Dynamic analogue iterative computing with memristors for real-time robotic autonomy

Yibai Xue[†,1], Jiancong Li[†,1,2], Zhili Cui[†,1], Yi Li[1]✉, Yingjie Yu[1], Yuyang Fu[1], Han Bao[1], Wenbo Yue[1], Longke Yan[2], Xin Zhao[2], Jia Chen[1], Kanhao Xue[1], Fengbin Tu[2]✉, Zhongrui Wang[3], and Xiangshui Miao[1]✉

[1]*School of Integrated Circuits, Hubei Key Laboratory for Advanced Memories, Huazhong University of Science and Technology, Wuhan, China*

[2]*Department of Electronic and Computer Engineering, The Hong Kong University of Science and Technology, Hong Kong SAR, China*

[3]*School of Microelectronics, Southern University of Science and Technology, Shenzhen, China*

✉Corresponding author. Email: liyi@hust.edu.cn; fengbintu@ust.hk; miaoxs@hust.edu.cn.

[†]These authors contributed equally to this work.

Table of contents:

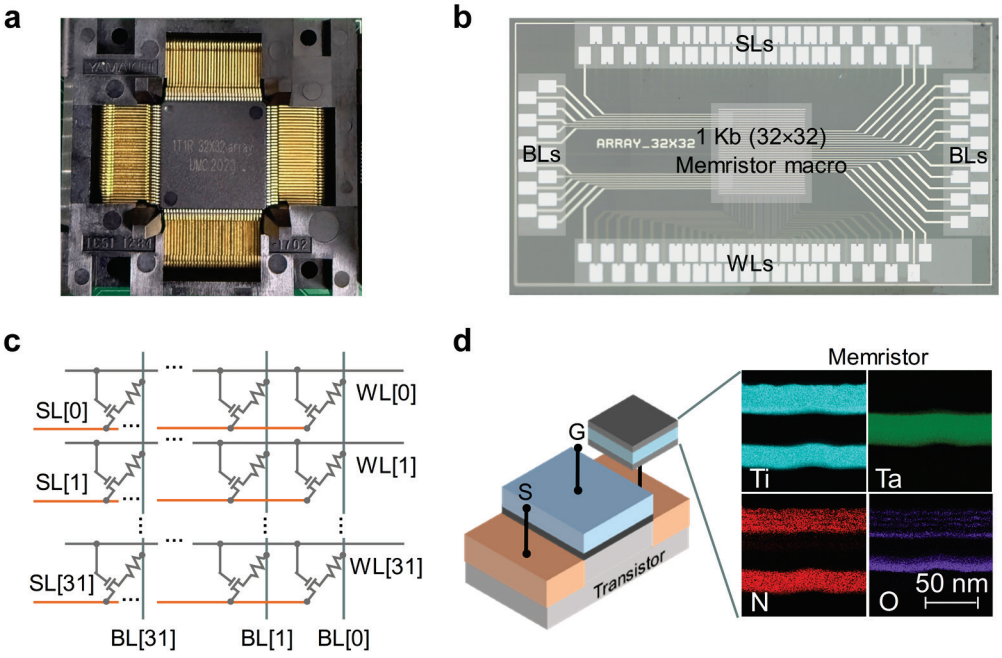
Note 1: Memristor macro and device characteristics

To realize the hardware implementation of the analogue iterative computing system for real-time robotic applications, we designed and fabricated a 1-Kb one-transistor-one-resistor (1T1R) memristor macro with a 32×32 array using a standard 180 nm CMOS process. The packaged macro for system-level integration is shown in **Supplementary Fig. 1a**, and the optical micrograph of the fully integrated chip is presented in **Supplementary Fig. 1b**. The array topology is detailed in **Supplementary Fig. 1c**. In this macro, each row shares a word line (WL) and a source line (SL) to control the access transistors, whereas each column shares a bit line (BL) connected to the top electrodes of the memristor devices. The cross-sectional schematic and the corresponding energy-dispersive X-ray spectroscopy (EDS) elemental maps in **Supplementary Fig. 1d** further confirm the vertical integration of the transition-metal-oxide memristive stack, consisting of Ti, Ta, O, and N layers, above the underlying CMOS access circuitry.

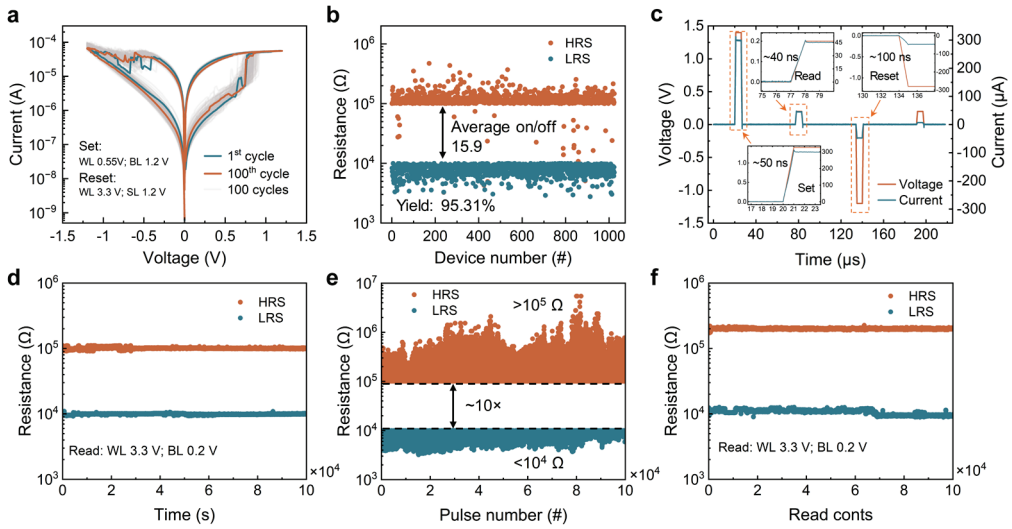
Before mapping analogue weights, we first characterized the fundamental binary switching behaviour of the 1T1R cells. As shown in **Supplementary Fig. 2a**, the devices exhibit uniform and repeatable bipolar resistive switching under direct-current voltage sweeps, with clear transitions between the high-resistance and low-resistance states. The array-level statistics in **Supplementary Fig. 2b** show a yield of 95.31% across the 1-Kb macro and an average on/off resistance ratio of 15.9. **Supplementary Fig. 2c** shows that reliable SET and RESET transitions can be achieved with pulse widths as short as 50 ns and 100 ns, respectively, together with a read latency of 40 ns. **Supplementary Fig. 2d** further shows robust data retention without degradation over 10^4 s. The devices also exhibit pulse endurance over 10^5 cycles while maintaining an approximately 10-fold memory window (**Supplementary Fig. 2e**), and read endurance over 10^5 continuous read operations (**Supplementary Fig. 2f**).

In addition to robust binary switching, ACIM requires continuous and high-precision conductance tuning for accurate matrix mapping. **Supplementary Fig. 3a** demonstrates the analogue programming behaviour of the devices, where the conductance can be gradually tuned in both potentiation and depression, without abrupt state transitions. In the 0-0.2 V read-voltage range, the devices enable linear readout of 256 distinct conductance states, as shown in **Supplementary Fig. 3b**. The retention measurements in **Supplementary Fig. 3c** show that representative 4-bit levels remain stable for over 10^4 s. Multilevel conductance measurements

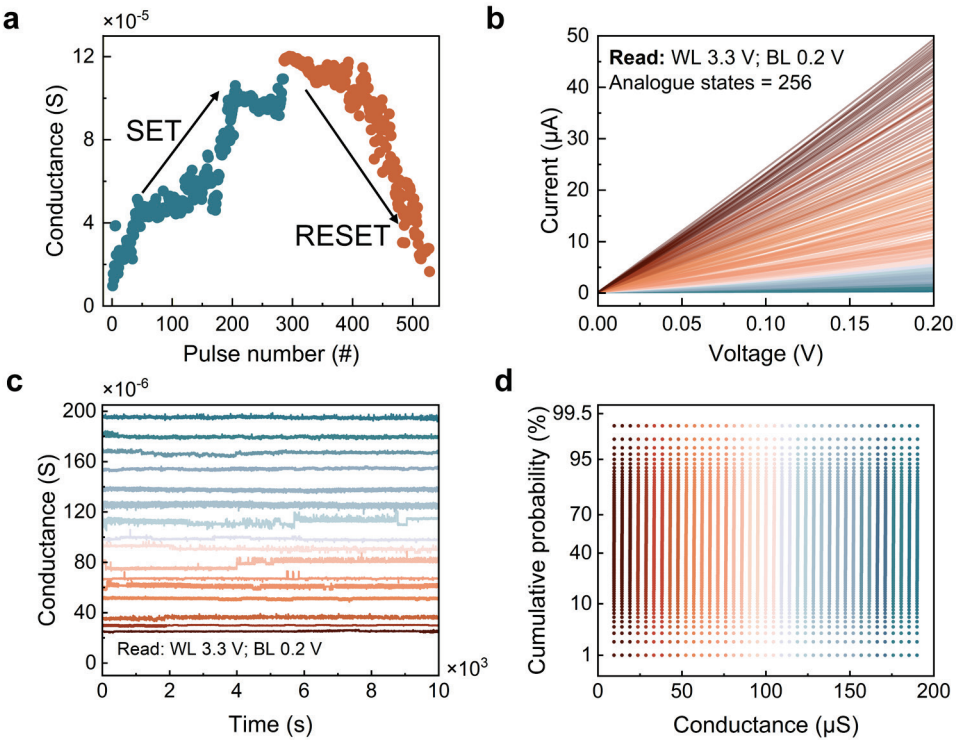
across more than 600 1T1R cells, presented in **Supplementary Fig. 3d**, indicate that the array can support more than 5-bit distinguishable conductance-state distributions.



Supplementary Fig. 1. 1-Kb (32×32) 1T1R memristor macro and array structure. a, Photograph of the packaged test chip. **b,** Optical micrograph of the 1-Kb memristor macro. **c,** Schematic illustration of the 32×32 crossbar array architecture. **d,** 3D schematic of the 1T1R cell and cross-sectional EDS elemental maps.



Supplementary Fig. 2. Binary switching characteristics of the memristor. a, Typical DC I-V switching curves. **b,** Distribution of high resistance state (HRS) and low resistance state (LRS) of all devices in the 1-Kb array, indicating high functional yield. **c,** Transient responses of set, reset, and read operations. **d,** Room-temperature retention of the HRS and LRS over 10^4 s. **e,** Pulse endurance over 10^5 cycles with an on/off ratio above 10. **f,** Read endurance over 10^5 read cycles.



