Characterizing neuronal population geometry with AI equation discovery

Reilly Tilbury[1], Dabin Kwon[1], Ali Haydaroglu[2], Jacob Ratliff[1], Valentin Schmutz[1], Matteo Carandini[2], Kevin Miller[3], Kim Stachenfeld[3], Kenneth D. Harris[1]

[1]UCL Queen Square Institute of Neurology, University College London, London, UK. [2]UCL Institute of Ophthalmology, University College London, London, UK. [3]Google DeepMind. *Correspondence: kenneth.harris@ucl.ac.uk.

The visual cortex contains millions of neurons, whose combined activity forms a population code representing visual stimuli. There is, however, a discrepancy between our understanding of this code at the single neuron and population levels: direct measurements suggest that the population code is high-dimensional, but current understanding of single-cell tuning curves implies low dimensionality. To reconcile this discrepancy, we used AI to find a new parsimonious, interpretable equation for tuning curves to oriented stimuli. Candidate equations were expressed as short computer programs, and evolved using Large Language Models (LLMs) to improve their fits to single-cell tuning. This resulted in an equation that not only improved single-cell fits, but also accurately modelled the population code's high-dimensional structure, even though population dimensionality was not the AI system's objective. This high-dimensionality occurred because the AI-derived tuning curves were not smooth at their peaks, which we proved is mathematically required for high-dimensional population coding of low-dimensional stimuli. We used the AI-generated tuning equation to demonstrate the advantages of high-dimensional codes in a simulated hyperacuity task, and showed that non-smooth peaks are also present in other brain systems, leading to high-dimensional population coding in head direction cells. These results suggest that the smoothness of neuronal tuning curves has an important role in setting the dimensionality of the population code, and demonstrate how AI equation discovery can accelerate scientific theory building in neuroscience and beyond.

The visual cortex comprises myriad neurons, whose activity together forms a "population code" representing image stimuli. The properties of the cortical population code determine the types of computations that can be performed by downstream brain structures. Theoretical work has long recognized the advantages of high-dimensional codes, which allow downstream neurons to flexibly read out diverse sensory features by just linear synaptic integration[1–5]. The cortex can form high-dimensional codes by embedding a manifold of possible sensory stimuli nonlinearly into a high-dimensional space of neuronal activity patterns. Nonlinear transformation of sensory inputs into high-dimensional codes, which allow linear readout of behaviorally-relevant features, has been proposed as a primary function of sensory cortex[6–9].

The dimensionality of the neural code for a particular stimulus set can be revealed using principal component analysis (PCA) of neuronal response vectors. The PCA eigenspectrum provides a sequence of numbers V_n measuring the variance of activity along the nth embedding dimension of the response manifold, whose dependence on n characterizes the population code's dimensionality. If the eigenspectrum decays exponentially, $V_n \sim e^{-n/M}$ or $V_n \sim e^{-n^2/2M^2}$, this indicates a low-dimensional code in which M sets a dimension scale beyond which variances rapidly vanish. If instead the eigenspectrum decays as a power law $V_n \sim n^{-\alpha}$, this indicates a higher-dimensional code with no characteristic dimension scale[10].

The population code emerges from the sensory tuning of individual neurons, but there is a discrepancy between our understanding of single neuron responses and population coding in visual cortex. This discrepancy is most evident for responses to gratings of varying direction. Single-neuron orientation tuning has been measured many times since the 1960s, typically by varying drift direction θ in steps of 15 or 30 deg, and fitting the results with a smooth function, such as the "double Gaussian"[11]:

$$f(\theta) = b + A_1 e^{-|\theta-\phi|^2/2\sigma^2} + A_2 e^{-|\theta-\phi-\pi|^2/2\sigma^2}$$

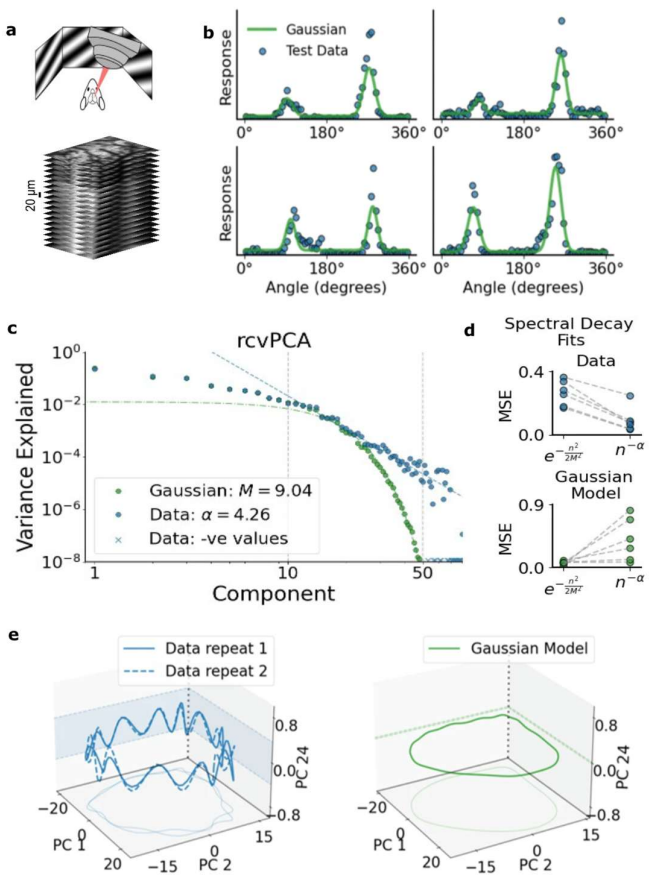


Figure 1 | Oriented stimuli produce a high-dimensional population code not captured by standard tuning curve models. a, Visual cortical population activity was recorded in awake mice passively presented with full-field drifting grating stimuli. **b,** Responses of four example cells as a function of grating direction (blue dots) together with double Gaussian model fits (green curves). **c,** Population eigenspectrum for an example experiment (blue dots; crosses indicate dimensions for which rcvPCA estimates are negative) and eigenspectrum of a population of double-Gaussian tuning curves fit to the recorded cells (green dots). Blue dashed line: $n^{-\alpha}$ fit to actual eigenspectrum; green dashed line: $e^{-n^2/2M^2}$ fit to eigenspectrum of Gaussian tuning curves, both with parameters estimated in the region 10 < n < 50. **d,** Mean-square error of $e^{-n^2/2M^2}$ and $n^{-\alpha}$ fits to eigenspectra of actual data (upper; each pair of blue points joined by dashed line is one recording) and Gaussian model (lower). **e,** Population responses and the Gaussian model predictions of the population projected onto principal components 1, 2, and 24. Left: two repeats of experimental data showing consistent high-dimensional variance. Right: Gaussian model showing low-dimensional structure.

This equation predicts a low-dimensional population code, but the actual code is high-dimensional. To show this, we analyzed 2-photon calcium recordings of visual cortical responses to drifting gratings of fine angular spacing (Fig. 1a-b). We applied regularized cross-validated PCA (rcvPCA; Extended Data Fig. 1; Methods), which estimates the eigenspectrum of stimulus responses discarding stimulus-independent variability. This revealed a power-law eigenspectrum decay with exponent ≈ 4 (Fig. 1c). Next, we fit tuning double-Gaussian curves to each neuron (Fig. 1b) and computed the PCA eigenspectrum of the population of fit tuning curves, obtaining exponential decay indicative of a low-dimensional code (Fig. 1c-d). This discrepancy could be visualized by projecting both actual population activity, and its fit by the double-Gaussian equation, onto the principal component axes (Fig. 1e). This discrepancy is not restricted to Gaussian tuning: mathematically, no family of tuning curves which is infinitely differentiable can produce eigenspectra with power-law decay (Appendix 1a,b; Ref. [9]). Because current orientation tuning models have this property, explaining high-dimensional population coding requires a new equation for single-cell tuning curves.

We had not found a suitable equation with traditional means, so turned to a new approach, which we term AI equation discovery (Fig. 2a). This approach differs to previous uses of AI in studying neural tuning, which involve training deep networks to match visual responses[12–20], resulting in networks which can reconstruct responses with high accuracy, but are challenging to interpret. AI equation discovery instead aims to produce a parsimonious, interpretable equation summarizing how neurons respond to stimuli.

We implemented this using an evolutionary approach [21–24] (Fig. 2a). Equations for tuning curves are expressed as Python programs predicting a cell's firing rate response to each stimulus, given cell-specific tuning parameters. The programs are scored by cross-validated mean-squared error, penalized by program complexity (Extended Data Fig. 2), and all programs and scores are entered into a database. Programs are evolved by repeatedly sampling two high-scoring "parent" programs from the database, and prompting an LLM to

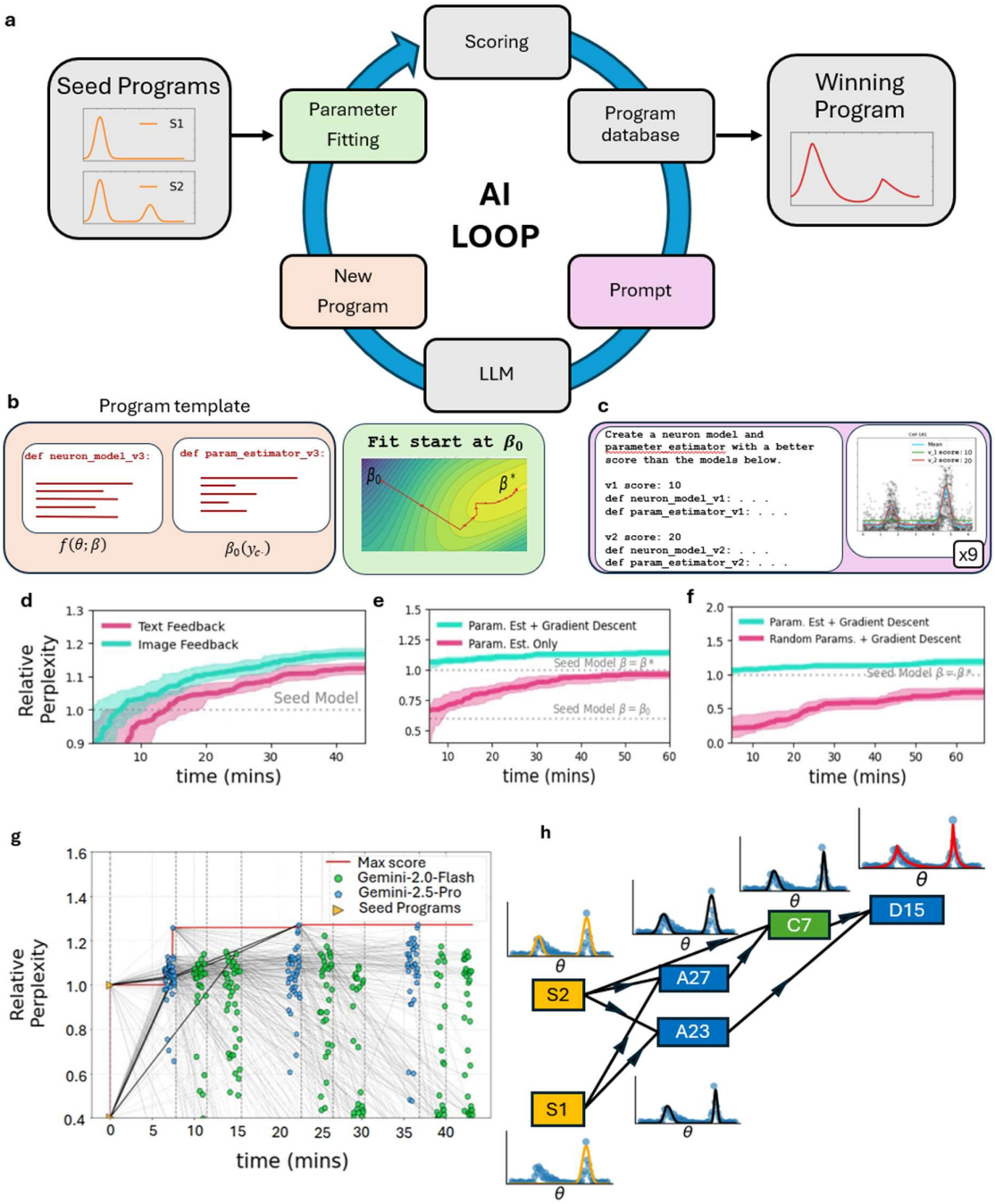


Fig. 2 | AI equation discovery system. a, System flowchart. Tuning curve equations are represented as Python programs, which are scored by complexity-penalized cross-validated squared error, and entered into a database. An LLM is repeatedly prompted to improve on current high-scoring example programs, and the LLM-generated programs are scored and entered into the database ("AI LOOP"). Finally, a winner is selected using held-out cells. **b,** Each program comprises a tuning function $f(\theta, \beta)$ that predicts the response of a cell with tuning parameters β to stimulus θ, and a parameter estimator $\hat{\beta}_c(y_{c\cdot})$ that provides a starting point for gradient optimization of each cell's tuning parameter β_c given its measured responses $y_{c\cdot}$. **c,** Prompts quote two high-scoring programs and their scores, and contain graphical diagnostics comparing these programs' predictions to actual responses for nine randomly selected cells. **d,** Ablation test comparing text-only feedback (magenta) vs. system including graphical diagnostics (cyan). **e,** Ablation test comparing system without gradient descent (magenta) to full model (cyan). **f,** Ablation test comparing system with random search initialization (magenta) to that with the parameter estimator (cyan). Solid lines in d-f show median fitness of the top cohorts (defined as the top 24 programs of each condition, pooled across 4 independent evolutions), error bars show 25th and 75th percentiles of the top cohorts **g,** Summary of programs generated during equation discovery. d-g, X-axis: wall-clock time of program generation. Y-axis: perplexity (i.e. exponential of program score), relative to that of double-Gaussian seed model. Red line: running-best score. Black lines: winner's family tree. **h,** Family tree of the winning model (left), and fits of each model in it for an example cell.

generate a new, higher-scoring "child" model. The prompt quotes the code of both parent programs, as well as

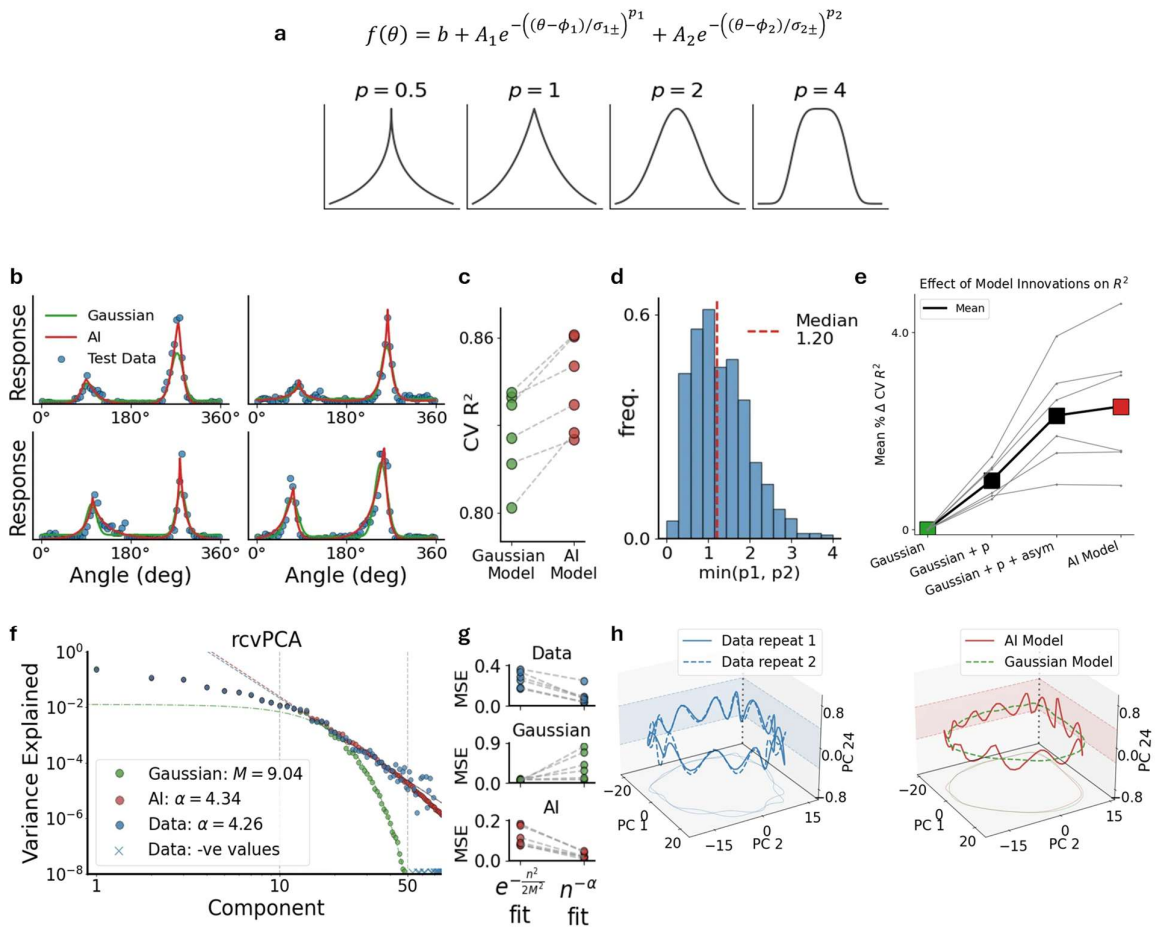


Fig. 3 | AI-discovered tuning function explains high dimensional population code. a, Shape of e^{-x^p} function peaks, illustrating cusp for p=0.5, triangular for p=1, smooth Gaussian for p=2 and flat for p=4. **b,** AI-derived tuning function $f(\theta) = b + A_1 e^{-\left((\theta-\phi_1)/\sigma_{1\pm}\right)^{p_1}} + A_2 e^{-\left((\theta-\phi_2)/\sigma_{2\pm}\right)^{p_2}}$ fit to the same example cells as in Fig. 1b. Blue points: experimental data; red curve: AI model; green curve: double-Gaussian model. **c,** Fit quality of double-Gaussian and AI models, as assessed by cross-validated R^2. Circles: individual experiments. **d,** Histogram over all cells of exponent p (minimum value over the two peaks, which determines the power-law exponent). **e,** Quantifying each parameter's effect by systematically ablating parameters from the equation. Thin curves show change in explained variance over double-Gaussian model, averaged over cells for single experiments; thick curve: mean. **f,** Eigenspectrum of AI model (red points) with eigenspectra of actual data (blue) and Gaussian model (green). **g,** mean-square error of $e^{-n^2/2M^2}$ and $n^{-\alpha}$ fits to each spectrum, as in Figure 1d. **h,** Population responses and AI model projected onto principal components 1, 2, and 24, as in Fig 1e. Left: two repeats of experimental data. Right: AI model (red) and double-Gaussian model (green dashed).

graphical diagnostics comparing the parent models to experimental data (Fig. 2b-c; Extended Data Fig. 3), a new feature which we hypothesized might help find better solutions than scores alone. The system ran for approximately 45 minutes of wall-clock time, wherein 576 programs were created and evaluated at a cost of $8.25 in LLM tokens. Ablation tests indicated significant improvements from giving the LLMs graphical model diagnostics, as well as another new feature whereby the LLMs also write code to estimate an approximate parameter fit, used as a gradient descent starting point to avoid local minima (Fig. 2d-f; Extended Data Fig. 4).

The winning model was identified as that with best cross-validated performance on a set of test cells held out from model evolution. The winning model's "family tree" showed that features were progressively added to the double-Gaussian model over several generations (Fig. 2g,h) resulting in a final equation

$$f(\theta) = b + A_1 e^{-\left(|\theta-\phi_1|/\sigma_{1\pm}\right)^{p_1}} + A_2 e^{-\left(|\theta-\phi_2|/\sigma_{2\pm}\right)^{p_2}}$$

This has three key differences to the original double-Gaussian model. Most importantly, the power of 2 in the Gaussian has been replaced by parameters p_1 and p_2, which control the shape of the two peaks (Fig. 3a). If $p =$

2, this yields the usual infinitely-differentiable Gaussian function. When $p < 2$, the curve is "peaky", and not infinitely-differentiable; when $p < 1$ it has cusp-shaped peaks, allowing it to fit large responses to exactly preferred stimuli which appeared to be outliers when fit by the Gaussian (Fig. 3b-d), but occurred specifically at each cell's preferred orientation, indicating a reliable feature of neuronal tuning (Extended Data Figs. 5,6). Second, the model allows the two preferred directions ϕ_1 and ϕ_2 to take arbitrary values rather than be 180° apart as in the original model. Finally, $\sigma_{1\pm}$ represents two parameters σ_{1+} and σ_{1-} allowing different widths to the left and right of the first tuning peak ($\sigma_{1\pm} = \sigma_{1-}$ when $\theta < \phi_1$ and $\sigma_{1\pm} = \sigma_{1+}$when $\theta > \phi_1$), and similarly for $\sigma_{2\pm}$. Ablation tests showed that all three features improved cross-validated explained variance (Fig. 3e), although this improvement was relatively modest as the new parameters improved the fit of only stimuli close to the tuning-curve peaks.

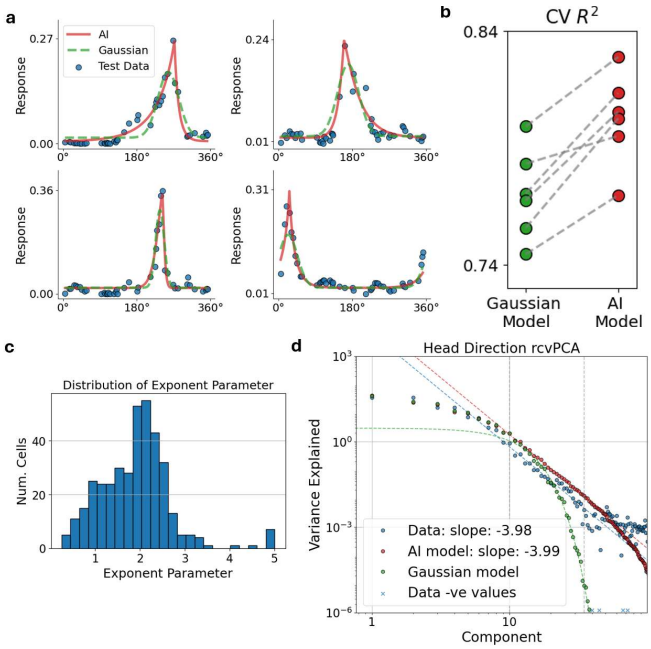


Figure 4 | Analysis of head-direction cell data. a, Head-direction tuning of four example cells from AD thalamus (blue points), together with AI (red) and Gaussian (dashed green) tuning curves. **b,** Fit quality of single-Gaussian and single-peaked AI model, as assessed by cross-validated R^2. Circles: individual mice. **c,** Histogram of p parameter across cells. **d,** Eigenspectrum of HD cell population activity (blue) together with Gaussian (green) and AI (red) models. Blue and Red dashed lines: $n^{-\alpha}$ fit to population eigenspectrum and AI model; green dashed line: $e^{-n^2/2M^2}$ fit to Gaussian model.

The new AI-derived tuning equation, unlike the traditional model, predicted the observed population code dimensionality. To show this, we fit AI-derived tuning curves to each cell. The PCA eigenspectrum of the population of AI-model tuning curves closely matched the n^{-4} powerlaw structure of the original data (Fig. 3f) and higher dimensionality could be visualized by principal component projection (Fig. 3g). Because the AI system found an equation for the tuning curves, we could explain this result analytically, proving that for $p < 2$, tuning curves of the form $e^{-|x|^p}$ generate population codes with PCA eigenspectrum decaying as $n^{-2(p+1)}$ due to their non-infinitely-differentiable peaks (Appendix 1b); for double-peaked tuning curves, p is the minimum of the values for the two peaks. The values of p fit to V1 data were centered around 1 (Fig. 3d), and thus consistent with the observed eigenspectrum power exponent of around 4. Notably, this dramatic improvement in predicting the high-dimensional structure of the population code occurred even though the LLM's objective function was based on single-cell error, not population geometry, and even though the AI system produced only a relatively modest improvement in its single-cell-based objective function.

To investigate whether peaky tuning might generate high-dimensional population codes in brain systems beyond visual cortex, we analyzed data from head-direction-cells (Fig. 4). Head-direction cells also encode an angular variable, the head direction of a subject physically moving through space. We analyzed data collected from head-direction cells in the thalamus and post-subiculum of mice freely exploring an open field environment[25], fitting each recorded cell with the AI-derived tuning equation (modified to have only one peak, matching the single-peaked nature of head-direction tuning curves). Individual cells had peaky tuning curves, which were better fit by the AI-derived model than Gaussian tuning (Fig. 4a-b). Apply rcvPCA to a population of head direction cells (pooling cells from all to enable more accurate estimation of higher dimensions) revealed a high-dimensional power-law eigenspectrum, that was again accurately captured by the AI-derived tuning equation (Fig. 4c).

To investigate the computational advantages of a high-dimensional population code for angular data, we took advantage of the fact that the AI system provided an explicit equation. We modeled a hyperacuity task, where a simulated agent is required to discriminate whether the stimulus orientation is in a region smaller than the tuning curve half-width, which can be performed by humans and animals[26–28]. We considered a population of symmetrical tuning curves of the form $e^{-(|\theta-\phi|/\sigma)^p}$, with fixed half-width σ =15° and evenly-distributed tuning curve centers, and asked how well a linear decoder of this activity could discriminate orientations closer or further than 3° from a target, for different values of p (Fig. 5a). We first simulated a downstream readout that learns by Hebbian plasticity, which showed better performance and faster learning for p=0.5 or 1.0 than p=2.0 or 2.5 (Fig. 5b).

