 2026-04-06, commit: 63ad966

Hypervectors

What is Hyperdimensional Computing?

Hyperdimensional computing (HDC) represents concepts as high-dimensional vectors and manipulates them with simple algebraic operations, typically the dimension (of any vectors) can be as high as thousands.

The key insight is that random vectors in high-dimensional spaces are **nearly orthogonal** — giving each concept a unique, distributed, and robust representation that tolerates potential ambiguity and interference.

In that sense, the traditional notion of curse of dimensionality becomes the bless of dimensionality.

Motivated readers should perform their own background research on this topic.

Sparse Binary Representation

Kongming uses **sparse binary** hypervectors. Each vector has a fixed, large number of dimensions (e.g., 65,536/64K or 1,048,576/1M), but only a very small fraction of them are “on” (set to 1). This sparsity is controlled by the Model configuration.

Furthermore, we focus on a special sparse binary configuration: **SparseSegmented** where each vector is divided into equal-sized *segments*, and exactly one bit is ON per segment.

Conceptually you can imagine each **SparseSegmented** hypervector as a list of phasers, where the offset of ON bit (within a host segment) represents the discretized phase.

In general, this unique constraint enables:

- **Compact storage**: only the offset of ON bit within its host segment need to be stored
- **Efficient operations**: Unlike neural nets, where weights are recorded in float numbers, binary operations can be stored and manipulated very efficiently with modern memory / CPUs.

Identity and Inverses

- The **identity** vector has all offsets set to 0. Binding with identity is a no-op. Actually as a special case, there is no storage cost;
- Binding a vector with its inverse yields the identity.

Similarity and distance measure

Two vectors are compared via **overlap** — the count of segments where both have the same ON bit. This is equivalent to a bitwise AND operation, which can be performed very efficiently in modern CPU.

For a model with cardinality k and segment size s, the expected overlap between two random vectors A and B is:

$$\mathrm{E}[O(A, B)] = Ns = 1$$

Given the model setup, this is typically 0, 1 or 2.

Semantically-related vectors have significantly higher overlap. A vector's overlap with itself equals its cardinality M.

The commonly-used distance measure (dis-similar measure) for binary vectors is Hamming Distance, equivalent to a bitwise XOR operation. As we discussed (and proved) in the paper, the **overlap** and **Hamming distance** between **sparse binary** hypervectors are two sides of the same coin, with the following equation:

$$2 \times O(A, B) + H(A, B) = 2M$$

Supported Models

A Model determines the total number of dimensions (width), how those dimensions are divided into segments (cardinality and sparsity), and therefore implies critical storage and compute characteristics.

Model	Width	Sparsity Bits	Segment Size	Carc (Ol
`MODEL_64K_8BIT`	65,536	8	256	256
`MODEL_1M_10BIT`	1,048,576	10	1,024	1,02
`MODEL_16M_12BIT`	16,777,216	12	4,096	4,09
`MODEL_256M_14BIT`	268,435,456	14	16,384	16,3
`MODEL_4G_16BIT`	4,294,967,296	16	65,536	65,5

Model properties

All model functions take a Model enum value and return the derived property:

> ℹ️ **Note**
>
> For simplicity, we use function names from Python. The counterparts from Go / Rust can be found by consulting their respective references.

Function	Description
`width`	Total dimension count (`2^width_bits`)
`sparsity`	Fraction of ON bits (`1 / segment_size`)
`cardinality`	Number of ON bits (= number of segments)
`segment_size`	Dimensions per segment

How to Choose a Model

- `MODEL_64K_8BIT` : Fast prototyping, tiny memory footprint. Good for tests and small-scale experiments.
- `MODEL_1M_10BIT` : General-purpose, balances performance and storage.
- `MODEL_16M_12BIT` : General-purpose, for the adventurous.
- `MODEL_256M_14BIT` / `MODEL_4G_16BIT` : Very high capacity, not there yet.

Larger models provide more orthogonal space (lower collision probability) at the cost of more memory per vector.

> ℹ️ **Note**
>
> The storage per hypervector estimation only applies to **SparseSegmented** (and a few other types) where raw offsets are needed. For certain scenarions, optimization can be employed to dramatically reduce storage requirements. **Sparkle**, for example, only stores the random seeds so that the offsets can be recovered on-the-fly at serialization time. Composite types (such as **Set**, **Sequence**) typically contain references to member **Sparkle** instances, and typically cost much less storage than a single **SparseSegmented** instance.

Last change: 2026-04-06, commit: 63ad966

Operators

Kongming provides two core algebraic operations on hypervectors.

Bind

Binding ($\otimes$) combines two vectors into a result that is dissimilar to both inputs. It is the multiplicative operation in the HDC algebra.

Mathematically

$$A \otimes B = B \otimes A \quad (\text{commutative})$$

$$(A \otimes B) \otimes C = A \otimes (B \otimes C) \quad (\text{associative})$$

$$A \otimes I = A \quad (\text{where I is an identity vector})$$

$$A \otimes A^{-1} = I \quad (\text{inverse})$$

$$O(A \otimes B, A) \approx O(A \otimes B, B) \approx \text{noise} \quad (\text{dissimilarity})$$

Implementation: segment-wise offset addition modulo segment size: check out original paper for details.

Check out code snippets from the API reference.

Release

Occasionally we use **release**, which is derived from **bind**, as the equivalent of division, as opposed to multiplication.

$$A \oslash B = A \otimes B^{-1}$$

Note that release is anti-commutative:

$$(A \oslash B)^{-1} = B \oslash A$$

Check out code snippets from the API reference.

Bundle

Bundling ($\oplus$) creates a superposition of vectors — the result is similar to all inputs. It is the additive operation within VSA algebra.

Mathematically

$$S = \sum_{i,\oplus} A_i$$

$$O(S, A_i) \gg O_{\text{random}} \quad (\text{similarity to each member})$$

$$O(S, X) \approx O_{\text{random}} \quad \text{for } X \notin \{A_i\} \quad (\text{dissimilarity to non-members})$$

Check out original paper for details on bundle operator.

Check out code snippets from the API reference.

Last change: 2026-04-06, commit: 63ad966

Composites

Composites combine multiple hypervectors into higher-level structures. Each composite type uses a different combination strategy, preserving different kinds of relationships between its members.

All composites follows the same contract (interface in Go and traits in Rust) and can be nested — a Set can contain Sparkles, Knots, or even other Sets.

Set

An **unordered** collection of concepts.

$$S = S_{marker} \otimes (\sum_{i,\oplus} M_i)$$

where S_{marker} is a special marker to distinguish a set from its individual members.

This mark is tuned for the domain, so that it will be shared among all sets within the same domain.

Use when: you need to represent “these things together” without order.

Check out code snippets from the API reference.

Sequence

An **ordered** collection.

$$S = S_{marker} \otimes (\sum_{i,\oplus} M_i \otimes S_{step}^i)$$

where S_{step} is a generic hypervector for positional encoding.

S_{marker} is a special marker to distinguish a sequence from its individual members. This mark is tuned for the domain, so that it will be shared among all sequences within the same domain.

Use when: order matters (e.g., words in a sentence, events in time).

Check out code snippets from the API reference.

Octopus

A **key-value** structure. Each key (a string) is converted to a Sparkle and bound with its corresponding value before bundling.

$$S = \sum_{i,\oplus} K_i \otimes V_i$$

Use when: you need to represent structured records with named attributes.

Check out code snippets from the API reference.

Knot

The result of **binding** (multiplicative composition) of hypervectors.

$$S = \prod_{i,\otimes} M_i$$

Binding is reversible: given a Knot of A and B, you can recover A by releasing B (binding with B's inverse).

Use when: you need a reversible association between concepts.

Check out code snippets from the API reference.

