ArrowSpace: Spectral Indexing of Embeddings using taumode ($\lambda\tau$)

Lorenzo Moriondo
Independent Researcher - tuned.org.uk
ORCID: 0000-0002-8804-2963

28 August 2025

Abstract

`ArrowSpace` is a library that implements a novel spectral indexing approach for vector similarity search, combining traditional semantic similarity with graph-based spectral properties. The library introduces taumode ($\lambda\tau$, lambda-tau) indexing, which blends Rayleigh quotient smoothness energy from graph Laplacians with edge-wise dispersion statistics to create bounded, comparable spectral scores. This enables similarity search that considers both semantic content and spectral characteristics of high-dimensional vector datasets.

1 Statement of Need

Traditional vector similarity search relies primarily on geometric measures, like cosine similarity or Euclidean distance which capture semantic relationships but ignore the spectral structure inherent in many datasets. For example in domains such as protein analysis, signal processing and molecular dynamics, the "roughness" or "smoothness" of feature signals across data relationships can provide valuable discriminative information that complements semantic similarity.

Existing vector databases and similarity search systems lack integrated spectral-aware indexing capabilities. While spectral methods exist in graph theory and signal processing (for spectral clustering see [8]), they are typically computationally expensive and they are not considered for database applications. With the increasing demand for vector searching though (in particular, at current state, for the components called "retrievers" in RAG applications [5]), the research on spectral indexing gains traction for database applications. `ArrowSpace` addresses this gap by providing:

1. **Spectral-aware similarity search** that combines semantic and spectral properties
2. **Bounded synthetic indexing** that produces comparable scores across datasets
3. **Memory-efficient representation** that avoids storing graph structures at query time
4. **High-performance Rust implementation** with potentially zero-copy operations and cache-friendly data layouts

2 Data Model and Algorithm

`ArrowSpace` provides an API to use taumode ($\lambda\tau$) that is a single, bounded, synthetic score per signal that blends the Rayleigh smoothness energy on a graph with an edgewise dispersion summary; enabling spectra-aware search and range filtering. Operationally, `ArrowSpace` stores dense features (inspired by CSR [7] and `smartcore` [12]) as rows over item nodes, computes a Laplacian on items, derives per-row Rayleigh energies, compresses them via a bounded map $E/(E+\tau)$, mixes in a dispersion term and uses the resulting $\lambda\tau$ both for similarity and to build a λ-proximity item graph used across the API. This way the $\lambda\tau$ (taumode) score can rely on a synthesis of the characteristics proper of diffusion models and geometric/topological representation of graphs.

2.1 Motivation

From an engineering perspective, there is increasing demand for vector database indices that can spot vector similarities beyond the current available methods (L2 distance, cosine distance, or more complex algorithms like HNSW that requires multiple graphs, or typical caching mechanism requiring hashing). New methods to search vector spaces can lead to more accurate and fine-tunable procedures to adapt the search to the specific needs of the domain the embeddings belong to.

2.2 Foundation

The starting score is Rayleigh as described in [4]. Chen emphasises that the Rayleigh quotient provides a variational characterisation of eigenvalues, it offers a way to find eigenvalues through optimisation rather than solving the characteristic polynomial. This perspective is fundamental in numerical linear algebra and spectral analysis. The treatment is particularly valuable for understanding how spectral properties of matrices emerge naturally from optimisation problems, which connects to applications in data analysis, graph theory, and machine learning.

Basic points:

- Definition: for a feature row x and item-Laplacian L, the smoothness is $E = \frac{x^\top Lx}{x^\top x}$, which is non-negative, scale-invariant in x, near-zero for constants on connected graphs, and larger for high-frequency signals; the Rayleigh quotient is the normalised Dirichlet Energy, it is the discrete Dirichlet energy normalised by signal power.

- Physical Interpretation: Dirichlet energy measure the "potential energy" or "stiffness" of a configuration while the Rayleigh quotient normalises this by the total "mass" or "signal power". the result is a scale-invariant measure of how much energy is required per unit mass (in our case the items-nodes).

- The numerator equals the sum of weighted edge differences $\sum_{(i,j)} w_{ij}(x_i - x_j)^2$, directly capturing roughness over the graph, a classical link between Laplacians and Dirichlet energy used throughout spectral methods.

