The Mathematics of Seeing

How a computer learns to recognise a face - and why it is just geometry

1. Every Face is Just a List of Numbers

Most days during my internship involve staring at feeds from theater screens lit only by projector glow. This setup must spot recording attempts instantly using one still image alone, judging if any attendee lifts a phone toward the movie. Magic? Not really - what seems like wizardry boils down mostly to number crunching. Begin with something ordinary: every human face can become an ordered sequence of digits.

Picture this. Every photo snapped by a digital camera arrives not as a drawing marked face but as a layout of small blocks known as **pixels**. Each block holds just one value - the light level of its spot in the scene - ranging from zero, which is deep black, up to two hundred fifty-five, meaning full white. Take an image sized at 100×100 - it adds up to ten thousand values lined out. Now shift to a regular HD movie screen size: 1920×1080 - and suddenly there 2,073,600 figures stacked together.

Here's the twist. Reading the number grid row after row - first line, next line, continuing down - builds a single sequence. That string of numbers? It's what math people name a vector. Any picture, no matter the size, turns into such a sequence. When an image holds n pixels, it becomes part of a structure known as $\mathbb{R}^n$, said aloud as "R-n." This idea appears like:

$$\boxed{\boldsymbol{x} = (\boldsymbol{p}_{11}, \boldsymbol{p}_{12}, \dots, \boldsymbol{p}_{1}\boldsymbol{w}, \boldsymbol{p}_{21}, \dots, \boldsymbol{p}^{\mathrm{h}}\boldsymbol{w})^{\mathrm{T}} \in \boldsymbol{R}^{\mathrm{n}}}$$

where the value p_{ij} stands for how bright a dot is at position i down, j across. Height and width in dots define h and w. When something belongs to a set, we write ∈. Each picture with that size fits one spot in this setup. A pair of images showing the same person, shot alike, form points near each other. Images of separate individuals land farther away in

space. To recognize a face, you locate which saved point sits closest. Geometry does the rest - no magic needed.