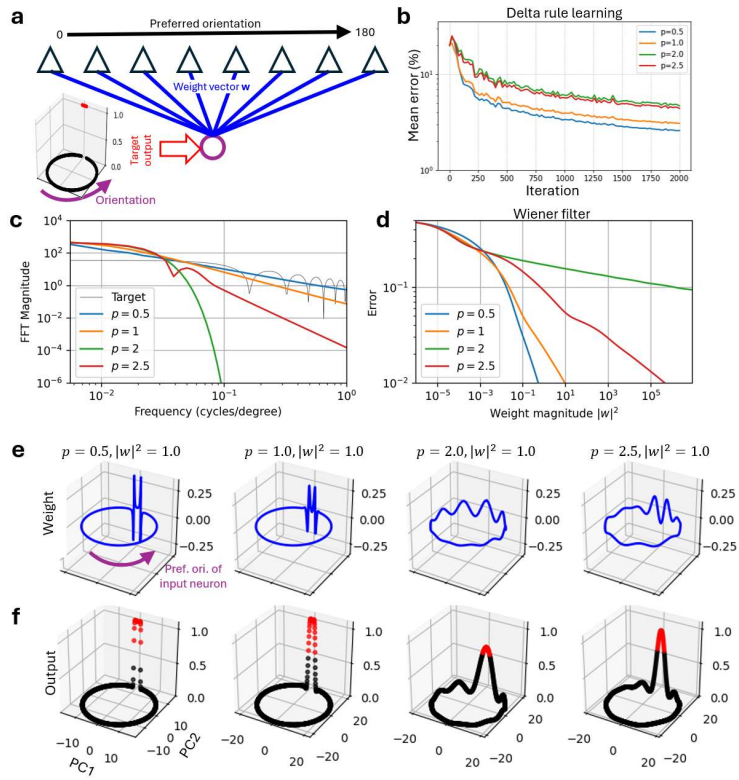


Figure 5 | Effect of tuning peakiness on a simulated hyperacuity task a, Simulation set up: a population of tuning curves of the form $e^{-(|\theta-\phi|/\sigma)^p}$ with evenly distributed preferred orientations, width σ =15°, and varying p, are linearly read out to discriminate a target function. The target function (inset) is 1 for orientations within 3° of a target angle (red points) and 0 otherwise (black points). **B**. Readout error vs. time for a Hebbian learning rule, for different values of p. **C**. Fourier transform of the target function (black) and tuning curves for different values of p (colors). **D**. Mean squared error achievable with optimal weight choice, as a function of weight norm ($|w|^2$) for different values of p. **E**. Optimal readout weights with fixed norm ($|w|^2$ = 1), for different values of p. X- and y-axes: sine and cosine of preferred angle ϕ of source neuron. Z-axis: weight value. **F**. Optimal predictions using the weights shown in E. X- and y-axes: first two principal components of population activity, reflecting sine and cosine of stimulus angle (note total variance for these components when p=0.5). Z-axis: optimal readout of boxcar target. Black points: angles for which target is 0, red points, angles for which target is 1.

The AI-derived equation allowed us to derive an analytical explanation of why. Because the neurons are arranged on a circle of preferred orientation ϕ, a weighted sum of their neural activity is the circular convolution of the tuning curve and the weight function, which in the Fourier domain becomes a multiplication: $F\left[\int w(\phi)f(\theta-\phi)d\phi\right] \approx F[w]F[f]$. Accurate readout with reasonably-sized weights therefore requires the Fourier transform of the tuning curve to be comparable in magnitude to that of the target function. The boxcar target function has Fourier decay ~ n^{-1} , while the modeled tuning curves decay as $n^{-(p+1)}$ for $p \neq 2$, but exponentially fast for $p = 2$ (Fig. 5c) Thus, accurate reconstruction of the target function with reasonable weights is possible for small values of p, but not for $p = 2$ (Fig. 5d). To illustrate this, we found the exact optimal solutions for fixed weight vector norm using a Wiener filter, and found that for p=0.5 or 1, accurate solutions are obtained by integrating the activity of neurons with preferred orientation near the target value (Fig. 5e-f). For p=2 however, the optimal weights are a widespread and oscillatory function of preferred orientation, and the reconstruction is substantially worse. Intriguingly, even the case of p=2.5, whose tuning curves are flatter than Gaussian but not infinitely differentiable, yielded better performance than the smooth Gaussian with p=2. Thus, the Gaussian tuning curve is exceptionally unsuited for making fine orientation discriminations such as required for hyperacuity, while peaked tuning curves allow better performance.

Finally, we considered what circuit mechanisms might generate peaky stimulus tuning, even in cases where external inputs are infinitely differentiable (Fig. 6). A single-cell nonlinearity cannot generate this robustly. Indeed, by Taylor's theorem, an infinitely-differentiable input tuning curve $f(\theta)$ can be approximated near its maximum θ_0 by a quadratic $f(\theta) \approx A(1-\beta^2(\theta-\theta_0)^2)$. Transforming this to $A(1-\beta|\theta-\theta_0|)$, which more closely matches the observed tuning curves, requires a transformation $f \mapsto A - \sqrt{A-f}$, which has infinite slope at $f = A$ and is undefined for $f > A$. There are no known single-cell processes that generate this kind of a transformation, and furthermore the maximum input A would not even be constant due to changes in input strength. Nevertheless, we illustrated how such a computation could be performed by a recurrent network, using an exactly solvable variant of the "stabilized supralinear network" model [29] (Fig. 6a-d; Appendix 1c). The model contains excitatory populations tuned to each stimulus, connected by supralinear recurrent excitation which would alone cause activity to explode to infinity, but is checked by high-gain feedback inhibition. We proved analytically that this model implements transformation of the form $f \mapsto A - \sqrt{A-f}$, robustly for different values of the maximum input A (Appendix 1c) and confirmed in simulation that it transforms Gaussian to peaky tuning curves (Fig. 6e-f). Furthermore, the model produces temporal dynamics similar to those seen in vivo: neurons whose preferred orientation is close to the current input have sustained activity, while neurons whose preferred orientation are further away have transient activity (Fig. 6e), leading to a narrowing of tuning curves with time[30,31].

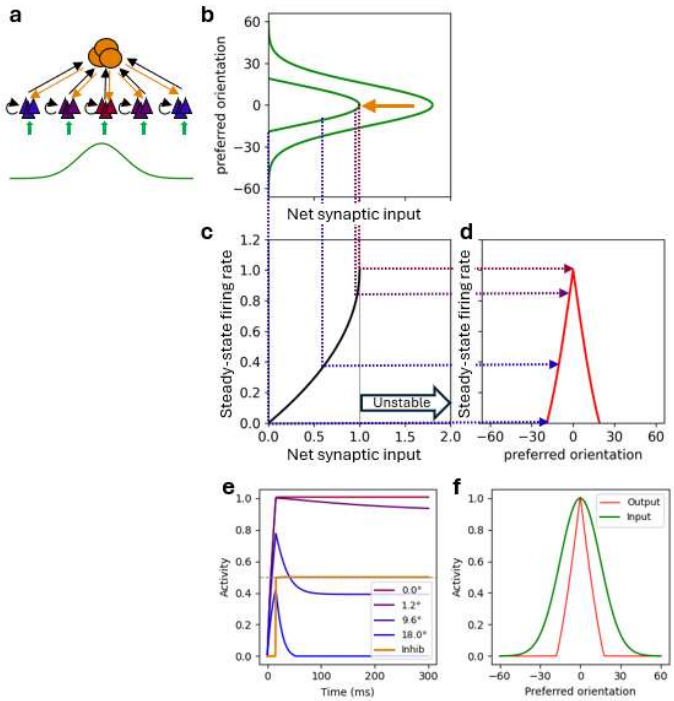


Figure 6 | Stabilized supralinear network transforms smooth inputs to peaky outputs. a, Hypothesized circuit mechanism. Excitatory neurons receiving similarly-tuned inputs (magenta to blue) are supralinearly recurrently connected, with activity stabilized by global feedback inhibition (orange). **b,** Synaptic input (x-axis) as a function of excitatory neurons' preferred orientation (y-axis) for a stimulus of 0°; note that in this plot, x depends on y. Rightmost green curve: external excitation; leftmost green curve: net synaptic input after feedback inhibition (orange arrow). **c,** In the exactly solvable model, supralinear recurrent excitation transforms a net synaptic input x into a steady-state firing rate of $1-\sqrt{1-x}$ (black curve). For net input >1 recurrent excitation is unstable, but feedback inhibition causes net input to the most excited cells to settle just below the threshold of instability. **d,** Predicted steady state firing rate in the exactly solvable model as a function of neuronal preferred orientation for a 0° stimulus. Dashed lines connecting b,c, and d indicate transformation of Gaussian-tuned inputs to peaky outputs, with colors reflected preferrerd orientation as in (a) and (e). **e,** Simulation results. Magenta-to-blue lines: firing rate of excitatory populations receiving inputs of preferred orientation indicated by color shade. Orange line: inhibitory firing rate. **f,** Simulated tuning curves for external input (green) and steady-state firing rate (red).

In summary, we used an AI-science system to discover an equation for direction tuning in mouse visual cortex. The AI system discovered an equation for "peaky" tuning curves, whose non-infinite-differentiability produced a high-dimensional population code, in contrast to the low-dimensional code produced by standard Gaussian tuning. Peaky tuning curves also underpin high-dimensionality in the mouse head-direction system, and future research can investigate whether they are found in a wider range of brain circuits. We illustrated how the high-dimensional codes produced by peaky tuning could allow flexible readout in a hyperacuity task and constructed an exactly solvable circuit model for producing them based on stabilized supralinear excitation. These results suggest that the differentiability of neuronal tuning curves has an important role in controlling

the dimensionality of the population code, and thus for understanding how cortical circuits process information.

These results also illustrate one way that AI reasoning can now contribute to scientific research. Fitting simple equations to experimental data can rapidly advance scientific understanding, and the equation found by AI in this study provided an analytical understanding we would not have gained from modeling tuning curves with artificial neural networks, however accurate these models had been. The equation the AI system found is simple, and, in principle, the humans on our team could have come up with it unaided. Nevertheless, we only found it after constructing the AI system. Finding equations summarizing experimental data is a task well-suited to AI science, because the performance of candidate equations can be numerically quantified. We suggest that AI agents will provide a powerful addition to human research teams in cases like this, in neuroscience and beyond.

References

1. Cover, T. M. Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition. *IEEE Transactions on Electronic Computers* **EC-14**, 326–334 (1965).
2. Marr, D. A theory of cerebellar cortex. *J.Physiol* **202**, 437–470 (1969).
3. Marr, D. A theory for cerebral neocortex. *Proc.R.Soc.Lond B Biol.Sci.* **176**, 161–234 (1970).
4. Barlow, H. B. & Rosenblith, W. A. Possible principles underlying the transformation of sensory messages. in *Sensory communication* (MIT Press, Cambridge MA, 1961).
5. Attneave, F. Some informational aspects of visual perception. *Psychol.Rev.* **61**, 183–193 (1954).
6. DiCarlo, J. J., Zoccolan, D. & Rust, N. C. How does the brain solve visual object recognition? *Neuron* **73**, 415–34 (2012).
7. Fusi, S., Miller, E. K. & Rigotti, M. Why neurons mix: high dimensionality for higher cognition. *Current Opinion in Neurobiology* **37**, 66–74 (2016).
8. Simoncelli, E. P. & Olshausen, B. A. Natural image statistics and neural representation. *Annu Rev Neurosci* **24**, 1193–1216 (2001).
9. Bordelon, B. & Pehlevan, C. Population codes enable learning from few examples by shaping inductive bias. 2021.03.30.437743 Preprint at https://doi.org/10.1101/2021.03.30.437743 (2022).
10. Stringer, C., Pachitariu, M., Steinmetz, N., Carandini, M. & Harris, K. D. High-dimensional geometry of population responses in visual cortex. *Nature* **571**, 361–365 (2019).
11. Swindale, N. V. Orientation tuning curves: empirical description and estimation of parameters. *Biol Cybern* **78**, 45–56 (1998).
12. Yamins, D. L. K. & DiCarlo, J. J. Using goal-driven deep learning models to understand sensory cortex. *Nat Neurosci* **19**, 356–365 (2016).
13. Yamins, D. L. K. *et al.* Performance-optimized hierarchical models predict neural responses in higher visual cortex. *Proceedings of the National Academy of Sciences* **111**, 8619–8624 (2014).
14. Cadena, S. A. *et al.* Deep convolutional models improve predictions of macaque V1 responses to natural images. *PLOS Computational Biology* **15**, e1006897 (2019).
15. Kindel, W. F., Christensen, E. D. & Zylberberg, J. Using deep learning to probe the neural code for images in primary visual cortex. *Journal of Vision* **19**, 29 (2019).
16. Turishcheva, P. *et al.* Retrospective for the Dynamic Sensorium Competition for predicting large-scale mouse primary visual cortex activity from videos. Preprint at https://doi.org/10.48550/arXiv.2407.09100 (2024).
17. Wang, E. Y. *et al.* Foundation model of neural activity predicts response to new stimulus types. *Nature* **640**, 470–477 (2025).
18. Lindsey, J. W. & Issa, E. B. Factorized visual representations in the primate visual system and deep neural networks. *eLife* **13**, RP91685 (2024).
19. Li, B. M., Cornacchia, I. M., Rochefort, N. L. & Onken, A. V1T: large-scale mouse V1 response prediction using a Vision Transformer. Preprint at https://doi.org/10.48550/arXiv.2302.03023 (2023).
20. Ecker, A. S. *et al.* A rotation-equivariant convolutional neural network model of primary visual cortex. Preprint at https://doi.org/10.48550/arXiv.1809.10504 (2018).
21. Novikov, A. *et al.* AlphaEvolve: A coding agent for scientific and algorithmic discovery. Preprint at https://doi.org/10.48550/arXiv.2506.13131 (2025).
22. Romera-Paredes, B. *et al.* Mathematical discoveries from program search with large language models. *Nature* **625**, 468–475 (2024).
23. Castro, P. S. *et al.* Discovering Symbolic Cognitive Models from Human and Animal Behavior. 2025.02.05.636732 Preprint at https://doi.org/10.1101/2025.02.05.636732 (2025).
24. Shojaee, P., Meidani, K., Gupta, S., Farimani, A. B. & Reddy, C. K. LLM-SR: Scientific Equation Discovery via Programming with Large Language Models. Preprint at https://doi.org/10.48550/arXiv.2404.18400 (2025).
25. Peyrache, A., Lacroix, M. M., Petersen, P. C. & Buzsáki, G. Internally organized mechanisms of the head direction sense. *Nat Neurosci* **18**, 569–575 (2015).
26. Andrews, D. P. Perception of contour orientation in the central fovea. I: short lines. *Vision Res* **7**, 975–997 (1967).
27. Westheimer, G. & McKee, S. P. Spatial configurations for visual hyperacuity. *Vision Res* **17**, 941–947 (1977).
28. Bradley, A., Skottun, B. C., Ohzawa, I., Sclar, G. & Freeman, R. D. Visual orientation and spatial frequency discrimination: a comparison of single neurons and behavior. *J Neurophysiol* **57**, 755–772 (1987).

29. Rubin, D. B., Van Hooser, S. D. & Miller, K. D. The Stabilized Supralinear Network: A Unifying Circuit Motif Underlying Multi-Input Integration in Sensory Cortex. *Neuron* **85**, 402–417 (2015).
30. Ringach, D. L., Hawken, M. J. & Shapley, R. Dynamics of orientation tuning in macaque primary visual cortex. *Nature* **387**, 281–284 (1997).
31. Gillespie, D. C., Lampl, I., Anderson, J. S. & Ferster, D. Dynamics of the orientation-tuned membrane potential response in cat primary visual cortex. *Nat Neurosci* **4**, 1014–1019 (2001).
32. Stringer, C., Michaelos, M., Tsyboulski, D., Lindo, S. E. & Pachitariu, M. High-precision coding in visual cortex. *Cell* **184**, 2767-2778.e15 (2021).
33. Stringer, C. *et al.* Spontaneous behaviors drive multidimensional, brainwide activity. *Science* **364**, 255 (2019).
34. Goldey, G. J. *et al.* Removable cranial windows for long-term imaging in awake mice. *Nat Protoc* **9**, 2515–2538 (2014).
35. Demas, J. *et al.* High-speed, cortex-wide volumetric recording of neuroactivity at cellular resolution using light beads microscopy. *Nat Methods* **18**, 1103–1111 (2021).
36. Schmutz, V. *et al.* High-dimensional neuronal activity from low-dimensional latent dynamics: a solvable model. 2025.06.03.657632 Preprint at https://doi.org/10.1101/2025.06.03.657632 (2025).

Acknowledgements

This work was funded by UKRI (Frontier Award EP/X022366/1 to MC), BBSRC (grant BB/W019884/1 to MC), the Wellcome Trust (Investigator Award 223144/Z/21/Z to MC and KDH), and the ERC (101097874 to KDH). AH is supported by a studentship from the Gatsby Charitable Foundation (GAT3755) and the Wellcome Trust (219627/Z/19/Z). VS is supported by a Royal Society Newton International Fellowship (NIF\R1\231927) and a fellowship from the Swiss National Science Foundation (grant no. 222150).

Methods

Experimental Methods and datasets analyzed

All experimental procedures were conducted according to the UK Animals Scientific Procedures Act (1986). Experiments were performed at University College London under personal and project licenses released by the Home Office following appropriate ethics review.

The data analyzed here came from four types of experiment. The first three are subtly different types of 2-photon calcium imaging of drifting grating data. The AI system discovered the tuning equation using only dataset 1, and datasets 2 and 3 were used to validate its generality. The fourth is head-direction cell recordings; AI equation discovery was not rerun on this data, instead the equation derived from dataset 1 was tested, after modification to contain only 1 peak.

Dataset 1

Dataset 1 was previously collected and shared by Stringer et al. [32], wherein the methods are fully described. This data contains activity of visual cortical excitatory neurons, recorded using a custom 2-photon mesoscope. Gratings of spatial frequency 0.05 cycles/degree were presented for 750 ms, at random floating-point directions between 0° and 360° on each trial.

For analysis, the floating-point direction angles were binned into 256 evenly-spaced direction bins, so that each bin contained at least 6 different trials. If more than 6 trials were present for a direction bin, stimuli were randomly discarded to make the remaining number of stimuli divisible by 6, and then averaged in random combination to make exactly 6 repeats per bin.

Dataset 2

Dataset 2 was collected in-house. We used a ThorLabs Bergamo 2 microscope to record activity in 2 mice, presenting full-field drifting sinusoidal gratings of 1024 directions evenly-sampling the 360° possible. The stimuli were shown in 3 blocks of exactly 1024 evenly-spaced orientations, in a different random order each time.

Transgenic adult mice expressing TdTomato in inhibitory neurons (Gad-Cre {Jax #010802} x Ai9 {Jax # 007909}) underwent surgery in a stereotaxic frame under isoflurane anaesthesia (5% for induction, 1 to 1.5%

maintenance), using previously-described methods[10,33]. Briefly, a 3 mm craniotomy was made in the skull over primary visual cortex using a biopsy punch. To express GCaMP6s, AAV1-Syn-GCaMP6s (Addgene #100843) was injected intracerebrally through the open craniotomy using a Nanoject II and sharpened glass pipette while the brain surface was covered with hydrated surgical gelatin sponge. A set of glass coverslips was implanted over the craniotomy, attached with veterinary cyanoacrylate glue and secured to the skull with Metabond dental cement alongside a metal headbar[34]. Animals were allowed to recover for at least one week before habituation to head fixation and imaging.

On the imaging day, animals were head fixed under a ThorLabs Bergamo 2 microscope and allowed to run freely on a cylindrical treadmill. A panel of 1024 evenly-spaced, sinusoidal drifting gratings spanning 360° was presented on a 3-monitor array covering a majority of the visual field. The stimuli were shown in 3 blocks of 1024 stimuli in a different random order each time. Gratings had a spatial frequency of 0.05 cycles per visual degree, a temporal frequency of 2 cycle per second, and 100% contrast. Stimuli were presented for 1 second with an intertrial delay of 1 second. Images were acquired in 12 planes spaced 30 microns apart vertically and sampled at a volume rate of 2Hz. Data were processed using Suite2p and all ROIs marked as cells were taken for analysis. Responses were taken as the average deconvolved calcium signal during the duration of visual stimulation.

For analysis, stimuli were binned into 256 evenly-spaced direction bins, resulting in 4 neighboring stimulus directions per bin per repeat. The responses of the two odd and even directions in each bin were averaged, resulting in 6 repeats per bin.

Dataset 3

Dataset 3 we used a custom light-beads microscope[35] to record responses to 2 blocks of repeats full-field drifting grating stimuli, of random integer directions ranging from 0° to 359°, in one mouse. This data has been previously analyzed in Ref. [36], where the methods are described.

These data were binned similarly to Dataset 1, but with 90 bins. Again, this resulted in exactly 6 repeats per bin.

Dataset 4

The fourth dataset was the activity of head-direction cells in freely-moving exploring mice, collected and shared by Peyrache et al. [25], comprising the CRCNS th-1 dataset. The dataset comprises 42 recording sessions obtained from seven freely moving mice while they foraged for food in an open arena (53 × 46 cm). Of these, we focused on six mice (12, 17, 20, 24, 25, and 28), comprising 39 sessions. Neural activity was recorded simultaneously from the antero-dorsal thalamic nucleus (AD) and post-subiculum (PoS) using chronically implanted, movable multi-site silicon probes. As the probes were advanced between sessions, we assumed that each session contained distinct units. From the total of 1,520 cells in these recordings, we used the criteria of the original study (concentration ratio, Raleigh test, and minimum firing rate threshold) to select 388 head direction cells.

Spike times (sampled at 20 kHz) were aligned to behavioural frames (39.0625 Hz) to determine head direction at each spike. The floating-point head direction at each timestamp was binned into 180 uniform bins spanning 0–360°. We created 6 repeats for each angle bin by dividing the timestamps into blocks of length ~5 mins (12,000 frames), separated by 10 s intervals to avoid dependencies between consecutive blocks. Each block was randomly assigned to one of the six repeats, randomly discarding blocks if necessary to ensure each repeat contained the same number of blocks. A firing rate was then computed for each angle bin and repeat, as the number of spikes in all assigned frames divided by total length of these frames. The firing rate histograms from all cells of all recordings were put together to form a "virtual population" containing enough cells to form estimates of dimensionality.

Regularized Cross-Validated PCA

We estimated the dimensionality of the visual cortical code to grating stimuli using an extension of the cross-validated Principal Components Analysis (cvPCA) method of Stringer et al. [10], which we term regularized cross-validated PCA (rcvPCA).

Ordinary PCA cannot be used for this purpose, as the variances it finds contain contributions from both stimulus-related ("signal") variance, and stimulus-independent ("noise") variance, but we are interested only in the former. The cvPCA method estimates the signal-related variance along a dimension $\mathbf{u}$ as a covariance between repeats $\hat{V}(\mathbf{u}) = \frac{1}{N_c}\sum_{c=1}^{N_c}(\mathbf{R_{1,c}} \cdot \mathbf{u})(\mathbf{R_{2,c}} \cdot \mathbf{u})$where $\mathbf{R_{i,c}}$ is the N_θ-dimensional vector of responses to repeat i of cell c across all stimuli. Because the stimulus-independent component of activity is independent across repeats, $\hat{V}(\mathbf{u})$ is an unbiased estimator of the stimulus-related variance along the direction $\mathbf{u}$ in stimulus space. Nevertheless, if the eigenvectors $\mathbf{u}_n$ are mis-estimated, even an unbiased estimate of the variance along these dimensions does not provide an accurate estimate of the stimulus-related principal component variances. In the original cvPCA method, stimuli were repeated twice, and the eigenvectors $\mathbf{u}_n$ were estimated by PCA of repeat 1 activity. The eigenvector estimates therefore contain contributions from both stimulus-dependent and stimulus-independent activity, and are not independent of the data used to assess variances, so may mis-estimate these eigenvectors.

The rcvPCA method improves over cvPCA by using three repeats of the stimuli. The first repeat is used to estimate the eigenvectors $\mathbf{u}_n$, using smoothing to obtain accurate estimates of stimulus-related covariance directions despite stimulus-independent variability. The second two repeats are used to estimate the stimulus-related variance along these eigenvectors, using the same formula $\hat{V}(\mathbf{u}) = \frac{1}{N_c}\sum_{c=1}^{N_c}(\mathbf{R_{2,c}} \cdot \mathbf{u})(\mathbf{R_{3,c}} \cdot \mathbf{u})$ To avoid artifactually lowering the estimated dimensionality, smoothing is not used for repeats 2 and 3.

Although rcvPCA requires a minimum of 3 repeats, activity was initially binned and averaged to create 6 repeats per angular bin, as described above for all datasets. This allowed multiple estimates of the eigenspectrum to be computed and averaged, thereby improving accuracy. We obtained our final eigenvalue estimates by averaging 60 runs of rcvPCA, one for each of the 6 ${}^5C_2$ ways of assigning one of the 6 repeats to be the first rcvPCA repeat and two of the remaining 5 to be the second and third (without regard for order).

Data Model

We model each response matrix as $\mathbf{R}_i = \Phi + \mathbf{E}_i$ for $i = 1,2,3$, where $\Phi \in \mathbb{R}^{N_c \times N_\theta}$ represents the stimulus-related signal component and $\mathbf{E}_i$ represents the noise component for repeat i. The noise matrices $\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3$ are independent and identically distributed with mean zero.