116

117 **Supplementary Fig. 3. Analogue switching characteristics of the memristor. a,** Conductance
118 potentiation and depression under consecutive identical pulses. **b,** Linear readout of 256 distinct
119 analogue conductance states. **c,** Retention behaviour of the 4-bit analogue states. **d,** Cumulative
120 distributions of over 5-bit conductance states across over 600 devices.

121

Note 2: Predictive model for the AC-PoP scheme

To enable accurate AC-PoP operations, it is essential to capture the state-dependent and highly nonlinear programming dynamics of the memristor array. We therefore developed a phenomenological predictive model, whose implementation flow is illustrated in **Supplementary Fig. 4**. The model relates the relative conductance change ($\Delta G/G_i$) to the applied programming pulse amplitude (V) through a modified sigmoid function:

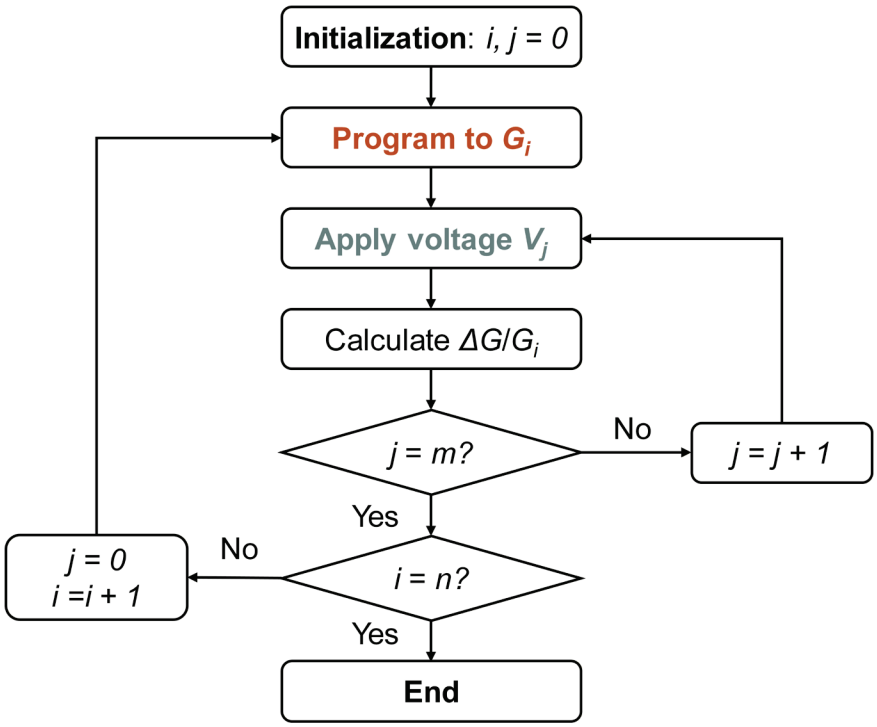
$$\frac{\Delta G}{G_i} = \frac{L(G_i)}{1+e^{-k(V-V_{th}(G_i))}} + c \tag{S1}$$

where L represents the state-dependent maximum amplitude of the conductance change, V_{th} denotes the threshold voltage required to trigger a substantial conductance state transition, k characterizes the abruptness of the switching process, and c is a baseline offset constant. Model parameters were extracted from the measured state-dependent conductance-update data in **Supplementary Fig. 5**. Specifically, we measured $\Delta G/G_i$ as a function of pulse amplitude V across a wide span of initial conductance states G_i, and the data were separated into set and reset branches. For each discrete G_i level, the corresponding sigmoid equation was fitted to obtain the state-dependent parameters $V_{th}(G_i)$ and $L(G_i)$ while k and c were held constant within each branch to avoid overfitting. The fitted parameter trends for the set and reset operations are shown in **Supplementary Figs. 6** and **7**, respectively, and the corresponding fitting parameters are summarized in **Supplementary Table 1**.

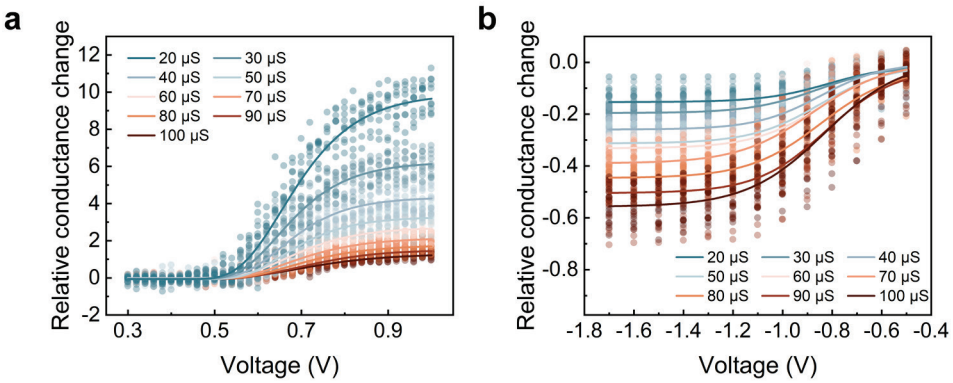
The extracted parameters of this predictive model not only mathematically capture the highly non-linear conductance change of the memristors, but also physically reflect the dynamic evolution of the conductive filament (CF) under an applied electric field[1,2]. The parameter L denotes the amplitude of the conductance change, which is intrinsically tied to the initial conductance state of the device. Specifically, a higher conductance state restricts further CF growth, making subsequent conductance enhancement increasingly difficult. This pre-existing state also dictates the varying amplitudes of conductance depression when identical voltages are applied. The parameter V_{th} represents the threshold voltage required to trigger a non-volatile transition to an adjacent conductance state. Voltages below this threshold induce negligible non-volatile changes, effectively preventing read disturbance, whereas voltages exceeding V_{th} drive a substantive switch to the next conductance state. Crucially, the dependence of L and V_{th} on the initial conductance

state macroscopically mirrors the pre-existing CF morphology. A higher G_i corresponds to a thicker, structurally robust CF. During the set process, further spatial expansion of a thick CF is fundamentally restricted by geometric confinement, naturally yielding a lower L. Furthermore, a thicker CF reduces the device's internal resistance, which exacerbates the voltage divider effect and demands a higher thermodynamic energy barrier to modulate, inherently shifting the required V_{th} to higher absolute values. The factor k indicates the abruptness of the state transition. It reflects the sensitivity of the internal CF morphological changes to the applied voltage. A larger k indicates a more rapid and abrupt switching once the applied voltage surpasses V_{th}.

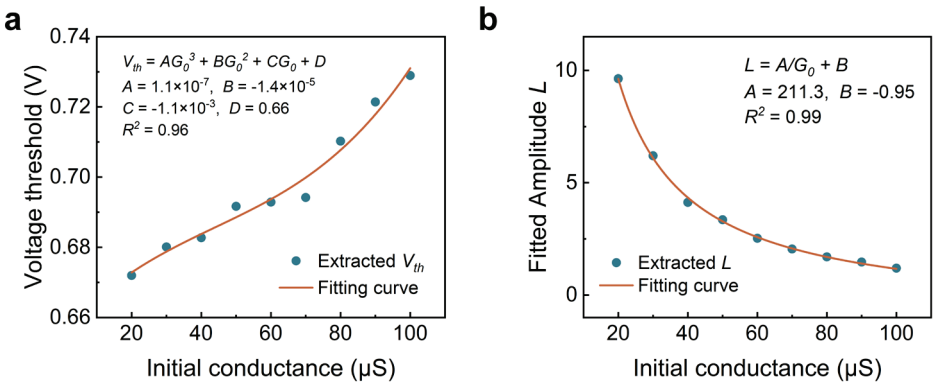
By integrating these conductance state dependent functions $V_{th}(G_i)$ and $L(G_i)$, we constructed comprehensive 3D predictive models (**Fig. 2c**) to quantitatively determine the required applied voltage for a target conductance change. Importantly, due to intrinsic memristor non-idealities, such as device-to-device (D2D) variations and inherent stochasticity, this model primarily captures the overall trends of conductance evolution and cannot guarantee perfectly precise weight updates with a single programming pulse. Therefore, we synergistically combined this predictive one-step programming with an analogue compensation technique. This integration effectively mitigates the precision loss induced by memristor non-idealities, establishing a rapid and highly precise weight update strategy.



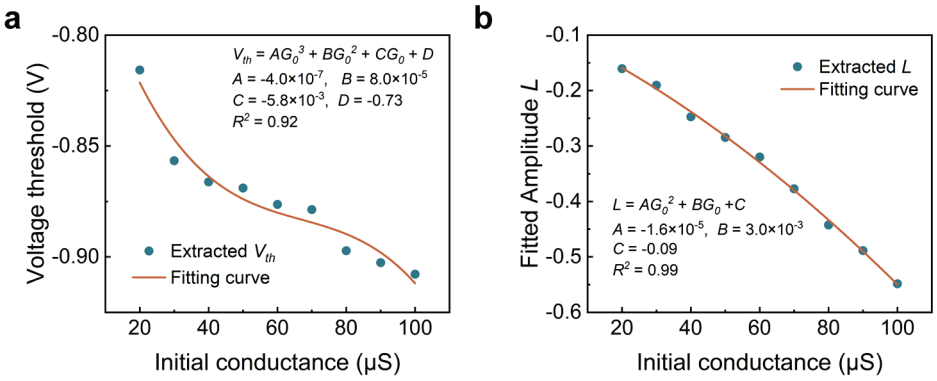
Supplementary Fig. 4. Flowchart of the $\Delta G/G_i$ measurement procedure. Each device was initialized to a predefined conductance state G_i, and voltage pulses V_j were then applied to quantify the conductance response under different G_i-V_j conditions.



Supplementary Fig. 5. Measured conductance responses used for predictive model construction. Relative conductance changes were measured as a function of programming voltage under different G_i during **(a)** set and **(b)** reset operations.



Supplementary Fig. 6. Fitted set-operation parameters as functions of the initial conductance. **a,** Threshold voltage V_{th} increases with G_i, indicating that larger set bias is required to further increase the conductance at higher-conductance states. **b,** Amplitude parameter L decreases with G_i, indicating a reduced conductance-update range as the device approaches higher conductance states.