Parcel

The result of **bundling** (additive composition).

$$S = \sum_{i,\oplus} M_i$$

Unlike direct bundling, Parcel keeps tracking of its members for serialization and introspection.

Use when: you need a superposition of concepts, with optional weights.

Check out code snippets from the API reference.

Summary

Type	Composition	Order?	Use Case
Set	Bundle + marker	No	Unordered groups
Sequence	Positional-bind + bundle + marker	Yes	Ordered lists
Octopus	Key-bind + bundle	Partial (by key)	Key-value records
Knot	Bind (multiply)	No	Reversible associations
Parcel	Bundle (add)	No	superpositions, weighted or unweighted

Last change: 2026-04-06, commit: 63ad966

Near Neighbor Search

Near Neighbor Search (NNS) generally retrieves chunks from the storage substrate in the increasing order of Hamming distance (from a query).

As we mentioned earlier, this is equivalent to a strictly decreasing order of overlap (between query and candidate). If overlap encodes the semantic relevance, this translates to a list of semantically similar candidates.

It leverages an underlying Associative Index for efficient recovery of candidates. The **Associative Index** is a semantic index that enables fast similarity-based lookup over stored hypervectors. Conceptually it turns a key-value substrate (item memory) into an associative memory — one where retrieval is by *content similarity*, not by exact content or key match.

This NNS module has a constant time complexity, with help from associative index. This implies the query time remain bounded, independent of the number of entries in the storage system. The secret sauce is the efficient random-access to underlying associative index.

Unlike approximate nearest neighbor methods (LSH, HNSW, etc.), the NNS module can computes **exact** overlap counts via the associative index. There is no approximation error and no index-specific parameters to tune.

Jump to the API reference for Near-Neighbor Search.

Last change: 2026-04-06, commit: 63ad966

HV

The core hypervector API. This module provides the building blocks for hyperdimensional computing: vector types, algebraic operators, and model configuration.

Section	Description
Common Utilities	Model, SparseOperation, similarity, identity, hashing
Operators	Bind, bundle, and BindDirect
HyperBinary Types	Interface + concrete types (Sparkle, Set, Sequence, etc.)
Customizing Run-time Behavior	Environment variables
Misc	Display, serialization

Last change: 2026-04-01, commit: edabb74

Common Utilities

Functions and types used across all HyperBinary types.

Section	Description
Models	Model enum and model functions
SparseOperation	Model + seeded RNG for deterministic vector generation
Seed128	128-bit seed embedding Domain + Pod
Domain & Pod	Semantic grouping (Domain) and slot identifier (Pod)
Utilities	Similarity, identity check, hashing

Last change: 2026-04-06, commit: fff9e78

Models

See Concepts: Hypervectors for the full overview.

Model Enum

Python Go Rust

```
model0 = hv.MODEL_64K_8BIT

model1 = hv.MODEL_1M_10BIT
```

Model Functions

Python Go Rust

```
hv.width(hv.MODEL_1M_10BIT)           # total dimensions
hv.cardinality(hv.MODEL_1M_10BIT)     # ON bit count
hv.sparsity(hv.MODEL_1M_10BIT)        # sparsity
hv.segment_size(hv.MODEL_1M_10BIT)    # dimensions per segment
```

See also: SparseOperation — Model + seeded RNG for deterministic vector generation.

Last change: 2026-04-04, commit: 34eaa34

Domain & Pod

A `Domain` models the semantic grouping for hypervectors, providing the high 64-bit half of a `Seed128`. A `Pod` is a slot within a Domain, providing the low 64-bit half. The (Domain, Pod) pair uniquely identifies a Sparkle.

Domain Constructors

Python Go Rust

```python
# From a name string (hashed to a 64-bit id)
d = hv.Domain("animals")

# Same as above
d = hv.Domain.from_name("animals")

# From a raw 64-bit id
d = hv.Domain.from_id(0x1234567890abcdef)

# From a domain prefix enum and a name suffix
# The id is computed as xxhash(prefix_label + "." + name)
d = hv.Domain.from_prefix_and_name(hv.DOMAIN_PREFIX_NLP, "concept")

# Accessors
d.id()              # u64
d.name()            # str (empty if constructed from id)
d.domain_prefix()   # int (0 = UNKNOWN if no prefix was set)
d.is_default()      # True if id == 0
```

Domain Prefix Constants

Constant	Label
hv.DOMAIN_PREFIX_USER	🎭

Constant	Label
`hv.DOMAIN_PREFIX_NLP`	💬

Domain prefixes provide namespacing for domains. When a prefix is set, the domain id is derived from the prefix label (and optional name), ensuring consistent hashing across languages.

Pod Constructors

Pods can be seeded by a string word, a raw uint64, or a prewired enum value.

Python Go Rust

```python
# From a word string (hashed to a 64-bit seed)
p = hv.Pod("cat")

# Same as above
p = hv.Pod.from_word("cat")

# From a raw 64-bit seed
p = hv.Pod.from_seed(42)

# From a prewired enum value
p = hv.Pod.from_prewired(hv.PREWIRED_SET_MARKER)
p = hv.Pod.from_prewired(hv.PREWIRED_STEP)

# Accessors
p.seed()       # u64
p.word()       # str (empty if constructed from seed or prewired)
p.prewired()   # int (0 if not prewired)
p.is_default() # True if seed == 0
```

Prewired Constants

Prewired pods are infrastructure-level constants with fixed seeds:

Constant	Label
hv.PREWIRED_NIL	∅
hv.PREWIRED_FALSE	❌
hv.PREWIRED_TRUE	✅
hv.PREWIRED_BEGIN	🚀
hv.PREWIRED_END	🏁
hv.PREWIRED_LEFT	⬅️
hv.PREWIRED_RIGHT	➡️
hv.PREWIRED_UP	⬆️
hv.PREWIRED_DOWN	⬇️
hv.PREWIRED_MIDDLE	⏺️
hv.PREWIRED_STEP	□
hv.PREWIRED_SET_MARKER	🫧
hv.PREWIRED_SEQUENCE_MARKER	📿

Last change: 2026-04-06, commit: 14bc3e4

Seed128

A `Seed128` is a 128-bit seed to drive a random number generator.

The current random number generator expects 2 64-bit seeds: the same (seed_high, seed_low) pair always produces the same sequence of random numbers, enabling reproducible and deterministic vector generation across runs and languages.

Constructors

Python Go Rust

```python
# From Domain and Pod arguments (each accepts Domain/Pod, int, or
str)
seed = hv.Seed128("animals", "cat")                # domain name +
pod word
seed = hv.Seed128(0, 42)                           # default domain
+ raw pod seed
seed = hv.Seed128("animals", 42)                   # domain name +
raw pod seed
seed = hv.Seed128(hv.Domain("animals"), hv.Pod("cat"))  # explicit
Domain/Pod objects

# Zero seed
seed_zero = hv.Seed128.zero()                      # (0, 0)

# Random seed from a SparseOperation
seed_rand = hv.Seed128.random(so)                  # consumes two
u64 from the RNG

# Accessors
seed.domain()                                      # Domain object
seed.pod()                                         # Pod object
seed.high()                                        # u64 (domain id)
seed.low()                                         # u64 (pod seed)
```

Usage

All composite constructors take a `Seed128`, as seed for the bundle operator:

Python Go Rust

```
seed = hv.Seed128("fruits", "fruit_set")

s = hv.Set(seed, a, b, c)
seq = hv.Sequence(seed, a, b, c)
```

Last change: 2026-04-06, commit: c19acaa

SparseOperation

A SparseOperation instance wraps a Model, a random number generator, and potentially other information related to the sparse operation in general.