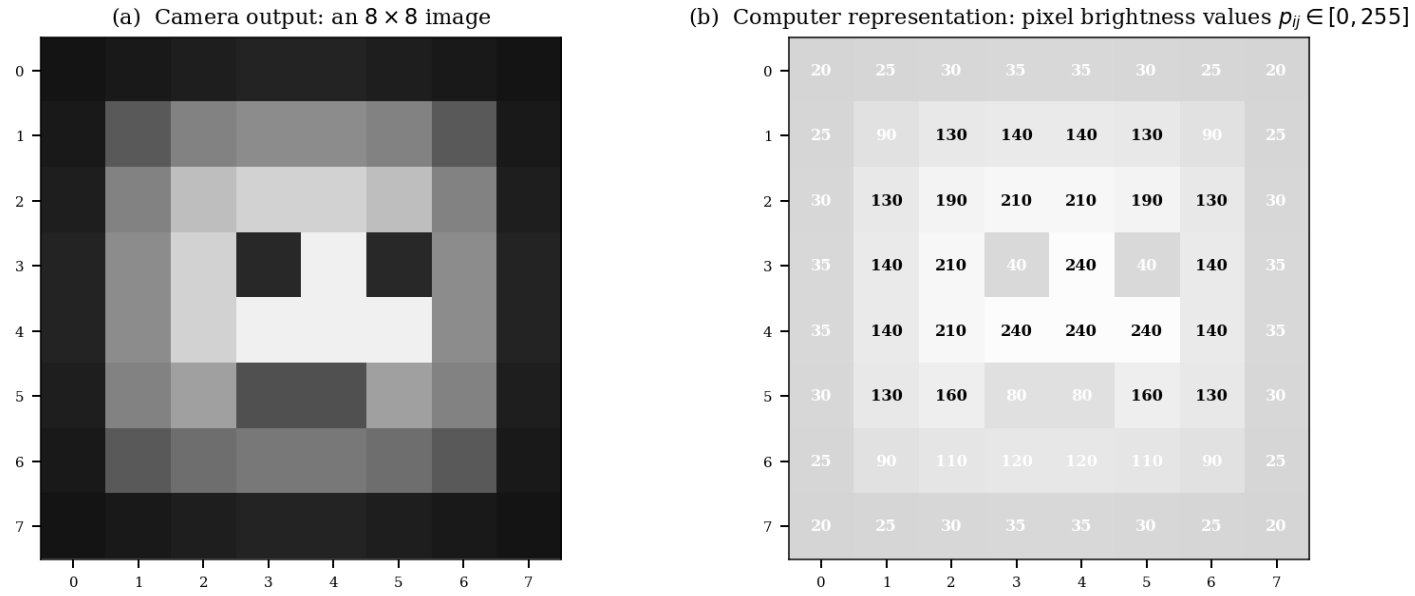

Figure 1. Left: a simplified 8×8 face-like image. Right: the same image as the computer stores it - a grid of integer brightness values $p_{ij} \in \{0, \ldots, 255\}$. Reading row by row produces a vector $x \in \mathbb{R}^{64}$. A real HD cinema frame gives $x \in \mathbb{R}^{2}{}_{jjjjjj}$.

2. How Do You Measure Whether Two Faces Match?

Given two face-vectors **a** and **b**, we need a way to measure how similar they are. The obvious approach is to subtract one from the other and see how big the difference is: this is called the **Euclidean distance**, written $\|\mathbf{a} - \mathbf{b}\|$. But it has a flaw. Photograph the same person in bright sunlight, then in a dim room, and every pixel value changes. The Euclidean distance between those two images will be enormous, even though they show the same face.

The fix is to ask not *how far apart* the two vectors are, but what **angle** they make with each other. This is called the **cosine similarity**, and it is built from the **dot product** - an operation where you multiply corresponding entries of two vectors together and add up the results:

$$\boxed{\boldsymbol{a} \cdot \boldsymbol{b} \;=\; \boldsymbol{a}_1\boldsymbol{b}_1 + \boldsymbol{a}_2\boldsymbol{b}_2 + \cdots + \boldsymbol{a}_n\boldsymbol{b}_n \;=\; \sum^{k} \boldsymbol{a}^{k}\,\boldsymbol{b}^{k}}$$

The cosine of the angle between the vectors is then:

$$\boxed{\boldsymbol{\cos\theta} \;=\; \frac{(\boldsymbol{a} \cdot \boldsymbol{b})}{(\|\boldsymbol{a}\| \cdot \|\boldsymbol{b}\|)}}$$

where $\|\mathbf{a}\| = \sqrt{(a \cdot a)}$ is the length of vector **a**. Dividing by the two lengths is the key move: if you double every pixel value in an image (i.e. make the whole image twice as bright), the numerator doubles but so do both lengths, so the ratio stays exactly the same. Uniform lighting changes become invisible. When cos θ is close to 1 the angle is tiny, meaning the faces are very similar. When it is close to 0 the angle is nearly 90 degrees, meaning they are very different.

Below is a worked example, using four-dimensional vectors (think of them as tiny 2×2 images):

(a) Face vectors in $\mathbb{R}^2$ (schematic)

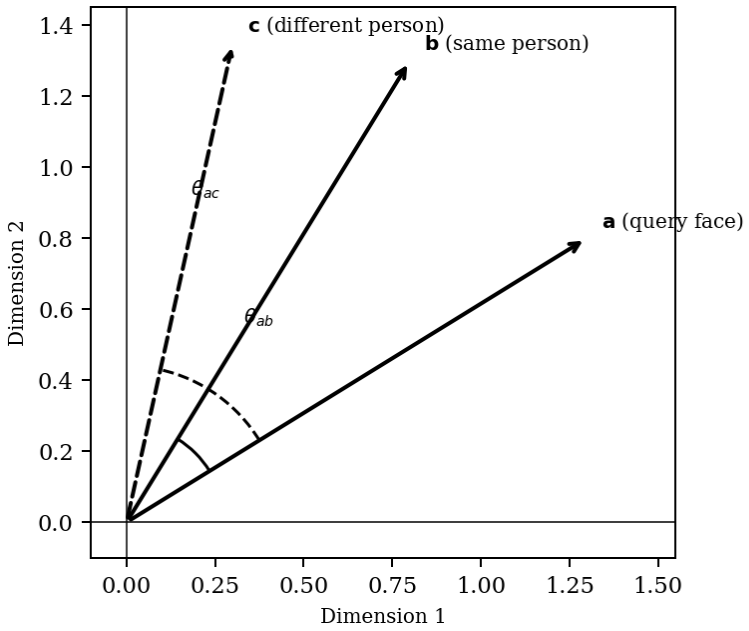

Figure 2. Left: face vectors drawn as arrows from the origin. A small angle $\theta_a{}^b$ between a and b means very similar faces; a large angle $\theta_a{}^c$ means different people. Right: the full worked calculation confirming $\cos(\theta_a{}^b) = 0.996 >> \cos(\theta_a{}^c) = 0.593$.