Estimating eigenvectors from repeat 1

We estimate the eigenvectors by first computing smoothed tuning curves on repeat 1 and performing an SVD on the matrix of stacked tuning curves.

Let $\theta_1, \dots, \theta_{N_\theta}$ be the stimulus angles, and define the circular distance

$$d(\theta_j, \theta_k) \;=\; \min(|\theta_j - \theta_k|,\, 2\pi - |\theta_j - \theta_k|).$$

For $w > 0$, we define the smoothing kernel $S_w \in \mathbb{R}^{N_\theta \times N_\theta}$ by a Gaussian:

$$(S_w)_{jk} = \frac{1}{Z_w}\exp\left(-\frac{d(\theta_j, \theta_k)^2}{2w^2}\right),$$

with Z_w chosen so that rows sum to one.

Define the smoothed training responses by $\widetilde{\mathbf{R}}_1 = \mathbf{R}_{\text{train}}\mathbf{S}_{w^*}$. We define the stimulus eigenvector estimates $\widehat{\boldsymbol{u}}_1, \dots, \widehat{\boldsymbol{u}}_{N_\theta}$ to be the L2-normalized eigenvectors of their covariance matrix $\frac{1}{N_c}\widetilde{\mathbf{R}}_1^\top\widetilde{\mathbf{R}}_1$.

Unbiased Cross-Validated Eigenvalue Estimates

We estimate the stimulus-related variance in the nth dimension as the covariance of population activity on repeats 2 and 3, projected onto the eigenvectors estimated from repeat 1:

$$\hat{\lambda}_n = \frac{1}{N_c}\widehat{\mathbf{u}}_n^\top\mathbf{R}_2^\top\mathbf{R}_3\,\widehat{\mathbf{u}}_n.$$

Since $R_2 = \Phi + E_2$ and $R_3 = \Phi + E_3$ with independent noise,

$$\mathbb{E}[\hat{\lambda}_n \mid \widehat{\mathbf{u}}_n] = \widehat{\mathbf{u}}_n^\top\Phi^\top\Phi\,\widehat{\mathbf{u}}_n.$$

Thus $\hat{\lambda}_n$ is an unbiased estimator of variance along $\widehat{\mathbf{u}}_n$, which will provide a good estimate of the true stimulus-related spectrum if the estimated eigenvector $\widehat{\mathbf{u}}_n$ is close to the true eigenvector. Note that the estimate $\hat{\lambda}_n$ may be negative, even though the true PC variances cannot be - however this is required for unbiased estimation. Indeed, if the true variance were zero, an estimator that could only give positive values would necessarily be biased upward.

Validation

To validate that the rcvPCA method could indeed accurately estimate the PCA spectrum of stimulus-related neural activity, under conditions of trial-to-trial variability similar to those found in our recordings, we created a ground-truth dataset with identical variability but for which the true spectrum was known (Extended data Fig. 1).

To do this we first estimated the noise of each cell, stimulus, and repeat, by subtracting from each cells response on each trial, its mean response over trials to the stimulus shown. We then added this to the deterministic stimulus response predicted by the AI model to create a simulation where the actual stimulus-related activity was known. We found that rcvPCA found a spectrum that was very close to the spectrum computed by simple PCA on the noise-free stimulus responses (Extended Data Fig. 1).

Evaluating tuning curve equations by a single-cell level error

The AI equation discovery algorithm identified the best-performing orientation tuning function using only the Stringer et al. [32] data. To ensure that the model had validity for other datasets, we evaluated its predictive accuracy also on two other datasets collected in house with a 2-photon microscopy (conventional and light-beads). In all cases, single-cell error was computed using cross-validated explained variance (r^2) relative to a Gaussian baseline.

Data were randomly partitioned along the stimulus axis into training, validation, and test sets. For each cell, the parameters of the AI and double-Gaussian models were fit on the training set by gradient descent, stopping at the iteration when validation mean squared error no longer improved. Gradient descent was then re-run for the same number of iterations using the combined training and validation data. Performance was evaluated on the held-out test set through calculating the explained variance of each cell, $r^2 = 1 - \sum_{\theta=1}^{N_\theta}\left(R_\theta - \widehat{R_\theta}\right)^2 / \sum_{\theta=1}^{N_\theta}(R_\theta - \bar{R})^2$, where $\widehat{R_\theta}$ = model prediction at angle i and $\bar{R}$ = response mean across angles.

This procedure was applied identically to both the Gaussian and AI-derived models to ensure comparability. The resulting test r^2 values provided a cross-validated measure of model fit.

LLM-guided equation discovery

The equation discovery engine aims to find a simple formula that can accurately model each neuron's tuning to sensory stimuli, using as few parameters as possible. It does this for a given experimental recording of

neuronal responses to many stimulus presentations, represented as a matrix $\mathbf{R} \in \mathbb{R}^{N_{\text{cells}} \times N_{\text{stim}}}$, and a stimulus vector $\boldsymbol{\theta} \in \mathbb{R}^{N_{stim}}$ where R_{cs} is the response of cell c on trial t on which stimulus θ_t was presented.

The goal is to learn a generative neuron model $f(\theta; \boldsymbol{\beta})$ that predicts any cell's mean response to any possible stimulus θ, given the cell's tuning parameters $\boldsymbol{\beta} \in \mathbb{R}^{N_{\text{params}}}$ (e.g. preferred angle, baseline rate, amplitude, …). A candidate program ϕ produced by our system consists of 2 python functions:

1. The neuron model $f(\theta, \boldsymbol{\beta})$
2. A parameter estimator $\widehat{\boldsymbol{\beta}}(\boldsymbol{\theta}, \mathbf{R}_c)$ of the parameter vector $\boldsymbol{\beta}_c \in \mathbb{R}^{N_{params}}$ for each cell, given its responses $\mathbf{R}_c \in \mathbb{R}^{N_{\text{stim}}}$ to stimuli $\boldsymbol{\theta} \in \mathbb{R}^{N_{\text{stim}}}$. These estimated parameters are used as a starting point for gradient-based optimization.

The LLM–derived parameter estimator generates sensible initial values for the new tuning-curve model, thereby reducing the risk that gradient descent parameter estimation will converge to a local minimum, and thus allowing functions f whose loss landscapes are non-convex (Extended Data Fig. 4). The resulting initial parameters substantially improved the performance and stability of subsequent gradient-based optimisation (Fig. 2f-g).

Core Operation: Stochastic Recombination Operator

The central operation in our evolutionary algorithm is a stochastic recombination operator $\mathcal{R}$ that generates new programs from parent pairs. Given two parent programs ϕ_1, ϕ_2 sampled from the population and ranked by score such that $S(\phi_1) \leq S(\phi_2)$, the operator produces a new program:

$$\phi_{\text{new}} \sim \mathcal{R}(\phi_1, \phi_2 \mid \text{LLM}, \text{prompt}, T)$$

Where:

- LLM is a large language model (e.g., **Gemini 2.0 Flash**) that generates code
- prompt is template that a specifies how to create a prompt from parent programs, together with their scores and graphical diagnostics
- T is a temperature parameter which anneals over time, and controls the LLM's stochasticity as well as the prompt text

The prompt is made up of the following components:

1. Temperature-dependent natural language instructions (if temperature is high, natural language instructions incentivize creativity, otherwise refinement is prioritized)
2. Source code of the two parent programs sorted by fitness (worst to best)
3. Diagnostic images showing parent model fits to data

The prompts follow the following format:

```
You are an AI scientist. Below are two programs sorted from worst to best. Your task is to generate a new mode
l that is better than both of these.

[Temperature-dependent creativity instructions]

[Image analysis instructions]

Program 1 Score = S(ϕ₁)
def neuron_model_v1(...): ...
def param_estimator_v1(...): ...

Program 2 Score = S(ϕ₂)
def neuron_model_v2(...): ...
def param_estimator_v2(...): ...
```

Further details of the parts in square brackets are provided below. In practice, neuron models f and parameter estimators $\hat{\beta}$ are generated in separate LLM calls, which we found empirically produces better output than asking an LLM to produce both functions simultaneously.

Examples of prompts and LLM-generated code arising in response to them are shown in Appendix 2.

Description of the winning model and parameter estimator

The full python code of the winning model and its parameter estimator can be found in Appendix 2. The winning model predicted each cell c's response to stimulus θ as the sum of up to two asymmetric generalised Gaussian peaks:

$$f_c(\theta) = b_c + A_{c1}\, e^{-\frac{1}{2}\left(\frac{|\theta-\varphi_{c1}|}{\sigma_{c1\pm}}\right)^{p_{c1}}} + A_{c2}\, e^{-\frac{1}{2}\left(\frac{|\theta-\varphi_{c2}|}{\sigma_{c2\pm}}\right)^{p_{c2}}}$$

Here b_c is cell c's baseline rate, A_{ci} its amplitude for peak i, p_{ci} the peakiness (shape) parameter, ϕ_{ci} the preferred angle, and $\sigma_{ci}\pm$ the asymmetric widths. The symbols $\sigma_{ci}\pm$ stand for two parameters σ_{ci} and σ_{ci} giving the widths to the left and right of the peak. In this notation,

$$\frac{|\theta-\varphi_{ci}|}{\sigma_{ci\pm}} = \begin{cases} \dfrac{(\theta-\phi_{ci})_{-\pi\ldots\pi}}{\sigma_{c1+}} & (\theta-\phi_{ci})_{-\pi\ldots\pi} \geq 0 \\[2ex] \dfrac{(\phi_{ci}-\theta)_{-\pi\ldots\pi}}{\sigma_{ci}} & (\theta-\phi_{ci})_{-\pi\ldots\pi} < 0 \end{cases}$$

Where $(\theta)_{-\pi\ldots\pi}$ means expressing the angle θ in the range $(-\pi,\pi]$ by adding or subtracting multiples of 2π as necessary. The AI model thus has a total of 11 parameters per cell: one peakiness, amplitude, center, and two widths for each peak, plus one baseline rate.

The AI-generated parameter estimator for this model first computed a smoothed firing rate histogram. To do so it divided the stimuli into $n = 180$ uniform bins, computing summed activity and stimulus count histograms, convolved both with a Gaussian kernel (σ = 2.5) to smooth the responses, and divided them to obtain a nonparametric tuning curve fit.

The baseline b_c was estimated as the minimum of the smoothed curve. Local maxima were identified by comparing each bin to its immediate neighbours. Peaks were sorted by amplitude, with the largest designated as the primary peak. A secondary peak was included only if its amplitude exceeded 10% of the primary and its angular separation was at least 90°. Cells without a second qualifying peak were treated as single-peaked (i.e. the second peak amplitude was set to exactly 0).

To estimate the asymmetric widths $\sigma_i\pm$, the parameter estimator identified the half-height points between baseline and peak:

$$f_i(\theta_h) = \hat{b} + \frac{1}{2}\hat{A}.$$

The corresponding angular distances to the peak, $\Delta\theta_h$, define the half-widths. Assuming a Gaussian form, the relationship between half-width and standard deviation is

$$e^{-(\Delta\theta_h/\sigma)^2} = \frac{1}{2},$$

which yields

$$\sigma = \frac{\Delta\theta_h}{\sqrt{2\ln 2}}$$

This procedure was applied independently to both sides of each peak to obtain asymmetric widths. Note that the LLM-generated code applied this formula even when $p \neq 2$, but good results were still obtained.

To estimate the peakiness parameter p, the parameter estimator used the ratio between the half-height ($\Delta\theta_h$) and quarter-height ($\Delta\theta_q$) angular distances. It estimated these using the formula

$$p = \frac{\ln 2}{\ln(\Delta\theta_q/\Delta\theta_h)}$$

Which can be obtained from the log of the generalized Gaussian $-(\Delta\theta/\sigma)^p = \ln k$, taking the ratio for $k = 1/2$ and $k = 1/4$. This analytical relationship provides an efficient, interpretable estimate of p directly from the smoothed tuning curves.

Data Partitioning Strategy

Only cells with reliable responses to the grating stimuli were used to fit and evaluate tuning curves. For each cell c, we define the concentration and activity as follows:

$$\text{concentration}(c) = \left|\frac{\sum_s R_{cs}\, e^{2i\theta_s}}{\sum_s R_{cs}}\right|,$$

$$\text{activity}(c) = \frac{\text{number of nonzero responses}}{\text{total number of responses}},$$

and only used cells with concentration ≥ 0.4 and activity ≥ 0.4.

To fit and evaluate the models, we employ a nested cross-validation scheme with two orthogonal splits:

Cell-Level Split (Outer Loop)

cells with reliable responses were divided into two groups:

- **Training cells** $\mathcal{C}_{\text{train}}$: 50% of cells (interleaved: $c_1, c_3, c_5, \ldots$) for program discovery.
- **Evaluation cells** $\mathcal{C}_{\text{test}}$: 50% of cells (interleaved: $c_2, c_4, c_6, \ldots$) reserved for final model selection and generalization assessment.

Trial-Level Split (Inner Loop)

- **Training trials** Θ_{train}: 50% of stimulus presentations (interleaved: $\theta_1, \theta_3, \theta_5, \ldots$) for parameter estimation and optimization
- **Testing trials** Θ_{test}: 50% of stimulus presentations (interleaved: $\theta_2, \theta_4, \theta_6, \ldots$) for scoring via cross-validated likelihood

Evaluation Protocol:

- **During evolutionary search:** All programs are discovered using $\mathcal{C}_{\text{train}}$ only. For each training cell, parameters are estimated and optimized on Θ_{train}, then evaluated on Θ_{test} to compute $S(\phi)$.
- **After search completion:** Top-scoring programs are evaluated on the held-out $\mathcal{C}_{\text{test}}$ using the same train-test split across trials. The program with best generalization to unseen cells is selected as the final model.

Program Scoring

To score a generated program $\phi = (f, \hat{\beta})$, we first fit tuning parameters for each cell. Let $\mathbf{R}_c^{(\text{train})}$ and $\mathbf{R}_c^{(\text{test})}$ be the response vectors of cell c for the training and test trials. The program's parameter estimator produces an initial parameter vector $\beta_0^{(c)} = \hat{\beta}(\Theta_{\text{train}}, \mathbf{R}_c^{(\text{train})})$, which is refined by gradient-descent to yield an estimate of:

$$\beta_c^* = \underset{\beta}{\text{argmin}}\ ||f(\Theta_{\text{train}}; \beta_c) - \mathbf{R}_c^{(\text{train})}||^2,$$

The program is then assigned a complexity-penalized, cross-validated score $S(\phi)$ by evaluating fit parameters across all cells on held-out test stimuli

$$S(\phi) \;=\; -\frac{1}{N_c}\sum_c ||f(\Theta_{\text{test}};\beta_c^*) - \mathbf{R}_c^{(\text{test})}||^2 \;-\; \lambda\, C(\phi).$$

We set $C(\phi)$ to be the number of free parameters: $C(\phi) = \dim(\beta_c)$, and we include a small value of $\lambda = 0.01$ to prevent an explosion in the number of parameters without meaningful performance gains. Since the maximum observed reduction in held-out MSE was 0.55, each additional free parameter decreases the score by approximately 2% of the attainable improvement.

Because programs are LLM-generated, we enforce:

- **Type/shape checks** on outputs of the new *neuron_model* and *parameter_estimator*;
  - The parameters introduced must be numeric, so they can optimized via gradient descent
  - The parameter estimator must return an *n_train_cells x n_params* array
  - The neuron model must return an *n_train_cells x n_train_stimuli* array
- **Timeouts** to discard programs that do not execute.

Programs that fail these checks receive a score of $-\infty$ and are excluded from parent selection.

Horizontal Structure: Islands

We use a parallel island model with $N_{\text{islands}} = 8$ semi-independent populations evolving in parallel[22]. Each island j maintains its own program pool $\mathcal{P}_j$. New candidate programs are generated by selecting two parent programs ϕ_1, ϕ_2 uniformly from $\mathcal{P}_j$ and prompting the LLM to produce a child $\phi_{\text{new}} = \mathcal{R}(\phi_1, \phi_2)$. Each generation produces $B = 6$ new programs per island, for a total of 48 LLM calls per generation – chosen to stay within the Gemini Tier 1 rate limits.

To balance exploration and exploitation, we implement migration after each generation. Each island j selects 2 of the newly generated programs to send to island $(j+1) \bmod N_{\text{islands}}$. Migrants are sampled according to a Boltzmann distribution with temperature T over program fitness:

$$P(\text{program } i \text{ migrates}) = \frac{\exp(\Delta S_i/T)}{\sum_k \exp(\Delta S_k/T)}$$

where ΔS_i is the normalized score difference from the island's best program. At high temperatures (early iterations), migration is more uniform, allowing many programs to spread; at low temperatures, only elite programs migrate, accelerating convergence. Received migrants are integrated into the destination island's population for recombination, and compete with local programs in the next pruning step.

Population Management and Deduplication

Each island j maintains a dynamic population $\mathcal{P}_j$ carrying out the following operations at the end of each generation:

1. **Deduplication**: Programs with high functional similarity (cosine similarity > 0.99 on evaluation matrices and score difference < 0.025) are removed to prevent genetic bottlenecks. Similarity is assessed both within islands and between island pairs—if two islands share $\geq 75\%$ of the critical population size in duplicates, the island containing the lower scoring solutions is reinitialized with the seed models.
2. **Pruning**: After each generation, populations are pruned by keeping only the 12 programs of highest score.

Temporal Structure: Evolution

Each island is initialized with **two seed programs**:

Single-peaked Gaussian:

$$f_1(\theta; \theta_{\text{pref}}, b, A, w) = b + Ae^{-\frac{d(\theta,\theta_{\text{pref}})^2}{2w^2}}$$

Double-peaked Gaussian:

$$f_2(\theta; \theta_{\text{pref}}, b, A_1, A_2, w) = b + A_1 e^{-\frac{d(\theta,\theta_{\text{pref}})^2}{2w^2}} + A_2 e^{-\frac{d(\theta,\theta_{\text{pref}}+\pi)^2}{2w^2}}$$

where

$$d(\theta_1, \theta_2) = \min(|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|)$$

is the circular distance function.

Both programs include simple parameter estimators $\hat{\beta}$ using circular statistics and peak detection on binned tuning curves. (See supplementary materials)

The temperature is a global parameter across islands, which decays according to an annealing schedule of $T(i) = e^{-i/N_{\text{iter}}}$, where i is the current generation. This temperature controls the evolution of programs via:

1. **LLM Temperature**: Controls the stochasticity of code generation
2. **Prompt Instructions**: Triggers a discrete phase transition from exploration mode to exploitation mode at a critical temperature T_c
3. **Migration Dynamics**: Modulates the probability distribution governing inter-island program exchange

Full copies of the prompts can be found in Appendix 2. Briefly, the *explore* mode instructs:

> "Use the models below as inspiration, but be creative and invent something new. Which features correlate with lower loss? Find these features and extrapolate them. You should also combine features from several models, and experiment with new ideas."

while the *exploit* mode emphasizes parsimony:

> "Use the models below as a template. Which features correlate with lower loss? Focus on exploiting the strengths of existing models and eliminating their weaknesses or redundancies. Are the parameter ranges correct? You will be penalized for complexity, so make the new model as simple as possible while still being better."

Programming language

To perform GPU-accelerated gradient descent we used JAX. However, we found during early development of the system that prompting the LLMs to generate code in JAX reduced creativity: LLM-generated JAX programs tended to use standard machine learning coding patterns whereas NumPy programs showed more scientifically creative solutions, perhaps due to the contents of the original LLM training data in these languages. We therefore prompted the LLMs to use NumPy syntax first, preserving creative LLM reasoning, then employed a second LLM to translate this code to JAX enabling to GPU-accelerated gradient-descent.

LLM Selection and Task Distribution

The system distributes computational load across three LLMs based on task complexity and creativity requirements:

- Lightweight Model (Default =*gemini-1.5-flash-8b*)
  - Tasked with translating code from NumPy to JAX, following the main LLM call.
- Mid-Range Model (Default = *gemini-2.0-flash*)

- This reasonably lightweight model is much cheaper and faster than a thinking model, and we used it for 2/3 of the neuron models generated.
- We also used it to generate all parameter estimators, which only require heuristics. The prompts provided explicit constraints (≤100 lines, no complex fitting libraries like *scipy.optimize*).

- Thinking Model (Default = *gemini-2.5-flash* or *gemini-2.5-pro*)
  - These were used to generate 1/3 of the neuron models, and were found to introduce more creative innovations. However, they were used sparingly as they are ~3 times slower, and ~20 times more expensive to run.

We reasoned that the mid-range model (*gemini-2.0-flash*) would produce modest but rapid refinements at volume, while periodic reasoning model injections would introduce occasional creative breakthroughs. This distribution of labor drastically reduces time and cost of programs generated. The run that generated the program presented in the main text took 45 minutes of machine time, and $8.25 of LLM tokens.

Multi-modal Prompt Engineering: Visual Feedback

In addition to the instructions and parent program code, the prompts contained a graphical input of diagnostic scatter plots, which showed each parent model's predictions overlaid on raw neural data, for 9 randomly selected cells (Extended Data Fig. 3). These images are embedded directly in the prompt alongside code, with instructions to:

- Identify systematic fit failures across multiple cells (e.g., *"model peaks are consistently too narrow in subplots 2, 5, 7"*)
- Analyze which model shapes (peak width, skewness, amplitude) best match data
- Propose parameter modifications addressing visual discrepancies
- Reference specific subplot examples in generated docstrings

For example, prompts instruct:

"Analyze the model's fit to the data in the image below. Look for places where the model (red curve for best, green for second-best) deviates most from the binned data mean (blue curve). This is where the models are weakest, and where you should focus improvements. Do the models' shapes align with the binned data mean and raw data scatter points? If not, how can you change the model to better match the data shape?"

This visual feedback loop improves the convergence rate (Fig. 2f) and also provides insight into the reasoning the LLM used when crafting the models, which can be found in the docstrings of the LLM-generated code (Appendix 2).

Pseudo-Code for AI Equation Discovery System

```
# INPUT
R ∈ ℝ^{N_cells × N_trials}  # neural responses
Θ ∈ ℝ^{N_trials}             # stimulus orientations
Seed Programs: ϕ₁=(f₁, β̂₁), ϕ₂=(f₂, β̂₂)
Hyperparams: N_islands, B, N_iter, λ, T(i), activity_thresh, conc_thresh

# PREPROCESS
Select cells with firing_prob > activity_thresh AND circular_conc > conc_thresh
Normalize each cell's response: ||R_c||₂ = 1
Split cells into TRAIN_CELLS and TEST_CELLS (50/50; disjoint)
Split trials into TRAIN_TRIALS and TEST_TRIALS (50/50; disjoint)

# FUNCTION FitAndScore(ϕ=(f, β̂); cells):
    For each cell c:
        β₀^(c) ← β̂(Θ_train, R_c) with timeout=5s
        If timeout: β₀^(c) ← defaults
        If invalid: return FAILED_COST

    Optimize β*_c by gradient descent on Θ_train (MSE loss)
    if convergence_failed: Return -infty
```

```
    Compute: Test_MSE = ∑_s (R_cs - f(θ_s; β*_c))²
    Score: S = Test_MSE - λ·dim(β)
Return S

# INITIALIZE
Add seeds to all islands after computing S(ϕ₁), S(ϕ₂) on TRAIN_CELLS

# EVOLUTION for i = 1..N_iter:
mode ← 'explore' if i < 0.5·N_iter else 'exploit'
For each island:
    Generate B candidates:
        Select parents ϕ₁, ϕ₂ randomly
        LLM generates f (with diagnostic image feedback)
        LLM generates β̂
        Translate to JAX; apply guardrails
        S ← FitAndScore((f, β̂); TRAIN_CELLS)
    Deduplicate (similarity > 0.99 AND |ΔS| < 0.025)
    Prune to critical population
    Migrate via Boltzmann distribution over S

# FINAL EVALUATION
For each surviving ϕ:
    S_test ← FitAndScore(ϕ; TEST_CELLS)
Rank by S_test and report top programs
```

Parameter ablation study

To assess how specific parameters of the AI model contributed to fit quality, we examined the effect on fit quality of successively ablating parameters. First, the full model ("p+asym+non-antipodal") was fit and evaluated as described above. We next considered a model ("p+asym") where the peak locations were constrained to be 180 degrees apart:

$$f(\theta) = b + A_1 e^{-|(\theta-\phi)/\sigma_{1\pm}|^{p_1}} + A_2 e^{-|(\theta-\phi-\pi)/\sigma_{2\pm}|^{p_2}}$$

We fit the parameters of this model for each cell by gradient descent as described above, starting from the full-model fits of all parameters except ϕ, which was started at the location of the full model's largest peak ϕ_1.

The next ablated model ("p") lost the asymmetric widths of the peaks, whilst allowing the two peaks to have different widths :

$$f(\theta) = b + A_1 e^{-|(\theta-\phi)/\sigma_1|^{p_1}} + A_2 e^{-|(\theta-\phi)/\sigma_2|^{p_2}}$$

We again fit the parameters of this model by gradient descent, starting from the same parameters as ("p+asym") except $\sigma_1 = (\sigma_{1+} + \sigma_{1-})/2$ and $\sigma_2 = (\sigma_{2+} + \sigma_{2-})/2$.