Some implementation starting points:

- Rayleigh energy $x^\top Lx/x^\top x$ measures how "wiggly" a feature signal is over an item graph; constants yield near-zero on connected graphs, while alternating patterns are larger, making it a principled spectral smoothness score for search and structure discovery.

- Pure Rayleigh can collapse near zero or be hard to compare across datasets; mapping energy to a bounded score and blending with a dispersion statistic produces a stable, comparable score that preserves spectral meaning while improving robustness for ranking and filtering.

2.2.1 Graph and data model

Rayleigh energy score is complemented for spectral indexing by computing the graph Laplacian [6] of the dataset:

- Items and features: `ArrowSpace` stores a matrix with rows = feature signals and columns = items; the item graph nodes are the columns, and Rayleigh is evaluated per feature row against that item-Laplacian, aligning spectral scores with dataset geometry.
- Item Laplacian: a Laplacian matrix is constructed over the graph of the items using a λ-proximity policy (ϵ threshold on per-item λ, union-symmetrised, k-capped, kernel-weighted); diagonals store degrees and off-diagonals are $-$weights, satisfying standard Laplacian invariants used by the Rayleigh quotient.

Example:

```
    // 1. Build item graph based on lambda proximity
    let items = vec![
    vec![1.0, 2.0],  // Item 0: $\lambda_0 = 0.3$
    vec![1.1, 2.1],  // Item 1: $\lambda_1 = 0.35$
    vec![3.0, 1.0],  // Item 2: $\lambda_2 = 0.8$
    ];
    let aspace = ArrowSpace::from_items(...)

    // 2. Connect items with $|\lambda_i - \lambda_j| \le \espilon$
    // Items 0,1 connected ($|0.3-0.35| = 0.05 \le \epsilon$)
    // Items 0,2 not connected $(|0.3-0.8| = 0.5 \gt \epsilon$)
    // Items 1,2 not connected $(|0.35-0.8| = 0.45 \gt \epsilon$)

    // 3. Resulting Laplacian (simplified):
    //      [  w   -w    0 ]
    // L = [ -w    w    0 ]  where w = kernel weight
    //      [  0    0    0 ]
```

2.2.2 Role of Laplacian

What the graph Laplacian contributes to Rayleigh energy:

1. Spectral Smoothness: Captures how features vary across item relationships
2. Graph Structure: Encodes similarity topology beyond simple pairwise distances
3. Efficient Computation: Sparse matrix enables fast spectral calculations
4. Theoretical Foundation: Connects to harmonic analysis and diffusion processes

2.3 taumode and bounded energy

The main idea for this design is to *build a score that synthesises the energy features and geometric features of the dataset* and apply it to vector searching.

Rayleigh and Laplacian as bounded energy transformation score become a bounded map: raw energy E is compressed to $E' = \frac{E}{E+\tau} \in [0, 1)$ using a strictly positive scale τ, stabilising tails and making scores comparable across rows and datasets while preserving order within moderate ranges.

Additional τ selection: taumode supports `Fixed`, `Mean`, `Median`, and `Percentile`; non-finite inputs are filtered and a small floor ensures positivity; the default `Median` policy provides robust scaling across heterogeneously distributed energies.

Rayleigh, Laplacian and τ selection enable the taumode score, so to use this score as an indexing score for dataset indexing.

2.3.1 Purpose of τ in the Bounded Transform

The τ parameter is crucial for the bounded energy transformation: $E' = E/(E + \tau)$. This maps raw Rayleigh energies from $[0, \infty)$ to $[0, 1)$, making scores:

- **Comparable across datasets** with different energy scales
- **Numerically stable** by preventing division issues with very small energies
- **Bounded** for consistent similarity computations

2.3.2 taumode Options and Their Use Cases

1. `taumode::Fixed(value)`

```
    taumode::Fixed(0.1)   // Use exactly $\tau = 0.1$
```

When to use:

- You have **domain knowledge** about the appropriate energy scale
- **Consistency** across multiple datasets is critical
- **Reproducibility** is paramount (no dependence on data distribution)

Example: If you know protein dynamics typically have Rayleigh energies around 0.05-0.2, you might fix $\tau = 0.1$.