Constructor

Python Go Rust

```python
so = hv.SparseOperation(hv.MODEL_1M_10BIT, 0, 42)
so1 = hv.SparseOperation(hv.MODEL_1M_10BIT, "domain", "pod")
```

Methods

Python Go Rust

```python
so.model()        # Model enum

so.width()        # width for this model

so.cardinality()  # cardinality for this model

so.sparsity()     # sparsity for this model

so.uint64()       # next random number
```

Usage: Generating Random Vectors

Python Go Rust

```python
so = hv.SparseOperation(hv.MODEL_1M_10BIT, 0, 42)
sparkle = hv.Sparkle.random(hv.Domain("domain"), so)
```

Last change: 2026-04-04, commit: 34eaa34

Utilities

Similarity

Python Go Rust

```
hv.overlap(a, b)    # Overlap
hv.hamming(a, b)    # Hamming distance
hv.equal(a, b)      # Equality check
```

Identity Check

Python Go Rust

```
v=hv.Sparkle.identity(model)
hv.is_identity(v)   # True if v is an identity vector
```

Hash Utilities

Python Go Rust

```
hv.hash64_from_string("hello")   # deterministic u64 hash from
string
hv.hash64_from_bytes(b"\x01\x02") # deterministic u64 hash from
bytes
hv.curr_time_as_seed()            # current time as a u64 seed
hv.kongming_studio_seed()         # fixed studio seed constant
```

Last change: 2026-04-04, commit: 34eaa34

HyperBinary Types

All vector types conform to a common interface. In Go this is the `HyperBinary` interface; in Rust it is the `HyperBinary` trait. The two implementations are kept at **feature parity**.

Python Go Rust

Python doesn't have the concept of interface/trait, but all `HyperBinary` derived types share a common set of methods.

```
v.model()         # Model enum
v.width()
v.cardinality()
v.hint()
v.stable_hash()   # int
v.seed128()
v.exponent()

v.core()          # SparseSegmented
v.power(p)        # HyperBinary
```

Concrete Types

Type	Description
SparseSegmented 🍡	Foundational vector — packed per-segment offsets
Sparkle ✨	Seeded, deterministic hypervector
Learner 💫	Online Hebbian learning
Set 🫧	Unordered collection
Sequence 📿	Ordered collection with positional encoding
Octopus 🐙	Key-value composite

Type	Description
Knot 🪢	Bound (multiplied) group
Parcel 🎁	Bundled (added) group

Last change: 2026-04-08, commit: 03128b0

SparseSegmented 🍡

The most foundational vector type — a sparse binary hypervector where each segment has exactly one ON bit at the recorded offset location. All other types (Sparkle, Set, Sequence, etc.) ultimately contain a `SparseSegmented` in memory for processing.

Structure

Field	Description
`model`	Sparsity configuration (Model)
`offsets`	Packed bit array of per-segment ON offsets. `nil` / `None` = identity vector
`hash`	Lazy-computed stable hash for equality checks

The offsets are bit-packed according to the model's sparsity bits — they do **not** align to byte boundaries. This trades a small CPU cost for compact, uniform storage that works both in memory and on disk.

Identity vector: when `offsets` is blank, the vector is the identity vector where all offsets are 0. Binding with identity is a no-op, and identity requires zero storage for offsets.

Constructors

Python Go Rust

```python
# Identity
ss = hv.SparseSegmented.identity(model)

# From raw offsets, typically discouraged...
ss = hv.SparseSegmented(model, offsets_bytes)
```

Key Methods

Python Go Rust

```python
ss.is_identity()  # True if identity vector

ss2 = ss.power(2)
inv = ss.power(-1)

# Similarity
hv.overlap(a, b)   # Count of matching ON bits
hv.hamming(a, b)   # Count of differing segments

ss.offsets()   # returns all offsets
```

Serialization

SparseSegmented serializes to `HyperBinaryProto` with hint `SPARSE_SEGMENTED`. The `offsets` field carries the raw packed bytes. Identity vectors serialize with empty offsets.

Last change: 2026-04-06, commit: 63ad966

Sparkle ✨

Sparkles are the atomic building block for higher-level constructs: essentially SparseSegmented annotated with domain and pod.

Domain is a logical namespace that groups related Sparkle instances. Pod acts as the secondary identifier for a Sparkle instance.

Sparkle is **deterministic**: the same (domain, pod) pair always produces the same offsets. For this reason, the (model, pod) pair uniquely identifies a Sparkle.

Sparkle Constructors

Python Go Rust

```python
# From a word string
s0 = hv.Sparkle.from_word(model, "animals", "cat")

# From a numeric seed
s1 = hv.Sparkle.from_seed(model, "animals", 42)

# From a prewired enum
s2 = hv.Sparkle.from_prewired(model, "animals",
hv.PREWIRED_SET_MARKER)

# Identity vector
s3 = hv.Sparkle.identity(model)

# Random (from SparseOperation)
so=hv.SparseOperation(hv.MODEL_1M_10BIT, "domain", "pod")
s4 = hv.Sparkle.random("animals", so)
```

Key Methods

Python Go Rust

```
s0.model()          # Model enum
s0.stable_hash()    # Deterministic hash
s0.exponent()       # Current exponent (1 for base vector)

s0_square=s0.power(2)      # Returns p-th power (new Sparkle)
hv.equal(s0, s0_square)    # s0_square = s0^2, different from
original s0.

core0=s0.core()      # Returns underlying SparseSegmented
core0.offsets()    # The raw offsets for each segment.
```

> ℹ️ **Note**
>
> `power(0)` always returns the identity sparkle. `power(-1)` returns the inverse.

Pretty-printing

Python

```
# Pretty-printing, or s.__str__()
print(s0)
# ✨:🔗animals,🌱cat

# More detailed information, or s.__repr__()
s
# hint: SPARKLE
# model: MODEL_1M_10BIT
# stable_hash: 9725717137035622833
# domain:
#   name: animals
# pod:
#   word: cat
```

During pretty-printing of Sparkle instances, you may notice special emoji for domain / pods.

💡 **emojis for domain / pod**

Emoji	Variant	Example
🔗	named domain	🔗`animals`, 🔗`PREFIX.name`
🌐	numeric domain	🌐`0x..c862`
🌱	named pod	🌱`cat`
🫛	numeric pod	🫛`0x..80e4`
🍀	pre-defined pod	🍀`SET_MARKER`
💪	Exponent / Power	💪`3`, 💪`-1`

Identity vectors display as `IDENT` (e.g., ✨`IDENT`).

> ℹ️ **Note**
>
> The underlying offsets are lazily generated from a seeded PRNG. Only the seeds are stored in serialization, which is a significant storage saving; offsets are recomputed during de-serialization.

Last change: 2026-04-06, commit: 63ad966

Learner 💫

Learners are designed to perform online bundling for a stream of observations, in the form of Hebbian-style learning.

The total storage / processing budget is fixed — what matters is the distribution of weights among observed vectors.

Constructors

Python Go Rust

```python
learner = hv.Learner(model, hv.Seed128(0, 42))

# a randomly-initialized learner.
learner = hv.Learner.random(so)
```

Feeding Observations

Python Go Rust

```python
learner.bundle(a)                  # single observation

learner.bundle_multiple(b, 3)      # with weight multiplier
```

Inspection

Python Go Rust

```
learner.age()              # number of observations seen
learner.weight(a)     # implicit weight for a probe vector
```

Last change: 2026-04-06, commit: 63ad966

Set 🫧

An unordered collection of hypervectors. See Composites: Set for the conceptual overview.

Constructor

Python Go Rust

```
s = hv.Set(hv.Seed128(0, 42), first, second, third)
```

Notable methods

Python

```
# All these will be approximately 1/3 of the total cardinality.
hv.overlap(s.unmasked(), first)
hv.overlap(s.unmasked(), second)
hv.overlap(s.unmasked(), third)
```

Last change: 2026-04-06, commit: 63ad966

Sequence 📿

An ordered collection of hypervectors with positional encoding. See Composites: Sequence for the conceptual overview.