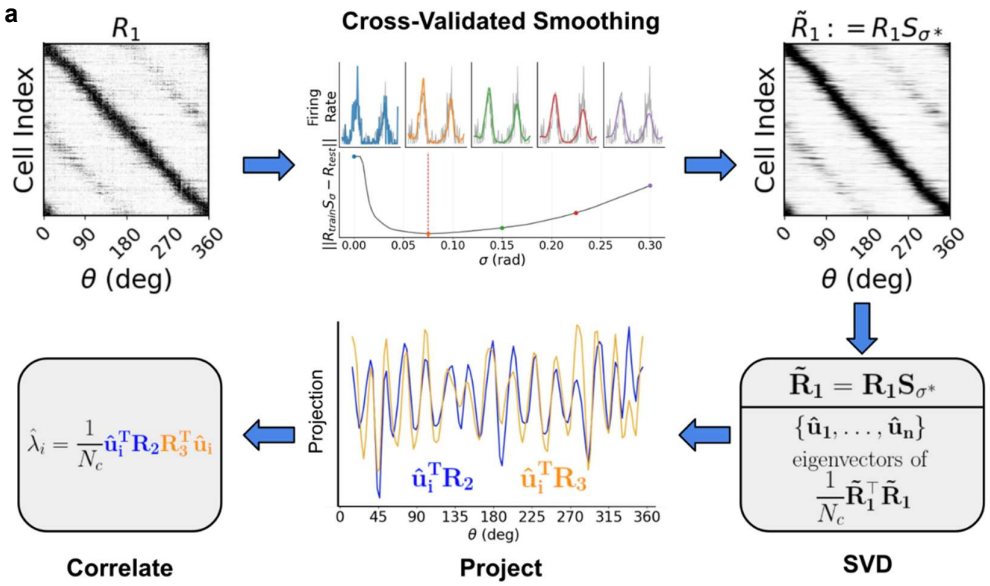
Creating new estimators for each ablated model did not improve performance and often led to local minima; thus, the original estimator was retained.

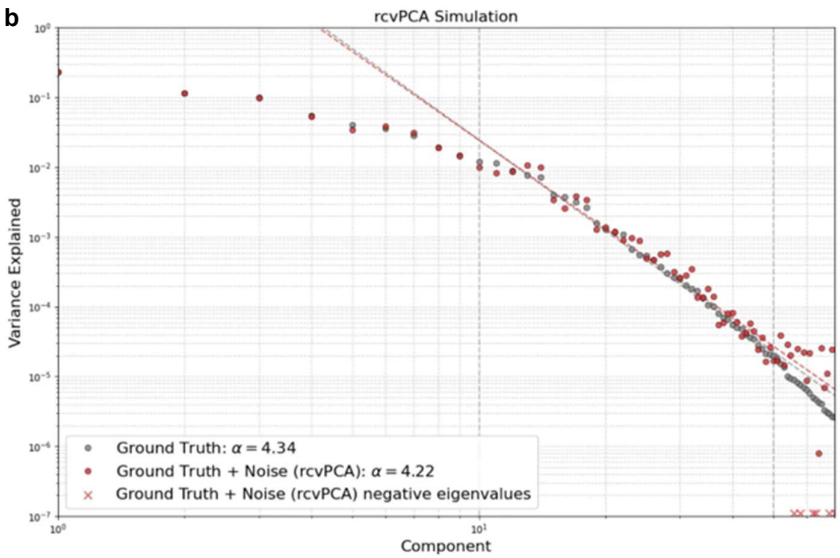
Analysis of head direction cell tuning curves

Because head-direction cells, unlike orientation tuning curves, are single peaks, we modified the AI model to have only a single tuning peak:

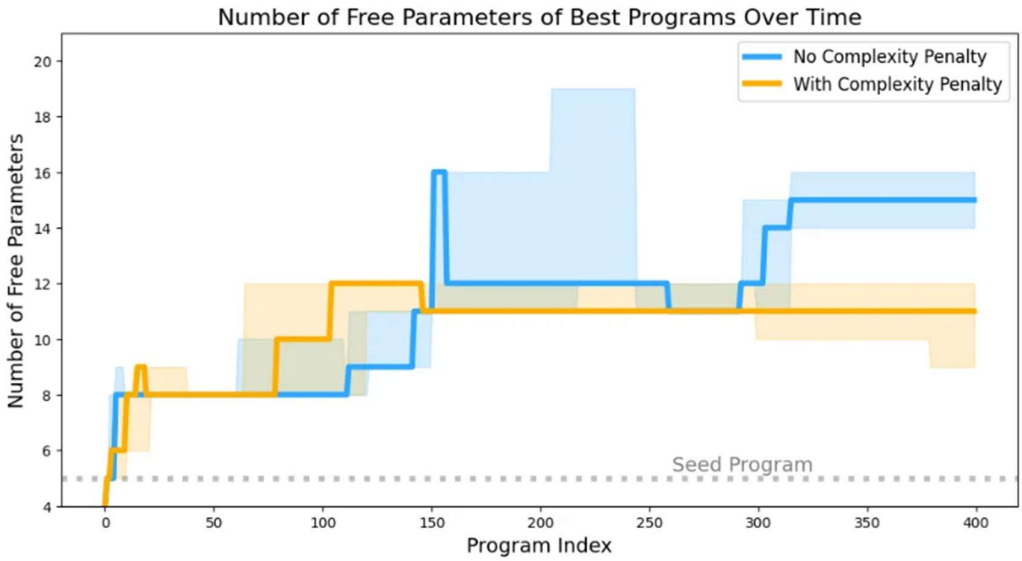
$$f(\theta) = b + Ae^{-|(\theta-\phi)/\sigma_\pm|^{p}}$$

Visually observing the fit model also revealed that cells with highly asymmetric widths suffered from discontinuity at the angle antipodal to the cell's preferred orientation. To correct for such discontinuity, we updated the tuning function by tiling the tuning functions either side of the original tuning function. Explicitly, for $g(\theta) = Ae^{-|(\theta-\phi)/\sigma_\pm|^{p}}$, we define $g_{\phi-2\pi}(\theta)$ and $g_{\phi+2\pi}$, and let $f(\theta) = g_{\phi-2\pi}(\theta) + g_{\phi}(\theta) + g_{\phi+2\pi}(\theta)$ for $\theta \in [0,2\pi]$. Full and ablated models were fit as described above.

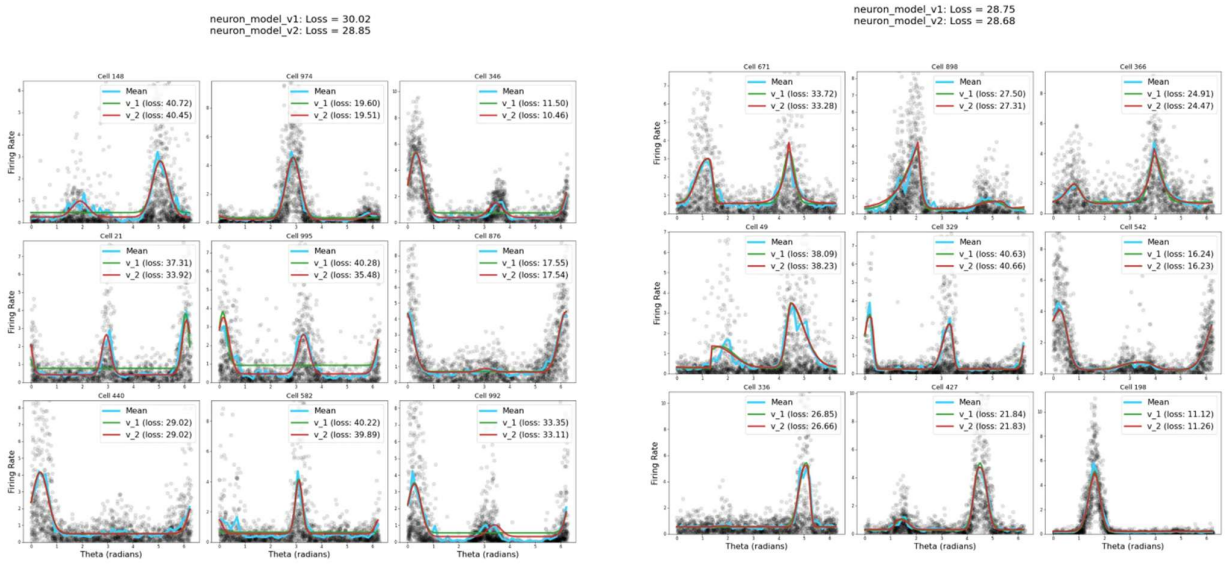




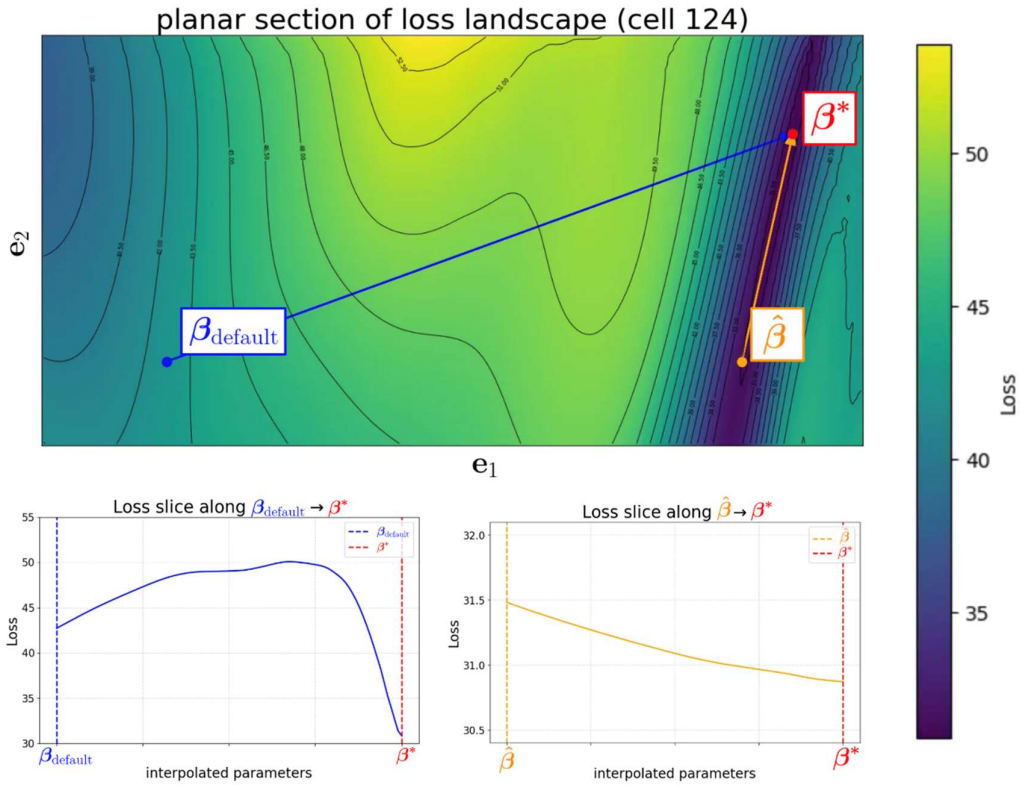
Extended Data Figure 1 | a, Illustration of the rcvPCA method**.** The first repeat (R_1) is used to nonparametrically estimate tuning curves for all cells by Gaussian smoothing. The smoothing width is determined by cross-validation to minimize prediction error, summed over cells, between the smoothed curves from R1 and responses on R_2 and R_3. Eigenvectors $\hat{\boldsymbol{u}}_i$ are estimated by applying singular value decomposition (SVD) to the matrix of smoothed tuning curves. The remaining two repeats (R_2 and R_3) are projected onto these eigenvectors, and the cross-covariances of the projections provide estimates of the stimulus-related variance spectrum. Note that smoothing is not applied to R_2 and R_3 as this would artificially lower estimated dimensionality. **b**, Testing rcvPCA with ground-truth data. We simulated data in which the signal-related spectrum was known, and trial-to-trial variability matched that of the experimental data, by adding experimentally measured noise to a known ground-truth tuning spectrum. Noise for each cell c, stimulus θ, and repeat i was computed as the residual for repeat i relative to the repeat-averaged response: $E_{c,\theta,i} = R_{c,\theta,i} - \bar{R}_{c,\theta}$, and added to a known tuning curve generated from the AI model's fit to all cells: $R'_{c,\theta,i} = \widehat{\Phi}_{c,\theta} + E_{c,\theta,i}$. Gray points: PCA spectrum of the ground truth data, computed by standard PCA on the AI model tuning curves $\widehat{\Phi}_{c,\theta}$. Gray dashed line: power-law fit to over eigenvalues 10-50, yielding an exponent of 4.34. Red points: spectrum estimated by rcvPCA from noisy data; red crosses are cases where the eigenvalue estimate was negative. Red dashed line: power-law fit to rcvPCA, recovering a slope of 4.22, close to the ground truth value.



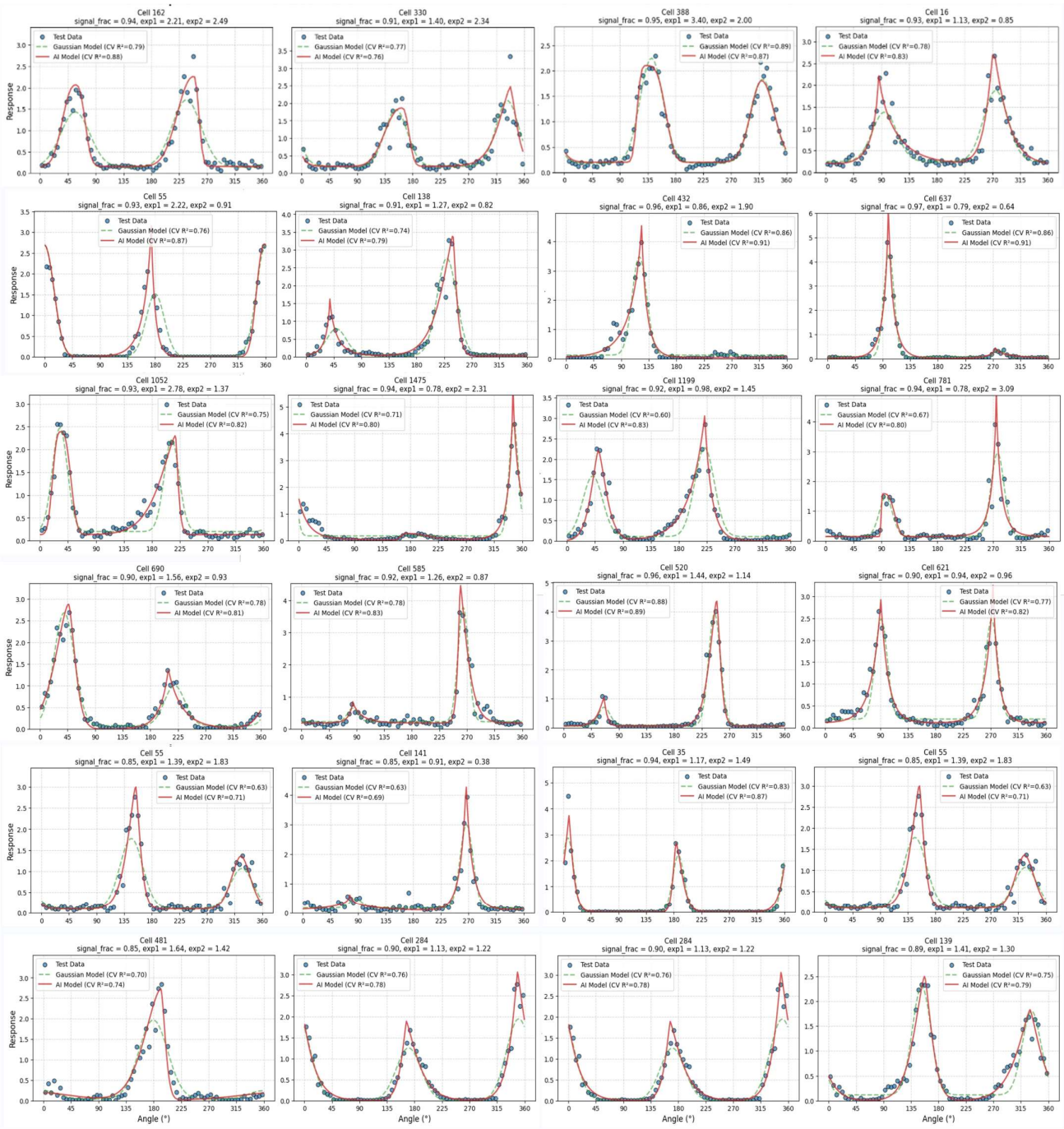
Extended Data Figure 2 | Effect of complexity penalty. The complexity penalty λ is proportional to the number of free parameters in the model. Setting $\lambda = 0$ causes the number of free parameters to grow over generations, yielding complex and opaque models. With the complexity penalty, model size remains stable, models remain interpretable and overfitting to the training cells is controlled.



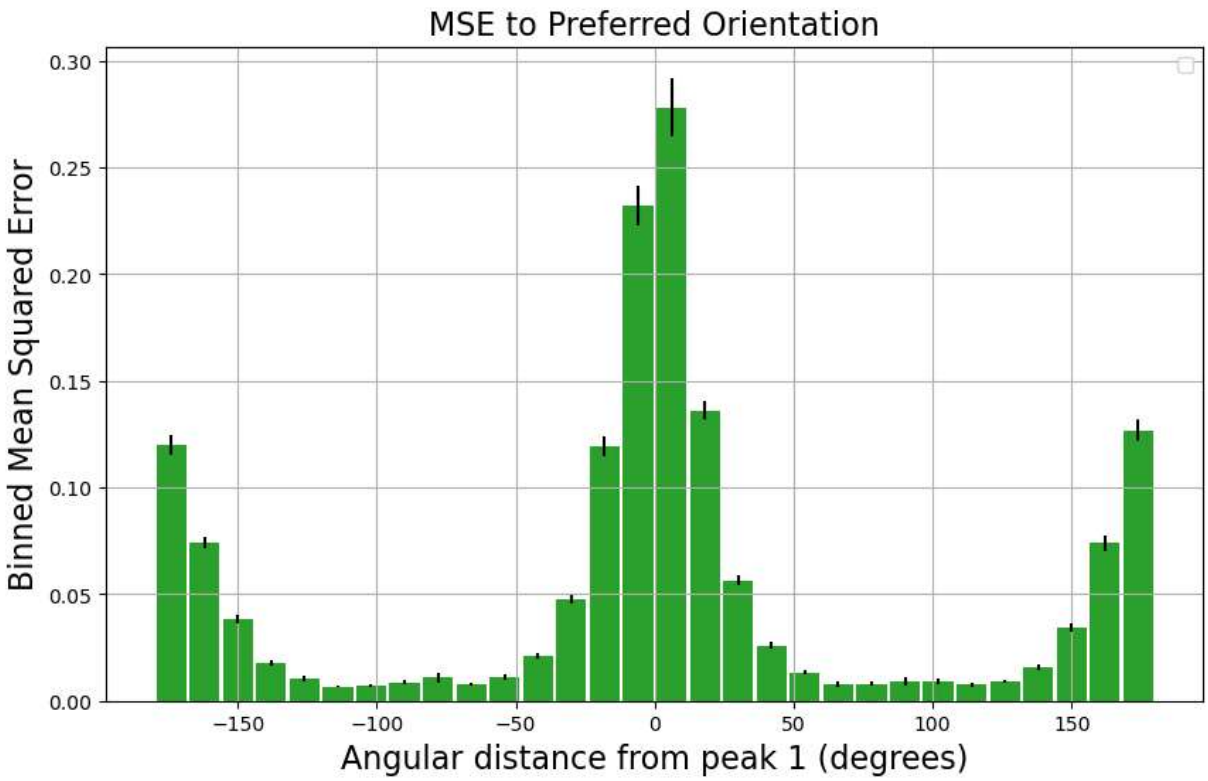
Extended Data Figure 3 | Graphical input to LLMs. In addition to textual prompts (described in Methods and quoted in full in Appendix), the LLMs are given graphical diagnostics of parent model fits to 9 randomly-selected example cells. This figure shows the graphical input given to the LLM on the first iteration (left), and on the iteration that generated the winning model (right), at actual resolution. Black points show single-trial responses. Cyan curve shows binned-mean responses for 90 evenly-spaced bins. Red and green curves show fits of the two parent models. Note that the two examples show diagnostics for different randomly-selected cells.



Extended Data Figure 4 | Avoiding local minima with the parameter estimator. Top: pseudocolor view of the winning AI model's loss landscape for one cell. A planar section through the N-dimensional parameter space of the model is obtained by defining the x-axis by projection onto vector $\boldsymbol{e_1}$ connecting the parameter estimator's output $\widehat{\boldsymbol{\beta}}$ and a default parameter values $\boldsymbol{\beta}_{default}$ (defined by the LLM as the default parameters in the tuning function), and defining the y-axis by projection onto a vector $\boldsymbol{e_2}$ orthogonally connecting the solution $\boldsymbol{\beta}^*$ to the line joining $\widehat{\boldsymbol{\beta}}$ and $\boldsymbol{\beta}_{default}$. Bottom: projection of the loss landscape onto lines connecting the two possible start points. The line connecting $\boldsymbol{\beta}_{default}$ to $\boldsymbol{\beta}^*$ traverses a local maximum, but the line connecting $\widehat{\boldsymbol{\beta}}$ to $\boldsymbol{\beta}^*$ does not.



Extended Data Figure 5 | Tuning curve model fits for example cells. Blue points: test-set responses as a function of grating direction for four example cells per mouse (one row per mouse; averaged over a random half the stimulus repeats, for any stimuli within the orientation bin). Red sold and green dashed curves show AI-derived and Gaussian model predictions, fits for each cell to training-set repeats (fit to the other half of repeats).



Extended Data Figure 6 | Gaussian model errors are at tuning curve peaks. Visual inspection of Gaussian model fits (Fig. 1B) suggested the Gaussian models failed to fit experimental data accurately specifically at the peaks of the tuning curves. To confirm this, we plotted the mean squared error of the Gaussian tuning curve fits as a function of angular distance from each cell's preferred orientation for an example mouse. Large errors are concentrated near the tuning curve peaks, demonstrating that the Gaussian model failed specifically at its peaks. Error bars: standard error across cells.

Appendix 1a: relationship of Fourier and PCA spectra

To understand the relationship between tuning curve shape and eigenspectrum, we consider a population of neurons with identically-shaped tuning curves $f_i(\theta)$, whose preferred orientations ϕ are evenly distributed around a circle. We let the size of the population tend to infinity, thus the activity of the population can be captured by a function of ϕ. For every θ, we represent this function of ϕ as a vector $\boldsymbol{h}$ in the Hilbert space $L^2([0,2\pi])$

$$\boldsymbol{h}(\theta) = \{f(\theta - \phi) | \phi \in [0,2\pi]\}$$

Whatever shape of the function f, the principal components of the population's response to evenly distributed orientation stimuli are sine and cosine functions of ϕ. To show this we expand f in a Fourier cosine/sine series:

$$f(\theta - \phi) = \sum_{n=0}^{\infty} c_n \cos\big(n(\theta - \phi)\big) + \sum_{n=1}^{\infty} s_n \sin\big(n(\theta - \phi)\big)$$

Note the cosine sum starts at $n = 0$ to include a constant term. Then the projection of $\boldsymbol{f}$ onto a cosine basis $\cos(m\phi)$ is

$$\begin{aligned} A_m(\theta) &= \int_{-\pi}^{\pi} f(\theta - \phi) \cos(m\phi)\, d\phi \\ &= \sum_{n=0}^{\infty} \int_{-\pi}^{\pi} c_n \cos\big(n(\theta - \phi)\big) \cos(m\phi)\ d\phi + \sum_{n=1}^{\infty} \int_{-\pi}^{\pi} c_n \sin\big(n(\theta - \phi)\big) \cos(m\phi)\ d\phi \end{aligned}$$

Using the trigonometric identities $\cos(a)\cos(b) = (\cos(a+b) + \cos(a-b))/2$ and $\cos(a)\sin(b) = (\cos(a+b) - \cos(a-b))/2$, we see that the only term that does not integrate to zero is

$$A_n(\theta) = \int_{-\pi}^{\pi} c_m \cos\big(m(\theta - \phi)\big) \cos(m\phi)\ d\phi = \begin{cases} \pi c_m \cos(m\theta), & m > 0 \\ 2\pi c_0, & m = 0 \end{cases}$$

Thus the expected product of two projections across an ensemble of random stimuli is

$$\mathbb{E}\big[A_m A_q\big] = \frac{1}{2\pi} \int_{-\pi}^{\pi} \pi c_m \cos(m\theta)\, \pi c_q \cos(q\theta)\, d\theta = \begin{cases} \pi^2 c_m^2/2 & m = q \neq 0 \\ 4\pi^2 c_0^2 & m = q = 0 \\ 0 & m \neq q \end{cases}$$

Similarly we obtain for the sine projections $B_m(\theta) = \int_{-\pi}^{\pi} f(\theta - \phi) \sin(m\phi)\, d\phi$

$$\mathbb{E}\big[B_m B_q\big] = \begin{cases} \pi^2 s_m^2/2 & m = q \neq 0 \\ 0 & m \neq q \end{cases}$$

And the expectation of sine times cosine is zero

$$\mathbb{E}\big[A_m B_q\big] = 0$$

Thus, applying PCA to this population will yield eigenvectors whose weights on neurons with preferred orientation ϕ is given by sine or cosine functions of ϕ, with an eigenspectrum proportional to the corresponding squared Fourier components of the tuning function $f(\theta - \phi)$.