2. `taumode::Median` **(Default)**

```
    taumode::Median   // Use median of all computed energies
```

When to use:

- **Robust scaling** - less sensitive to outliers than mean
- **Heterogeneous energy distributions** with potential skewness
- **General-purpose** applications where you want automatic adaptation

Why it's default: The median provides a stable central tendency that works well across diverse datasets without being thrown off by extreme values.

3. `taumode::Mean`

```
    taumode::Mean   // Use arithmetic mean of energies
```

When to use:

- **Normally distributed** energy values
- You want the transform to **preserve relative distances** around the center
- **Mathematical simplicity** is preferred

Caution: Sensitive to outliers - a few very high-energy features can skew the entire transformation.

4. `taumode::Percentile(p)`

```
taumode::Percentile(0.25)  // Use 25th percentile
taumode::Percentile(0.75)  // Use 75th percentile
```

When to use:

- **Fine-tuned control** over the energy threshold
- **Emphasising different regimes:**
  - Low percentiles (0.1-0.3): Emphasise discrimination among low-energy (smooth) features
  - High percentiles (0.7-0.9): Emphasise discrimination among high-energy (rough) features

2.3.3 Practical Impact on Search

The choice of taumode affects how the bounded energies E' distribute in $[0, 1)$:

```
// Low-energy feature with different $\tau$ values
let energy = 0.01;
let tau_small = 0.001;  // $E' = 0.01/0.011 \approx 0.91$ (high
    sensitivity)
let tau_large = 0.1;    // $E' = 0.01/0.11 \approx 0.09$ (low sensitivity)
```

Effect on Lambda-Aware Similarity In the lambda-aware similarity score: $s = \alpha \cdot \text{cosine} + \beta \cdot (1/(1 + |\lambda_q - \lambda_i|))$

- **Smaller** $\tau \rightarrow$ More compressed E' values $\rightarrow$ **Less discrimination** between different energy levels
- **Larger** $\tau \rightarrow$ More spread E' values $\rightarrow$ **Greater emphasis** on spectral differences

2.3.4 Implementation Robustness

The code includes several safeguards. About the τ scale, it is limited to a floor. This has proved useful to find similarity in vectors at a range interval scale of 10^{-7}:

```
pub const TAU_FLOOR: f64 = 1e-9;
```

All the tests for finiteness and boundedness of taumode are present in the tests in the repository.

Recommendation Strategy

1. **Start with** `taumode::Median` (default) - works well generally
2. **Use** `taumode::Fixed` when you need reproducibility across runs/datasets
3. **Try** `taumode::Percentile(0.25)` if you want to emphasise smooth features
4. **Try** `taumode::Percentile(0.75)` if rough/high-frequency features are most important
5. **Avoid** `taumode::Mean` unless you're confident about normal distribution

The choice fundamentally determines **how much the spectral component (λ) influences similarity** relative to semantic cosine similarity, making it a key hyperparameter for tuning search behavior in your specific domain.

3 Summary and Conclusion

3.1 taumode ($\lambda\tau$) Indexing

The core innovation of `ArrowSpace` is the $\lambda\tau$ synthetic index, which combines:

- **Rayleigh Energy**: For each feature signal x over an item graph with Laplacian L, computes the smoothness energy $E = (x^T Lx)/(x^T x)$
- **Bounded Transform**: Maps raw energy E to $E' = E/(E + \tau)$ using a robust τ selection policy (Median, Mean, Percentile, or Fixed)
- **Dispersion Term**: Captures edge-wise concentration of spectral energy using Gini-like statistics
- **Synthetic Score**: Blends E' and dispersion via $\lambda = \alpha \cdot E' + (1 - \alpha) \cdot G$, producing bounded scores

Here the references to these concepts in the code:

3.1.1 Rayleigh Energy Implementation

The **Rayleigh energy computation** $E = (x^T Lx)/(x^T x)$ is implemented in `src/operators.rs`:

```rust
    /// Rayleigh quotient x^T L x / x^T x for Laplacian L (CSR).
    pub fn rayleigh_lambda(gl: &GraphLaplacian, x: &[f64]) -> f64 {
      assert!(!x.is_empty(), "vector cannot be empty");
      let den: f64 = x.iter().map(|&xi| xi * xi).sum();
      if den <= 0.0 {
        return 0.0;
      }
      let mut num = 0.0;
      for i in 0..gl.nnodes {
        let xi = x[i];
        let start = gl.rows[i];
        let end = gl.rows[i + 1];
        let s: f64 = (start..end).map(|idx| gl.vals[idx] *
            x[gl.cols[idx]]).sum();
        num += xi * s;
      }
      num / den
    }
```

3.1.2 Bounded Transform Implementation

The **bounded transform** $E' = E/(E + \tau)$ is implemented in `src/taumode.rs`:

```rust
    // Select tau over the per-item energies and map to bounded scores
    let tau = select_tau(&e_item_raw, tau_mode);
    let mut synthetic_items = Vec::with_capacity(n_items);
    for i in 0..n_items {
      let e_bounded = {
        let e = e_item_raw[i].max(0.0);
        e / (e + tau)  // <-- Bounded transform here
      };
      let g_clamped = g_item_raw[i].clamp(0.0, 1.0);
      let s = alpha * e_bounded + (1.0 - alpha) * g_clamped;
      synthetic_items.push(s);
    }
```

3.1.3 Dispersion Term Implementation

The **Gini-like dispersion statistic** is computed in `src/taumode.rs`:

```
    // G_f: sum of squared normalised edge shares
    let mut g_sq_sum = 0.0;
    if edge_energy_sum > 0.0 {
      for i in 0..n_items {
        let xi = x[i];
        let (s, e) = (gl.rows[i], gl.rows[i + 1]);
        for idx in s..e {
          let j = gl.cols[idx];
          if j == i {
            continue;
          }
          let w = (-gl.vals[idx]).max(0.0);
          if w > 0.0 {
            let d = xi - x[j];
            let contrib = w * d * d;
            let share = contrib / edge_energy_sum;  // Edge energy share
            g_sq_sum += share * share;  // Gini-like concentration
          }
        }
      }
    }
    let g_f = g_sq_sum.clamp(0.0, 1.0);
    dispersions_f.push(g_f);
```

3.1.4 Synthetic Score Blending

The **final synthetic score** $\lambda = \alpha \cdot E' + (1 - \alpha) \cdot G$ is computed in `src/taumode.rs`:

```
    let s = alpha * e_bounded + (1.0 - alpha) * g_clamped;
    synthetic_items.push(s);
```

3.2 Graph Construction

`ArrowSpace` builds similarity graphs from vector data using lambda-proximity connections:

- **Item Graphs**: Connects items whose aggregated λ values differ by at most ϵ
- **K-Capping**: Limits neighbors per node while maintaining graph connectivity
- **Union Symmetrisation**: Ensures undirected Laplacian properties
- **Kernel Weighting**: Uses monotone kernels $w = 1/(1 + (|\Delta\lambda|/\sigma)^p)$ for edge weights

3.3 Memory-Efficient Design

The library consider by-design several optimisations for performance:

- **Column-Major Storage**: Dense arrays with features as rows, items as columns (for production-readiness [12] will be used)
- **Potentially Zero-Copy Operations**: Slice-based access without unnecessary allocations as already present in [12]
- **Single-Pass Computation**: $\lambda\tau$ indices computed once, graph can be discarded

- **Cache-Friendly Layout**: Contiguous memory access patterns for potential SIMD optimization

4 Implementation

`ArrowSpace` is implemented in Rust (edition 2024) with the following architecture:

4.1 Core Components

- `ArrowSpace`: Dense matrix container with per-item $\lambda\tau$ scores
- `ArrowItem`: Individual vector with spectral metadata and similarity operations
- `GraphFactory`: Constructs various graph types from vector data
- `ArrowSpaceBuilder`: Fluent API for configuration and construction