Constructor

Python Go Rust

```python
# Constructing a sequence, with logical index start at 1 (default to
0).
seq = hv.Sequence(hv.Seed128(0, 42), first, second, third, start=1)
```

Last change: 2026-04-06, commit: 63ad966

Octopus 🐙

A key-value composite where each value is bound with its key's Sparkle. See Composites: Octopus for the conceptual overview.

Constructor

Python Go Rust

```python
oct = hv.Octopus(hv.Seed128(0, 42), ["color", "shape"], red, circle)
```

Key Methods

Python Go Rust

```python
oct.value_by_key("color") # returns the value, or raises ValueError
```

Last change: 2026-04-04, commit: 34eaa34

Knot 

The result of binding (multiplicative composition) of hypervectors. Unlike `BindDirect`, Knot tracks its member parts for serialization and debugging. See Composites: Knot.

Constructor

Python Go Rust

```
# Not directly constructed in Python. Use hv.bind() instead.
k = hv.bind(a, b)
```

Last change: 2026-04-03, commit: 49d0013

Parcel 🎁

The result of bundling (additive composition) of hypervectors. Unlike `BundleDirect`, Parcel tracks its members and bundling seed for serialization and debugging. See Composites: Parcel.

Constructors

Python Go Rust

```
p = hv.bundle(hv.Seed128(10, 1), a, b, c)
```

Last change: 2026-04-03, commit: 49d0013

Operators

See Concepts: Operators for the full overview.

Bind

Python Go Rust

```
bound = hv.bind(a, b)
released = hv.release(bound, b)  # this will recover `a`

hv.equal(a, b)                   # hash equality
```

Release

Extracts one component from a binding: $A \oslash B = A \otimes B^{-1}$

Python Go Rust

```
bound = hv.bind(role, filler)
recovered = hv.release(bound, role)  # ≈ filler
```

Bundle

Python Go Rust

```
p = hv.bundle(hv.Seed128(10, 1), a, b, c)
```

Last change: 2026-04-04, commit: 34eaa34

Customizing runtime behavior

Environment Variables

All environment variables are read once on first access and cannot be changed at runtime. Unset variables use the documented default.

`KONGMING_RNG`

Selects the pseudo-random number generator backend used for hypervector generation.

Value	Description
`xoshiro++` (default)	xoshiro256++ — simple, fast, cross-language deterministic
`pcg`	PCG-DXSM — classic/compat mode (matches pre-v3.7.5 behavior)

Changing this affects all generated vectors: Sparkle offsets, Learner bundling, Cyclone patterns. Vectors generated with different backends are **not** comparable.

`KONGMING_REPR_FORMAT`

Controls `__repr__()` / `Repr()` output format.

Value	Description
`YAML` (default)	Multi-line YAML dump
`PROTO`	Multi-line protobuf debug string

KONGMING_LEARNER_SAMPLING

Controls the bundling strategy used by Learner.

Value	Description
FISHER_YATES (default)	Fisher-Yates shuffle — selects exactly the right number of segments per round
CLASSIC	Per-segment probabilistic sampling — each segment is independently sampled with a fixed probability

```
# Example: use PCG for backward compatibility with pre-v3.7.5
vectors
export KONGMING_RNG=pcg

# Example: switch repr to protobuf debug format
export KONGMING_REPR_FORMAT=PROTO

# Example: use classic sampling in Learner
export KONGMING_LEARNER_SAMPLING=CLASSIC
```

Querying the Current Environment

Use `global_env()` to inspect all active settings at runtime. Returns a `GlobalEnv` protobuf message — new fields added to the proto automatically appear.

```
>>> hv.global_env()
rng_hint: XOSHIRO256PP
learner_sampling: FISHER_YATES
repr_format: YAML
```

Last change: 2026-04-04, commit: 17a1885

Misc

Display

All HyperBinary types have a compact, emoji-prefixed string representation for quick visual inspection. See HyperBinary Types for type symbols and Sparkle for field labels.

Python `__str__` and `__repr__`

`__str__` (triggered by `print()`) returns the compact emoji form:

```
>>> a = hv.Sparkle.with_word(hv.MODEL_64K_8BIT, hv.d0(), "hello")
>>> print(a)
✨:🌐0x..c862,🫛0x..80e4
```

`__repr__` (triggered by evaluating a variable in the shell or notebook) returns a detailed, developer-friendly YAML representation, controlled by the `KONGMING_REPR_FORMAT` environment variable:

```
>>> a
hint: SPARKLE
model: MODEL_64K_8BIT
stable_hash: 12345678
domain:
  id: ...
pod:
  seed: 12345
```

Set `KONGMING_REPR_FORMAT=PROTO` for protobuf debug output instead of the default YAML. See Environment Variables for all supported variables.

Go / Rust Display

Python Go Rust

```
print(sparkle)      # compact emoji form via __str__
repr(sparkle)       # detailed YAML/proto form via __repr__
```

Serialization

Python Go Rust

```
# HyperBinary → protobuf bytes
msg = hv.to_message(sparkle)

# protobuf bytes → HyperBinary
obj = hv.from_message(msg)

# raw proto bytes → HyperBinary
obj = hv.from_proto_bytes(data)

# proto bytes → YAML string (for debugging)
hv.format_to_yaml(data)
```

Last change: 2026-04-02, commit: 03bd04a

Memory

The memory package provides persistent and in-memory storage for hypervectors with semantic indexing and near-neighbor search.

The core abstraction is a **Chunk** — an immutable identity (Sparkle) paired with a mutable semantic code (any HyperBinary). Chunks are stored in a **Substrate** (pluggable storage backend), queried via **ChunkSelectors**, and created via **ChunkProducers**.

Section	Description
Chunk	The fundamental storage unit
Substrate & Views	Storage backends and transactional views
Selectors	Query builders for reading chunks
Producers	Write builders for creating chunks

Last change: 2026-04-01, commit: 4532a28

Chunk

The fundamental storage unit in the memory system. A Chunk pairs an **immutable identity** (always a Sparkle) with a **mutable semantic code** (any HyperBinary type).

The unique id for a chunk facilitates the compositionality, as this chunk is either present or absent. At the same time, the potentially learnable code (for a chunk) offers opportunities to learn and adapt, just like weights from traditional neural nets.

Structure

Field	Type	Description
`id`	Sparkle	Immutable identity — determines storage key
`code`	HyperBinary	Semantic content (can be updated). If absent, defaults to `id`
`note`	string	Human-readable annotation, primarily for debugging
`extra`	protobuf Any	Extensible payload for application-specific data, primarily for debugging

Inspection

Chunks are typically created via producers (see Producers), but can be inspected after retrieval (see Selectors).

Python

```
# chunk = memory.first_picked_chunk(view,
memory.by_item_key("animals", "cat"))

chunk.id         # Sparkle
chunk.code       # HyperBinary
chunk.note       # str
chunk.extra      # Optional[bytes]
```

Last change: 2026-04-06, commit: bdbfd5e

Substrate & Views

A **Substrate** is a pluggable storage backend. It provides transactional **views** for reading and writing chunks.

View Pattern

All storage access goes through views:

- **SubstrateView** — read-only, supports key lookup and prefix scanning
- **SubstrateMutableView** — extends SubstrateView with write staging and atomic commit (to underlying storage)

Python

```python
# Read-only view (context manager)
with storage.new_view() as view:
    # Check if chunk exists, without actually reading it back.
    exists = view.chunk_exists("animals", "cat")

    cat_chunk = view.read_chunk("animals", "cat")

# Mutable view (auto-commits on clean exit, rollback on exception)
with storage.new_mutable_view() as view:
    view.write_chunk(sparkle, code=some_code, note="my note")

    # commits automatically
```

Storage Backends

InMemory

Volatile, in-process storage. All data lost on exit. Best for testing and ephemeral caches.