Supplementary Fig. 7. Fitted reset-operation parameters as functions of the initial conductance. a, Threshold voltage V_{th} shifts towards larger absolute values with increasing G_i, indicating that a stronger reset bias is required to induce further conductance decrease at higher-conductance states. **b,** Amplitude parameter L decreases with G_i, indicating a reduced conductance-update range during reset at higher initial conductance states.

Set operation				
Initial *G* (μS)	***V_{th}* (V)**	***L***	***k***	***c***
20	0.67	9.63		
30	0.68	6.20		
40	0.68	4.12		
50	0.69	3.35		
60	0.69	2.52	17.13	−0.096
70	0.69	2.05		
80	0.71	1.70		
90	0.72	1.47		
100	0.73	1.20		
Reset operation				
Initial *G* (μS)	***V_{th}* (V)**	***L***	***k***	***c***
20	-0.82	-0.16		
30	-0.86	-0.19		
40	-0.87	-0.25		
50	-0.87	-0.28		
60	-0.88	-0.32	-7.23	-0.0131
70	-0.88	-0.38		
80	-0.90	-0.44		
90	-0.90	-0.49		
100	-0.91	-0.55		

Supplementary Table 1. Fitting parameters for the predictive model

Note 3: The AC-PoP implementation and comparison with WVP

(1) Demonstration of the AC-PoP Scheme

The AC-PoP update for a matrix element was implemented using the pulse sequence shown in **Supplementary Fig. 8**. For a target conductance increment $\Delta G=G(t_1)-G(t_0)$, the initial residual was defined as $r_0=\Delta G$. In the first AC-PoP cycle, the system measured the initial conductance state with a 5 μs read pulse. The predictive model then calculated the programming voltage V_0 for the main device according to r_0, and a 2 μs programming pulse was applied. Since the D2D and C2C variations cause the programmed conductance to deviate from the target value, this first programming step leaves a residual error r_1. In each subsequent AC-PoP cycle, the residual error r_i was converted into the target conductance increment for the next compensation device, and the corresponding programming V_i was calculated by the same predictive model. In the gain-weighted compensation architecture, the residual was evaluated in the effective mapped-weight domain as

$$r_{i+1} = r_i - K_i \Delta G_i \tag{S2}$$

where K_i is the scaling factor of the i-th compensation macro and ΔG_i is the conductance increment implemented by the corresponding programming pulse.

In the implemented circuit, the three macros were read through TIAs with feedback resistors of 10 kΩ, 1 kΩ, and 100 Ω, corresponding to relative readout scaling factors (K_0, K_1, K_2) of 10, 1, and 0.1, respectively. This scaling maps the conductance updates from different macros onto different effective weight ranges, allowing the macro with the larger scaling factor to provide the dominant update and the macros with smaller scaling factors to implement progressively finer residual corrections. Therefore, the sequential compensation scheme converts a single iterative and variation-sensitive programming operation into a coarse-to-fine update process, improving the effective update precision without requiring each memristor device to be programmed exactly to the target value. For the matrix-update demonstration in **Fig. 2e**, the programming-voltage distributions generated by the predictive model are shown in **Supplementary Fig. 9**, and the corresponding differential conductance matrices before and after AC-PoP are detailed in **Supplementary Fig. 10**.

(2) Illustration of the WVP method

For baseline evaluation, a conventional WVP procedure was applied to the same matrix-update target in **Fig. 2e** using identical pulse widths. The WVP workflow is presented in **Supplementary Fig. 11**. In each WVP iteration, the conductance of the selected device was first measured as g and compared with the target conductance g_{target}. If the deviation $|g_{\text{target}} - g|$ was smaller than the preset tolerance, the programming operation was regarded as successful. Otherwise, the sign of the error determined the programming direction, where a set pulse was applied when $g<g_{target}$ and a reset pulse was applied when $g>g_{target}$. The updated conductance was then read again to determine whether an additional programming pulse was required, and the write-verify process proceeded through successive iterations until the conductance fell within the target tolerance window. With a predefined 5% write-error tolerance, the effectiveness of the WVP procedure was verified on two memristor macros, as shown in **Supplementary Fig. 12**, which presents representative iterative programming traces and the accurate mapping of a "HUST" pattern.

For the matrix-update task shown in **Fig. 2e**, the matrix $A(t)$ was constrained within ±100 μS, and the target update ΔA was constrained within ±30 μS. The corresponding differential conductance matrices obtained by WVP are provided in **Supplementary Fig. 13** for comparison with AC-PoP. During the update experiment on the 8×8 array, the applied programming and read pulses were counted for both methods under the same pulse-width conditions. The programming and read pulse widths were fixed at 5 μs and 2 μs, respectively. For WVP, the programming voltage was set between 0.8 and 2.0 V for set operations and between −0.3 and −2.5 V for reset operations, while the read voltage was fixed at 0.2 V. For AC-PoP, the programming voltage was set between 0.6 and 1.8 V for set operations and between −0.3 and −2.0 V for reset operations, matching the applicable voltage range of the predictive model.

(3) Performance comparison of AC-PoP with WVP

The results of the update task were obtained from ten independent experiments. Across the three AC-PoP cycles involving six differential compensation macros, WVP required an average of 26607.1 program-read cycles, whereas AC-PoP required only 77.1 device updates on average. In this hardware demonstration, each AC-PoP update consisted of one programming operation followed by one read operation to verify the final update accuracy. In practical operation, if post-update verification is not required, the read operation after the last AC-PoP cycle can be omitted.

The total update energy was calculated from the voltage applied in each operation and the corresponding measured resistance state. For the j-th pulse, the device energy was evaluated as,

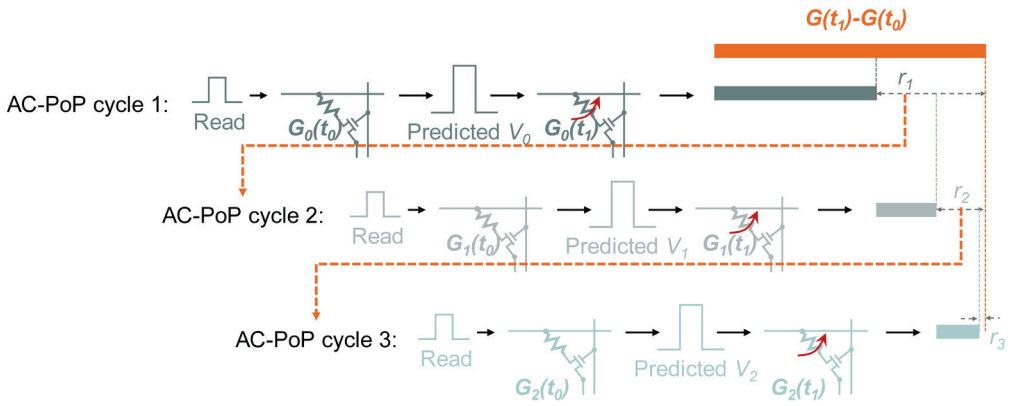
$$E_j = \int V_j(t) I_j(t) dt \tag{S3}$$

For rectangular voltage pulses, this expression can be approximated using the measured resistance state R_j as,

$$E \approx \sum_j \frac{V_j^2}{R_j} \tau_j \tag{S4}$$

where V_j, R_j, and τ_j are the applied voltage, measured resistance, and pulse width of the j-th read or programming operation, respectively. This calculation includes both programming and read pulses, and therefore represents the total energy consumed during the complete matrix-update process. Using this method, the average total energy of WVP was 6.00×10^{-6} J, whereas AC-PoP consumed only 5.93×10^{-8} J. Therefore, AC-PoP reduced the update energy consumption by approximately 101× compared with conventional WVP, as shown in **Fig. 2g**.

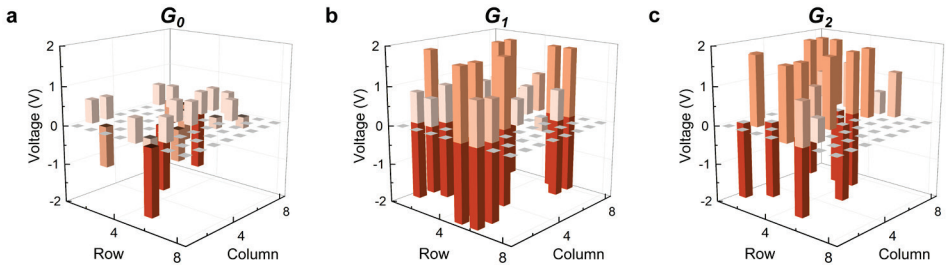
The operation latency was calculated from the accumulated physical pulse duration. Since each program-read cycle consisted of a 5 μs programming pulse and a 2 μs read pulse, the average latency of WVP was T_{WVP}=26607.1×(5+2) μs=0.18625 s. For AC-PoP, the average latency was $T_{AC\text{-}PoP}$=77.1×(5+2) μs=540 μs. Thus, AC-PoP reduced the update latency by approximately 345× compared with conventional WVP, as presented in **Fig. 2h**.



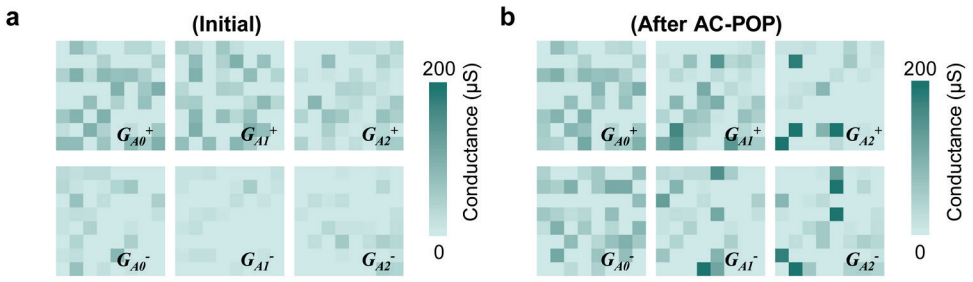
271

272 **Supplementary Fig. 8. Multi-device sequential residual compensation in AC-PoP.** The target
273 conductance increment $\Delta G=G(t_1)-G(t_0)$, is sequentially decomposed across G_0, G_1 and G_2. Starting
274 from the measured states $G_0(t_0)$, $G_1(t_0)$, and $G_2(t_0)$, single predicted programming pulses V_0, V_1,
275 and V_2 are applied sequentially. The residual error is progressively reduced to r_1, r_2, and finally r_3
276 according to $r_{i+1} = r_i - K_i \Delta G_i$.