(b) Cosine similarity: worked calculation

Numerical example (in $\mathbb{R}^4$):

$\mathbf{a} = (210,\ 190,\ 40,\ 240)^\top$
$\mathbf{b} = (205,\ 188,\ 45,\ 238)^\top$ [same person]
$\mathbf{c} = (80,\ 120,\ 200,\ 30)^\top$ [different person]

$$\mathbf{a} \cdot \mathbf{b} = 210\times205 + 190\times188 + 40\times45 + 240\times238 = 43050 + 35720 + 1800 + 57120 = 137690$$

$$\|\mathbf{a}\| = \sqrt{210^2 + 190^2 + 40^2 + 240^2} = \sqrt{44100 + 36100 + 1600 + 57600} = \sqrt{139400} \approx 373.4$$

$$\|\mathbf{b}\| \approx 370.3,\quad \|\mathbf{c}\| \approx 247.6$$

$$\cos\theta_{ab} = \frac{137690}{373.4 \times 370.3} \approx 0.996$$

$$\mathbf{a} \cdot \mathbf{c} = 80\times210 + 120\times190 + 200\times40 + 30\times240 = 16800 + 22800 + 8000 + 7200 = 54800$$

$$\cos\theta_{ac} = \frac{54800}{373.4 \times 247.6} \approx 0.593$$

$\cos\theta_{ab} \approx 0.996 \gg \cos\theta_{ac} \approx 0.593$
$\Rightarrow$ **b** is correctly identified as the same person.

The system correctly identifies **b** as the same person as **a** (cosine near 1), and **c** as someone different (cosine well below 1). That is the mathematical core of face recognition. The rest is about doing this efficiently when n is 2 million rather than 4.

3. The Curse of 2,073,600 Dimensions

Each cosine computation costs about $2n$ arithmetic operations. With $N = 10{,}000$ stored faces and $n = 2{,}073{,}600$ pixels, comparing one new image against the entire database costs:

$$\mathbf{2 \times n \times N = 2 \times 2{,}073{,}600 \times 10{,}000}$$

$$\mathbf{= \boxed{4.1 \times 10^{10}\ operations/frame}}$$

At 30 frames per second that is over a trillion operations per second. Not viable for a real-time system. We need to compress the vectors dramatically - but without losing the information that lets us tell people apart. The key insight is that this should be possible, because face-vectors do not wander randomly through $\mathbb{R}^n$. They are highly structured: all human faces share two eyes, a nose, a mouth, bilateral symmetry. This means face-vectors cluster in a much thinner **subspace** of $\mathbb{R}^n$ - a lower-dimensional slice, like a flat sheet inside a vast room. Principal Component Analysis (PCA) is the mathematical tool that finds this slice.

4. Principal Component Analysis: Finding the Subspace

4.1 The Mean Face

Begin with a training set of m labelled photographs. Compute the **mean face** μ - the average of all training vectors, entry by entry:

$$\boxed{\boldsymbol{\mu} = \frac{1}{m} \times (\boldsymbol{x}_1 + \boldsymbol{x}_2 + \cdots + \boldsymbol{x}_m)}$$

Then subtract μ from every training image to produce centred vectors $\tilde{\mathbf{x}}_i = \mathbf{x}_i - \boldsymbol{\mu}$. This is asking: how does each face differ from the average face? Stack these as columns of a matrix $\tilde{\mathbf{X}} \in \mathbb{R}^{n \times m}$. The **sample covariance matrix** is:

$$\boxed{\boldsymbol{C} = \left(\frac{1}{m-1}\right) \times \tilde{\boldsymbol{X}} \times \tilde{\boldsymbol{X}}^T \in \mathbb{R}^{nxn}}$$

The entry at row j, column k of $\mathbf{C}$ tells you how much pixels j and k tend to vary together. If they are typically bright or dark at the same time, the entry is large. The covariance matrix encodes everything PCA needs to know about the structure of human faces.