Python

```
storage = memory.InMemory(hv.MODEL_64K_8BIT, "my_store")
```

Embedded

Persistent, single-machine storage backed by an embedded key-value store. Suitable for local development and moderate-scale deployments.

Python

```
storage = memory.Embedded(hv.MODEL_64K_8BIT, "/path/to/store")
```

ScyllaDB (Distributed)

Distributed storage via Cassandra-compatible ScyllaDB. For high-scale, multi-node deployments.

Python

```
# Not exposed yet...
```

Last change: 2026-04-06, commit: 63ad966

Selectors

ChunkSelectors are composable query builders for reading chunks from the substrate. Each selector defines how to locate and return matching chunks.

Last change: 2026-04-06, commit: 63ad966

NNS (Near-Neighbor Search)

Wraps one or more attractors to perform near-neighbor search.

Python

```
result = memory.first_picked(
    view, memory.nns(
    memory.set_members(memory.by_item_key("sets", "my_set")),
))
```

Last change: 2026-04-06, commit: 63ad966

Each attractor conceptually provides “the center of attraction” for candidates: the NNS accepts one or more attractors, to perform the actual near-neighbor search work, by interacting with underlying associative index.

Implementation-wise, attractors are specialized selectors.

Forward attractors

Roughly forward attractors try to find parts from a given a composite.

Attractor	Query	Attracts
SetMembersAttractor	Releases SET_MARKER from a Set	All members of the Set
SequenceMemberAttractor	Releases with SEQUENCE_MARKER + positional marker	Sequence member at a specific position
TentacleAttractor	Release with key	Octopus value for a given key

Python

```
memory.set_members(memory.by_item_key("sets", "my_set"))

memory.sequence_member(memory.by_item_key("seqs", "my_seq"), pos=2)

memory.tentacle(memory.by_item_key("records", "person"), key="name")
```

Reverse Attractors

Roughly reverse attractors try to locate composites given a part.

Attractor	Query	Attracts
SetAttractor	Binds member with SET_MARKER	All Sets that contain a given member
SequenceAttractor	Binds member with SEQUENCE_MARKER + position	All Sequences containing the given member at a specific position
OctopusAttractor	Binds value with key	Octopuses with a given key-value pair

Python

```
memory.set_attractor(memory.by_item_key("animals", "cat"),
domain="sets")

memory.sequence_attractor(memory.by_item_key("animals", "cat"),
pos=0, domain="seqs")

# NOTE the value is specified with another selector.
memory.octopus_attractor(key="color",
value=memory.by_item_key("colors", "red"))
```

Analogical Reasoner

Analogical reasoner tries to perform analogical reasoning, like “A is to B as C is to ?”.

Given the analogy “king is to queen as man is to ?”

Python

```
king   = hv.Sparkle(model, "role", "king")
queen  = hv.Sparkle(model, "role", "queen")
man    = hv.Sparkle(model, "role", "man")

# Analogy: "king is to queen as man is to ?"
#   src     = king   (the known source of the relationship)
#   feature = queen  (the known feature/attribute of src)
#   dst     = man    (the target; we want to find its corresponding
feature)
#
# The attractor computes: dst ⊗ feature ⊗ src⁻¹
# This produces a code that should overlap with "woman"
memory.nns(
    memory.analogical_reasoner(memory.with_code(man), src=king,
feature=queen)
)
```

Last change: 2026-04-06, commit: 63ad966

Other Selectors

ByItemKey

Exact lookup by domain + pod.

Python

```
sel = memory.by_item_key("animals", "cat")
```

ByItemDomain

All chunks in a given domain (prefix scan).

Python

```
sel = memory.by_item_domain("animals")
```

WithCode / WithSparkle

Literal selector — returns a hypervector directly, no storage lookup.

Python

```
sel = memory.with_code(some_hv)

sel = memory.with_sparkle("animals", "cat")
```

Joiner

Union of multiple selectors — returns results from each of the inner selectors.

Python

```
sel = memory.joiner(
    memory.by_item_key("animals", "cat"),
    memory.by_item_key("animals", "dog"),
)
```

Range

Limits results to `[start, start+limit)`. `limit=0` (default) implies no limit, and iteration continue until there is no more results.

Python

```
sel = memory.range_sel(
    memory.by_item_domain("animals"), start=0, limit=10)
```

OnlyDomain

Filters inner selector results by given domain.

Python

```
sel = memory.only_domain(
    "animals", inner_selector)
```

Last change: 2026-04-06, commit: 63ad966

Working with Results

FirstPicked — get the first match

Returns the first chunk matching the selector. Returns an error if nothing is found.

Python

```
# Returns the first matching Chunk (with .id, .code, .note, .extra)
chunk = memory.first_picked(view, selector)
print(chunk.id, chunk.code, chunk.note)
```

Iterator — iterate all matches

Python

Not available as iterator in Python. Use `mem_get()` for batch reads.

mem_get — high-level batch read

Python

```
results = storage.mem_get(selector)
for hv in results:
    print(hv)
```

Last change: 2026-04-06, commit: 63ad966

Producers

ChunkProducers are write builders that create and persist chunks in the substrate. Each producer encapsulates the logic for constructing a specific type of chunk.

> ℹ️ **Note**
> Some producers only update existing chunks (e.g., `ClusterUpdater`) without creating new ones. In those cases, `Produce` returns the updated chunk rather than a newly created one.

Producer Options

Producer options are additional information supplied to producer constructor to tweak behavior.

Python

```python
# `note` indiciates additional note for the new terminal chunk.
memory.new_terminal("d", "p", note="annotation")

# `semantic_indexing` indicates we need to index the semantic code
# (on top of the id vector).
memory.from_set_members("d", "p", members, semantic_indexing=True)
```

Concrete Producers

NewTerminal

Creates a chunk whose code equals its identity (a bare Sparkle). Useful for registering atoms/symbols.

Python

```
with storage.new_mutable_view() as view:
    memory.mem_set(view, memory.new_terminal("fruits", "apple",
note="an apple"))
```

NewLearner

Creates a fresh Learner chunk for online learning.

Python

```
with storage.new_mutable_view() as view:
    memory.mem_set(view, memory.new_learner("learners",
"my_learner", note="a learner"))
```

FromSetMembers

Creates a Set from stored members.

Python

```
with storage.new_mutable_view() as view:
    memory.mem_set(
        view,
        memory.from_set_members(
            "sets",
            "fruit_set",
            memory.by_item_domain("fruits"),
        ))
```

FromSequenceMembers

Creates a Sequence from stored members with positional encoding.

Python

```
with storage.new_mutable_view() as view:
    memory.mem_set(
        view,
        memory.from_sequence_members(
            "seqs",
            "greeting",
            memory.joiner(
                memory.by_item_key("words", "hello"),
                memory.by_item_key("words", "world"),
            ),
            start=0),
    )
```

FromKeyValues

Creates an Octopus (key-value composite) from keys and value selectors.

Python

```
with storage.new_mutable_view() as view:
    memory.mem_set(
        view,
        memory.from_key_values(
            "records",
            "obj1",
            keys=["color", "shape"],
            values=memory.joiner(
                memory.by_item_key("colors", "red"),
                memory.by_item_key("shapes", "circle"),
            ),
        ))
```

ClusterUpdater

Feeds an observed chunk into an existing Learner, updating its accumulated code via bundling.

Python

```
with storage.new_mutable_view() as view:
    memory.mem_set(view,
        memory.cluster_updater(
            learner=memory.by_item_key("learners", "my_learner"),
            observed=memory.by_item_key("fruits", "apple"),
            multiple=1,
        ))
```

Last change: 2026-04-06, commit: 63ad966

Notebook Quick Start

This guide walks through using Kongming HV in a Jupyter notebook, cell by cell.