277



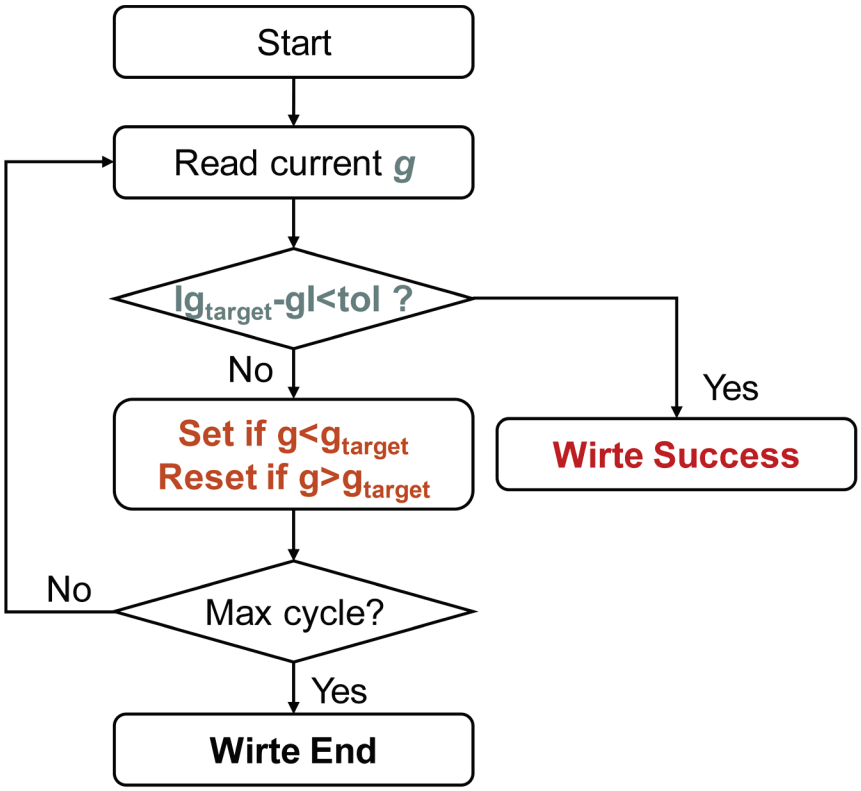
Supplementary Fig. 9. Model-predicted programming-voltage distributions used in AC-PoP. The three maps show the programming voltages generated by the predictive model and experimentally applied in the sequential compensation cycles for the matrix-update task in Fig. 2e.



283

284 **Supplementary Fig. 10. Differential conductance matrices before and after AC-PoP.** The
285 conductance states of the six differential macros are shown before and after the AC-PoP update
286 for the matrix-update task in Fig. 2e.

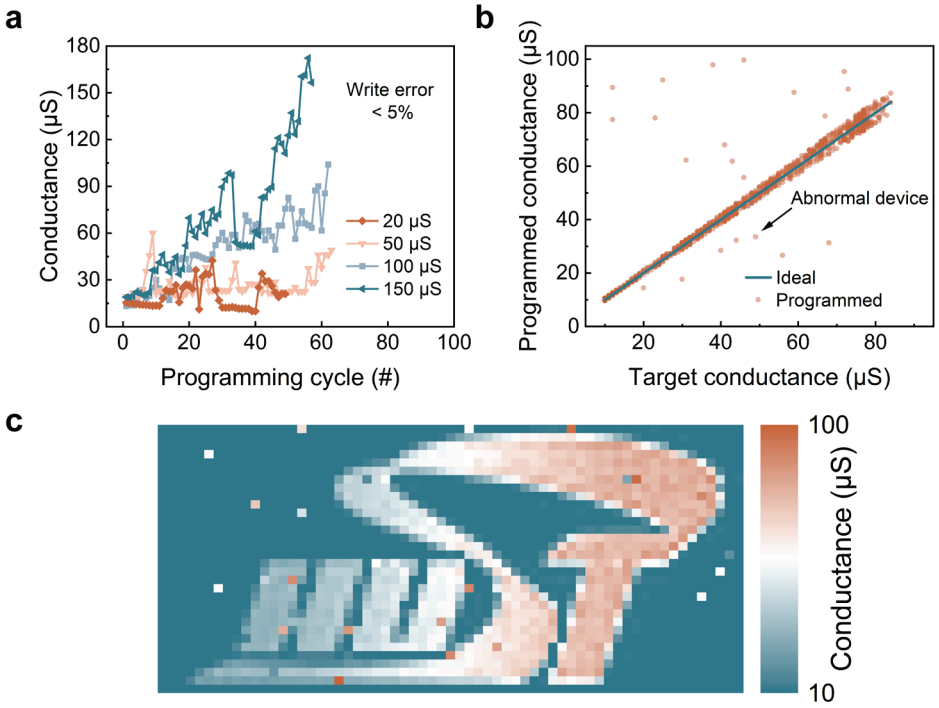
287



288

289 **Supplementary Fig. 11. Workflow of conventional write-verify programming.** The device
290 conductance g is read and compared with the target value gtarget, after which a set pulse is applied
291 for $g<g_{target}$ or a reset pulse is applied for $g>g_{target}$ until the write error falls within the preset
292 tolerance.

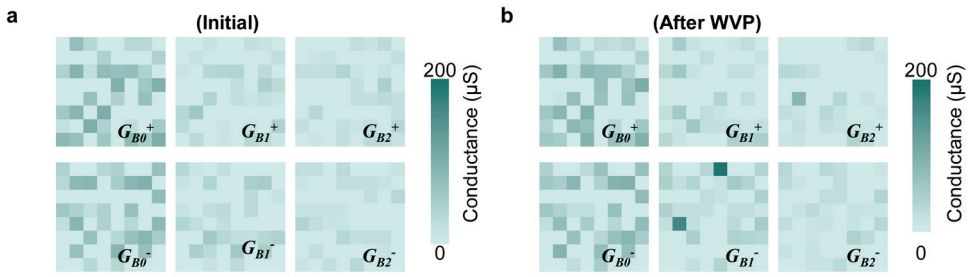
293



294

295 **Supplementary Fig. 12. Representative WVP characterization results. a,** Iterative
296 programming trajectories for representative target conductance states. **b,** Comparison between the
297 ideal and programmed conductance values measured on two 32×32 memristor macros. **c,** Accurate
298 mapping of a representative "HUST" pattern after WVP.

299



Supplementary Fig. 13. Differential conductance matrices before and after WVP. The conductance states of the six differential macros are shown before and after WVP for the same matrix-update task in Fig. 2e, providing the baseline comparison with AC-PoP.

Note 4: Hardware implementation and experimental characterization of the mAIC system

(1) Hardware implementation of the mAIC system

The hardware architecture of the mAIC system is shown in **Supplementary Fig. 14**. The mAIC system solves matrix equations by implementing the Jacobi iteration formula,

$$x_{i+1} = Bx_i + f \tag{S5}$$

where the iteration matrix B is mapped onto the differential CIM boards by AC-PoP, and the vector term f is provided as an external analogue excitation. The hardware consists of three stacked differential CIM boards, a motherboard for interconnection, feedback routing, and PCB power distribution, a 32-channel analogue adder board, and NI PXIe modules for input-vector generation and high-speed output acquisition.

The three differential CIM boards execute the signed MVM operation, with the corresponding signal-transfer path shown in **Supplementary Fig. 15**. On each board, the STM32F103ZCT6 MCU configures the transfer-gate network to select the WL, BL, and SL connections of the target memristor macros for operation. The current iteration state x_i is applied as input voltages and converted into complementary voltage pairs by AD823 operational-amplifier-based analogue inverter circuits. Each inverter is implemented as a unity-gain inverting amplifier. The non-inverting input of the AD823 is grounded, the input voltage V_{IN} is applied to the inverting input through a resistor R, and an equal feedback resistor R is connected between the output and the inverting input. The output voltage is therefore,

$$V_{\text{out}} = -\frac{R_f}{R_{\text{in}}} V_{\text{IN}} = -V_{\text{IN}}, \quad R_f = R_{\text{in}} = R \tag{S6}$$

This circuit generates the complementary voltage $-V_{IN}$ from the applied input V_{IN}, enabling the two paired memristor macros to be driven by differential input-voltage pairs for signed MVM.

The iteration matrix B is mapped by AC-PoP into three compensation conductance matrices, G_0, G_1, and G_2, which are programmed on three CIM boards. The output currents from the differential memristor macros are converted into voltages by AD823 operational-amplifier-based TIAs. In each TIA, the non-inverting input of the AD823 is grounded, and the feedback resistor R_K converts the input current into an output voltage,

$$V_{\text{TIA}} = -R_K I_{\text{out}} \tag{S7}$$

The TIA feedback resistors R_K are set to 10 kΩ, 1 kΩ, and 100 Ω for three CIM boards, corresponding to the scaling factors K_0=10, K_1=1, K_2=0.1, respectively. Therefore, the effective signed MVM implemented by the three differential CIM boards is $Bx_i=(K_0G_0+K_1G_1+K_2G_2)x_i$.

The scaled voltage outputs from the selected N compensation CIM differential macros are routed through the motherboard to the 32-channel AD823-based analogue adder. Here, N is the number of AC-PoP cycles used in the current implementation, corresponding to N=1,2, or 3. Each adder channel is implemented as an AD823 operational-amplifier summing circuit with a non-inverting bias input. The CIM output voltages are summed at the inverting node, while the vector term f is applied as a 32-channel analogue voltage V_f generated by the NI PXIe-6739 module and coupled to the non-inverting input through the R_2-R_1 resistor network. With $R_1=R_2/N$, the non-inverting input voltage is $V_+=V_f/(N+1)$, which gives a unity contribution of V_f at the adder output after the closed-loop gain of the summing amplifier. Thus, after accounting for the sign inversion of the summing stage, the adder combines the scaled CIM outputs with f to complete the Jacobi iteration in Eq. S5.