Appendix 1b: relationship between peak shape and Fourier spectrum

To evaluate the eigenspectrum of our tuning functions, we must therefore calculate their Fourier series. We first consider a simplified version of the tuning function that retains only one symmetrical peak:

$$f(\theta) = e^{-|\theta/\sigma|^p}$$

The Fourier series for even this simplified tuning curve is not exactly computable, but we are able to estimate the asymptotic decay of its Fourier spectrum by exploiting the relationship between a function's smoothness and the

asymptotic decay of its Fourier spectrum [1,2]. We will do so following the approach of Erdelyi [3]. Around θ= 0, we can expand in a power series:

$$f(\theta) = \sum_{m=0}^{\infty} \frac{(-1)^m \ |\theta/\sigma|^{pm}}{m!} \tag{1}$$

We will show that the Fourier spectrum is dominated by the least-smooth term in this series, $|\theta/\sigma|^p$, which has Fourier decay $O(n^{-1-p})$.

We can quantify the smoothness of a function by its differentiability: the more times it can be differentiated, the smoother it is. There are several subtly different measures of differentiability, for example whether we require that the derivative be continuous. Here we will use the concept of *weak differentiability*. We say that a function $f(x)$ has weak derivative $f'(x)$ if the integration by parts formula holds, i.e. for any infinitely-differentiable $\eta(x)$ with $\eta(a) = \eta(b) = 0$:

$$\int_a^b f(x)\eta'(x)dx = -\int_a^b f'(x)\eta(x)dx$$

The weak derivative does not need to be continuous, or even defined at every point. For example if $p > 0$ the function $|x|^p$ has weak derivative $p|x|^{p-1}\mathrm{sgn}(x)$, which it is not defined at 0 when $p < 1$. Nevertheless, the weak derivative can be used in the integration-by-parts formula. In general, $|x|^p$ is weakly differentiable at least $\lfloor p \rfloor + 1$ times, where $\lfloor p \rfloor$ is the largest integer $\leq p$.

The Fourier series $\tilde{f}(n)$ of a function $f(\theta)$ which is m times weakly differentiable must decay as $O(n^{-m})$, i.e. there exists $C > 0$ such that $\left|\tilde{f}(n)\right| \leq Cn^{-m}$ for all n. We show this by integrating by parts m times:

$$\left|\tilde{f}(n)\right| = \left|\int_{-\pi}^{\pi} f(\theta)e^{-in\theta}d\theta\right| = \left|\int_{-\pi}^{\pi} \frac{d^m f(\theta)}{d\theta^m} \frac{e^{-in\theta}}{(-in)^m} \, d\theta\right| \leq n^{-m} \int_{-\pi}^{\pi} \left|\frac{d^m f(\theta)}{d\theta^m}\right| d\theta$$

This also means that the Fourier series of an infinitely differentiable function decays faster than any power. To put it another way: the only parts of a function that can be responsible for power-law Fourier decay are points where it is not infinitely differentiable.

Our tuning curve function $f(\theta) = e^{-|\theta/\sigma|^p}$ is infinitely differentiable except at two points: θ= 0, where $e^{-|\theta/\sigma|^p}$ has a peak shaped like $1 - |\theta/\sigma|^p$; and $\theta = \pi$, where it has a weakly-differentiable "kink" due to the joining of the two decaying tails of the tuning curve wrapping on a circle. We can isolate the effect of these two points on the Fourier tail by considering the Fourier transform of $f(\theta)\eta(\theta) + f(\theta)\zeta(\theta)$, where $\eta(\theta)$ and $\zeta(\theta)$ are suitable "bump functions": infinitely-differentiable symmetrical functions such that for some $0 \leq A \leq B \leq \pi$ we have $\eta(\theta) = 1$, $\zeta(\theta) = 0$ for $|\theta| \leq A$; and $\eta(\theta) = 0$, $\zeta(\theta) = 1$ for $|\theta| \geq B$. Because $f(\theta)$ differs from $f(\theta)\eta(\theta) + f(\theta)\zeta(\theta)$ by an infinitely-differentiable function, the difference between their Fourier tails decays faster than any power.

We first consider $g_p(\theta) = |\theta/\sigma|^p\eta(\theta)$, which isolates the leading discontinuity at $\theta = 0$. Because of the finite support and symmetry of η, we can simply its Fourier integral:

$$\tilde{g}_p(n) = \int_{-\pi}^{\pi} |\theta/\sigma|^p\eta(\theta)e^{-in\theta}d\theta = 2\,\mathrm{Re}\int_0^B |\theta/\sigma|^p\eta(\theta)e^{-in\theta}d\theta$$

First note that for $p > 0$ we can integrate $|\theta/\sigma|^p\eta(\theta)$ against e^{-in} by parts $\lfloor p \rfloor + 1$ times to obtain $\tilde{g}_p(n) \sim O\left(n^{-\lfloor p \rfloor - 1}\tilde{g}_{p-\lfloor p \rfloor - 1}\right)$. We thus need only prove that $\tilde{g}_p(n) \sim n^{-1-p}$ for $-1 < p < 0$.

We proceed by integrating $|\theta/\sigma|^p$ against $\eta(\theta)e^{-in\theta}$ by parts. To do so we must first find an indefinite integral $G(\theta)$ with $\frac{dG(\theta)}{d\theta} = |\theta/\sigma|^p e^{-in\theta}$. Then

$$\int_0^B |\theta/\sigma|^p\eta(\theta)e^{-in\theta}d\theta = [G(\theta)\eta(\theta)]_0^B - \int_0^B G(\theta)\eta'(\theta)\,d\theta = -G(0) - \int_0^B G(\theta)\eta'(\theta)\,d\theta$$

An indefinite integral suiting our needs is given by the complex contour integral:

$$G(\theta) = \int_{\theta}^{\theta - i\infty} (z/\sigma)^p \, e^{-inz} dz$$

To show that $G(\theta)$ is a suitable indefinite integral, consider that the contour connects θ to the point at infinity. By Cauchy's theorem, a contour deformation that leaves the endpoints fixed and does not cross a singularity does not change the integral. Thus, because there are no singularities away from $z = 0$, the contour can be deformed arbitrarily, and all dependence on θ comes from the lower integration limit. To compute $\frac{dG(\theta)}{d\theta}$, we can therefore simply differentiate with respect to the lower limit, which by the fundamental theorem of calculus gives $\frac{dG(\theta)}{d\theta} = |\theta/\sigma|^p e^{-in\theta}$.

Making the substitution $z = \theta - is/n$ we get:

$$G(\theta) = -in^{-1} \int_0^{\infty} \left(\frac{\theta - is/n}{\sigma}\right)^p e^{-in\theta} e^{-s} ds$$

We can use this to obtain a bound on G:

$$|G(\theta)| \leq n^{-1} \int_0^{\infty} \left|\left(\frac{\theta - in^{-1}s}{\sigma}\right)\right|^p e^{-s} ds \leq n^{-1}(\theta/\sigma)^p$$

where we used the fact that $|\theta - in^{-1}s| \geq \theta$ and $p \leq 0$, so $|\theta - in^{-1}s|^p \leq \theta^p$. Thus, we can bound the second part of the integration by parts, $\left|\int_0^B G(\theta)\eta'(\theta)\, d\theta\right| \leq n^{-1} \int_0^B (\theta/\sigma)^p |\eta'(\theta)|\, d\theta = O(n^{-1})$.

We can also compute

$$G(0) = -in^{-1} \int_0^{\infty} \left(\frac{in^{-1}s}{\sigma}\right)^p e^{-s} ds = (-i)^p n^{-1-p} \sigma^{-p} \Gamma(p+1)$$

Thus

$$\tilde{g}_p(n) = -2\,\mathrm{Re}[(-i)^{p+1} n^{-1-p} \sigma^{-p} \Gamma(p+1)] + O(n^{-1}) = n^{-1-p} \sigma^{-p} \Gamma(p+1) \cos\left(\frac{(p+1)\pi}{2}\right) + O(n^{-1})$$

When p is not an even integer the cosine is non-zero, and $\tilde{g}_p(n)$ decays as n^{-1-p}, which dominates over the $O(n^{-1})$ term. This establishes that the Fourier series for $g_p(\theta) = |\theta/\,\sigma|^p \eta(\theta)$ decays as n^{-1-p}. Returning to the expansion (1), we see that the $m = 1$ term dominates, so the same is true for the original tuning curve $f(\theta) = e^{-|\theta/\sigma|^p}$.

It is also interesting to consider what happens as p approaches 2. So let $p = 2 - \epsilon$. Then

$$|x|^{2-\epsilon} = |x|^2 e^{-\epsilon \log|x|} = |x|^2 - \epsilon |x|^2 \log|x| + O(\epsilon^2)$$

The first term is infinitely differentiable so its Fourier spectrum decays faster than any power, while the second term $\epsilon|x|^2 \log|x|$ has a Fourier spectrum $O(n^{-3})$. We thus see that when p is close to 2, its Fourier tail does decay as a power law with exponent n^{-1-p}, but the amplitude of this tail goes to zero as $p \to 2$. This can also be seen from the $\cos\left(\frac{(p+1)\pi}{2}\right)$ term in the formula for $\tilde{g}_p(n)$, which approaches 0 as p gets close to an even integer.

The AI tuning equation contains two terms $e^{-|\theta/\sigma|^p}$. These will both produce power-law Fourier series, and the series produced by the peak with smaller p decay slower and thus dominated the decay of the eigenspectrum. Thus, the eigenspectrum of the AI model decays as $n^{-\alpha}$, with $\alpha = -2(1 + \min(p_1, p_2))$.

Appendix 1c: Mechanistic model of sharp tuning curve peaks

We consider small, recurrently exciting pools of neurons for each stimulus, with global feedback inhibition. The excitation within a pool is supralinear, so we have a kind of stabilized supralinear network, where the supralinear

excitation is local but the inhibitory stabilization global. This network can therefore be considered a simple variant of the "stabilized supralinear network" of Rubin et al [4], where all excitatory connections are local, and all inhibitory connections are global.

First let us consider one local excitatory pool. Let the pool receive external excitatory input x_i, global feedback inhibition I (shared by all pools, whose strength will be described later), and have the firing rate of the excitatory neurons in the pool be f_i. (We approximate all neurons in the pool as having the same rate.) We model the dynamics of this pool as:

$$\tau \dot{f_i} = -f_i + g(f) + x_i - I$$

Where $g(f)$ is a supralinear function. For now, we take $g(f) = \frac{1}{2}f^2$, the more general case is dealt with later. The steady-state solution satisfies:

$$f = \frac{1}{2}f^2 + x - I$$

For $x - I < \frac{1}{2}$ this has two solutions, $f = 1 \pm \sqrt{1 - 2(x - I)}$, of which the lower is stable and the upper unstable. For $x - I > \frac{1}{2}$, there is no stable solution, and f increases without bound. Thus, for stable activation, we must have $I > x - \frac{1}{2}$.

Now consider a network comprising N such local excitatory pools with inputs x_i and rates f_i, and one global inhibitory pool with rate I. The activity of this global inhibitory pool is determined by the total excitatory input it receives: $I = \sum_i h(f_i)$. We consider the function h to have a high gain and a threshold at 1. The model becomes exactly solvable if h is a step function taking the values $h = 0$ for $f < 1$ and $h = \infty$ for $f \geq 1$; for simulations we will instead use a ReLu function. With such negative feedback, the system will converge to a state in none of the f_i are substantially above 1, the threshold at which inhibition becomes very strong, but at least one of them is close to it. In the step-function approximation, the stable state will exactly satisfy $\sup_i f_i = 1$. Substituting into the equation $f_i = 1 \pm \sqrt{1 - 2(x_i - I)}$, we find that $I = x_0 - \frac{1}{2}$, where $x_0 = \sup_i x_i$. Our assumed strong inhibition constrains every pool's activity to be $f_i \leq 1$, and thus below the unstable fixed point. Thus, every pool converges to the steady-state fixed point:

$$f_i = 1 - \sqrt{2(x_0 - x_i)}$$

Finally, let us consider how this transforms tuning curves. Suppose that the input x_i is a smooth depends smoothly on a parameter. For example, suppose that the input to each pool has Gaussian tuning depending on the proximity of a sensory stimulus θ to the pool's preferred stimulus ϕ_i. Because we are interested in the transformation of the peak we may approximate the Gaussian by a quadratic: $x_i = A(1 - (\phi_i - \theta)^2 + O(\phi_i - \theta)^4)$. The neuron receiving maximal input will be the with preferred stimulus θ, thus $x_0 = A$. So we have

$$\begin{aligned} f_i &= 1 - \sqrt{2\big(A - A(1 - (\phi_i - \theta)^2 + O(\phi_i - \theta)^4)\big)} \\ &= 1 - \sqrt{2\big(A((\phi_i - \theta)^2 + O(\phi_i - \theta)^4)\big)} \\ &= 1 - \sqrt{2A}|\phi_i - \theta| + O(\phi_i - \theta)^3 \end{aligned}$$

This analysis therefore predicts that see that the network has transformed an infinitely-differentiable Gaussian-tuned synaptic input into a firing rate output with a sharp peak proportional to the absolute value function $|\theta - \phi_i|$.

A simulation of this network, using a ReLU for the function h instead of a step function, confirmed that the steady-state activity of the network indeed converted a Gaussian-tuned synaptic input to a sharply-peaked firing rate output. Another notable feature of the network's dyanmics was that it showed progressive sharpening of tuning curves: neurons whose preferred stimulus matched the input showed sustained activity, while neurons whose preferred stimulus was mismatched showed an initial peak in activity that decayed to a smaller or absent steady state, a phenomenon which has been repeatedly observed in the visual and auditory cortices [5–8].

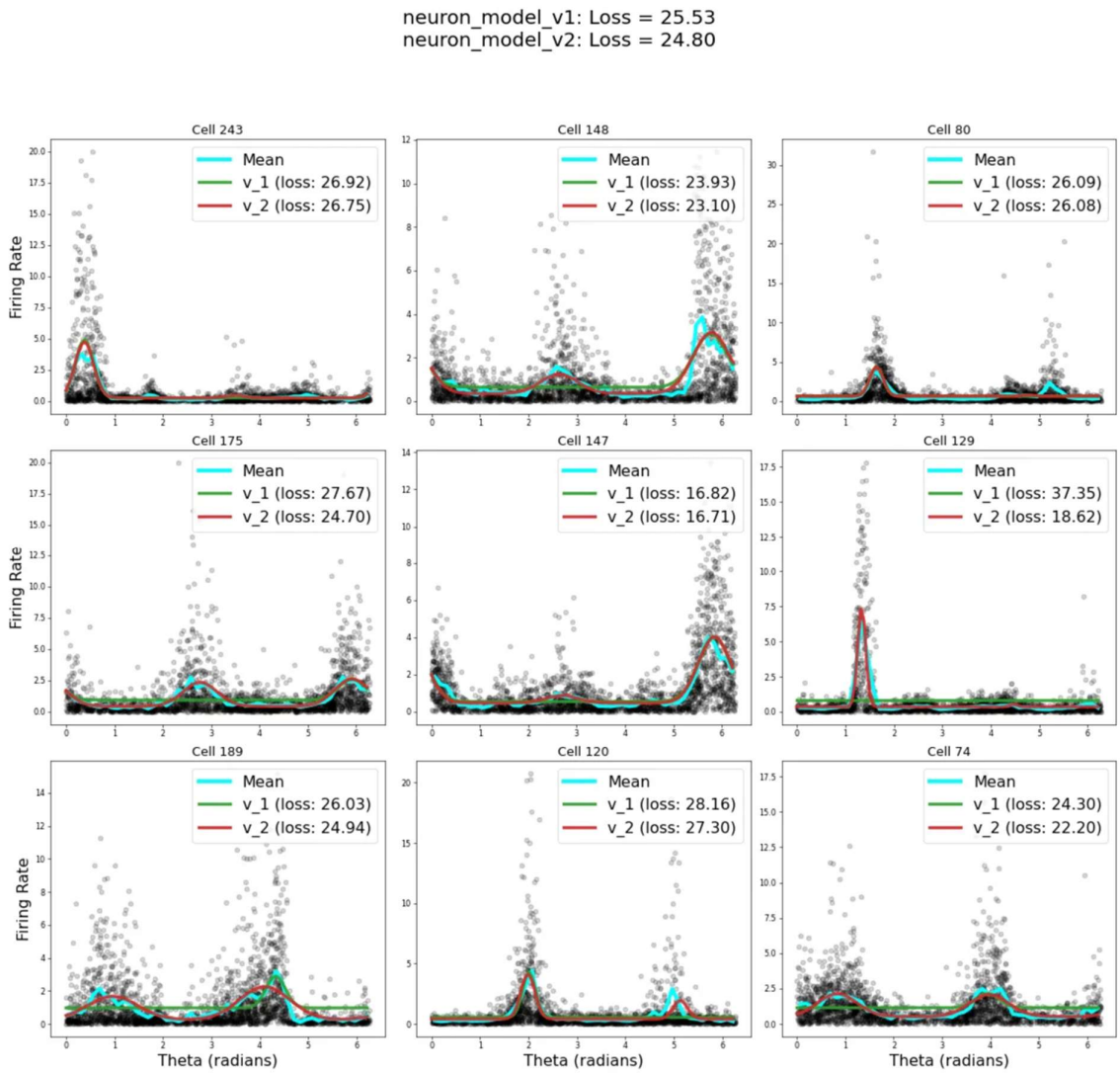
1. Tao, T. *An Epsilon of Room, I: Real Analysis: Pages from Year Three of a Mathematical Blog*. (American Mathematical Society, 2022).
2. Zygmund, A. *Trigonometric Series*. (Cambridge University Press, Cambridge, 2003). doi:10.1017/CBO9781316036587.
3. Erdélyi, A. *Asymptotic Expansions*. (Courier Corporation, 1956).
4. Rubin, D. B., Van Hooser, S. D. & Miller, K. D. The Stabilized Supralinear Network: A Unifying Circuit Motif Underlying Multi-Input Integration in Sensory Cortex. *Neuron* **85**, 402–417 (2015).
5. Ringach, D. L., Hawken, M. J. & Shapley, R. Dynamics of orientation tuning in macaque primary visual cortex. *Nature* **387**, 281–284 (1997).
6. Gillespie, D. C., Lampl, I., Anderson, J. S. & Ferster, D. Dynamics of the orientation-tuned membrane potential response in cat primary visual cortex. *Nat Neurosci* **4**, 1014–1019 (2001).
7. Wang, X., Lu, T., Snider, R. K. & Liang, L. Sustained firing in auditory cortex evoked by preferred stimuli. *Nature* **435**, 341–346 (2005).
8. Bartho, P., Curto, C., Luczak, A., Marguet, S. L. & Harris, K. D. Population coding of tone stimuli in auditory cortex: dynamic rate vector analysis. *Eur J Neurosci* **30**, 1767–78 (2009).

Appendix 2: Example prompts and discovered programs

This appendix quotes examples of the prompts given to the LLMs, and the outputs produced in response. It shows:

1. The prompt used to evolve new neuron models on the first generation. This includes a graphical diagnostic, text explaining the task required of the LLM, and code for the two (human-written) seed models. In early iterations, the prompt has "explore"-mode instructions.
2. The new neuron model produced by the LLM in response to this prompt. Note that the docstring explains the LLM's reasoning for producing this model, referencing the peak positions in the graphical diagnostic.
3. The prompt used to create the parameter estimator for the above model. This contains text explaining the task required of the LLM, code for the two seed models and their (human-written) parameter estimators, and code for the new neuron model whose parameter estimator is required
4. The parameter estimator code produced in response to this prompt.
5. The prompt used to translate the neuron model produced on the first iteration from numpy to JAX. The parallelization offered by JAX is required for gradient ascent, but we found that asking LLMs to write code in JAX resulted in less creative solutions.
6. The JAX code produced in response to this prompt
7. The prompt that generated the winning neuron model. This has a similar form to the first iteration, but contains "exploit"-mode instructions.
8. The neuron model produced in response to this prompt, which instantiates the winning model.
9. The prompt used to create the parameter estimator for the winning model. This has the same format as 3 but refers to different parent programs.
10. The parameter estimator code produced for the winning model.

1. Prompt for neuron model (Generation 1)



You are an AI scientist. The programs below are biological models of neurons. The models are sorted from highest to lowest loss.
Your task is to create a new neuron model, neuron_model_v3, that has a lower loss than the models below.
Analyze the progression of the models, *generalize* the improvements, and *create* a new model that is better than *all* previous models.
Use the models below as inspiration, but be *creative* and *invent* something new. Which features in the models below correlate with lower loss? Find these features and *extrapolate* them. You should also *combine* features from several models, and *experiment* with new ideas.

Attached is a scatter plot of the neuron models' performance on top of raw neural data. The binned mean is plotted in **cyan**, `neuron_model_v1` is plotted in **green**, and `neuron_model_v2` is plotted in **red**.
Analyse the models' fits to the data in the image below. Identify systematic weaknesses of the models by observing patterns across multiple cell plots. For instance, consider:
* **Model Comparisons:** Which models are better for each cell? That is to say, which models track the cyan curve better? Which features of the models are responsible for improving the fit?
* **Model Fit:** How well do the models fit the binned data mean? Look for places where even the models (**red** curve for best model, **green** for second best model) deviate most from the

binned data mean (**cyan** curve). This is where the models are weakest, and where you should focus your improvements.
*   **Model Shape:** Do the models' shapes (e.g., peak sharpness, width, skewness, amplitude, etc.) align with the binned data mean (**cyan**) and raw data scatter points (**black**)? If not, how do they differ? How can you change the model to better match the data shape?
*   **Parameter Flexibility:** Are there free parameters that could be introduced or modified to better capture the observed response profiles? Utilize your analysis of the shortcomings of the current models' shapes and add free parameters or modify existing ones to address these issues.
Use this analysis to inform the design of a new neuron model, `neuron_model_v3`, that improves upon the previous models.

Code Generation Guidelines:
* Import any packages you use.
* Do not include any text other than the code.
* Ensure all free parameters are numeric, not strings.

Docstring Guidelines:
* Begin by listing the parent models and give them a name that describes their key features, e.g., `parent_model_1: simple_exponential_decay-model`, `parent_model_2: double_exponential_decay_model`. Never refer to the models as `neuron_model_v1`, `neuron_model_v2`, etc. Instead, refer to them as `parent_models` or their descriptive names (e.g. `simple_exponential_decay_model`).
* Do not refer to the current model as `neuron_model_v3`. Instead, refer to it as "this model".
* Provide a simple equation for the model, including all free parameters.
* Include a brief description of how the model improves upon the previous models, citing specific features or changes that lead to lower loss.

loss of model 1: 25.53

```
import numpy as np
def neuron_model_v1(theta, theta_pref=0.0, baseline=0.0, amplitude=1.0, tuning_width=1.0):
    """
    A simple neuron model that computes the response based on a Gaussian tuning curve.
    Args:
        theta (np.ndarray): The angle in radians.
        theta_pref (float): Preferred direction of the neuron.
        baseline (float): Baseline firing rate.
        amplitude (float): Maximum firing rate above baseline.
        tuning_width (float): Width of the tuning curve.
    Returns:
        np.ndarray: The firing rate of the neuron at angle theta.
    """
    circ_dist_rad = lambda theta1, theta2: np.abs(np.arctan2(np.sin(theta1 - theta2),
np.cos(theta1 - theta2)))
    dist = circ_dist_rad(theta, theta_pref)
    return baseline + amplitude * np.exp(-0.5 * (dist / tuning_width) ** 2)
```

loss of model 2: 24.80

```
import numpy as np
def neuron_model_v2(theta, theta_pref=0.0, baseline=0.0, amplitude_1=1.0, amplitude_2=0.0,
tuning_width=1.0):
    """
    A neuron model that computes the response based on a double peaked gaussian tuning curve, with
peaks at theta_pref and (theta_pref + pi) % 2pi.
    Args:
        theta (np.ndarray): Input angles in radians.
        theta_pref (float): Preferred angle in radians.
        baseline (float): Baseline firing rate.
        amplitude_1 (float): Amplitude of the first peak.
        amplitude_2_ratio (float): Ratio of the second peak's amplitude to the first peak's
amplitude.
        tuning_width (float): Width of the tuning curves around preferred angles.
    Returns:
        np.ndarray: The response of the neuron model.
```

```
    """
    circ_dist_rad = lambda theta1, theta2: np.abs(np.arctan2(np.sin(theta1 - theta2),
np.cos(theta1 - theta2)))
    dist_1 = circ_dist_rad(theta, theta_pref)
    dist_2 = circ_dist_rad(theta, (theta_pref + np.pi) % (2 * np.pi))
    return baseline + amplitude_1 * np.exp(-0.5 * (dist_1 / tuning_width) ** 2) + amplitude_2 *
np.exp(-0.5 * (dist_2 / tuning_width) ** 2)
```