Section	Description
Notebook Platforms	Setup differences between Jupyter, Colab, and Binder
Interactive Notebooks	Links to existing notebooks
Walkthrough	Step-by-step: vocabulary, similarity, learning, binding

Tips

- **Reproducibility**: Use fixed seeds in `SparseOperation` for deterministic results across reruns.
- **Visualization**: Use `pandas` DataFrames for overlap matrices — they render nicely in Jupyter.
- **Performance**: The Rust backend is fast. Building 10,000 vectors takes under a second on `MODEL_64K_8BIT`.
- **Model choice**: Start with `MODEL_64K_8BIT` for exploration. Switch to `MODEL_1M_10BIT` or larger for production workloads.

Last change: 2026-04-01, commit: 4d22850

Notebook Platforms

Setup and behavior differ across Jupyter, Google Colab, and Binder. This page covers the key differences.

Try Online

Notebook	Platform	Link
`first.ipynb`	Google Colab	Open in Colab
`first.ipynb`	Binder	launch binder
`memory.ipynb`	Google Colab	Open in Colab
`lisp.ipynb`	Google Colab	Open in Colab

Comparison

	Jupyter (local)	Google Colab	Binder
Account	None	Google account required	None
Install	`pip install` in terminal beforehand	`!pip install` in first cell	Pre-installed via `requirements.txt`
Restart needed	No	Yes — after first install	No
Startup time	Instant	Fast (~5s)	Slow (2–5 min cold start)

	Jupyter (local)	Google Colab	Binder
Persistence	Local filesystem	Google Drive (optional mount)	Ephemeral — lost on timeout
GPU	If available locally	Free tier available	Not available
Custom packages	Full control	`!pip install` per session	Via `requirements.txt` only

Jupyter (Local)

Install once in your terminal, then use in any notebook:

```
pip install kongming-rs-hv
```

```
# Cell 1 — no restart needed
from kongming import hv
```

For development workflows with frequent code changes, use autoreload:

```
%load_ext autoreload
%autoreload 2
```

Google Colab

Colab runs in the cloud with a fresh environment each session. Install in the first cell:

```
# Cell 1 — install
!pip install kongming-rs-hv
```

After the first install, Colab requires a **runtime restart**:

1. Go to **Runtime** → **Restart runtime** (or use the button Colab shows after install)
2. Then run the remaining cells

```
# Cell 2 — after restart
from kongming import hv
model = hv.MODEL_64K_8BIT
```

Subsequent sessions on the same notebook will need the install cell again — Colab does not persist pip packages across sessions.

Saving work: Use `google.colab.drive` to mount Google Drive for persistent storage:

```
from google.colab import drive
drive.mount('/content/drive')
# Then use paths like /content/drive/MyDrive/...
```

Binder

Binder builds a Docker image from your repo's `requirements.txt` and launches a Jupyter server. No account needed.

- **First launch**: Takes 2–5 minutes to build the environment
- **Subsequent launches**: Faster if the image is cached
- **No install needed**: `kongming-rs-hv` is pre-installed from `requirements.txt`
- **Ephemeral**: All work is lost when the session times out (~10 min idle)

```
# Cell 1 — works immediately, no install
from kongming import hv
```

> ⚠️ **Limitation**
>
> You cannot install additional packages not in `requirements.txt` (the environment is read-only).

Choosing a Platform

Use case	Recommended
Daily development	Jupyter (local)
Quick demo / sharing	Google Colab
Zero-setup exploration	Binder
Teaching / workshops	Google Colab (students have accounts)
Persistent storage needed	Jupyter (local) or Colab + Drive

Last change: 2026-04-08, commit: ab5cf46

Interactive Notebooks

For deeper walkthroughs, open these notebooks directly:

Notebook	Description	Colab
`first.ipynb`	Introduction to hypervectors, bind/bundle operations, and composites	
`memory.ipynb`	In-memory and persistent storage, near-neighbor search with attractors, and export to disk	
`lisp.ipynb`	VSA-based LISP interpreter where every data structure is a hypervector	

See also: LISP Interpreter — a full application built on the core API.

Last change: 2026-04-08, commit: ab5cf46

Walkthrough

A step-by-step introduction to Kongming HV in a notebook, cell by cell.

Setup

```
# Cell 1: Install and import
# !pip install kongming-rs-hv pandas

from kongming import hv
import pandas as pd

model = hv.MODEL_64K_8BIT
so = hv.SparseOperation(model, 0, 1)
```

Building a Vocabulary

```
# Cell 2: Create vectors for a set of words
words = ["cat", "dog", "fish", "bird", "tree", "rock"]
vectors = {w: hv.Sparkle.from_word(model, "vocab", w) for w in
words}

print(f"Created {len(vectors)} vectors")
print(f"Model: {model}, Cardinality: {hv.cardinality(model)}")
```

Output:

```
Created 6 vectors
Model: 1, Cardinality: 256
```

Similarity Matrix

```
# Cell 3: Compute pairwise overlap
data = {}
for w1 in words:
    data[w1] = {w2: hv.overlap(vectors[w1], vectors[w2]) for w2 in
words}

pd.DataFrame(data, index=words)
```

Output:

	cat	dog	fish	bird	tree	rock
cat	256	1	0	2	1	1
dog	1	256	1	0	1	2
fish	0	1	256	1	0	1
bird	2	0	1	256	1	0
tree	1	1	0	1	256	1
rock	1	2	1	0	1	256

The diagonal is 256 (cardinality = perfect self-overlap). Off-diagonal values are near 0-2 (random noise), confirming the vectors are near-orthogonal.

Learning from Observations

```
# Cell 4: Create a learner and feed it observations
learner = hv.Learner(model, hv.Seed128(0, so.uint64()))

# "cat" seen 3 times, "dog" once, "bird" once
for _ in range(3):
    learner.bundle(vectors["cat"])
learner.bundle(vectors["dog"])
learner.bundle(vectors["bird"])

print(f"Learner age: {learner.age()}")
```

Output:

```
Learner age: 5
```

Probing the Learner

```
# Cell 5: Check what the learner remembers
results = []
for w in words:
    ov = hv.overlap(learner, vectors[w])
    results.append({"word": w, "overlap": ov})

df = pd.DataFrame(results).sort_values("overlap", ascending=False)
df
```

Output:

word	overlap
cat	~75
dog	~30
bird	~30
fish	~1
tree	~1
rock	~1

“cat” has the highest overlap (seen 3x). “dog” and “bird” (seen 1x each) have moderate overlap. Unseen words are at noise level (~1).

Binding: Role-Filler Pairs

```
# Cell 6: Create a structured representation
#   "a cat that is red"
color_role = hv.Sparkle.from_word(model, "role", "color")
animal_role = hv.Sparkle.from_word(model, "role", "animal")

red = hv.Sparkle.from_word(model, "color", "red")
blue = hv.Sparkle.from_word(model, "color", "blue")
cat = vectors["cat"]

# Bind role with filler, then bundle the pairs
learner2 = hv.Learner(model, hv.Seed128(0, so.uint64()))
learner2.bundle(hv.Sparkle.bind(color_role, red))
learner2.bundle(hv.Sparkle.bind(animal_role, cat))

# Probe: "what color?"
query = hv.Sparkle.bind(learner2, color_role.power(-1))
print(f"red overlap:  {hv.overlap(query, red)}")   # high
print(f"blue overlap: {hv.overlap(query, blue)}")   # ~1
print(f"cat overlap:  {hv.overlap(query, cat)}")    # ~1
```

Last change: 2026-04-06, commit: 63ad966

Python Quick Start

Section	Description
Installation	PyPI install, supported platforms, import paths
Quick Example	Minimal code showing bind, bundle, and overlap
Walkthrough	Vectors, similarity, random generation, power, learning

See also: Notebook Quick Start for interactive Jupyter walkthroughs.