4.2 Eigenfaces: The Eigenvector Decomposition

$\mathbf{C}$ is a real symmetric matrix. The **spectral theorem** from linear algebra guarantees that it can be decomposed into a set of **eigenvectors** e_k and associated **eigenvalues** λ_k, satisfying:

$$\boxed{\boldsymbol{C} \times \boldsymbol{e}^k = \boldsymbol{\lambda}^k \times \boldsymbol{e}^k, \quad \boldsymbol{k} = \mathbf{1, 2, \ldots, n}}$$

Think of an eigenvector as a special direction in pixel space: multiplying it by **C** does not rotate it at all, just stretches it by a factor λ_k. The eigenvector with the largest eigenvalue λ_1 points in the direction along which the training faces vary the most. The eigenvector with the second-largest eigenvalue points in the direction of greatest remaining variation orthogonal to the first. And so on. When you visualise these eigenvectors as images, they look like ghostly, blurred face-like patterns. They are called **eigenfaces**.

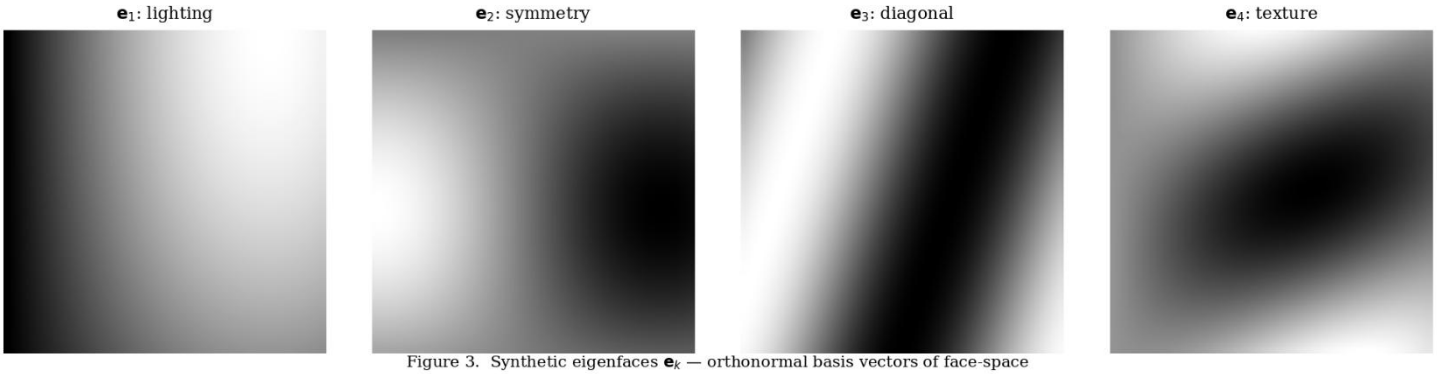

Figure 3. Synthetic eigenfaces $\mathbf{e}_k$ — orthonormal basis vectors of face-space

Figure 3. Four synthetic eigenfaces e^k, shown as images. Each is a direction in pixel space capturing an independent type of variation across the training set. Early eigenfaces tend to encode large-scale features like global lighting; later ones capture finer structure.

The fraction of total variation captured by the first *k* eigenfaces is:

$$V(k) = \frac{(\lambda_1 + \lambda_2 + \cdots + \lambda^k)}{(\lambda_1 + \lambda_2 + \cdots + \lambda_n)}$$

In real face datasets, $V(k)$ reaches 90% for *k* between 50 and 150, compared to $n \approx 10{,}000$. That means just 50 to 150 numbers can describe a face nearly as well as 10,000. The other 9,850-plus dimensions are mostly noise.