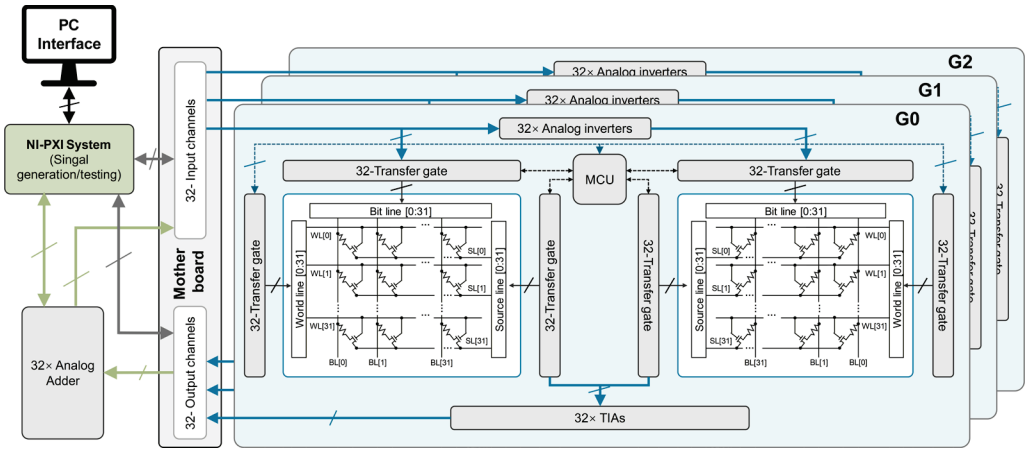
The resulting voltage vector represents x_{i+1} and is fed back through the motherboard to the CIM-board inputs for the next iteration. Meanwhile, two NI PXIe-6368 data-acquisition modules connected to the motherboard provide 32 acquisition channels for real-time voltage recording of x at a sampling rate of 2 MHz, until the analogue feedback loop reaches a stable converged voltage state.

(2) Experimental validation of the mAIC system.

For experimental validation, a predefined random 16×16 matrix-equation-solving instance was implemented on the mAIC system. The original numerical coefficient matrix was first transformed into the Jacobi iteration matrix B, and then mapped to the available memristor conductance range of 10–100 μS. The target and programmed conductance representations are shown in **Supplementary Fig. 16**. After AC-PoP programming with N=3, the programmed matrix closely matched the target matrix, with an absolute conductance mapping error below 0.1 μS for all matrix elements. The corresponding differential conductance distributions for the N=3 implementation are shown in **Supplementary Fig. 17**.

Based on the AC-PoP mapping, we further evaluated the continuous-time solving dynamics for N=1,2, and 3 (**Supplementary Fig. 18**). The measured voltage trajectories show nearly

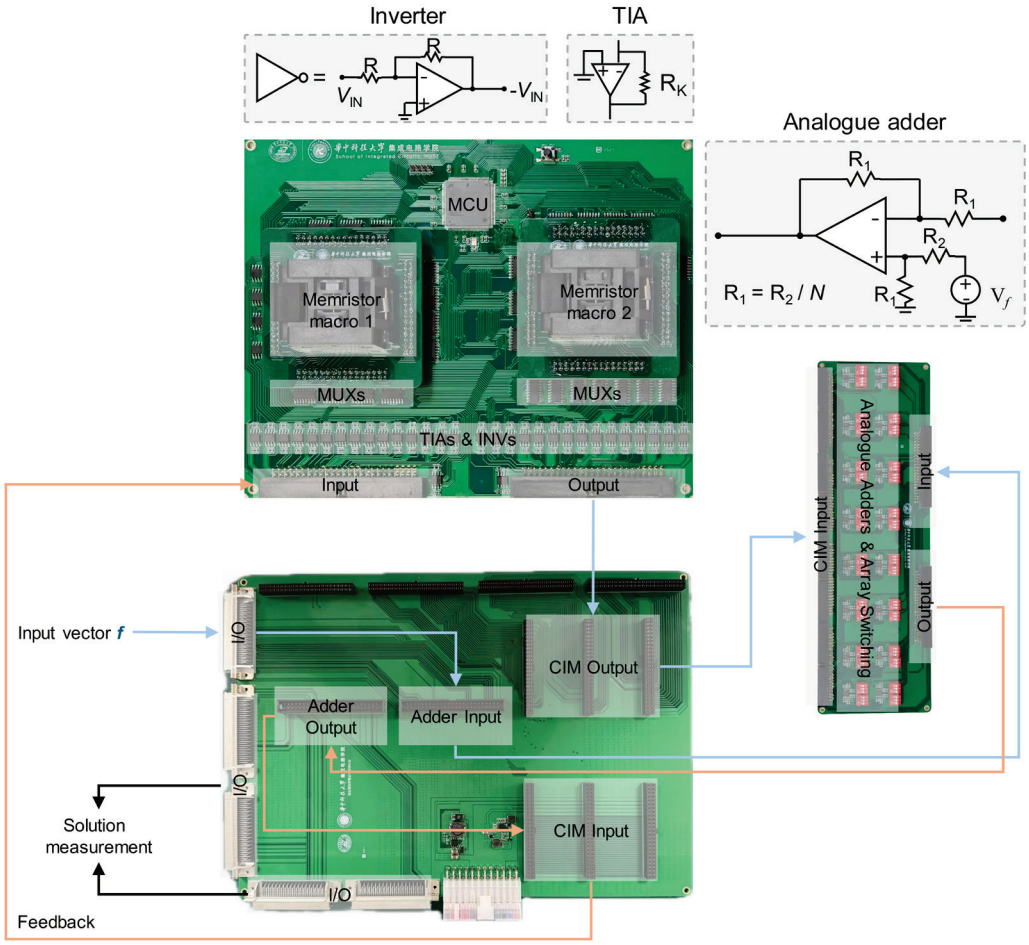
identical convergence trends and comparable settling times, indicating that AC-PoP operates as a parallel matrix-compensation mechanism and does not alter the intrinsic convergence behaviour of the analogue Jacobi feedback loop. Meanwhile, the mean absolute solution error decreases from 5.39×10^{-2} at N=1 to 1.92×10^{-3} at N=2, and further to 5.96×10^{-4} at N=3. Since the convergence behaviour remains almost unchanged from N=1 to N=3, whereas the final error decreases with improved conductance mapping accuracy, the solution error in this range is mainly dominated by matrix-mapping. These results show that AC-PoP improves the matrix-mapping accuracy for higher solution accuracy, while preserving the fully parallel, continuous-time convergence dynamics of the mAIC system.



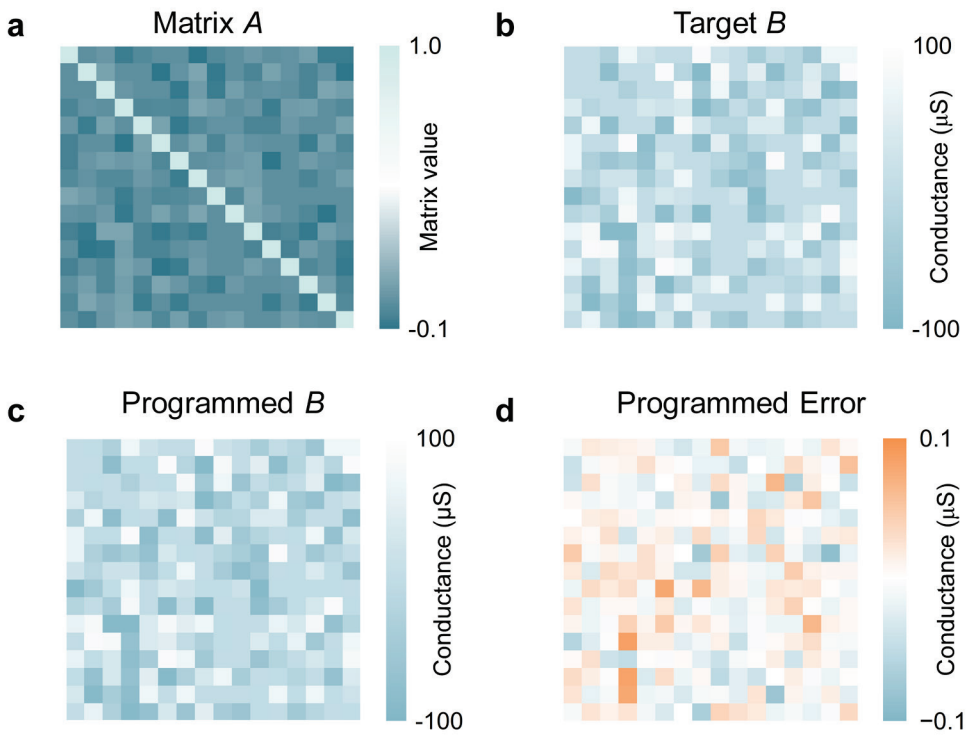
373

374 **Supplementary Fig. 14. Hardware architecture of the mAIC system.** The mAIC system
375 implements the Jacobi iteration $x_{i+1}=Bx_i+f$ using three stacked differential CIM boards, a shared
376 motherboard, a 32-channel analogue adder board, and an external NI PXIe system. The three CIM
377 boards implement the AC-PoP programming and signed MVM, while the motherboard provides
378 interconnection, feedback routing, and power distribution. The NI PXIe modules provide external
379 vector excitation and high-speed voltage readout for monitoring the solving dynamics.

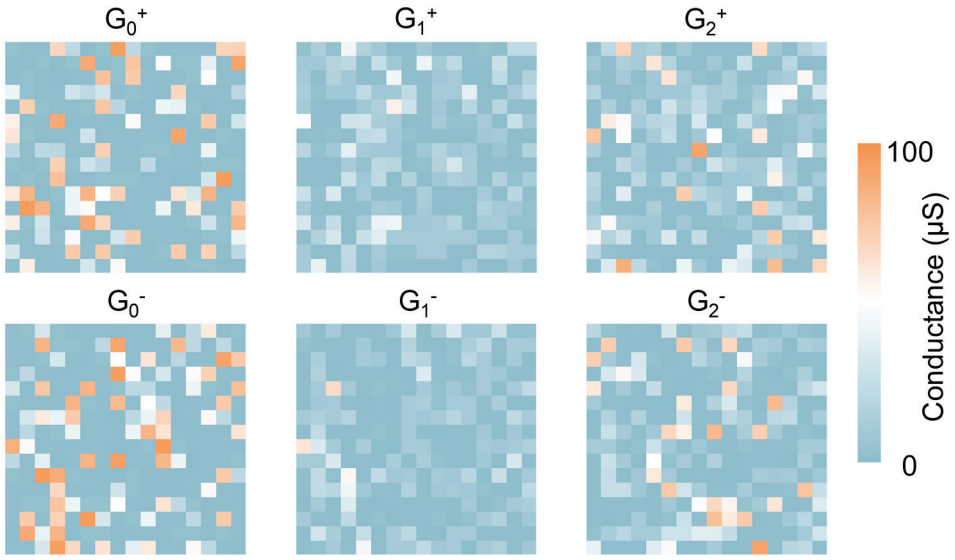
380



Supplementary Fig. 15. Signal-transfer path and analogue circuit design of the mAIC system. Each differential CIM board integrates two memristor macros, an MCU-controlled transfer-gate network, AD823-based analogue inverters, and AD823-based TIAs for signed MVM and output scaling. The motherboard provides CIM input distribution, CIM output collection, adder input/output routing, NI PXIe I/O interfacing, and feedback routing from the adder output to the CIM input. The analogue adder and array-switching board implement 32-channel AD823-based summing circuits to combine the scaled CIM outputs with the vector term *f*. Representative inverter, TIA, and adder circuits are also shown.