Last change: 2026-04-01, commit: 0551c6e

Installation

PyPI

```
pip install kongming-rs-hv
```

Supported Platforms

Platform	Architectures	Python Versions
Linux	x86_64	3.10–3.14
macOS	Apple Silicon & Intel	3.10–3.14
Windows	x86_64	3.10–3.14

Verifying Installation

```
import kongming
print(kongming.__version__)  # e.g. should be "3.6.5", as of Apr.
2026. Yours should be newer.

from kongming import hv
print(hv.MODEL_64K_8BIT)  # should print 1
```

Import Paths

The package exposes two main modules:

```
from kongming import hv       # hypervector operations
from kongming import memory   # storage and selectors
```

Model constants are available directly on `hv`:

```
hv.MODEL_64K_8BIT      # 1
hv.MODEL_1M_10BIT      # 2
hv.MODEL_16M_12BIT     # 3
hv.MODEL_256M_14BIT    # 4
hv.MODEL_4G_16BIT      # 5
```

Last change: 2026-04-02, commit: da51f66

Quick Example

A minimal example showing the core operations:

```
from kongming import hv

# Create hypervectors
a = hv.Sparkle.from_word(hv.MODEL_64K_8BIT, hv.d0(), "hello")
b = hv.Sparkle.from_word(hv.MODEL_64K_8BIT, hv.d0(), "world")
print(f'Overlap: {hv.overlap(a, b)}')  # Near orthogonal (~1)

# Bind: result is dissimilar to both inputs
bound = hv.bind(a, b)
print(f'{hv.overlap(bound, a)=}, {hv.overlap(bound, b)=}')  # ~1, ~1

# Bundle: result is similar to both inputs
bundled = hv.bundle(hv.Seed128(10, 1), a, b)
print(f'{hv.overlap(bundled, a)=}, {hv.overlap(bundled, b)=}')  #
high, high
```

What's Happening

1. `Sparkle.from_word` generates a deterministic hypervector from a word. Same word always produces the same vector.
2. Two unrelated vectors have near-zero overlap (~1) — random high-dimensional vectors are nearly orthogonal.
3. `hv.bind(a, b)` produces a vector dissimilar to both (low overlap). Binding is reversible.
4. `hv.bundle(seed, a, b)` produces a vector similar to both (high overlap). Different seeds produce different but equally valid results.

Last change: 2026-04-02, commit: da51f66

Walkthrough

A deeper exploration of the Python API, covering vector creation, similarity, random generation, power/permutation, and online learning.

Creating Vectors

```
from kongming import hv

model = hv.MODEL_64K_8BIT

# Create sparkles (atomic vectors) from words
cat = hv.Sparkle.from_word(model, "animals", "cat")
dog = hv.Sparkle.from_word(model, "animals", "dog")

# Same inputs always produce the same vector
cat2 = hv.Sparkle.from_word(model, "animals", "cat")
assert cat.stable_hash() == cat2.stable_hash()
```

Measuring Similarity

```
# Random vectors have ~1 overlap
print(hv.overlap(cat, dog))   # ≈ 1 (near-orthogonal)

# A vector is maximally similar to itself
print(hv.overlap(cat, cat))   # = 256 (= cardinality)
```

Using SparseOperation for Random Generation

```
so = hv.SparseOperation(model, 123, 456)

# Generate random sparkles
a = hv.Sparkle.random("my_domain", so)
b = hv.Sparkle.random("my_domain", so)

# Each call to so produces a new random seed
print(hv.overlap(a, b))  # ≈ 1
```

Power and Permutation

```
# Power creates a permuted vector
s = hv.Sparkle.from_word(model, "pos", "step")
s2 = s.power(2)
s3 = s.power(3)

# Different powers are near-orthogonal
print(hv.overlap(s, s2))   # ≈ 1
print(hv.overlap(s, s3))   # ≈ 1

# Inverse: power(-1) undoes power(1)
s_inv = s.power(-1)
# bind(s, s_inv) ≈ identity
```

Online Learning with Learner

```
learner = hv.Learner(model, hv.Seed128(0, 42))

# Feed observations one at a time
learner.bundle(cat)
learner.bundle(cat)   # seen twice — stronger signal
learner.bundle(dog)

# The learned vector is more similar to cat (seen 2x)
print(hv.overlap(learner, cat))  # higher
print(hv.overlap(learner, dog))  # lower but above random
```

Last change: 2026-04-06, commit: 63ad966

Mexican Dollar

The “What’s the Dollar of Mexico?” problem is a classic demonstration of analogical reasoning with hypervectors. It shows how structured knowledge about countries can be encoded, and how algebraic operations can answer analogy questions without explicit programming.

The Problem

Given knowledge about three countries:

Country	Code	Capital	Currency
USA	USA	Washington DC	Dollar
Mexico	MEX	Mexico City	Peso
Sweden	SWE	Stockholm	Krona

We want to answer questions like:

- “What is the Dollar of Mexico?” → **Peso**
- “What is the Washington DC of Mexico?” → **Mexico City**
- “What is the Dollar of Sweden?” → **Krona**

How It Works

Each country is encoded as a bundled set of role-filler bindings:

$$\text{US} = \sum_{\oplus} (\text{code} \otimes \text{usa},\ \text{capital} \otimes \text{dc},\ \text{currency} \otimes \text{dollar})$$

$$\text{Mexico} = \sum_{\oplus} \left(\text{code} \otimes \text{mex},\ \text{capital} \otimes \text{mexico\_city},\ \text{currency} \otimes \text{peso}\right)$$

$$\text{Sweden} = \sum_{\oplus} (\text{code} \otimes \text{swe},\ \text{capital} \otimes \text{stockholm},\ \text{currency} \otimes \text{krona})$$

To find “the Dollar of Mexico”, we compute a **transfer vector** from US to Mexico:

$$T_{\text{US} \to \text{Mexico}} = \text{Mexico} \oslash \text{US}$$

Then apply it to Dollar:

$$\text{result} = \text{dollar} \otimes T_{\text{US} \to \text{Mexico}}$$

The result will have high overlap with **Peso** — the analogical answer.

The same transfer works for Sweden:

$$T_{\text{US} \to \text{Sweden}} = \text{Sweden} \oslash \text{US}$$

$$\text{result} = \text{dollar} \otimes T_{\text{US} \to \text{Sweden}} \approx \text{krona}$$

Code (Manual)

The algebraic approach — compute the transfer vector directly:

```
from kongming import hv

model = hv.MODEL_64K_8BIT
so = hv.SparseOperation(model, "knowledge", 0)

# Create role markers
country_code = hv.Sparkle.from_word(model, "role", "country_code")
capital      = hv.Sparkle.from_word(model, "role", "capital")
currency     = hv.Sparkle.from_word(model, "role", "currency")

# Create fillers
usa         = hv.Sparkle.from_word(model, "country", "usa")
mex         = hv.Sparkle.from_word(model, "country", "mex")
swe         = hv.Sparkle.from_word(model, "country", "swe")
dc          = hv.Sparkle.from_word(model, "capital", "dc")
mexico_city = hv.Sparkle.from_word(model, "capital", "mexico_city")
stockholm   = hv.Sparkle.from_word(model, "capital", "stockholm")
dollar      = hv.Sparkle.from_word(model, "currency", "dollar")
peso        = hv.Sparkle.from_word(model, "currency", "peso")
krona       = hv.Sparkle.from_word(model, "currency", "krona")

# Encode each country as role-filler bundles
us_record = hv.bundle(hv.Seed128.random(so),
    hv.bind(country_code, usa),
    hv.bind(capital, dc),
    hv.bind(currency, dollar),
)
mexico_record = hv.bundle(hv.Seed128.random(so),
    hv.bind(country_code, mex),
    hv.bind(capital, mexico_city),
    hv.bind(currency, peso),
)
sweden_record = hv.bundle(hv.Seed128.random(so),
    hv.bind(country_code, swe),
    hv.bind(capital, stockholm),
    hv.bind(currency, krona),
)

# Transfer vector: Mexico / US
transfer_to_mexico = hv.release(mexico_record, us_record)

# "What's the Dollar of Mexico?"
mexican_dollar = hv.bind(dollar, transfer_to_mexico)
print(f"peso overlap:  {hv.overlap(mexican_dollar, peso)}")    #
high!
print(f"dollar overlap: {hv.overlap(mexican_dollar, dollar)}")  # ~1
(noise)
```

```
print(f"krona overlap:  {hv.overlap(mexican_dollar, krona)}")   # ~1
(noise)

# "What's the Washington DC of Mexico?"
mexican_dc = hv.bind(dc, transfer_to_mexico)
print(f"mexico_city overlap: {hv.overlap(mexican_dc, mexico_city)}")
# high!

# Transfer to Sweden works the same way
transfer_to_sweden = hv.release(sweden_record, us_record)
swedish_dollar = hv.bind(dollar, transfer_to_sweden)
print(f"krona overlap: {hv.overlap(swedish_dollar, krona)}")  #
high!
```