4.2 Usage Example

```rust
use ArrowSpace::builder::ArrowSpaceBuilder;
use ArrowSpace::core::ArrowItem;

// Build ArrowSpace from item vectors
let items = vec![
vec![1.0, 2.0, 3.0],  // Item 1
vec![2.0, 3.0, 1.0],  // Item 2
vec![3.0, 1.0, 2.0],  // Item 3
];

let (aspace, _graph) = ArrowSpaceBuilder::new()
.with_rows(items)
.with_lambda_graph(1e-3, 6, 2.0, None)
.build();

// Query with lambda-aware similarity
let query = ArrowItem::new(vec![1.5, 2.5, 2.0], 0.0);
let results = aspace.search_lambda_aware(&query, 5, 0.8, 0.2);
```

5 Performance Characteristics

5.1 Computational Complexity

- **Index Construction**: $O(N^2)$ for similarity graph (already identified a solution to make this into $O(N \log N)$); $O(F \cdot \text{nnz}(L))$ for $\lambda\tau$ computation.
- **Query Time**: $O(N)$ for linear scan, $O(1)$ for $\lambda\tau$ lookup
- **Memory Usage**: $O(F \cdot N)$ for dense storage, $O(N)$ for $\lambda\tau$ indices

5.2 Benchmarks

The library includes comprehensive benchmarks comparing `ArrowSpace` with baseline cosine similarity:

- **Single Query**: ∼15% overhead for $\lambda\tau$-aware search vs pure cosine
- **Batch Queries**: Scales linearly with batch size, maintains constant per-query overhead
- **Memory Footprint**: 4-8 bytes per $\lambda\tau$ index vs graph storage

6 Scientific Applications

`ArrowSpace` has been designed with several scientific domains in mind:

6.1 Protein Structure Analysis

The examples demonstrate protein-like vector databases with molecular dynamics features (inspired by [13]):

```
    // Trajectory features for spectral analysis
    fn trajectory_features(domain: &ProteinDomain) -> Vec<f64> {
      let mut features = Vec::new();
      for frame in &domain.trajectory {
        features.push(frame.rmsd);
        features.push(frame.energy / 1000.0);
        features.push(frame.temperature / 300.0);
        // ... additional biophysical features
      }
      features
    }

    let items: Vec<Vec<f64>> = domains
    .into_iter()
    .map(extract_features)
    .collect();

    let (aspace, _gl) = ArrowSpaceBuilder::new()
    .with_rows(items) // $N\timesF -> auto-transposed to F\timesN$
    .build();
```

6.2 Results

`ArrowSpace` has substantial potential for raw improvements plus all the advantages provided to downstream more complex operations like matching, comparison and search due to the λ spectrum. The time complexity for a range-based lookup is the same as a sorted set $O(\log(N) + M)$. As demonstrated in the `proteins_lookup` example, starting from a collection of λs with a standard deviation of 0.06, it is possible to sort out the single nearest neighbour with a range query on an query interval of $\lambda \pm 10^{-7}$.

6.3 Testing and Validation

The library includes extensive test coverage:

- **Unit Tests**: Core algorithms, edge cases, mathematical properties
- **Integration Tests**: End-to-end workflows, builder patterns
- **Property Tests**: Scale invariance, non-negativity, boundedness
- **Domain Tests**: Molecular dynamics simulations, fractal analysis
- **Performance Tests**: Benchmarks against baseline implementations

6.4 Theoretical properties and tests

- Invariants: tests enforce non-negativity of Rayleigh, near-zero for constant vectors on connected graphs, scale-invariance $\lambda(cx) = \lambda(x)$, and conservative upper bounds via diagonal degrees, aligning with standard spectral graph theory expectations [4].

- Laplacian structure: CSR symmetry, negative off-diagonals, non-negative diagonals, degree–diagonal equality, and deterministic ordering are validated to ensure stable Rayleigh evaluation and reproducible $\lambda\tau$ synthesis across builds [3].

6.5 Practical guidance

- Defaults: a practical starting point is $\epsilon \approx 10^{-3}$, $k \in [3, 10]$, $p = 2.0$, $\sigma = \epsilon$, and `taumode::Median` with $\alpha \approx 0.7$; this keeps the λ-graph connected but sparse and yields bounded $\lambda\tau$ values that mix energy and dispersion robustly for search [14, 9].