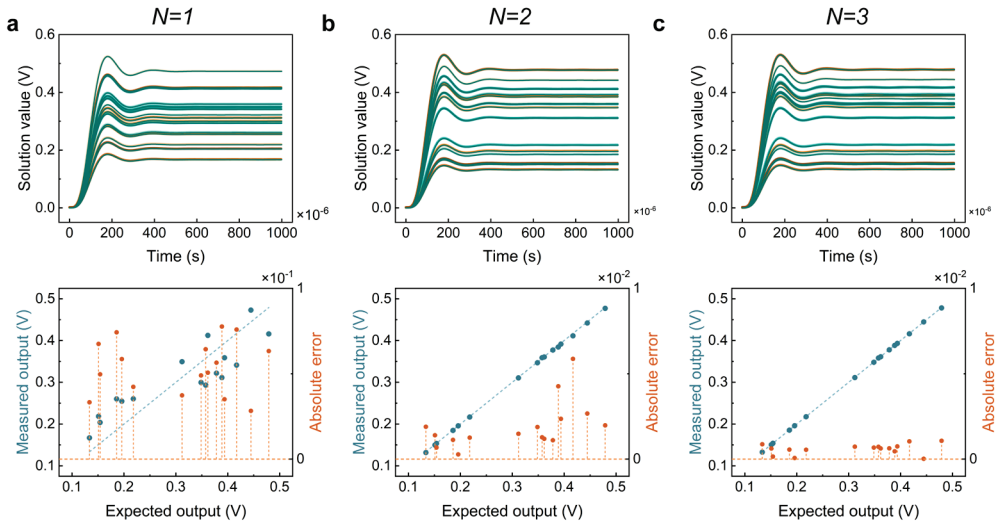
Supplementary Fig. 16. Conductance mapping of a predefined 16 × 16 random matrix equation. a, Original numerical coefficient matrix *A*. **b,** Target conductance matrix *B* obtained by mapping to the hardware conductance range. **c,** Programmed conductance after AC-PoP programming with *N*=3. **d,** Absolute programming error between the target and programmed matrices, with all errors below 0.1 μS.



398

399 **Supplementary Fig. 17. Differential conductance matrix distributions for the *N*=3 AC-PoP**
400 **implementation.** The programmed conductance matrix in Supplementary Fig. 16 is represented
401 by three groups of differential macros.

402



Supplementary Fig. 18. Hardware solution of the predefined random 16×16 matrix equation using the mAIC system. Measured voltage convergence traces and corresponding absolute errors for AC-PoP cycles with **(a)** N=1, **(b)** N=2, and **(c)** N=3. The mean absolute solution errors are 5.39×10^{-2}, 1.92×10^{-3}, and 5.96×10^{-4}, respectively. The convergence trends and settling behaviors remain nearly unchanged, while the final solution error decreases with increasing N.

Note 5: Solution accuracy analysis of the mAIC system

The solution accuracy of the mAIC system is limited by hardware non-idealities at both the memristor-array level and the peripheral analogue-circuit level. Under ideal conditions, the original matrix equation solved by the mAIC system is

$$Ax^* = b \tag{S8}$$

where A is the original coefficient matrix, b is the input vector, and x^* is the ideal solution. In a practical mAIC system, conductance-mapping errors, voltage-input errors, device variations, leakage currents, IR-drop, finite amplifier gain, peripheral analogue offsets, and readout noise perturb the implemented matrix equation, causing the converged solution to deviate from x^*. The resulting steady-state solving error depends on both the magnitude of these perturbations and the sensitivity of the original coefficient matrix A. The following perturbation analysis links these hardware-induced errors to the final solution accuracy and explains the observed dependence on the number of AC-PoP cycles N and the matrix condition number $\kappa(A)$.

These hardware non-idealities are modelled as an equivalent matrix perturbation ΔA and an equivalent input-vector perturbation Δb. Effects such as programming errors, D2D variation, IR-drop, leakage, and finite amplifier gain are mainly absorbed into ΔA, whereas input-voltage errors, peripheral offsets, and readout noise are mainly absorbed into Δb, although some non-idealities may contribute to both. Assuming that A is nonsingular and that the hardware-induced perturbations are sufficiently small, the effective matrix equation implemented by the practical mAIC system can therefore be expressed as:

$$(A + \Delta A)\hat{x} = b + \Delta b \tag{S9}$$

where $\hat{x}$ is the solution obtained from the practical mAIC system after convergence. The solution error is defined as

$$\Delta x = \hat{x} - x^* \tag{S10}$$

Substituting $\hat{x} = x^* + \Delta x$ into Eq. S9 and using the ideal relation $Ax^* = b$, we obtain

$$(A + \Delta A)(x^* + \Delta x) = b + \Delta b \tag{S11}$$

436 which gives

$$A\Delta x + \Delta A x^{*} + \Delta A \Delta x = \Delta b \tag{S12}$$

438 When the hardware perturbations are sufficiently small, the second-order term $\Delta A \Delta x$ can be
439 neglected. The first-order error relation is therefore

$$A\Delta x \approx \Delta b - \Delta A x^{*} \tag{S13}$$

441 and the solution error can be approximated as

$$\Delta x \approx A^{-1}(\Delta b - \Delta A x^{*}) \tag{S14}$$

443 Taking norms on both sides of Eq. S14 gives

$$\| \Delta x \| \lesssim \| A^{-1} \| \left(\| \Delta b \| + \| \Delta A \| \ \| x^{*} \| \right) \tag{S15}$$

445 After normalization by $\| x^{*} \|$, the first-order relative solution error can be bounded as

$$\frac{\| \Delta x \|}{\| x^{*} \|} \lesssim \| A^{-1} \| \left(\frac{\| \Delta b \|}{\| x^{*} \|} + \| \Delta A \| \right) \tag{S16}$$

447 Using $b = A x^{*}$ and $\| b \| \le \| A \| \ \| x^{*} \|$, this relation can be written in the normalized form

$$\frac{\| \Delta x \|}{\| x^{*} \|} \lesssim \kappa(A) \left(\frac{\| \Delta A \|}{\| A \|} + \frac{\| \Delta b \|}{\| b \|} \right) \tag{S17}$$

449 where

$$\kappa(A) = \| A \| \ \| A^{-1} \| \tag{S18}$$

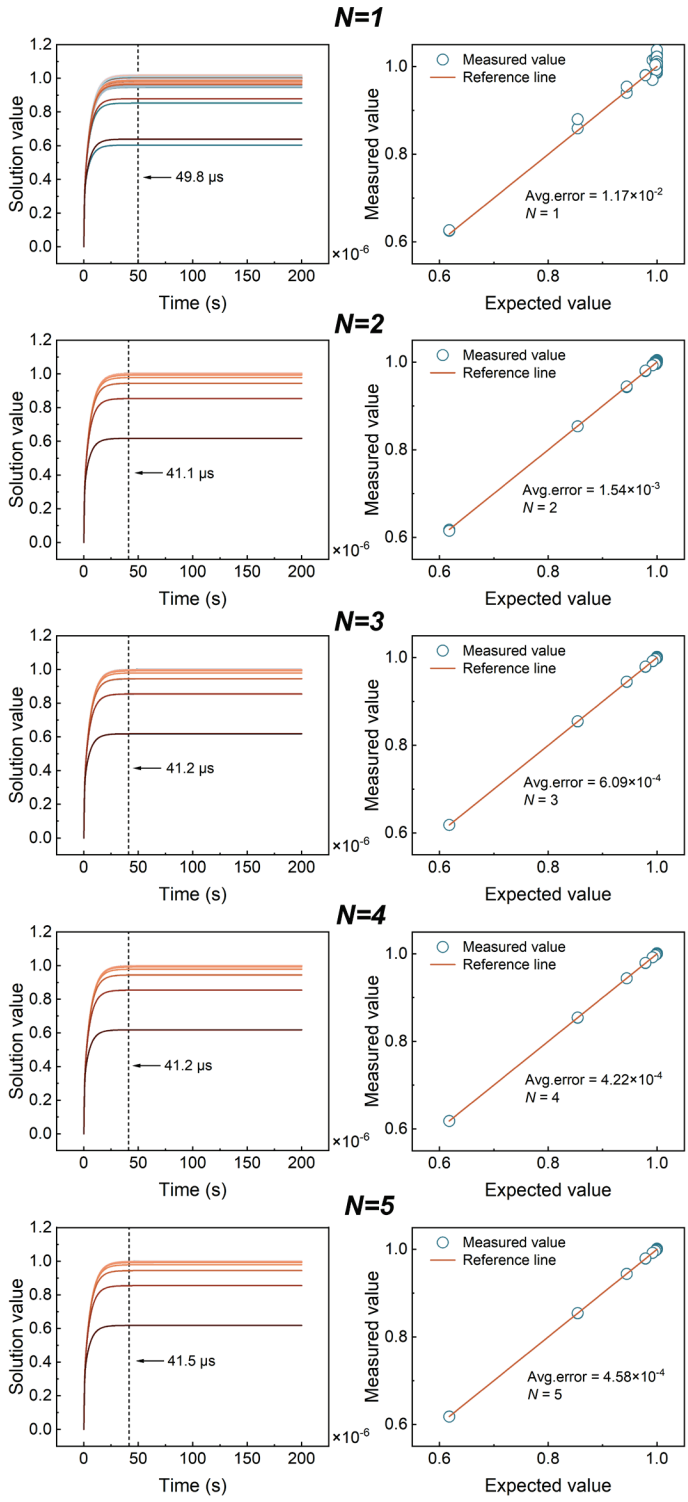
451 is the condition number of the original coefficient matrix A. This expression shows that the steady-
452 state solving error is governed by two factors: the magnitude of hardware-induced perturbations
453 and the sensitivity of the original matrix equation.