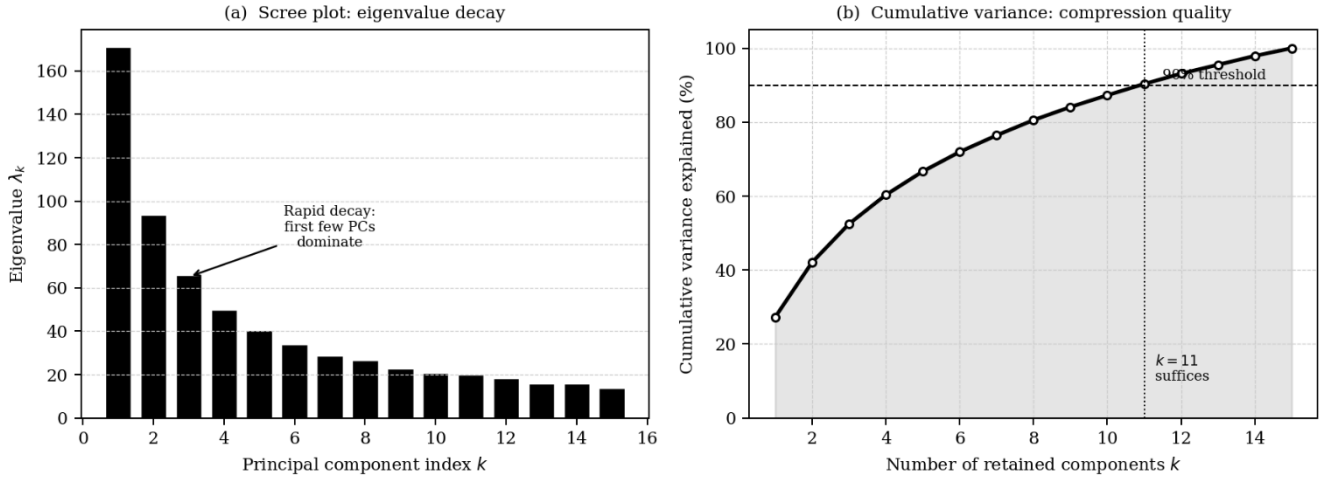

Figure 4. Left: the eigenvalue λ^k for each principal component decays rapidly - the first few eigenfaces matter far more than the rest. Right: cumulative explained variance V(k). The 90% threshold is reached at a small value of k, justifying a dramatic compression of the data.

4.3 A Faster Route to the Eigenfaces

For n = 10,000, the covariance matrix **C** is 10,000×10,000 = 100 million entries, requiring 800 MB of storage and $O(n^3) = O(10^{12})$ operations to diagonalise. Turk and Pentland (1991) observed that **C** has at most $m - 1$ non-zero eigenvalues (one fewer than the number of training images). So instead of working with the huge $n \times n$ matrix, you diagonalise the much smaller $m \times m$ dual matrix $\boldsymbol{L} = \left(\frac{1}{m-1}\right) \widetilde{\boldsymbol{X}}^{\mathrm{T}} \widetilde{\boldsymbol{X}}$, then recover the eigenfaces as:

$$\boxed{\boldsymbol{e}^{\mathrm{k}} = \frac{\boldsymbol{X} \cdot \boldsymbol{v}^{\mathrm{k}}}{\sqrt{(\boldsymbol{m}-\mathbf{1}) \times \boldsymbol{\lambda}^{\mathrm{k}}}}}$$

For m = 1,000 training images this drops the cost from $O(10^{12})$ to $O(10^9)$ operations - a thousand times faster, for exactly the same eigenfaces.

5. Recognising a Face in Five Steps

With eigenfaces in hand, recognition is clean. During a **training phase**, we project each training image onto the k eigenfaces to produce a short vector called its **face code**. Given eigenface matrix **E** (whose columns are the eigenfaces), and a training image **x**:

$$\boxed{\boldsymbol{w} = \boldsymbol{E}^{\mathrm{T}} \times (\boldsymbol{x} - \boldsymbol{\mu}) \in \mathbb{R}^{\mathrm{k}}}$$

The face code **w** records how much of each eigenface direction is present in the image. Because the eigenfaces are orthonormal ($\mathbf{E}^{\mathrm{T}}\mathbf{E} = \mathbf{I}$), this projection is lossless within the eigenface subspace. What is lost - the **reconstruction error** - is exactly:

$$\boxed{\|\boldsymbol{x} - \boldsymbol{\mu} - \boldsymbol{E} \times \boldsymbol{w}\|^2 = \sum_{\boldsymbol{j}=\boldsymbol{k}+\mathbf{1}}^{\boldsymbol{n}} (\boldsymbol{e}_{\boldsymbol{j}}^{\boldsymbol{T}} \times (\boldsymbol{x} - \boldsymbol{\mu}))^2}$$

This is identically zero if we keep all n eigenfaces, and decreases as k grows. In practice, accepting a small reconstruction error in exchange for dramatically shorter face codes is the trade-off that makes the whole system viable.

At **recognition time**, a new image is projected to its face code, and the predicted identity is whichever stored code is closest to it:

$$\boxed{\hat{\mathrm{i}} = \boldsymbol{argmin}_{\mathrm{i}} \, \| \boldsymbol{w} - \boldsymbol{w}_{\mathrm{i}} \|}$$

This is the **1-nearest-neighbour classifier** in $\mathbb{R}^{\mathrm{k}}$. A threshold δ can be applied: if the nearest stored code is still more than δ away, the system returns *unknown identity* rather than forcing a bad match.