Code (with AnalogicalReasoner)

When records are stored in memory (as Octopus composites), the `AnalogicalReasoner` selector handles the transfer automatically:

```
from kongming import hv, memory

model = hv.MODEL_64K_8BIT
store = memory.InMemory(model)

keys = ["capital", "currency", "country_code"]

# Store individual fillers — NNS needs them as searchable items
fillers = {}
for word in ["dc", "USD", "USA", "mexicoCity", "MXN", "MEX",
             "stockholm", "SEK", "SWE"]:
    s = hv.Sparkle.from_word(model, 0, word)
    store.put(s)
    fillers[word] = s

# Store country records as Octopus composites
store.put(hv.Octopus(
    hv.Seed128("country", "USA"), keys,
    fillers["dc"], fillers["USD"], fillers["USA"],
))
store.put(hv.Octopus(
    hv.Seed128("country", "MEX"), keys,
    fillers["mexicoCity"], fillers["MXN"], fillers["MEX"],
))
store.put(hv.Octopus(
    hv.Seed128("country", "SWE"), keys,
    fillers["stockholm"], fillers["SEK"], fillers["SWE"],
))

# Retrieve stored records
us_code  = store.get("country", "USA").code
mex_code = store.get("country", "MEX").code
swe_code = store.get("country", "SWE").code

view = store.new_view()

# "What is the USD of Mexico?"
result = memory.first_picked_chunk(view,
    memory.nns(
        memory.analogical_reasoner(
            memory.with_code(mex_code),
            us_code,
            fillers["USD"],
        )
    )
)
print(result.id)  # → ✨:🌱MXN
```

```
# "What is the Washington DC of Mexico?"
result = memory.first_picked_chunk(view,
    memory.nns(
        memory.analogical_reasoner(
            memory.with_code(mex_code),
            us_code,
            fillers["dc"],
        )
    )
)
print(result.id)  # → ✨:🌱mexicoCity

# "What is the Dollar of Sweden?"
result = memory.first_picked_chunk(view,
    memory.nns(
        memory.analogical_reasoner(
            memory.with_code(swe_code),
            us_code,
            fillers["USD"],
        )
    )
)
print(result.id)  # → ✨:🌱SEK
```

The `AnalogicalReasoner` computes the transfer vector internally and uses near-neighbor search to find the best match in memory — no manual algebra needed.

Why It Works

The transfer vector $T = \text{Mexico} \oslash \text{US}$ captures the *structural mapping* between the two records. When applied to any filler from the US record, it maps it to the corresponding filler in the Mexico record — because the role-filler binding structure is preserved by the algebra.

This is a form of **analogical reasoning**: no explicit rules, no lookup tables — just algebraic operations on high-dimensional vectors.

See Also

- Concepts: Operators — algebraic foundations
- Operators — bind, release, bundle
- Octopus — key-value composite used for country records
- Memory: Selectors — `analogical_reasoner`, `nns`, `with_code`
- Near-neighbor search — how the reasoner finds answers

Last change: 2026-04-05, commit: 63ab0cc

LISP Interpreter

Last change: 2026-04-02, commit: da51f66

Bulk Storage Benchmark

This example populates a storage with a large number of random terminal chunks, then queries a few by key to verify correctness. It demonstrates how to batch-create items and measure throughput.

Note associative index is also prepared in the process, and near-neighbor search is available immediately upon successful conclusion of all writing.

Motivated readers can further improve this script to test various producers or selectors.

Script

```
#!/usr/bin/env python3
"""Populate local storage with random terminal chunks and verify
retrieval."""

import argparse
import shutil
import tempfile
import time
from kongming import hv, memory


def main():
    parser = argparse.ArgumentParser(description="Bulk storage
benchmark")
    parser.add_argument(
        "-n", "--count", type=int, default=10_000,
        help="Number of terminal chunks to create (default: 10000)",
    )
    parser.add_argument(
        "--model", type=int, default=hv.MODEL_1M_10BIT,
        help="HV model (default: MODEL_1M_10BIT)",
    )
    parser.add_argument(
        "--domain", type=str, default="bench",
        help="Domain name for all chunks (default: bench)",
    )
    parser.add_argument(
        "--backend", type=str,
        choices=["inmemory", "embedded"],
        default="inmemory",
        help="Storage backend (default: inmemory)",
    )
    parser.add_argument(
        "--path", type=str, default=None,
        help="Disk path for embedded backend (default: temp
directory)",
    )
    args = parser.parse_args()

    # --- Create storage ---
    tmpdir = None
    if args.backend == "embedded":
        if args.path:
            path = args.path
```

```
        else:
            tmpdir = tempfile.mkdtemp()
            path = f"{tmpdir}/bench_store"
        storage = memory.Embedded(args.model, path)
        print(f"Backend: Embedded (path={path})")
    else:
        storage = memory.InMemory(args.model)
        print("Backend: InMemory (BTreeMap, pure in-memory)")

    # --- Write phase ---
    print(f"Writing {args.count:,} terminal chunks …")
    t0 = time.perf_counter()
    for i in range(args.count):
        storage.mem_set(memory.new_terminal(args.domain, str(i)))

    elapsed = time.perf_counter() - t0
    rate = args.count / elapsed
    print(f"  done in {elapsed:.2f}s  ({rate:,.0f} chunks/s)")
    print(f"  item_count = {storage.item_count():,}")

    # --- Read phase: spot-check a few items ---
    spot_checks = range(0, args.count, args.count // 100)
    print(f"Spot-checking keys: {spot_checks}")
    for idx in spot_checks:
        expected = hv.Sparkle.from_word(args.model, args.domain,
str(idx))
        chunk = storage.get(args.domain, str(idx))
        if not hv.equal(chunk.id, expected):
            print(f"mismatch at key {idx}: {chunk}")

    print("All checks passed.")

    # --- Cleanup ---
    if tmpdir:
        del storage
        shutil.rmtree(tmpdir)


if __name__ == "__main__":
    main()
```

Usage

```
# Default: 10K chunks, in-memory storage substrate.
python bulk_storage.py

# Embedded (disk-backed storage substrate).
python bulk_storage.py --backend embedded

# Embedded with a specific path (tip: use a tmpfs mount for near-in-
memory speed)
python bulk_storage.py --backend embedded --path /dev/shm/my_bench

# Custom count
python bulk_storage.py -n 100000

# Different model, 1 implies MODEL_64K_8BIT model, etc.
python bulk_storage.py -n 10000 --model 1
```

Last change: 2026-04-08, commit: 03128b0