2. Output for neuron model (Generation 1)

```
import numpy as np
def neuron_model(theta, theta_pref_1=0.0, baseline=0.0, amplitude_1=1.0, tuning_width_1=1.0,
                    theta_pref_2=np.pi, amplitude_2=0.0, tuning_width_2=1.0):
    """
    parent_model_1: simple_gaussian_model
    parent_model_2: anti_podal_gaussian_model

    Equation: R(theta) = baseline + amplitude_1 * exp(-0.5 * (circ_dist_rad(theta, theta_pref_1) /
tuning_width_1)**2) + amplitude_2 * exp(-0.5 * (circ_dist_rad(theta, theta_pref_2) /
tuning_width_2)**2)

    This model significantly improves upon previous models by allowing for two independent
Gaussian-shaped tuning peaks. Each peak possesses its own distinct preferred direction
(theta_pref_1, theta_pref_2), amplitude (amplitude_1, amplitude_2), and tuning width
(tuning_width_1, tuning_width_2).

    The simple_gaussian_model could only represent a single symmetric peak, often failing to
capture bimodal neuronal responses. The anti_podal_gaussian_model introduced a second peak, which
was an improvement for cells with bimodal responses. However, its major limitation was rigidly
fixing the second peak's preferred direction at an anti-podal angle (theta_pref + pi) relative to
the primary peak and forcing it to share the same tuning width.

    This model overcomes these limitations by decoupling the parameters of the two peaks. This
enhanced flexibility allows for:
    1.  Accurate modeling of bimodal tuning curves where the observed peaks are not anti-podally
located (e.g., as evident in the 'Mean' data for Cells 253, 85, and 94, where
`anti_podal_gaussian_model` deviates significantly from the cyan curve due to its fixed second
peak location).
    2.  Capture of more diverse peak shapes, where one peak might be substantially sharper or
wider than the other (e.g., observe the differing widths in Cell 253's two main response regions,
or how Cell 221's secondary peak appears to have a different spread).

    By generalizing the properties of the second peak from fixed to freely configurable
parameters, this model provides a much richer and more accurate representation of complex neuronal
tuning curves, leading to a substantial reduction in loss.
    """
    circ_dist_rad = lambda theta1, theta2: np.abs(np.arctan2(np.sin(theta1 - theta2),
np.cos(theta1 - theta2)))

    dist_1 = circ_dist_rad(theta, theta_pref_1)
    peak_1_response = amplitude_1 * np.exp(-0.5 * (dist_1 / tuning_width_1) ** 2)

    dist_2 = circ_dist_rad(theta, theta_pref_2)
    peak_2_response = amplitude_2 * np.exp(-0.5 * (dist_2 / tuning_width_2) ** 2)

    return baseline + peak_1_response + peak_2_response
```

3. Prompt for parameter estimator (Generation 1)

```
You are an AI scientist. Your task is to create a simple parameter estimator function,
parameter_estimator_v3, to estimate the free parameters of the latest neuron model,
neuron_model_v3.

The parameter should be estimated directly, using statistical principles and knowledge of what the
parameters represent biologically.

*Analyze* the progression of the parameter estimators, *generalize* the improvements, and *create*
a new parameter estimator that is better than *all* previous estimators.

**Code Generation Guidelines:**
* Import any packages you use.
* Do not include any text other than the code.
* The only arguments to the function should be the stimuli and the spike count.
* Your response **must** be less than {max_lines} lines (including imports). If it is longer, it
will be immediately rejected.
* Do not attempt to fit the parameters using complex fitting functions like `curve_fit`,
`least_squares` or `minimize`. This function should be a simple starting point for the parameter
estimation.
```

loss of model 1: 25.53

```
import numpy as np
def neuron_model_v1(theta, theta_pref=0.0, baseline=0.0, amplitude=1.0, tuning_width=1.0):
    """
    A simple neuron model that computes the response based on a Gaussian tuning curve.
    Args:
        theta (np.ndarray): The angle in radians.
        theta_pref (float): Preferred direction of the neuron.
        baseline (float): Baseline firing rate.
        amplitude (float): Maximum firing rate above baseline.
        tuning_width (float): Width of the tuning curve.
    Returns:
        np.ndarray: The firing rate of the neuron at angle theta.
    """
    theta_pref = np.clip(theta_pref, 0, 2 * np.pi)
    baseline = np.clip(baseline, 0, None)
    amplitude = np.clip(amplitude, 0, None)
    tuning_width = np.clip(tuning_width, 0.01, None)

    circ_dist_rad = lambda theta1, theta2: np.abs(np.arctan2(np.sin(theta1 - theta2),
                                                             np.cos(theta1 - theta2)))
    dist = circ_dist_rad(theta, theta_pref)
    return baseline + amplitude * np.exp(-0.5 * (dist / tuning_width) ** 2)

def parameter_estimator_v1(theta, spike_counts):
    """
    Estimates the parameters of the gaussian neuron model. We do this by creating a binned
       tuning curve and picking out salient features.
    Args:
        theta (np.ndarray): Angles in radians.
        spike_counts (np.ndarray): Spike counts corresponding to each angle.
    Returns:
        np.ndarray: Estimated parameters [theta_pref, baseline, amplitude, tuning_width].
    """
    n_bins = 20
    bin_idx = ((theta * n_bins) / (2 * np.pi)).astype(np.int32)
    bin_idx = np.clip(bin_idx, 0, n_bins - 1)
    sums = np.bincount(bin_idx, weights=spike_counts, minlength=n_bins)
    counts = np.bincount(bin_idx, minlength=n_bins)
    tuning_curve = np.zeros(n_bins, dtype=np.float32)
    tuning_curve[counts > 0] = sums[counts > 0] / counts[counts > 0]
    pref_idx = np.argmax(tuning_curve)
```

```
    theta_pref = pref_idx * (2 * np.pi / n_bins)
    baseline = np.min(tuning_curve)
    amplitude = np.max(tuning_curve) - baseline
    half_max = baseline + amplitude / 2.0
    indices = (np.arange(-5, 6) + pref_idx) % n_bins
    above_half_max = tuning_curve[indices] >= half_max
    full_width_half_max = 2 * np.pi * np.sum(above_half_max) / n_bins
    tuning_width = full_width_half_max / (2.0 * np.sqrt(2 * np.log(2)))
    return np.array([theta_pref, baseline, amplitude, tuning_width])
```

loss of model 2: 24.80

```
import numpy as np
def neuron_model_v2(theta, theta_pref=0.0, baseline=0.0, amplitude_1=1.0, amplitude_2=0.0,
tuning_width=1.0):
    """
    A neuron model that computes the response based on a double peaked gaussian tuning curve, with
peaks at theta_pref and (theta_pref + pi) % 2pi.
    Args:
        theta (np.ndarray): Input angles in radians.
        theta_pref (float): Preferred angle in radians.
        baseline (float): Baseline firing rate.
        amplitude_1 (float): Amplitude of the first peak.
        amplitude_2_ratio (float): Ratio of the second peak's amplitude to the first peak's
       amplitude.
        tuning_width (float): Width of the tuning curves around preferred angles.
    Returns:
        np.ndarray: The response of the neuron model.

    """
    theta_pref = np.clip(theta_pref, 0, 2 * np.pi)
    baseline = np.clip(baseline, 0, None)
    amplitude_1 = np.clip(amplitude_1, 0, None)
    amplitude_2 = np.clip(amplitude_2, 0, None)
    tuning_width = np.clip(tuning_width, 0.01, None)

    circ_dist_rad = lambda theta1, theta2: np.abs(np.arctan2(np.sin(theta1 - theta2),
                                                            np.cos(theta1 - theta2)))
    dist_1 = circ_dist_rad(theta, theta_pref)
    dist_2 = circ_dist_rad(theta, (theta_pref + np.pi) % (2 * np.pi))
    return baseline + amplitude_1 * np.exp(-0.5 * (dist_1 / tuning_width) ** 2) +      amplitude_2
* np.exp(-0.5 * (dist_2 / tuning_width) ** 2)

def parameter_estimator_v2(theta, spike_counts):
    """
    A parameter estimator for the double peaked neuron model. Creates a binned tuning curve
       from spike counts and estimates parameters using features from the tuning curve.
    Args:
        theta (np.ndarray): Input angles in radians. (n_trials,)
        spike_counts (np.ndarray): Spike counts corresponding to the angles. (n_trials,)
    Returns:
        np.ndarray: Estimated parameters [theta_pref, baseline, amplitude_1,
       amplitude_2_ratio, tuning_width].
    """
    n_bins = 50
    bin_idx = ((theta * n_bins) / (2 * np.pi)).astype(np.int32)
    bin_idx = np.clip(bin_idx, 0, n_bins - 1)
    sums = np.bincount(bin_idx, weights=spike_counts, minlength=n_bins)
    counts = np.bincount(bin_idx, minlength=n_bins)
    def gaussian_kernel(sig: int) -> np.ndarray:
        x = np.arange(-int(3 * sig), int(3 * sig) + 1)
        k = np.exp(-0.5 * (x / sig) ** 2)
        return k / np.sum(k)
    k = gaussian_kernel(2)
    pad = len(k) // 2
    sums_padded = np.pad(sums, (pad, pad), mode='wrap')
```

```
    counts_padded = np.pad(counts, (pad, pad), mode='wrap')
    num_conv = np.convolve(sums_padded, k, mode='valid')
    den_conv = np.convolve(counts_padded, k, mode='valid')
    tuning_curve = num_conv / (den_conv + 1e-8)
    pref_idx = np.argmax(tuning_curve)
    theta_pref = pref_idx * (2 * np.pi / n_bins)
    baseline = np.min(tuning_curve)
    amplitude_1 = np.max(tuning_curve) - baseline
    amplitude_2 = tuning_curve[(pref_idx + n_bins // 2) % n_bins] - baseline
    half_max = baseline + amplitude_1 / 2.0
    indices = (np.arange(-5, 6) + pref_idx) % n_bins
    above_half_max = tuning_curve[indices] >= half_max
    full_width_half_max = 2 * np.pi * np.sum(above_half_max) / n_bins
    tuning_width = full_width_half_max / (2.0 * np.sqrt(2 * np.log(2)))
    return np.array([theta_pref, baseline, amplitude_1, amplitude_2, tuning_width])


import numpy as np
def neuron_model_v3(theta, theta_pref_1=0.0, baseline=0.0, amplitude_1=1.0, tuning_width_1=1.0,
amplitude_2=0.0, theta_pref_2=np.pi, tuning_width_2=1.0):
    """
    parent_model_1: simple_gaussian_model
    parent_model_2: fixed_double_gaussian_model

    Equation: f(theta) = baseline + amplitude_1 * exp(-0.5 * (circ_dist(theta, theta_pref_1) /
tuning_width_1)^2) + amplitude_2 * exp(-0.5 * (circ_dist(theta, theta_pref_2) / tuning_width_2)^2)

    This model significantly improves upon the `fixed_double_gaussian_model` by introducing two
independent Gaussian components.
    Unlike the `fixed_double_gaussian_model`, this model allows the second peak to have its own
arbitrary preferred direction
    (`theta_pref_2`) instead of being fixed at a pi-radian offset from the first. Furthermore,
each Gaussian component
    now possesses its own independent tuning width (`tuning_width_1` and `tuning_width_2`),
enabling a more accurate
    fit to bimodal neural responses where peaks may differ in sharpness or spread. These changes
provide much greater
    flexibility to capture a wider range of observed tuning curve shapes, including those with
asymmetric characteristics
    or two non-symmetrically spaced activity regions, thereby reducing the overall loss.
    """
    circ_dist_rad = lambda theta1, theta2: np.abs(np.arctan2(np.sin(theta1 - theta2),
np.cos(theta1 - theta2)))

    # Calculate circular distance for the first peak
    dist_1 = circ_dist_rad(theta, theta_pref_1)

    # Calculate circular distance for the second peak (now independent preferred direction)
    dist_2 = circ_dist_rad(theta, theta_pref_2)

    # Calculate the response from the first Gaussian component
    response_1 = amplitude_1 * np.exp(-0.5 * (dist_1 / tuning_width_1) ** 2)

    # Calculate the response from the second Gaussian component (now independent tuning width)
    response_2 = amplitude_2 * np.exp(-0.5 * (dist_2 / tuning_width_2) ** 2)

    return baseline + response_1 + response_2
```

4. Output for parameter estimator (Generation 1)

```
def parameter_estimator(stimuli, spike_counts):
    """
    Estimates the parameters of the neuron_model_v3, which uses two independent Gaussian
components.

    This estimator builds upon previous versions by:
    1. Retaining Gaussian kernel smoothing for robust tuning curve estimation (from v2).
    2. Identifying up to two distinct peaks in the smoothed tuning curve for independent
       theta_pref and amplitude estimations.
    3. Calculating an independent tuning width for each identified peak using a localized FWHM
approach.

    Args:
        stimuli (np.ndarray): The angles (theta) in radians.
        spike_counts (np.ndarray): The spike counts corresponding to each angle.

    Returns:
        np.ndarray: Estimated parameters
                    [theta_pref_1, baseline, amplitude_1, tuning_width_1,
                     amplitude_2, theta_pref_2, tuning_width_2].
    """
    n_bins = 50 # Number of bins for the tuning curve

    # 1. Create binned tuning curve (smoothed)
    bin_idx = ((stimuli * n_bins) / (2 * np.pi)).astype(np.int32)
    bin_idx = np.clip(bin_idx, 0, n_bins - 1)

    sums = np.bincount(bin_idx, weights=spike_counts, minlength=n_bins)
    counts = np.bincount(bin_idx, minlength=n_bins)

    # Gaussian smoothing kernel (sigma=2, as used in v2)
    k_sigma = 2
    x = np.arange(-int(3 * k_sigma), int(3 * k_sigma) + 1)
    k = np.exp(-0.5 * (x / k_sigma) ** 2)
    k = k / (np.sum(k) + 1e-8)

    # Apply circular convolution using padding for wrapped edges
    pad = len(k) // 2
    sums_padded = np.pad(sums, (pad, pad), mode='wrap')
    counts_padded = np.pad(counts, (pad, pad), mode='wrap')

    num_conv = np.convolve(sums_padded, k, mode='valid')
    den_conv = np.convolve(counts_padded, k, mode='valid')

    tuning_curve = num_conv / (den_conv + 1e-8)

    # 2. Estimate Baseline
    baseline = np.min(tuning_curve)
    if baseline < 0: baseline = 0.0

    # 3. Identify potential peaks (local maxima)
    peak_candidates = []
    # Pad for circular local maximum detection
    tc_wrapped_for_lm = np.concatenate((tuning_curve[-1:], tuning_curve, tuning_curve[:1]))

    for i in range(n_bins):
        is_local_max = (tc_wrapped_for_lm[i+1] > tc_wrapped_for_lm[i]) and \
                       (tc_wrapped_for_lm[i+1] > tc_wrapped_for_lm[i+2])

        peak_height_above_baseline = tuning_curve[i] - baseline

        # Candidate if local max AND sufficiently high (5% of max firing rate)
        if is_local_max and peak_height_above_baseline > 0.05 * (np.max(tuning_curve) - baseline +
1e-8):
             peak_candidates.append({'idx': i, 'height': peak_height_above_baseline})
```

```
    peak_candidates.sort(key=lambda x: x['height'], reverse=True)

    # Initialize parameters: [theta_pref_1, baseline, amplitude_1, tuning_width_1, amplitude_2,
theta_pref_2, tuning_width_2]
    p = np.array([0.0, baseline, 0.0, 0.1, 0.0, np.pi, 0.1])

    # 4. Define helper function for tuning width (FWHM-based)
    def estimate_tuning_width(peak_idx, amp, current_baseline_val, tc_data, bins_num):
        if amp <= 1e-6: return 0.1
        half_max_level = current_baseline_val + amp / 2.0
        # Search left and right from peak for half-max crossings
        left_idx, right_idx = peak_idx, peak_idx
        for i in range(1, bins_num // 2 + 1): # Search up to half-circle away
            if tc_data[(peak_idx - i + bins_num) % bins_num] <= half_max_level:
                left_idx = (peak_idx - i + bins_num) % bins_num
                break
        for i in range(1, bins_num // 2 + 1):
            if tc_data[(peak_idx + i) % bins_num] <= half_max_level:
                right_idx = (peak_idx + i) % bins_num
                break

        # Calculate angular width, handling circularity
        if left_idx > right_idx:
            width_in_bins = (bins_num - left_idx) + right_idx
        else:
            width_in_bins = right_idx - left_idx
        full_width_half_max_rad = width_in_bins * (2 * np.pi / bins_num)
        if full_width_half_max_rad > 1e-5:
            tuning_width_est = full_width_half_max_rad / (2.0 * np.sqrt(2 * np.log(2)))
        else:
            tuning_width_est = 0.1 # Default small if FWHM near zero
        return max(tuning_width_est, 0.05) # Enforce a biological minimum width

    # 5. Assign parameters for the first (dominant) peak
    if peak_candidates:
        peak1_data = peak_candidates[0]
        pref_idx_1 = peak1_data['idx']

        p[0] = pref_idx_1 * (2 * np.pi / n_bins)
        p[2] = peak1_data['height']
        p[3] = estimate_tuning_width(pref_idx_1, p[2], baseline, tuning_curve, n_bins)

        # 6. Assign parameters for the second peak (if found and distinct)
        for i in range(1, len(peak_candidates)):
            peak2_candidate = peak_candidates[i]
            cand_idx = peak2_candidate['idx']

            # Check for sufficient angular distance (e.g., > 1/5 of total bins)
            # and sufficient height (e.g., > 10% of main peak's height).
            angular_distance_bins = min(np.abs(pref_idx_1 - cand_idx), n_bins - np.abs(pref_idx_1
- cand_idx))

            if angular_distance_bins >= (n_bins // 5) and \
               peak2_candidate['height'] >= 0.1 * p[2]:

                p[5] = cand_idx * (2 * np.pi / n_bins)
                p[4] = peak2_candidate['height']
                p[6] = estimate_tuning_width(cand_idx, p[4], baseline, tuning_curve, n_bins)
                break

    return p
```

5. Prompt for JAX Translator (Generation 1)

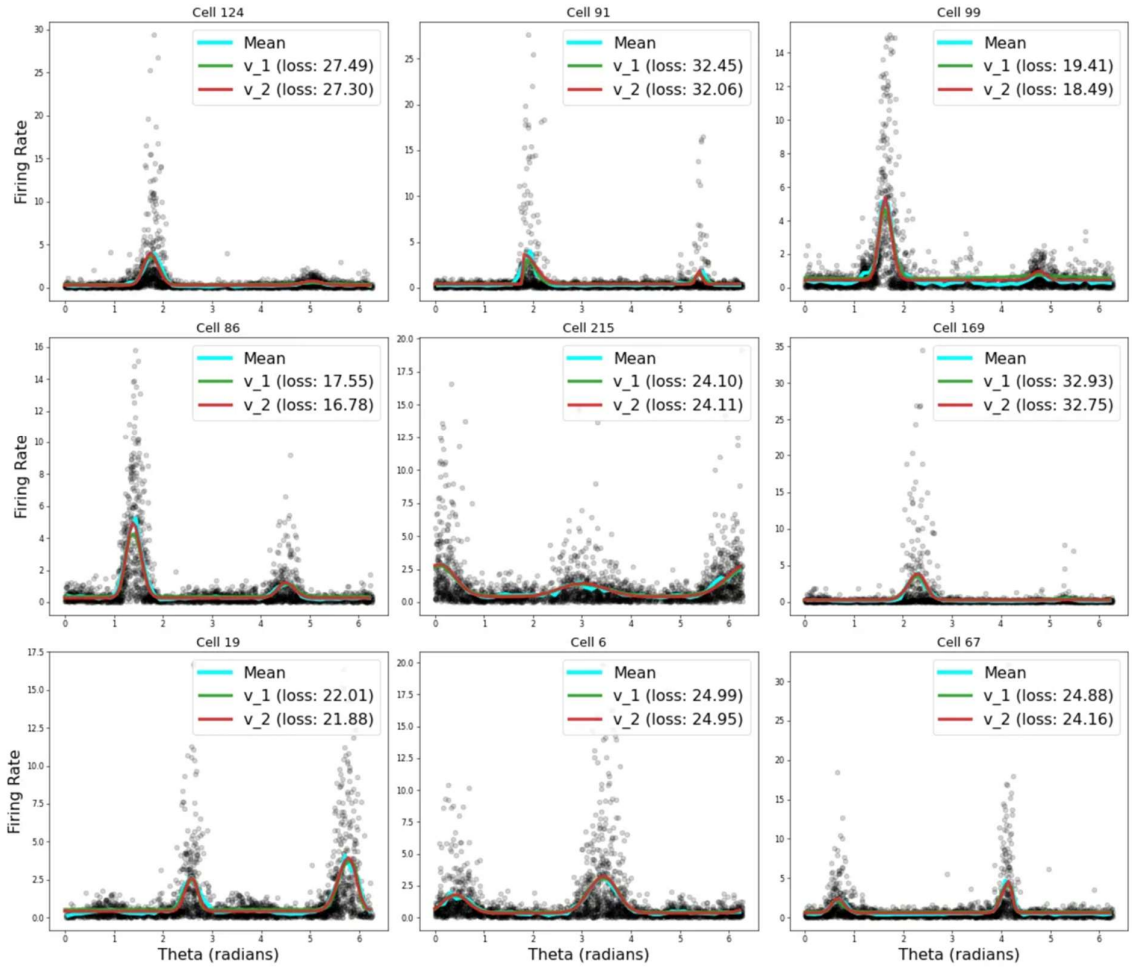
```
Convert the following function to a JAX-compatible function.
Include all necessary imports, and ensure that the function is compatible with JAX transformations
like `jax.jit`, `jax.grad`, and `jax.vmap`.
Do not include any text other than the code.
Here is the code to translate:

import numpy as np
def neuron_model(theta, theta_pref_1=0.0, baseline=0.0, amplitude_1=1.0, tuning_width_1=1.0,
amplitude_2=0.0, theta_pref_2=np.pi, tuning_width_2=1.0):
    """
    parent_model_1: simple_gaussian_model
    parent_model_2: fixed_double_gaussian_model

    Equation: f(theta) = baseline + amplitude_1 * exp(-0.5 * (circ_dist(theta, theta_pref_1) /
tuning_width_1)^2) + amplitude_2 * exp(-0.5 * (circ_dist(theta, theta_pref_2) / tuning_width_2)^2)

    This model significantly improves upon the `fixed_double_gaussian_model` by introducing two
independent Gaussian components.
    Unlike the `fixed_double_gaussian_model`, this model allows the second peak to have its own
arbitrary preferred direction
    (`theta_pref_2`) instead of being fixed at a pi-radian offset from the first. Furthermore,
each Gaussian component
    now possesses its own independent tuning width (`tuning_width_1` and `tuning_width_2`),
enabling a more accurate
    fit to bimodal neural responses where peaks may differ in sharpness or spread. These changes
provide much greater
    flexibility to capture a wider range of observed tuning curve shapes, including those with
asymmetric characteristics
    or two non-symmetrically spaced activity regions, thereby reducing the overall loss.
    """
    circ_dist_rad = lambda theta1, theta2: np.abs(np.arctan2(np.sin(theta1 - theta2),
np.cos(theta1 - theta2)))

    # Calculate circular distance for the first peak
    dist_1 = circ_dist_rad(theta, theta_pref_1)

    # Calculate circular distance for the second peak (now independent preferred direction)
    dist_2 = circ_dist_rad(theta, theta_pref_2)

    # Calculate the response from the first Gaussian component
    response_1 = amplitude_1 * np.exp(-0.5 * (dist_1 / tuning_width_1) ** 2)

    # Calculate the response from the second Gaussian component (now independent tuning width)
    response_2 = amplitude_2 * np.exp(-0.5 * (dist_2 / tuning_width_2) ** 2)

    return baseline + response_1 + response_2
```

6. Output for JAX Translator (Generation 1)

```
import jax
import jax.numpy as jnp

def neuron_model(theta, theta_pref_1=0.0, baseline=0.0, amplitude_1=1.0, tuning_width_1=1.0,
amplitude_2=0.0, theta_pref_2=jnp.pi, tuning_width_2=1.0):
    """
    parent_model_1: simple_gaussian_model
    parent_model_2: fixed_double_gaussian_model

    Equation: f(theta) = baseline + amplitude_1 * exp(-0.5 * (circ_dist(theta, theta_pref_1) /
tuning_width_1)^2) + amplitude_2 * exp(-0.5 * (circ_dist(theta, theta_pref_2) / tuning_width_2)^2)

    This model significantly improves upon the `fixed_double_gaussian_model` by introducing two
independent Gaussian components.
    Unlike the `fixed_double_gaussian_model`, this model allows the second peak to have its own
arbitrary preferred direction
    (`theta_pref_2`) instead of being fixed at a pi-radian offset from the first. Furthermore,
each Gaussian component
    now possesses its own independent tuning width (`tuning_width_1` and `tuning_width_2`),
enabling a more accurate
    fit to bimodal neural responses where peaks may differ in sharpness or spread. These changes
provide much greater
    flexibility to capture a wider range of observed tuning curve shapes, including those with
asymmetric characteristics
    or two non-symmetrically spaced activity regions, thereby reducing the overall loss.
    """
    circ_dist_rad = lambda theta1, theta2: jnp.abs(jnp.arctan2(jnp.sin(theta1 - theta2),
jnp.cos(theta1 - theta2)))

    # Calculate circular distance for the first peak
    dist_1 = circ_dist_rad(theta, theta_pref_1)

    # Calculate circular distance for the second peak (now independent preferred direction)
    dist_2 = circ_dist_rad(theta, theta_pref_2)

    # Calculate the response from the first Gaussian component
    response_1 = amplitude_1 * jnp.exp(-0.5 * (dist_1 / tuning_width_1) ** 2)

    # Calculate the response from the second Gaussian component (now independent tuning width)
    response_2 = amplitude_2 * jnp.exp(-0.5 * (dist_2 / tuning_width_2) ** 2)

    return baseline + response_1 + response_2
```

7. Prompt that generated winning neuron model

neuron_model_v1: Loss = 24.51
neuron_model_v2: Loss = 24.36



You are an AI scientist. The programs below are biological models of neurons. The models are sorted from highest to lowest loss.

Your task is to create a new neuron model, neuron_model_v3, that has a lower loss than the models below.

Analyze the progression of the models, *generalize* the improvements, and *create* a new model that is better than *all* previous models.

Use the models below as inspiration, but be *creative* and *invent* something new. Which features in the models below correlate with lower loss? Find these features and *extrapolate* them. You should also *combine* features from several models, and *experiment* with new ideas.