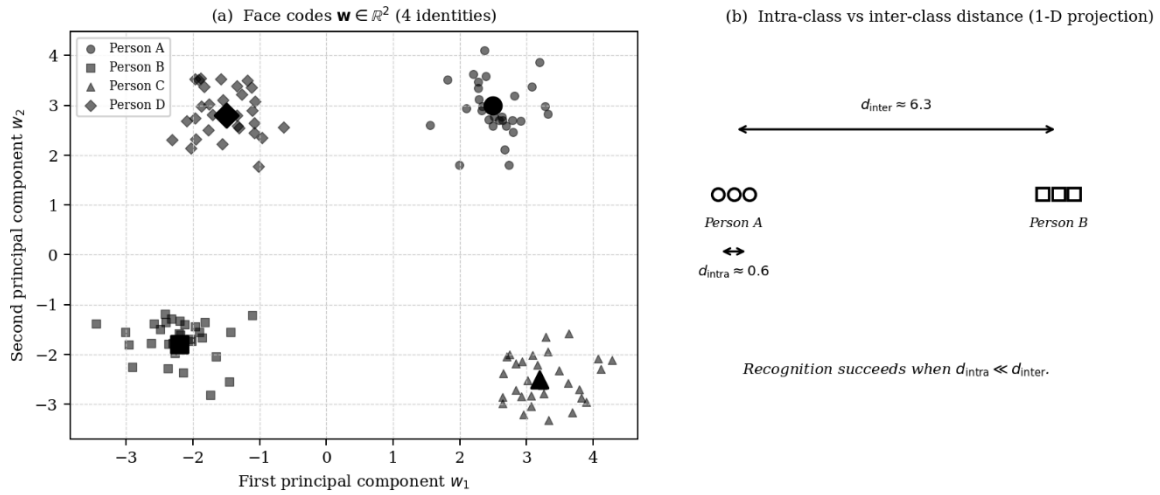

Figure 5. Left: face codes w projected into two dimensions for four identities. Each cluster represents the same person photographed many times; they group tightly (small d_intra) with good separation between clusters (large d_inter). Right: recognition succeeds reliably when d_intra << d_inter.

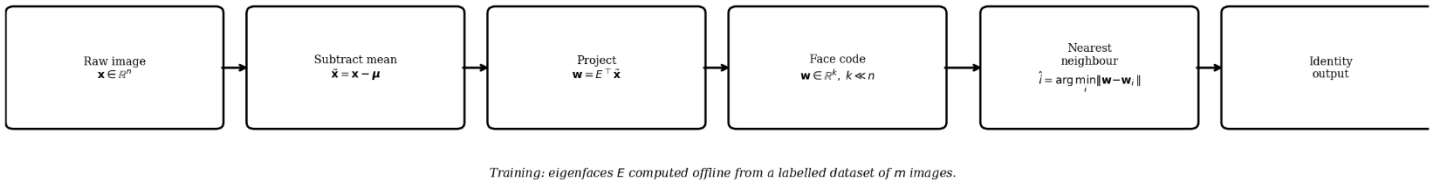

Figure 6. The complete pipeline from raw pixel data to a recognised identity. The eigenface matrix E is computed once, offline. At query time, five steps reduce 2,073,600 numbers to around 150, then find the nearest stored code.

6. How Much Faster Is This? A Concrete Calculation

Take a database of $N = 10{,}000$ faces, queried at 30 fps, with $k = 150$ eigenfaces. Two costs: projecting the new image onto eigenfaces ($2nk$ operations), then searching the database ($2kN$ operations):

$$\boxed{\boldsymbol{Cost\_eigen = 2 * n * k + 2 * k * N}}$$

$$= 2 \times 2{,}073{,}600 \times 150 + 2 \times 150 \times 10{,}000$$

$$= 622{,}080{,}000 + 3{,}000{,}000$$

$$\approx \boxed{6.25 \times 10^8\ operations/frame}$$

Compare this to pixel-space comparisons:

$$\boxed{\begin{aligned} \boldsymbol{Cost_{pixel}} &= \boldsymbol{2 * n * N} \\ &= \boldsymbol{2 \times 2{,}073{,}600 \times 10{,}000} \\ &= \boldsymbol{4.1 \times 10^{10}\ operations/frame} \end{aligned}}$$