- Usage patterns: build ArrowSpace from item rows (auto-transposed internally), let the builder construct the λ-graph and compute synthetic $\lambda\tau$, then use lambda-aware similarity for ranking or ϵ-band ordered sets for range-by-score retrieval; in-place algebra over items supports superposition experiments while preserving spectral semantics through recompute [4, 2, 3].

7 Conclusion

`ArrowSpace` provides a novel approach to vector similarity search by integrating spectral graph properties with traditional semantic similarity measures. The $\lambda\tau$ indexing system offers a memory-efficient way to capture spectral characteristics of vector datasets while maintaining practical query performance. The library's design emphasises both mathematical rigor and computational efficiency, making it suitable for scientific applications requiring spectral-aware similarity search.

The combination of Rust's performance characteristics with innovative spectral indexing algorithms positions `ArrowSpace` as a valuable tool for researchers and practitioners working with high-dimensional vector data where both semantic content and structural properties matter.

Lambda-aware similarity: for query and item ArrowItems, the score combines semantic cosine and λ proximity via $s = \alpha \cos(q, i) + \beta(1/(1 + |\lambda_q - \lambda_i|))$, making search sensitive to both content and spectral smoothness class; setting $\alpha = 1, \beta = 0$ recovers plain cosine.

Range and top-k: `ArrowSpace` exposes lambda-aware top-k, radius queries, and pairwise cosine matrices; examples validate that λ-aware rankings agree with cosine when $\beta = 0$ and diverge meaningfully when blending in λ proximity, with tests covering Jaccard overlap and commutativity of algebraic operations.

The definition of a core library to be used to develop a database solution based on spectral indexing is left to another paper that will include further improvements in terms of algorithms and idioms to make this approach to indexing feasible and efficient in modern cloud installations.

References

[1] Lorenzo Moriondo, *ArrowSpace-rs: Spectral Vector Search with Lambda-Tau Indexing.* Rust codebase for spectral indexing, 2025. https://github.com/Mec-iS/arrowspace-rs

[2] Sridhar Mahadevan, *CMPSCI 791BB: Advanced ML - Spectral Graph Theory.* University of Massachusetts, Amherst, 2006. https://people.cs.umass.edu/~mahadeva/cs791bb/lectures-s2006/lec4.pdf

[3] Peter Grindrod, *Laplacians of Complex Networks.* University of Bristol. https://people.maths.bris.ac.uk/~maajg/teaching/complexnets/laplacians.pdf

[4] Guangliang Chen, *Math 253: Rayleigh Quotient Lecture Notes.* San Jose State University, 2020. https://www.sjsu.edu/faculty/guangliang.chen/Math253S20/lec4RayleighQuotient.pdf

[5] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela, *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.* arXiv preprint arXiv:2005.11401, 2020. Accepted at NeurIPS 2020.

[6] Daniel A. Spielman, *Spectral Graph Theory - Lecture 7.* Yale University, 2007. https://www.cs.yale.edu/homes/spielman/462/2007/lect7-07.pdf

[7] Terence Kelly, *Compressed Sparse Row Format for Representing Graphs.* ;login: The Magazine of USENIX & The Advanced Computing Systems Association, 45(4):76–83, 2020. Programming Workbench Column.

[8] Ulrike von Luxburg, *A tutorial on spectral clustering.* Statistics and Computing, 17:395–416, 2007.

[9] Dexter Chua, *Heat Kernel and Spectral Dimension.* 2025. https://dec41.user.srcf.net/exp/heat_kernel/heat_kernel.pdf

[10] Xiao Bai, *Heat Kernel Analysis On Graphs.* PhD thesis, University of York, 2007.

[11] Xiao Bai and Edwin R. Hancock, *Heat Kernels, Manifolds and Graph Embedding.* Pattern Recognition, University of California, Davis, 2010.

[12] Vlad Orlov, *Smartcore: A numerical library in pure Rust.* 2019. https://github.com/smartcorelib/smartcore

[13] Philip Nelson, *Physical Models of Living Systems.* University of Pennsylvania, 2015. https://www.physics.upenn.edu/biophys/PMLS/

[14] *Dirichlet energy.* Wikipedia, 2024. https://en.wikipedia.org/wiki/Dirichlet_energy