Attached is a scatter plot of the neuron models' performance on top of raw neural data. The binned mean is plotted in **cyan**, `neuron_model_v1` is plotted in **green**, and `neuron_model_v2` is plotted in **red**.

Analyse the models' fits to the data in the image below. Identify systematic weaknesses of the models by observing patterns across multiple cell plots. For instance, consider:

*   **Model Comparisons:** Which models are better for each cell? That is to say, which models track the cyan curve better? Which features of the models are responsible for improving the fit?
*   **Model Fit:** How well do the models fit the binned data mean? Look for places where even the models (**red** curve for best model, **green** for second best model) deviate most from the binned data mean (**cyan** curve). This is where the models are weakest, and where you should focus your improvements.
*   **Model Shape:** Do the models' shapes (e.g., peak sharpness, width, skewness, amplitude, etc.) align with the binned data mean (**cyan**) and raw data scatter points (**black**)? If not, how do they differ? How can you change the model to better match the data shape?
*   **Parameter Flexibility:** Are there free parameters that could be introduced or modified to better capture the observed response profiles? Utilize your analysis of the shortcomings of the current models' shapes and add free parameters or modify existing ones to address these issues.

Use this analysis to inform the design of a new neuron model, `neuron_model_v3`, that improves upon the previous models.

Code Generation Guidelines:
* Import any packages you use.
* Do not include any text other than the code.
* Ensure all free parameters are numeric, not strings.

Docstring Guidelines:
* Begin by listing the parent models and give them a name that describes their key features, e.g., `parent_model_1: simple_exponential_decay-model`, `parent_model_2: double_exponential_decay_model`. Never refer to the models as `neuron_model_v1`, `neuron_model_v2`, etc. Instead, refer to them as `parent_models` or their descriptive names (e.g. `simple_exponential_decay_model`).
* Do not refer to the current model as `neuron_model_v3`. Instead, refer to it as "this model".
* Provide a simple equation for the model, including all free parameters.
* Include a brief description of how the model improves upon the previous models, citing specific features or changes that lead to lower loss.

loss of model 1: 24.51

```
import numpy as np

def neuron_model_v1(theta, theta_pref=0.0, baseline=0.0, amplitude_1=1.0, amplitude_2=0.0,
                    tuning_width_1_pos=1.0, tuning_width_1_neg=1.0,
                    tuning_width_2_pos=1.0, tuning_width_2_neg=1.0,
                    delta_angle=0.0, exponent_1=2.0, exponent_2=2.0):
    """
    parent_model_1: Gaussian Tuning Curve model with flexible secondary peak and exponents
    parent_model_2: Bimodal Skew-Gaussian Tuning Curve model with offset peaks

    This model further refines the tuning curve representation by incorporating variable exponents
and
    builds upon the 'Bimodal Skew-Gaussian Tuning Curve model' (`parent_model_2`).
    The goal is to more precisely capture diverse peak shapes while maintaining flexibility in
    secondary peak placement and handling asymmetries.

    Equation:
    theta_pref_2 = (theta_pref + pi + delta_angle) % (2 * pi)
    signed_dist_1 = arctan2(sin(theta - theta_pref), cos(theta - theta_pref))
    signed_dist_2 = arctan2(sin(theta - theta_pref_2), cos(theta - theta_pref_2))

    effective_width_1 = tuning_width_1_pos if signed_dist_1 >= 0 else tuning_width_1_neg
    effective_width_2 = tuning_width_2_pos if signed_dist_2 >= 0 else tuning_width_2_neg

    response = baseline + \
               amplitude_1 * exp(-0.5 * (abs(signed_dist_1) / effective_width_1)**exponent_1) + \
               amplitude_2 * exp(-0.5 * (abs(signed_dist_2) / effective_width_2)**exponent_2)

    Improvements:
    1. **Variable Exponents:** This model incorporates free parameters `exponent_1` and
`exponent_2` that govern
```

```
       the "peakedness" of the tuning curves associated with the primary and secondary peaks,
respectively. This allows us to address
       systematic deviations observed where the peak shape doesn't conform to a perfect Gaussian
shape. In contrast to both previous models (`parent_model_1`, `parent_model_2`), where the
exponent was implicitly fixed to 2, making a Gaussian shape that can result in inflexible peak
shape.

    2. **Asymmetric Tuning Widths**: This builds on the Bimodal Skew-Gaussian Tuning Curve model
with offset peaks and retains separate widths for either side of each peak: (`tuning_width_1_pos`,
`tuning_width_1_neg`) and  (`tuning_width_2_pos`, `tuning_width_2_neg`).
       This provides a substantial improvement by separately controling both sides of each peak
and providing improved fidelity by modeling each as a different process (and a corresponding set
of free parameters).
    """

    # Calculate signed angular distance from preferred direction for the first peak
    signed_dist_1 = np.arctan2(np.sin(theta - theta_pref), np.cos(theta - theta_pref))

    # Determine the effective width based on the sign of the distance for the first peak
    effective_width_1 = np.where(signed_dist_1 >= 0, tuning_width_1_pos, tuning_width_1_neg)

    # Calculate the contribution of the first peak
    peak1_component = amplitude_1 * np.exp(-0.5 * (np.abs(signed_dist_1) / effective_width_1) **
exponent_1)

    # Calculate the preferred direction for the second peak, allowing for a delta_angle offset
    theta_pref_2 = (theta_pref + np.pi + delta_angle) % (2 * np.pi)

    # Calculate signed angular distance from the second preferred direction for the second peak
    signed_dist_2 = np.arctan2(np.sin(theta - theta_pref_2), np.cos(theta - theta_pref_2))

    # Determine the effective width based on the sign of the distance for the second peak
    effective_width_2 = np.where(signed_dist_2 >= 0, tuning_width_2_pos, tuning_width_2_neg)

    # Calculate the contribution of the second peak
    peak2_component = amplitude_2 * np.exp(-0.5 * (np.abs(signed_dist_2) / effective_width_2) **
exponent_2)

    # Sum components to get the total response
    return baseline + peak1_component + peak2_component
```

loss of model 2: 24.36

```
import numpy as np
def neuron_model_v2(theta,
                    theta_pref_1=0.0,
                    baseline=0.0,
                    amplitude_1=1.0,
                    width_left_1=1.0,
                    width_right_1=1.0,
                    theta_pref_2=np.pi,
                    amplitude_2=0.0,
                    tuning_width_2=1.0):
    """
    parent_model_1: simple_gaussian_tuning_model
    parent_model_2: fixed_bimodal_gaussian_tuning_model

    Equation: response = baseline + amplitude_1 * exp(-0.5 * (signed_diff_1 /
width_effective_1)^2) + amplitude_2 * exp(-0.5 * (circ_dist_2 / tuning_width_2)^2)

    This model improves upon previous models by addressing two key limitations:
    1.  **Asymmetric Peak Shapes**: The primary peak (peak 1) now utilizes an asymmetric Gaussian
profile. This is achieved by introducing two distinct tuning width parameters: `width_left_1` for
angles 'left' (counter-clockwise) of the preferred direction `theta_pref_1`, and `width_right_1`
for angles 'right' (clockwise). This enhanced flexibility allows for a more accurate fit to the
```

```
common observed skewness and varying spread of neuronal tuning curves on either side of the peak
(e.g., noticeable in Cell 1, Cell 85, Cell 35), providing a more nuanced representation than
previous symmetric models.
    2.  **Flexible Secondary Peak Location**: Building on the bimodal improvement in the
`fixed_bimodal_gaussian_tuning_model`, this model introduces a fully independent preferred
direction parameter for the second peak, `theta_pref_2`. Unlike the previous model which fixed the
second peak's location at exactly 180 degrees opposite the first (`theta_pref + pi`),
`theta_pref_2` can be independently optimized. This is crucial for neurons where a secondary
response exists but is not precisely antipodal to the primary peak (e.g., observed in Cell 1, Cell
115, Cell 222, where the second peak is near 0/2pi but its exact angle relative to the main peak
varies).

    The combination of asymmetric shaping for the primary peak and arbitrary angular positioning
for the secondary peak provides a more powerful and adaptive framework for modeling diverse
neuronal response profiles, thereby aiming for a significantly lower overall model loss.
    """

    # Helper function for signed circular difference, essential for asymmetric tuning
    # Result is in range (-pi, pi]
    def _signed_circ_diff_rad(angle_radians, preferred_angle_radians):
        delta = angle_radians - preferred_angle_radians
        return np.arctan2(np.sin(delta), np.cos(delta))

    # Helper function for absolute circular distance
    # Result is in range [0, pi]
    def _circ_dist_rad(angle_radians, preferred_angle_radians):
        return np.abs(np.arctan2(np.sin(angle_radians - preferred_angle_radians),
np.cos(angle_radians - preferred_angle_radians)))

    # --- Peak 1: Asymmetric Gaussian component ---
    signed_diff_1 = _signed_circ_diff_rad(theta, theta_pref_1)
    # Select width based on the sign of the circular difference (left vs. right of peak)
    width_1_effective = np.where(signed_diff_1 < 0, width_left_1, width_right_1)
    # Ensure widths are positive to prevent numerical instability (e.g., division by zero)
    width_1_effective = np.maximum(width_1_effective, 1e-6)
    # Calculate the first peak's contribution
    response_peak_1 = amplitude_1 * np.exp(-0.5 * (signed_diff_1 / width_1_effective) ** 2)

    # --- Peak 2: Symmetric Gaussian component with flexible preferred angle ---
    circ_dist_2 = _circ_dist_rad(theta, theta_pref_2)
    # Ensure tuning width is positive
    tuning_width_2 = np.maximum(tuning_width_2, 1e-6)
    # Calculate the second peak's contribution
    response_peak_2 = amplitude_2 * np.exp(-0.5 * (circ_dist_2 / tuning_width_2) ** 2)
    # Total model response
    return baseline + response_peak_1 + response_peak_2
```

8. Output for winning neuron model

```
import numpy as np
def neuron_model(theta,
                    theta_pref_1=0.0,
                    baseline=0.0,
                    amplitude_1=1.0,
                    width_ccw_1=1.0,
                    width_cw_1=1.0,
                    exponent_1=2.0,
                    theta_pref_2=np.pi,
                    amplitude_2=0.0,
                    width_ccw_2=1.0,
                    width_cw_2=1.0,
                    exponent_2=2.0):
    """
    parent_model_1: VariableExponent_AsymmetricBimodalModel
    parent_model_2: IndependentLocation_AsymmetricPrimaryModel

    Equation:
    Helper Functions:
    _signed_circ_diff_rad(angle, preferred_angle) = arctan2(sin(angle - preferred_angle),
cos(angle - preferred_angle))

    Component 1:
    signed_diff_1 = _signed_circ_diff_rad(theta, theta_pref_1)
    effective_width_1 = width_ccw_1 if signed_diff_1 < 0 else width_cw_1
    component_1 = amplitude_1 * exp(-0.5 * (abs(signed_diff_1) / effective_width_1)**exponent_1)

    Component 2:
    signed_diff_2 = _signed_circ_diff_rad(theta, theta_pref_2)
    effective_width_2 = width_ccw_2 if signed_diff_2 < 0 else width_cw_2
    component_2 = amplitude_2 * exp(-0.5 * (abs(signed_diff_2) / effective_width_2)**exponent_2)

    Response:
    response = baseline + component_1 + component_2

    This model creates a comprehensive bimodal tuning curve by intelligently combining and
enhancing the most effective
    features from its parent models, targeting specific shortcomings observed in the provided
neural data.

    Improvements:
    1.  **Unified Peak Shape Flexibility and Asymmetry**: This model brings together the concept
of variable exponents
        (`exponent_1`, `exponent_2` from `parent_model_1`) and asymmetric tuning widths for both
peaks (`width_ccw_1`, `width_cw_1`, `width_ccw_2`, `width_cw_2`,
        inheriting from `parent_model_1`'s `tuning_width_pos/neg` for both peaks and
`parent_model_2`'s `width_left/right` for the primary).
        This provides unparalleled control over the "peakedness" (e.g., sharper than Gaussian for
high exponents,
        or fatter tails for low exponents to better fit broader shoulders as seen in Cell 91, 169)
and the side-to-side
        asymmetry of *both* the primary and secondary response profiles. `parent_model_2` lacked
this crucial flexibility for both peaks.

    2.  **Fully Independent Secondary Peak Placement**: Crucially, this model retains
`parent_model_2`'s enhancement of allowing the secondary
        preferred direction (`theta_pref_2`) to be fully independent. Unlike `parent_model_1`,
which constrained the
        secondary peak to be an offset from the primary's antipodal angle, `theta_pref_2` can be
optimized freely. This is vital
        for capturing neurons with complex bimodal tuning where the secondary peak is not strictly
180 degrees from the primary
```

```
        (as observed in cells like Cell 1 and Cell 222 in previous models' descriptions, implying
non-antipodal secondary peaks).

    3.  **Enhanced Fit Accuracy:** By integrating these powerful and complementary features, this
model is designed to
        overcome systematic weaknesses of its predecessors. It provides a more robust framework to
capture
        the detailed nuances of neuronal responses, from highly skewed and sharp primary peaks to
broad and
        asymmetrically shaped secondary responses that can appear at any angle, thereby leading to
a
        significantly lower overall loss.
    """

    # Helper function for signed circular difference, essential for asymmetric tuning
    # Result is in range (-pi, pi]
    def _signed_circ_diff_rad(angle_radians, preferred_angle_radians):
        delta = angle_radians - preferred_angle_radians
        return np.arctan2(np.sin(delta), np.cos(delta))

    # --- Peak 1: Asymmetric, Variable Exponent Gaussian component ---
    signed_diff_1 = _signed_circ_diff_rad(theta, theta_pref_1)

    # Select width based on the sign of the circular difference (counter-clockwise vs. clockwise)
    width_1_effective = np.where(signed_diff_1 < 0, width_ccw_1, width_cw_1)

    # Ensure widths are positive to prevent numerical instability (e.g., division by zero or log
of non-positive)
    width_1_effective = np.maximum(width_1_effective, 1e-6)

    # Calculate the first peak's contribution with variable exponent
    # abs(signed_diff_1) is used because the exponent should operate on the magnitude of the
difference.
    peak1_component = amplitude_1 * np.exp(-0.5 * (np.abs(signed_diff_1) / width_1_effective) **
exponent_1)

    # --- Peak 2: Asymmetric, Variable Exponent Gaussian component with independent preferred
angle ---
    signed_diff_2 = _signed_circ_diff_rad(theta, theta_pref_2)

    # Select width based on the sign of the circular difference for the second peak
    width_2_effective = np.where(signed_diff_2 < 0, width_ccw_2, width_cw_2)

    # Ensure widths are positive
    width_2_effective = np.maximum(width_2_effective, 1e-6)

    # Calculate the second peak's contribution with variable exponent
    peak2_component = amplitude_2 * np.exp(-0.5 * (np.abs(signed_diff_2) / width_2_effective) **
exponent_2)

    # Total model response
    return baseline + peak1_component + peak2_component
```

9. Parameter Estimator Prompt for Winning Model

```
You are an AI scientist. Your task is to create a simple parameter estimator function,
parameter_estimator_v3, to estimate the free parameters of the latest neuron model,
neuron_model_v3.

The parameter should be estimated directly, using statistical principles and knowledge of what the
parameters represent biologically.

*Analyze* the progression of the parameter estimators, *generalize* the improvements, and *create*
a new parameter estimator that is better than *all* previous estimators.

**Code Generation Guidelines:**
* Import any packages you use.
* Do not include any text other than the code.
* The only arguments to the function should be the stimuli and the spike count.
* Your response **must** be less than {max_lines}  lines (including imports). If it is longer, it
will be immediately rejected.
* Do not attempt to fit the parameters using complex fitting functions like `curve_fit`,
`least_squares` or `minimize`. This function should be a simple starting point for the parameter
estimation.
```

loss of model 1: 24.51

```
import numpy as np

def neuron_model_v1(theta, theta_pref=0.0, baseline=0.0, amplitude_1=1.0, amplitude_2=0.0,
                    tuning_width_1_pos=1.0, tuning_width_1_neg=1.0,
                    tuning_width_2_pos=1.0, tuning_width_2_neg=1.0,
                    delta_angle=0.0, exponent_1=2.0, exponent_2=2.0):
    """
    parent_model_1: Gaussian Tuning Curve model with flexible secondary peak and exponents
    parent_model_2: Bimodal Skew-Gaussian Tuning Curve model with offset peaks

    This model further refines the tuning curve representation by incorporating variable exponents
and
    builds upon the 'Bimodal Skew-Gaussian Tuning Curve model' (`parent_model_2`).
    The goal is to more precisely capture diverse peak shapes while maintaining flexibility in
    secondary peak placement and handling asymmetries.

    Equation:
    theta_pref_2 = (theta_pref + pi + delta_angle) % (2 * pi)
    signed_dist_1 = arctan2(sin(theta - theta_pref), cos(theta - theta_pref))
    signed_dist_2 = arctan2(sin(theta - theta_pref_2), cos(theta - theta_pref_2))

    effective_width_1 = tuning_width_1_pos if signed_dist_1 >= 0 else tuning_width_1_neg
    effective_width_2 = tuning_width_2_pos if signed_dist_2 >= 0 else tuning_width_2_neg

    response = baseline + \
               amplitude_1 * exp(-0.5 * (abs(signed_dist_1) / effective_width_1)**exponent_1) + \
               amplitude_2 * exp(-0.5 * (abs(signed_dist_2) / effective_width_2)**exponent_2)

    Improvements:
    1. **Variable Exponents:** This model incorporates free parameters `exponent_1` and
`exponent_2` that govern
       the "peakedness" of the tuning curves associated with the primary and secondary peaks,
respectively. This allows us to address
       systematic deviations observed where the peak shape doesn't conform to a perfect Gaussian
shape. In contrast to both previous models (`parent_model_1`, `parent_model_2`), where the
exponent was implicitly fixed to 2, making a Gaussian shape that can result in inflexible peak
shape.
```

```
    2. **Asymmetric Tuning Widths**: This builds on the Bimodal Skew-Gaussian Tuning Curve model
with offset peaks and retains separate widths for either side of each peak: (`tuning_width_1_pos`,
`tuning_width_1_neg`) and  (`tuning_width_2_pos`, `tuning_width_2_neg`).
       This provides a substantial improvement by separately controling both sides of each peak
and providing improved fidelity by modeling each as a different process (and a corresponding set
of free parameters).
    """

    # Calculate signed angular distance from preferred direction for the first peak
    signed_dist_1 = np.arctan2(np.sin(theta - theta_pref), np.cos(theta - theta_pref))

    # Determine the effective width based on the sign of the distance for the first peak
    effective_width_1 = np.where(signed_dist_1 >= 0, tuning_width_1_pos, tuning_width_1_neg)

    # Calculate the contribution of the first peak
    peak1_component = amplitude_1 * np.exp(-0.5 * (np.abs(signed_dist_1) / effective_width_1) **
exponent_1)

    # Calculate the preferred direction for the second peak, allowing for a delta_angle offset
    theta_pref_2 = (theta_pref + np.pi + delta_angle) % (2 * np.pi)

    # Calculate signed angular distance from the second preferred direction for the second peak
    signed_dist_2 = np.arctan2(np.sin(theta - theta_pref_2), np.cos(theta - theta_pref_2))

    # Determine the effective width based on the sign of the distance for the second peak
    effective_width_2 = np.where(signed_dist_2 >= 0, tuning_width_2_pos, tuning_width_2_neg)

    # Calculate the contribution of the second peak
    peak2_component = amplitude_2 * np.exp(-0.5 * (np.abs(signed_dist_2) / effective_width_2) **
exponent_2)

    # Sum components to get the total response
    return baseline + peak1_component + peak2_component

def _estimate_peak_shape(tuning_curve, peak_idx, baseline, amplitude, n_bins):
    angle_step = 2 * np.pi / n_bins
    SQRT_2_LOG_2 = np.sqrt(2 * np.log(2))
    default_width = 0.5
    default_exponent = 2.0

    if amplitude < 0.1:
        return default_width, default_width, default_exponent

    target_val = baseline + amplitude / 2.0

    pos_half_width_bins = next((k for k in range(1, n_bins // 4 + 1) if tuning_curve[(peak_idx +
k) % n_bins] < target_val), n_bins // 4)
    neg_half_width_bins = next((k for k in range(1, n_bins // 4 + 1) if tuning_curve[(peak_idx - k
+ n_bins) % n_bins] < target_val), n_bins // 4)

    tw_pos = pos_half_width_bins * angle_step / SQRT_2_LOG_2
    tw_neg = neg_half_width_bins * angle_step / SQRT_2_LOG_2

    fqm_level = baseline + amplitude/4
    pos_qtr_width_bins = next((k for k in range(1, n_bins // 4 + 1) if tuning_curve[(peak_idx + k)
% n_bins] < fqm_level), n_bins//4)
    neg_qtr_width_bins = next((k for k in range(1, n_bins // 4 + 1) if tuning_curve[(peak_idx - k
+ n_bins) % n_bins] < fqm_level), n_bins//4)

    tw_pos_qtr = pos_qtr_width_bins * angle_step/ SQRT_2_LOG_2
    tw_neg_qtr = neg_qtr_width_bins * angle_step/ SQRT_2_LOG_2


    if (tw_pos <= 0) or (tw_pos_qtr <= 0):
       exponent=default_exponent
    else:
```

```
        exponent = np.log(2) / np.log(tw_pos_qtr / tw_pos)

    return tw_pos, tw_neg, exponent

def parameter_estimator_v1(stimuli, spike_counts):
    n_bins = 120
    kernel_sigma = 3.0
    min_peak_amplitude = 0.3
    default_width = 0.5
    min_width = 0.05
    default_exponent = 2.0
    min_exponent = 0.5
    max_exponent = 5.0

    bin_idx = ((stimuli * n_bins) / (2 * np.pi)).astype(np.int32)
    bin_idx = np.clip(bin_idx, 0, n_bins - 1)

    sums = np.bincount(bin_idx, weights=spike_counts, minlength=n_bins)
    counts = np.bincount(bin_idx, minlength=n_bins)

    x_k = np.arange(-int(3 * kernel_sigma), int(3 * kernel_sigma) + 1)
    k = np.exp(-0.5 * (x_k / kernel_sigma) ** 2)
    k = k / np.sum(k)

    pad = len(k) // 2
    sums_padded = np.pad(sums, (pad, pad), mode='wrap')
    counts_padded = np.pad(counts, (pad, pad), mode='wrap')

    num_conv = np.convolve(sums_padded, k, mode='valid')
    den_conv = np.convolve(counts_padded, k, mode='valid')
    tuning_curve = num_conv / (den_conv + 1e-8)

    baseline = np.maximum(0.0, np.min(tuning_curve))
    angle_step = 2 * np.pi / n_bins

    pref_idx_1 = np.argmax(tuning_curve)
    theta_pref_1 = pref_idx_1 * angle_step
    amplitude_1 = tuning_curve[pref_idx_1] - baseline
    amplitude_1 = np.maximum(0.0, amplitude_1)

    tw1_pos, tw1_neg, exponent_1 = _estimate_peak_shape(tuning_curve, pref_idx_1, baseline,
amplitude_1, n_bins)
    tw1_pos = np.clip(tw1_pos, min_width, np.pi)
    tw1_neg = np.clip(tw1_neg, min_width, np.pi)
    exponent_1 = np.clip(exponent_1, min_exponent, max_exponent)

    search_window_half_bins = n_bins // 8
    expected_center_idx_2 = (pref_idx_1 + n_bins // 2) % n_bins
    peak2_candidate_indices = [(expected_center_idx_2 + i + n_bins) % n_bins for i in range(-
search_window_half_bins, search_window_half_bins + 1)]
    pref_idx_2 = peak2_candidate_indices[np.argmax(tuning_curve[peak2_candidate_indices])]
    theta_pref_2 = pref_idx_2 * angle_step

    amplitude_2 = tuning_curve[pref_idx_2] - baseline
    amplitude_2 = np.maximum(0.0, amplitude_2)

    tw2_pos, tw2_neg, exponent_2 = _estimate_peak_shape(tuning_curve, pref_idx_2, baseline,
amplitude_2, n_bins)
    tw2_pos = np.clip(tw2_pos, min_width, np.pi)
    tw2_neg = np.clip(tw2_neg, min_width, np.pi)
    exponent_2 = np.clip(exponent_2, min_exponent, max_exponent)

    delta_angle = np.arctan2(np.sin(theta_pref_2 - (theta_pref_1 + np.pi)), np.cos(theta_pref_2 -
(theta_pref_1 + np.pi)))
```

```
    return np.array([theta_pref_1, baseline, amplitude_1, amplitude_2,
                     tw1_pos, tw1_neg, tw2_pos, tw2_neg, delta_angle, exponent_1, exponent_2])
```