The compression ratio is:

$$R = \frac{Cost_{pixel}}{Cost\_eigen} = \frac{4.1 \times 10^{10}}{6.25 \times 10^{8}} \approx 6,560$$

A saving of roughly **6,500-fold** - for free, just by understanding the geometry. In the limiting case where the database is very large ($N >> k$), the ratio approaches:

$$R \to \frac{n}{2k} = \frac{2,073,600}{300} = 6,912$$

That is the number that convinced me this was worth writing about. A 6,912-fold reduction in computation is not an engineering optimisation someone thought up: it falls directly from the same mathematics that told us 150 eigenfaces capture 90% of facial variation. The maths gives you the compression *and* the speedup in a single package.

7. What Modern Systems Do Differently

The eigenface method was state of the art in 1991. Modern systems, including Face ID on your phone, use deep neural networks instead - but here is something I find genuinely surprising: the *geometry* is identical. A neural network trained for face recognition learns a mapping $f: \mathbb{R}^n \to \mathbb{R}^k$ that is highly non-linear, but the final recognition step is still a nearest-neighbour search in $\mathbb{R}^k$. The network is trained using a **triplet loss**:

$$L = max(0, \quad \|f(x_a) - f(x_p)\|^2 - \|f(x_a) - f(x_n)\|^2 + \alpha)$$

where x_a is an anchor image, x_p is a *positive* example (same person as the anchor), x_n is a *negative* example (different person), and $\alpha > 0$ is a margin. Minimising L pulls same-person embeddings together and pushes different-person embeddings apart. It is enforcing, by gradient descent, exactly the condition $d_{intra} << d_{inter}$ from Figure 5 - now for a non-linear mapping rather than a linear projection.

High dimensionality, far from being a problem, turns out to be what makes recognition reliable. If embedding vectors are roughly uniformly distributed on the unit sphere S^{k-1}, the probability that a random person's embedding falls within angular distance θ of your stored code is:

$$P(cos\,\theta > 1 - \delta) \approx C^k \times \delta^{\frac{k-1}{2}}$$

which decreases *exponentially* in k for fixed δ. Apple reports a false-acceptance probability for Face ID of approximately 10^{-6} - one in a million. This is not a coincidence of engineering. It is a consequence of this geometry: in a sufficiently high-dimensional space, with a sufficiently good embedding, no two people's face codes overlap.

8. Conclusion

The next time someone unlocks their phone with their face, here is what is actually happening in the next 30 milliseconds: the camera produces a vector of ~2 million numbers; a trained mapping projects it to roughly 150 numbers; a nearest-neighbour search compares those 150 numbers against a stored template; and the phone unlocks if the cosine similarity is close enough to 1. The whole thing - from image to identity - is dot products and geometry.

What I find remarkable is how much the mathematics gives you for free. The same spectral decomposition of the covariance matrix that tells you *which* 150 numbers to keep also tells you that working with those 150 numbers is 6,912 times cheaper than working with 2 million. The compression and the speedup are not separate ideas - they are the same idea, seen from two angles.

I started this essay asking how a machine represents a face. The answer turned out to be: as a point in a very high-dimensional space, close to other points that show the same person, and far from points that show anyone else. Recognition is just finding the nearest point. And the whole machinery that makes this fast and accurate? It is linear algebra, doing what linear algebra always does: finding the right way to look at the problem.

References

[1] Turk, M. A. and Pentland, A. P. (1991). Eigenfaces for recognition. Journal of Cognitive Neuroscience, 3(1), pp. 71–86.

[2] Jolliffe, I. T. (2002). Principal Component Analysis, 2nd edn. Springer.

[3] Schroff, F., Kalenichenko, D. and Philbin, J. (2015). FaceNet: a unified embedding for face recognition and clustering. Proceedings of CVPR 2015, pp. 815–823.

[4] Belhumeur, P. N., Hespanha, J. P. and Kriegman, D. J. (1997). Eigenfaces vs. Fisherfaces: recognition using class specific linear projection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 19(7), pp. 711–720.

[5] Apple Inc. (2017). Face ID Security. Apple Platform Security Guide.

[6] Viola, P. and Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. Proceedings of CVPR 2001, Vol. 1, pp. 511–518.