454 The dependence on the number of AC-PoP cycles N is evaluated in **Supplementary Figs. 19**
455 and **20**. Increasing N progressively compensates the residual conductance-mapping error and
456 therefore reduces the effective matrix perturbation ΔA. As a result, the converged solution obtained

from the mAIC system becomes closer to the ideal solution x^*. The voltage-convergence traces and solution comparisons in **Supplementary Fig. 19** show improved agreement between the hardware solution and the ideal solution as *N* increases. The solution error in **Supplementary Fig. 20** decreases rapidly at small *N*, indicating that the matrix-mapping error is the dominant error source in this regime. At larger *N*, the error reduction gradually saturates. This saturation suggests that, after the conductance-mapping error is sufficiently compensated, the remaining solution error is limited by circuit-level non-idealities that are not fully removed by additional AC-PoP cycles, such as amplifier offset, finite readout gain, input-voltage error, parasitic effects, and noise. Therefore, increasing *N* improves the solution precision only until matrix-mapping error is no longer the dominant contributor. Based on this trade-off between solution accuracy and compensation overhead, *N*=3 was used in the experimental demonstrations.

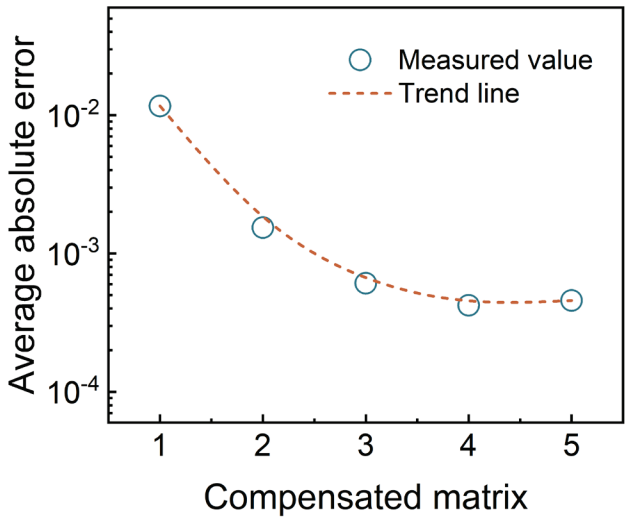
The influence of matrix conditioning is evaluated in **Supplementary Figs. 21** and **22** using matrices with different condition numbers of the original coefficient matrix *A*. For well-conditioned matrices, the same level of hardware perturbation produces only a small deviation in the converged solution. As $\kappa(A)$ increases, perturbations in *A* and *b* are more strongly amplified, leading to larger deviations between $\hat{x}$ and x^*. The solution comparisons in **Supplementary Fig. 21** show this degradation in solving accuracy for matrices with larger condition numbers. **Supplementary Fig. 22** further summarizes the extracted solution error as a function of $\kappa(A)$. The measured error shows an approximately linear increasing trend with $\kappa(A)$ with R^2=0.95 for a linear fit. This observation is consistent with the first-order perturbation bound, in which $\kappa(A)$ acts as an error-amplification factor. These results indicate that, even under comparable hardware perturbations, ill-conditioned matrices produce larger steady-state solution errors.

Together, these results indicate that the solving precision of the mAIC system is determined by the residual hardware perturbations and their amplification by the target matrix. Increasing the number of AC-PoP cycles *N* reduces the matrix-mapping component of the perturbation, whereas the condition number $\kappa(A)$ determines how strongly the remaining perturbations are amplified in the final solution. This analysis explains the improved solution accuracy obtained by AC-PoP compensation and the reduced precision observed for ill-conditioned matrices.

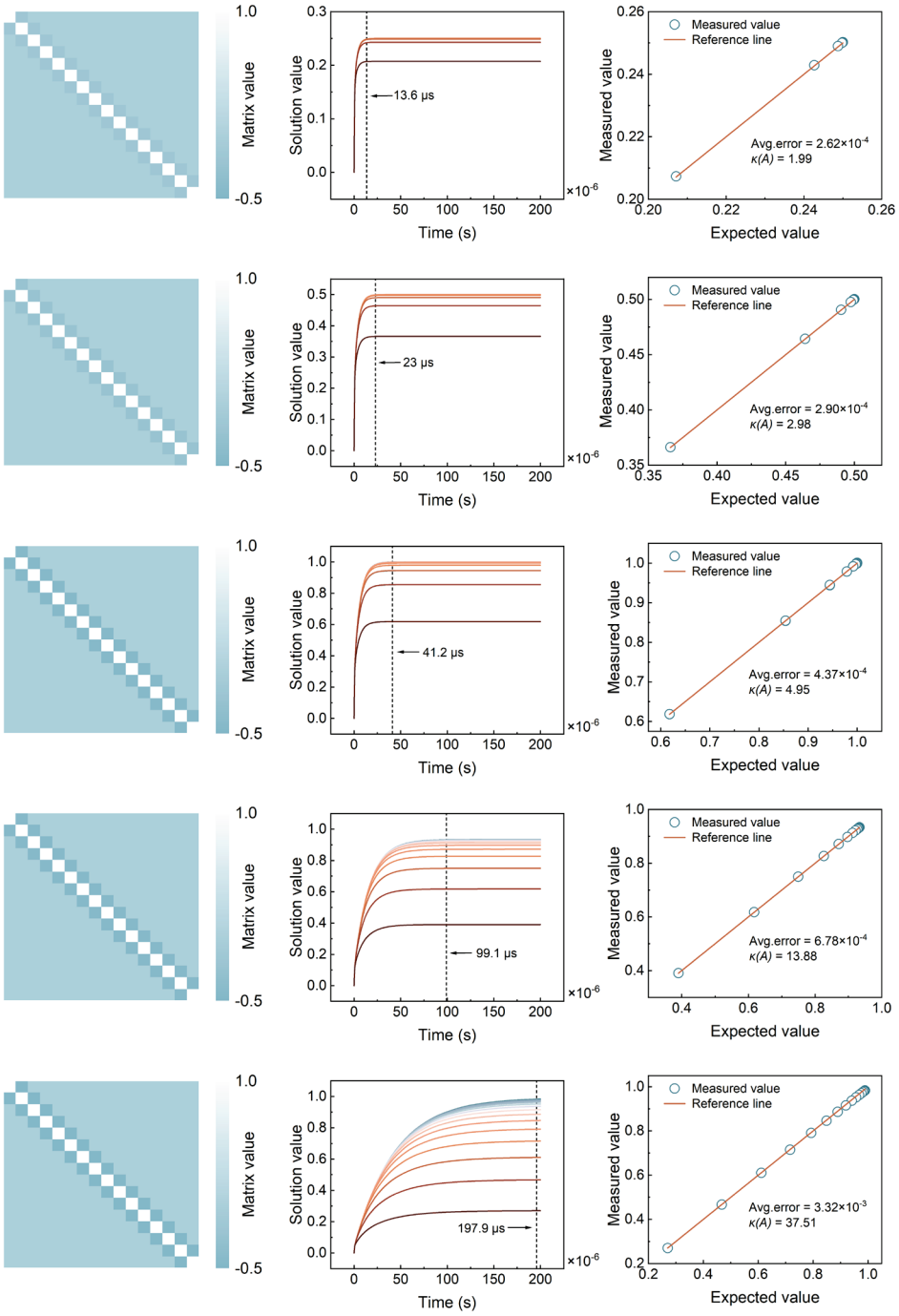


485

486 **Supplementary Fig. 19. Solution accuracy under different numbers of AC-PoP cycles *N*.**
487 Voltage convergence traces and solution comparisons obtained with different numbers of *N*.
488 Increasing *N* improves the agreement between the mAIC solution and the ideal solution.

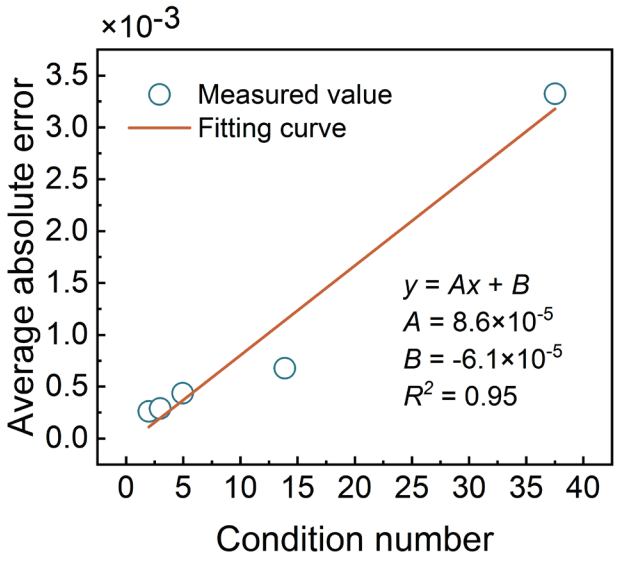


Supplementary Fig. 20. Dependence of solution error on the number of AC-PoP cycles. Relative solution error as a function of *N*. The error decreases rapidly at small *N* and saturates at larger *N*, motivating the use of *N*=3 as a trade-off between accuracy and compensation overhead.



494

495 **Supplementary Fig. 21. Solution accuracy for matrices with different condition numbers.**
496 Representative coefficient matrices, voltage-convergence traces, and solution comparisons for
497 matrices with different $\kappa(A)$. Larger $\kappa(A)$ produces larger deviations between the mAIC and
498 ideal solutions.



499

500 **Supplementary Fig. 22. Dependence of solution error on matrix condition number.** Relative
501 solution error as a function of $\kappa(A)$. The approximately linear trend with R^2=0.95 is consistent
502 with perturbation analysis, where $\kappa(A)$ amplifies hardware-induced errors.

503

Note 6: Stability of the analogue feedback iteration

The following analysis focuses on the fixed-point iteration stability and assumes that the analogue building blocks are internally stable and that the feedback loop can be described by a quasi-static linear update. The mAIC feedback circuit implements the Jacobi iterative form, $x_{i+1} = Bx_i + f$, where x_i is the voltage vector applied to the array input at the i-th iteration, matrix B is the iteration matrix implemented by the mapped conductance matrices, and f is the fixed vector term supplied by the external voltage source. As illustrated in **Supplementary Fig. 23**, the input voltage vector is applied to the differential memristor arrays to perform signed MVM. The output currents are converted into voltage signals by the TIAs, weighted by the selected gain factors, and then combined with the vector term f by the analogue adder to generate the next-state voltage x_{i+1}. This voltage vector is routed back to the array input for the next iteration.