loss of model 2: 24.36

```
import numpy as np
def neuron_model_v2(theta,
                    theta_pref_1=0.0,
                    baseline=0.0,
                    amplitude_1=1.0,
                    width_left_1=1.0,
                    width_right_1=1.0,
                    theta_pref_2=np.pi,
                    amplitude_2=0.0,
                    tuning_width_2=1.0):
    """
    parent_model_1: simple_gaussian_tuning_model
    parent_model_2: fixed_bimodal_gaussian_tuning_model

    Equation: response = baseline + amplitude_1 * exp(-0.5 * (signed_diff_1 /
width_effective_1)^2) + amplitude_2 * exp(-0.5 * (circ_dist_2 / tuning_width_2)^2)

    This model improves upon previous models by addressing two key limitations:
    1.  **Asymmetric Peak Shapes**: The primary peak (peak 1) now utilizes an asymmetric Gaussian
profile. This is achieved by introducing two distinct tuning width parameters: `width_left_1` for
angles 'left' (counter-clockwise) of the preferred direction `theta_pref_1`, and `width_right_1`
for angles 'right' (clockwise). This enhanced flexibility allows for a more accurate fit to the
common observed skewness and varying spread of neuronal tuning curves on either side of the peak
(e.g., noticeable in Cell 1, Cell 85, Cell 35), providing a more nuanced representation than
previous symmetric models.
    2.  **Flexible Secondary Peak Location**: Building on the bimodal improvement in the
`fixed_bimodal_gaussian_tuning_model`, this model introduces a fully independent preferred
direction parameter for the second peak, `theta_pref_2`. Unlike the previous model which fixed the
second peak's location at exactly 180 degrees opposite the first (`theta_pref + pi`),
`theta_pref_2` can be independently optimized. This is crucial for neurons where a secondary
response exists but is not precisely antipodal to the primary peak (e.g., observed in Cell 1, Cell
115, Cell 222, where the second peak is near 0/2pi but its exact angle relative to the main peak
varies).

    The combination of asymmetric shaping for the primary peak and arbitrary angular positioning
for the secondary peak provides a more powerful and adaptive framework for modeling diverse
neuronal response profiles, thereby aiming for a significantly lower overall model loss.
    """

    # Helper function for signed circular difference, essential for asymmetric tuning
    # Result is in range (-pi, pi]
    def _signed_circ_diff_rad(angle_radians, preferred_angle_radians):
        delta = angle_radians - preferred_angle_radians
        return np.arctan2(np.sin(delta), np.cos(delta))

    # Helper function for absolute circular distance
    # Result is in range [0, pi]
    def _circ_dist_rad(angle_radians, preferred_angle_radians):
        return np.abs(np.arctan2(np.sin(angle_radians - preferred_angle_radians),
np.cos(angle_radians - preferred_angle_radians)))

    # --- Peak 1: Asymmetric Gaussian component ---
    signed_diff_1 = _signed_circ_diff_rad(theta, theta_pref_1)

    # Select width based on the sign of the circular difference (left vs. right of peak)
    width_1_effective = np.where(signed_diff_1 < 0, width_left_1, width_right_1)

    # Ensure widths are positive to prevent numerical instability (e.g., division by zero)
    width_1_effective = np.maximum(width_1_effective, 1e-6)
```

```
    # Calculate the first peak's contribution
    response_peak_1 = amplitude_1 * np.exp(-0.5 * (signed_diff_1 / width_1_effective) ** 2)

    # --- Peak 2: Symmetric Gaussian component with flexible preferred angle ---
    circ_dist_2 = _circ_dist_rad(theta, theta_pref_2)

    # Ensure tuning width is positive
    tuning_width_2 = np.maximum(tuning_width_2, 1e-6)

    # Calculate the second peak's contribution
    response_peak_2 = amplitude_2 * np.exp(-0.5 * (circ_dist_2 / tuning_width_2) ** 2)

    # Total model response
    return baseline + response_peak_1 + response_peak_2


def parameter_estimator_v2(theta, spike_counts):
    """
    Estimates the parameters of the neuron_model_v3 which includes an asymmetric primary
    Gaussian peak and a symmetric, independently positioned secondary Gaussian peak.

    Generalizations and Improvements:
    1. Robust Tuning Curve Estimation: Continues to use binning combined with Gaussian
       smoothing of both spike sums and trial counts, a refinement from v2 for noise robustness.
    2. Dynamic Peak Identification: Instead of relying on fixed antipodal locations,
       it identifies the two highest local maxima in the smoothed tuning curve to estimate
       theta_pref_1 and theta_pref_2, making the second peak's position truly flexible.
       A heuristic checks for significance and separation for the secondary peak.
    3. Asymmetric Width Estimation: Introduces a method to separately estimate
       width_left_1 and width_right_1 for the primary peak by finding the half-maximum
       points independently on the left and right sides of the peak. This addresses
       the asymmetric shaping introduced in neuron_model_v3.
    4. Flexible Secondary Peak Width: If a significant secondary peak is found, its
       tuning_width_2 is estimated based on its own half-maximum points. Otherwise,
       it defaults to an average of the primary peak's widths, providing a reasonable
       initial guess in the absence of clear evidence.

    Args:
        theta (np.ndarray): Angles in radians.
        spike_counts (np.ndarray): Spike counts corresponding to each angle.

    Returns:
        np.ndarray: Estimated parameters [theta_pref_1, baseline, amplitude_1,
                                          width_left_1, width_right_1, theta_pref_2,
                                          amplitude_2, tuning_width_2].
    """
    n_bins = 150  # Increased bins for finer resolution, combined with smoothing

    # Map angles to bins, ensuring circularity
    bin_idx = ((theta * n_bins) / (2 * np.pi)).astype(np.int32)
    bin_idx = np.clip(bin_idx, 0, n_bins - 1)

    # Bin the spike counts and trial counts
    sums = np.bincount(bin_idx, weights=spike_counts, minlength=n_bins)
    counts = np.bincount(bin_idx, minlength=n_bins)

    def _gaussian_kernel(sigma: float) -> np.ndarray:
        """Helper to create a Gaussian kernel for smoothing."""
        radius = int(3 * sigma)
        x = np.arange(-radius, radius + 1)
        k = np.exp(-0.5 * (x / sigma) ** 2)
        return k / (np.sum(k) + 1e-8)  # Add epsilon for numerical stability

    # Apply Gaussian smoothing to sums and counts
    kernel_sigma = 2.0  # Smoothing kernel width
    kernel = _gaussian_kernel(kernel_sigma)
```

```
pad = len(kernel) // 2

sums_padded = np.pad(sums, (pad, pad), mode='wrap')
counts_padded = np.pad(counts, (pad, pad), mode='wrap')

num_conv = np.convolve(sums_padded, kernel, mode='valid')
den_conv = np.convolve(counts_padded, kernel, mode='valid')

# Calculate smoothed tuning curve, handling divisions by zero
tuning_curve = num_conv / (den_conv + 1e-8)

# 1. Estimate Baseline
baseline = np.min(tuning_curve)

# 2. Identify Local Maxima (Peaks)
local_maxima = []
for i in range(n_bins):
    prev_val = tuning_curve[(i - 1 + n_bins) % n_bins]
    next_val = tuning_curve[(i + 1) % n_bins]
    if tuning_curve[i] >= prev_val and tuning_curve[i] >= next_val:
        local_maxima.append((tuning_curve[i], i))

local_maxima.sort(key=lambda x: x[0], reverse=True)

# Default values, consistent with neuron_model_v3 for robustness
theta_pref_1, amplitude_1 = 0.0, 0.0
width_left_1, width_right_1 = 1.0, 1.0
theta_pref_2, amplitude_2, tuning_width_2 = np.pi, 0.0, 1.0
min_model_width = 1e-6 # Minimum width for stability, enforced by neuron_model_v3

if local_maxima:
    # Primary Peak (Peak 1)
    peak_1_val, peak_1_idx = local_maxima[0]
    amplitude_1 = peak_1_val - baseline
    theta_pref_1 = peak_1_idx * (2 * np.pi / n_bins)

    if amplitude_1 > 1e-3: # Only calculate widths if amplitude is significant
        sqrt_2ln2 = np.sqrt(2.0 * np.log(2.0))
        half_max_level_1 = baseline + amplitude_1 / 2.0

        # Search left for half-max
        dist_bins_left = 0
        for i in range(1, n_bins // 2 + 1):
            current_idx = (peak_1_idx - i + n_bins) % n_bins
            if tuning_curve[current_idx] <= half_max_level_1:
                dist_bins_left = i
                break
        width_left_1 = (dist_bins_left * (2 * np.pi / n_bins)) / sqrt_2ln2

        # Search right for half-max
        dist_bins_right = 0
        for i in range(1, n_bins // 2 + 1):
            current_idx = (peak_1_idx + i) % n_bins
            if tuning_curve[current_idx] <= half_max_level_1:
                dist_bins_right = i
                break
        width_right_1 = (dist_bins_right * (2 * np.pi / n_bins)) / sqrt_2ln2

        # Apply minimum width constraint
        width_left_1 = np.maximum(width_left_1, min_model_width)
        width_right_1 = np.maximum(width_right_1, min_model_width)
    else: # Amplitude not significant, set widths to default
        width_left_1, width_right_1 = 1.0, 1.0

    # Secondary Peak (Peak 2)
    if len(local_maxima) > 1:
```

```
            peak_2_val, peak_2_idx = local_maxima[1]
            amplitude_2_candidate = peak_2_val - baseline
            theta_pref_2_candidate = peak_2_idx * (2 * np.pi / n_bins)

            # Check for significance and reasonable separation from primary peak
            peak_sep_rad = np.abs(np.arctan2(np.sin(theta_pref_1 - theta_pref_2_candidate),
                                             np.cos(theta_pref_1 - theta_pref_2_candidate)))

            if amplitude_2_candidate > (amplitude_1 * 0.1) and peak_sep_rad > (np.pi / 8): # Min
10% primary amp & > 22.5 deg sep
                amplitude_2 = amplitude_2_candidate
                theta_pref_2 = theta_pref_2_candidate

                # Estimate tuning_width_2 (symmetric for simplicity)
                half_max_level_2 = baseline + amplitude_2 / 2.0
                dist_bins_2_half = 0
                for i in range(1, n_bins // 2 + 1):
                    l_idx = (peak_2_idx - i + n_bins) % n_bins
                    r_idx = (peak_2_idx + i) % n_bins
                    if tuning_curve[l_idx] <= half_max_level_2 or tuning_curve[r_idx] <=
half_max_level_2:
                        dist_bins_2_half = i # Taking the broader of left/right for FWHM (approx)
                        break
                if dist_bins_2_half == 0: dist_bins_2_half = n_bins // 8 # Default if no half-max
                tuning_width_2 = (dist_bins_2_half * (2 * np.pi / n_bins)) / sqrt_2ln2
                tuning_width_2 = np.maximum(tuning_width_2, min_model_width)
            else: # Secondary peak not significant or too close
                amplitude_2 = 0.0
                theta_pref_2 = (theta_pref_1 + np.pi) % (2 * np.pi) # Default to antipodal
                tuning_width_2 = np.mean([width_left_1, width_right_1]) # Default average of
primary
        else: # Only one significant peak found
            amplitude_2 = 0.0
            theta_pref_2 = (theta_pref_1 + np.pi) % (2 * np.pi) # Default to antipodal
            tuning_width_2 = np.mean([width_left_1, width_right_1]) # Default average of primary

    return np.array([theta_pref_1, baseline, amplitude_1,
                     width_left_1, width_right_1, theta_pref_2,
                     amplitude_2, tuning_width_2])

import numpy as np
def neuron_model_v3(theta,
                    theta_pref_1=0.0,
                    baseline=0.0,
                    amplitude_1=1.0,
                    width_ccw_1=1.0,
                    width_cw_1=1.0,
                    exponent_1=2.0,
                    theta_pref_2=np.pi,
                    amplitude_2=0.0,
                    width_ccw_2=1.0,
                    width_cw_2=1.0,
                    exponent_2=2.0):
    """
    parent_model_1: VariableExponent_AsymmetricBimodalModel
    parent_model_2: IndependentLocation_AsymmetricPrimaryModel

    Equation:
    Helper Functions:
    _signed_circ_diff_rad(angle, preferred_angle) = arctan2(sin(angle - preferred_angle),
cos(angle - preferred_angle))

    Component 1:
    signed_diff_1 = _signed_circ_diff_rad(theta, theta_pref_1)
    effective_width_1 = width_ccw_1 if signed_diff_1 < 0 else width_cw_1
    component_1 = amplitude_1 * exp(-0.5 * (abs(signed_diff_1) / effective_width_1)**exponent_1)
```

```
    Component 2:
    signed_diff_2 = _signed_circ_diff_rad(theta, theta_pref_2)
    effective_width_2 = width_ccw_2 if signed_diff_2 < 0 else width_cw_2
    component_2 = amplitude_2 * exp(-0.5 * (abs(signed_diff_2) / effective_width_2)**exponent_2)

    Response:
    response = baseline + component_1 + component_2

    This model creates a comprehensive bimodal tuning curve by intelligently combining and
enhancing the most effective
    features from its parent models, targeting specific shortcomings observed in the provided
neural data.

    Improvements:
    1.  **Unified Peak Shape Flexibility and Asymmetry**: This model brings together the concept
of variable exponents
        (`exponent_1`, `exponent_2` from `parent_model_1`) and asymmetric tuning widths for both
peaks (`width_ccw_1`, `width_cw_1`, `width_ccw_2`, `width_cw_2`,
        inheriting from `parent_model_1`'s `tuning_width_pos/neg` for both peaks and
`parent_model_2`'s `width_left/right` for the primary).
        This provides unparalleled control over the "peakedness" (e.g., sharper than Gaussian for
high exponents,
        or fatter tails for low exponents to better fit broader shoulders as seen in Cell 91, 169)
and the side-to-side
        asymmetry of *both* the primary and secondary response profiles. `parent_model_2` lacked
this crucial flexibility for both peaks.

    2.  **Fully Independent Secondary Peak Placement**: Crucially, this model retains
`parent_model_2`'s enhancement of allowing the secondary
        preferred direction (`theta_pref_2`) to be fully independent. Unlike `parent_model_1`,
which constrained the
        secondary peak to be an offset from the primary's antipodal angle, `theta_pref_2` can be
optimized freely. This is vital
        for capturing neurons with complex bimodal tuning where the secondary peak is not strictly
180 degrees from the primary
        (as observed in cells like Cell 1 and Cell 222 in previous models' descriptions, implying
non-antipodal secondary peaks).

    3.  **Enhanced Fit Accuracy:** By integrating these powerful and complementary features, this
model is designed to
        overcome systematic weaknesses of its predecessors. It provides a more robust framework to
capture
        the detailed nuances of neuronal responses, from highly skewed and sharp primary peaks to
broad and
        asymmetrically shaped secondary responses that can appear at any angle, thereby leading to
a
        significantly lower overall loss.
    """

    # Helper function for signed circular difference, essential for asymmetric tuning
    # Result is in range (-pi, pi]
    def _signed_circ_diff_rad(angle_radians, preferred_angle_radians):
        delta = angle_radians - preferred_angle_radians
        return np.arctan2(np.sin(delta), np.cos(delta))

    # --- Peak 1: Asymmetric, Variable Exponent Gaussian component ---
    signed_diff_1 = _signed_circ_diff_rad(theta, theta_pref_1)

    # Select width based on the sign of the circular difference (counter-clockwise vs. clockwise)
    width_1_effective = np.where(signed_diff_1 < 0, width_ccw_1, width_cw_1)

    # Ensure widths are positive to prevent numerical instability (e.g., division by zero or log
of non-positive)
    width_1_effective = np.maximum(width_1_effective, 1e-6)
```

```
    # Calculate the first peak's contribution with variable exponent
    # abs(signed_diff_1) is used because the exponent should operate on the magnitude of the
difference.
    peak1_component = amplitude_1 * np.exp(-0.5 * (np.abs(signed_diff_1) / width_1_effective) **
exponent_1)

    # --- Peak 2: Asymmetric, Variable Exponent Gaussian component with independent preferred
angle ---
    signed_diff_2 = _signed_circ_diff_rad(theta, theta_pref_2)

    # Select width based on the sign of the circular difference for the second peak
    width_2_effective = np.where(signed_diff_2 < 0, width_ccw_2, width_cw_2)

    # Ensure widths are positive
    width_2_effective = np.maximum(width_2_effective, 1e-6)

    # Calculate the second peak's contribution with variable exponent
    peak2_component = amplitude_2 * np.exp(-0.5 * (np.abs(signed_diff_2) / width_2_effective) **
exponent_2)

    # Total model response
    return baseline + peak1_component + peak2_component
```

10. Output for Parameter Estimator of Winning Model

```
def parameter_estimator(stimuli, spike_counts):
    """
    Estimates parameters for the neuron_model_v3 based on stimulus angles and observed spike
counts.
    This estimator employs statistical principles to identify and characterize tuning curve peaks,
    including baseline, amplitude, asymmetric widths, preferred directions, and exponents for
    two potential peaks.

    Parameters are estimated directly from a smoothed firing rate curve, aiming for a simple
    yet robust initial estimation suitable for generalized Gaussian-like tuning profiles.

    Parameters:
    stimuli (np.ndarray): An array of stimulus angles in radians (0 to 2*pi).
    spike_counts (np.ndarray): An array of spike counts corresponding to each stimulus.

    Returns:
    np.ndarray: An array containing the estimated parameters in the following order:
                [theta_pref_1, baseline, amplitude_1, width_ccw_1, width_cw_1,
                 exponent_1, theta_pref_2, amplitude_2, width_ccw_2, width_cw_2, exponent_2]
    """
    # --- Configuration Constants ---
    n_bins = 180  # Number of angular bins for tuning curve estimation
    kernel_sigma = 2.5 # Sigma for Gaussian smoothing kernel
    min_peak_amplitude = 0.5 # Min amplitude (spikes/stimulus) above baseline for a peak to be
considered valid
    min_model_width = 1e-6 # Minimum allowed width for numerical stability (as per model)
    default_width_value = 1.0 # Default width for non-significant peaks or non-determinable widths
    min_exponent = 0.5 # Minimum allowed exponent value
    max_exponent = 5.0 # Maximum allowed exponent value
    default_exponent_value = 2.0 # Default exponent (Gaussian)
    min_second_peak_ratio = 0.1 # Min amplitude of secondary peak relative to primary
    min_second_peak_separation = np.pi / 4 # Min angular separation between primary and secondary
peaks

    # --- 1. Binning and Smoothing ---
    # Convert stimuli to bin indices, handle wrap-around implicitly by modulo in bincount
    bin_idx = ((stimuli * n_bins) / (2 * np.pi)).astype(np.int32)
    bin_idx = np.clip(bin_idx, 0, n_bins - 1)

    sums = np.bincount(bin_idx, weights=spike_counts, minlength=n_bins)
    counts = np.bincount(bin_idx, minlength=n_bins)

    # Create Gaussian smoothing kernel
    kernel_radius = int(3 * kernel_sigma)
    x_kernel = np.arange(-kernel_radius, kernel_radius + 1)
    kernel = np.exp(-0.5 * (x_kernel / kernel_sigma) ** 2)
    kernel /= (np.sum(kernel) + 1e-8) # Normalize kernel

    # Pad arrays for circular convolution
    pad = len(kernel) // 2
    sums_padded = np.pad(sums, (pad, pad), mode='wrap')
    counts_padded = np.pad(counts, (pad, pad), mode='wrap')

    # Convolve to get smoothed sum of spikes and counts
    num_conv = np.convolve(sums_padded, kernel, mode='valid')
    den_conv = np.convolve(counts_padded, kernel, mode='valid')

    # Calculate smoothed tuning curve (avoid division by zero)
    tuning_curve = np.zeros_like(num_conv, dtype=float)
    valid_den_mask = den_conv > 1e-8
    tuning_curve[valid_den_mask] = num_conv[valid_den_mask] / den_conv[valid_den_mask]

    angle_step = 2 * np.pi / n_bins
```

```
# --- 2. Baseline Estimation ---
# Baseline cannot be negative
baseline_est = np.maximum(0.0, np.min(tuning_curve))

# --- Helper function for estimating peak shape parameters ---
def _get_peak_params_simple(peak_idx_val, peak_idx, bsl, tc, n_bns, ang_step,
                            min_w, def_w, min_exp, max_exp, def_exp, min_amp_thresh):
    amp = peak_idx_val - bsl
    if amp < min_amp_thresh:
        return amp, def_w, def_w, def_exp # Return default parameters if amplitude too small

    # Find bins where tuning curve drops below half-max
    target_half_val = bsl + amp / 2.0
    half_ccw_bins, half_cw_bins = 0, 0
    for k in range(1, n_bns // 2 + 1):
        if half_ccw_bins == 0 and tc[(peak_idx - k + n_bns) % n_bns] <= target_half_val:
            half_ccw_bins = k
        if half_cw_bins == 0 and tc[(peak_idx + k) % n_bns] <= target_half_val:
            half_cw_bins = k
        if half_ccw_bins > 0 and half_cw_bins > 0:
            break

    # Calculate width (v1-style, implicitly assuming Gaussian form for width parameter base)
    SQRT_2_LOG_2 = np.sqrt(2 * np.log(2))
    width_ccw = (half_ccw_bins * ang_step) / SQRT_2_LOG_2 if half_ccw_bins > 0 else def_w
    width_cw = (half_cw_bins * ang_step) / SQRT_2_LOG_2 if half_cw_bins > 0 else def_w

    # Clip widths to valid range
    width_ccw = np.clip(width_ccw, min_w, np.pi)
    width_cw = np.clip(width_cw, min_w, np.pi)

    # Find bins where tuning curve drops below quarter-max for exponent estimation
    target_qtr_val = bsl + amp / 4.0
    qtr_ccw_bins, qtr_cw_bins = 0, 0
    for k in range(1, n_bns // 2 + 1):
        if qtr_ccw_bins == 0 and tc[(peak_idx - k + n_bns) % n_bns] <= target_qtr_val:
            qtr_ccw_bins = k
        if qtr_cw_bins == 0 and tc[(peak_idx + k) % n_bns] <= target_qtr_val:
            qtr_cw_bins = k
        if qtr_ccw_bins > 0 and qtr_cw_bins > 0:
            break

    # Exponent estimation (log(2) / log(dist_qtr / dist_half))
    exponent_estimates = []
    if half_ccw_bins > 0 and qtr_ccw_bins > half_ccw_bins: # Ensure ratio > 1 for valid log
        exponent_estimates.append(np.log(2) / np.log(qtr_ccw_bins / half_ccw_bins))
    if half_cw_bins > 0 and qtr_cw_bins > half_cw_bins:
        exponent_estimates.append(np.log(2) / np.log(qtr_cw_bins / half_cw_bins))

    # Take mean of valid estimates, else default
    exponent = np.mean(exponent_estimates) if exponent_estimates else def_exp
    exponent = np.clip(exponent, min_exp, max_exp)

    return amp, width_ccw, width_cw, exponent

# --- 3. Peak Identification and Parameter Estimation ---
# Find all local maxima in the smoothed tuning curve
local_maxima = []
for i in range(n_bins):
    prev_val = tuning_curve[(i - 1 + n_bins) % n_bins]
    next_val = tuning_curve[(i + 1) % n_bins]
    if tuning_curve[i] >= prev_val and tuning_curve[i] >= next_val:
        local_maxima.append((tuning_curve[i], i))
local_maxima.sort(key=lambda x: x[0], reverse=True) # Sort by peak amplitude (descending)
```

```
    # Initialize all output parameters with model defaults (if no peak is found)
    theta_pref_1 = 0.0
    amplitude_1 = default_width_value # amplitude for primary peak default to 1 for non-zero
contribution if peak is small
    width_ccw_1, width_cw_1, exponent_1 = default_width_value, default_width_value,
default_exponent_value

    theta_pref_2 = np.pi # Antipodal to default theta_pref_1
    amplitude_2 = 0.0 # Default no second peak
    width_ccw_2, width_cw_2, exponent_2 = default_width_value, default_width_value,
default_exponent_value

    # Process Primary Peak
    if local_maxima:
        peak_1_val, peak_1_idx = local_maxima[0]
        # Estimate parameters for the primary (highest) peak
        amplitude_1, width_ccw_1, width_cw_1, exponent_1 = \
            _get_peak_params_simple(peak_1_val, peak_1_idx, baseline_est, tuning_curve, n_bins,
angle_step,
                                    min_model_width, default_width_value, min_exponent,
max_exponent,
                                    default_exponent_value, min_peak_amplitude)
        theta_pref_1 = peak_1_idx * angle_step

        # Process Secondary Peak (if a suitable candidate exists)
        if len(local_maxima) > 1:
            # Iterate through other local maxima to find a suitable secondary peak
            for i in range(1, len(local_maxima)):
                peak_2_val_candidate, peak_2_idx_candidate = local_maxima[i]
                current_amplitude_2_candidate = peak_2_val_candidate - baseline_est
                current_theta_pref_2_candidate = peak_2_idx_candidate * angle_step

                # Check amplitude significance relative to primary peak's estimated amplitude
                if current_amplitude_2_candidate < (amplitude_1 * min_second_peak_ratio):
                    continue # Skip if too small compared to the primary peak

                # Check angular separation from the *primary* peak
                # np.arctan2(sin(a-b), cos(a-b)) gives circular distance in [-pi, pi]
                peak_sep_rad = np.abs(np.arctan2(np.sin(theta_pref_1 -
current_theta_pref_2_candidate),
                                                 np.cos(theta_pref_1 -
current_theta_pref_2_candidate)))
                if peak_sep_rad < min_second_peak_separation:
                    continue # Skip if too close to the primary peak

                # Valid secondary peak found, estimate its parameters
                amplitude_2, width_ccw_2, width_cw_2, exponent_2 = \
                    _get_peak_params_simple(peak_2_val_candidate, peak_2_idx_candidate,
baseline_est, tuning_curve, n_bins, angle_step,
                                            min_model_width, default_width_value, min_exponent,
max_exponent,
                                            default_exponent_value, min_peak_amplitude)
                theta_pref_2 = current_theta_pref_2_candidate

                # Break after finding the first suitable secondary peak (highest amplitude among
remaining candidates)
                break

    # --- Pack and return parameters in the correct order for neuron_model_v3 ---
    return np.array([theta_pref_1, baseline_est, amplitude_1, width_ccw_1, width_cw_1,
                     exponent_1, theta_pref_2, amplitude_2, width_ccw_2, width_cw_2, exponent_2])
```