For a fixed programmed matrix, this feedback process repeatedly applies the same linear state-update relation. If x^* denotes the steady-state solution satisfying $x^* = Bx^* + f$, the deviation from this steady state, $e_i = x_i - x^*$, evolves as:

$$e_{i+1} = Be_i \tag{S19}$$

Thus, convergence of the feedback voltage state requires the iteration error to decay under repeated multiplication by matrix B. The convergence behaviour is therefore governed by the spectral radius of the iteration matrix, with stable convergence obtained when

$$\rho(B) < 1 \tag{S20}$$

where $\rho(B)$ is the spectral radius of the iteration matrix B. Under this condition, the error introduced in the feedback voltage state is progressively attenuated. When $\rho(B)$ approaches one, the attenuation becomes slower and the circuit requires a longer settling time. When $\rho(B) \geq 1$, the feedback voltage state cannot converge to a bounded steady state.

To verify this spectral-radius dependence, we constructed a group of iteration matrices with similar sparse structures but different spectral radii. The representative matrices and the corresponding voltage-convergence traces are shown in **Supplementary Fig. 24**. When the spectral radius is small, the voltage states rapidly approach their steady values within a short settling time. As the

spectral radius increases, the convergence traces become progressively slower, indicating weaker error attenuation in the feedback loop. These results confirm that the convergence speed of the mAIC feedback circuit is primarily determined by the spectral radius of the implemented iteration matrix.

This trend is consistent with the error-propagation relation derived above. During repeated feedback, the dominant error component is associated with the eigenvalue of matrix *B* with the largest magnitude. Therefore, a larger *ρ(B)* produces slower error decay and longer voltage settling. **Supplementary Fig. 25** summarizes the extracted settling time as a function of *ρ(B)*. The fitted curve shows a nonlinear increase in settling time as *ρ(B)* approaches one, which agrees with the convergence condition $\rho(B)<1$ and explains the slower convergence observed in **Supplementary Fig. 24**.

For the damped Jacobi formulation used in the path-planning task, the feedback loop implements the effective iteration:

$$x_{i+1} = \tilde{B}x_i + \tilde{f} \tag{S21}$$

where $\tilde{B} = (1-\omega)I + \omega B$, and $\tilde{f} = \omega f$.

Here, ω is the relaxation factor. The stability condition is therefore determined by the spectral radius of the effective iteration matrix,

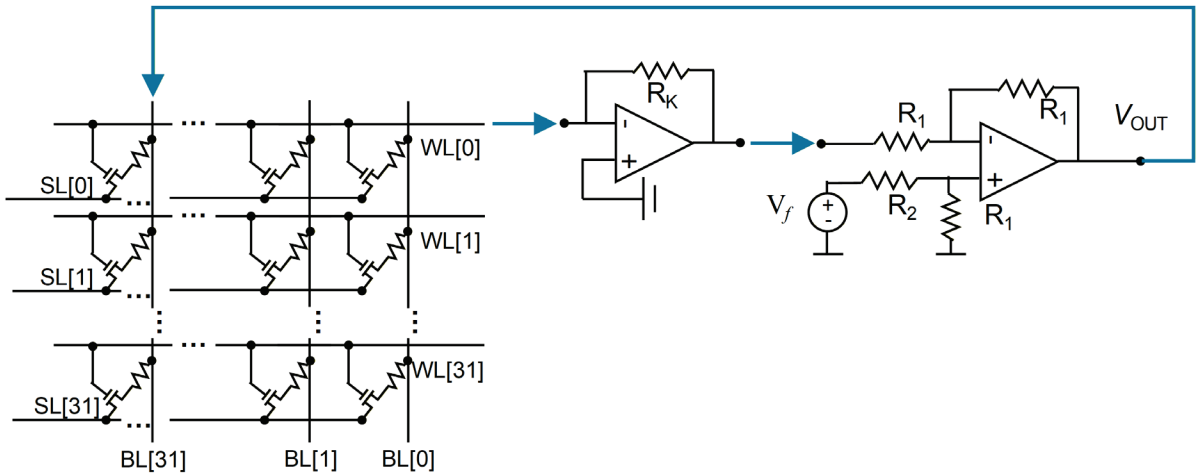
$$\rho(\tilde{B}) < 1 \tag{S22}$$

If λ_i is an eigenvalue of matrix *B*, the corresponding eigenvalue of $\tilde{B}$ is

$$\tilde{\lambda}_i = (1-\omega) + \omega\lambda_i \tag{S23}$$

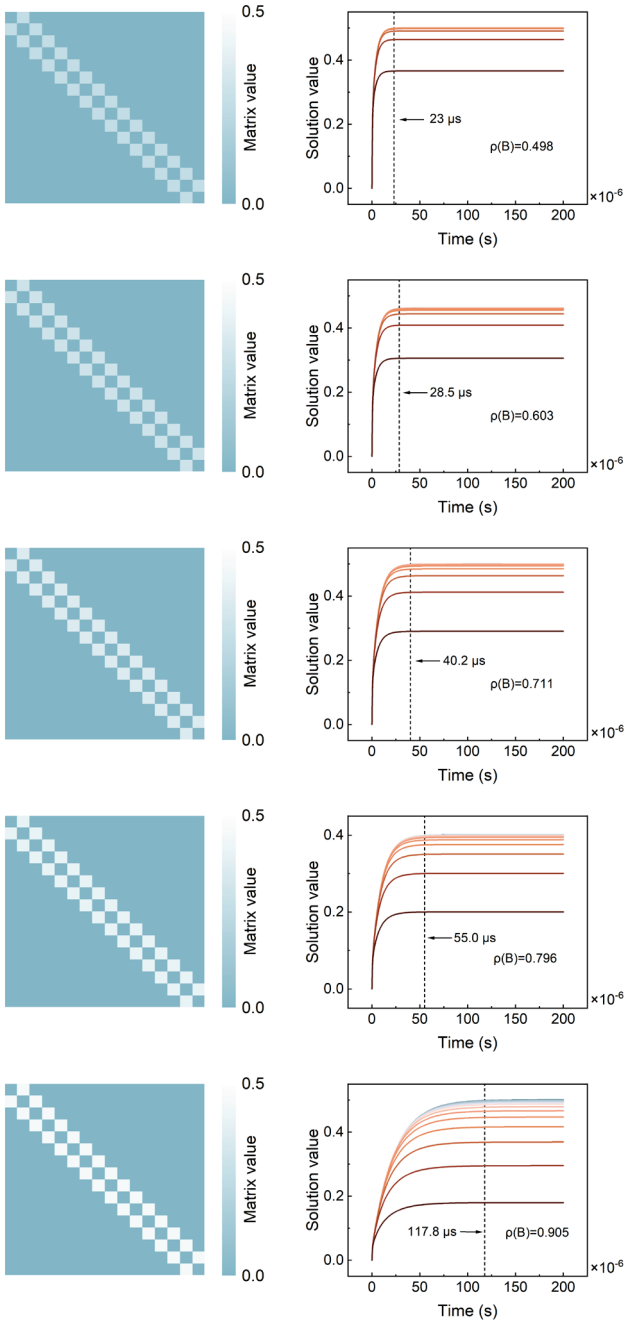
The relaxation factor changes the eigenvalue distribution of the effective iteration matrix and can therefore be used to tune the convergence behaviour for a given implemented matrix. By selecting ω, the convergence behaviour can be adjusted for the implemented matrix, with a trade-off between stability margin and convergence speed.

In hardware, the damped matrix $\tilde{B}$ is implemented by remapping the target conductance matrices, whereas the damped vector term $\tilde{f}$ is supplied by rescaling the external voltage inputs. Thus, the damping factor changes the matrix actually implemented by the feedback loop rather than acting as a post-processing step after analogue solving. This formulation allows the convergence behaviour to be adjusted through the effective iteration matrix while keeping the feedback topology unchanged.



561

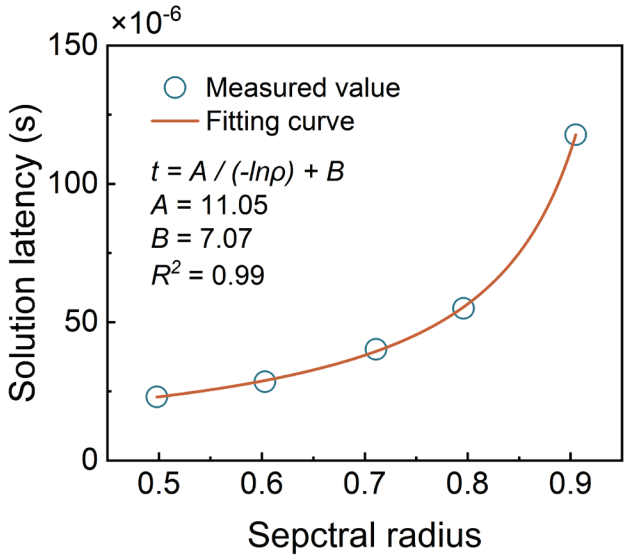
562 **Supplementary Fig. 23. Equivalent circuit model of the mAIC feedback iteration.** Equivalent
563 analogue feedback circuit implementing the fixed-point update $x_{i+1} = Bx_i + f$. The labelled
564 resistors set the TIA scaling factors and the adder weighting ratios.



565

566 **Supplementary Fig. 24. Voltage convergence under iteration matrices with different spectral**
567 **radii.** Representative iteration matrices and corresponding voltage-convergence traces for
568 different spectral radii $\rho(\mathbf{B})$. The voltage settling becomes slower as $\rho(\mathbf{B})$ increases towards one,
569 consistent with weaker attenuation of the feedback-state error.

570



571

572 **Supplementary Fig. 25. Relationship between spectral radius and settling time.** Extracted
573 convergence time as a function of the spectral radius $\rho(B)$. The fitted curve shows a nonlinear
574 increase in settling time as $\rho(B)$ approaches one, consistent with the convergence condition $\rho(B)<1$.

575

Note 7: SLAM backend construction and performance benchmark

(1) SLAM equation construction and solution

The SLAM backend localization task evaluated in **Fig. 3** was constructed from the first 20 frames of KITTI sequence 00[3]. The original vehicle trajectory was projected onto the ground plane and temporally down-sampled to form a compact two-dimensional SLAM backend localization problem. Each pose was represented by a planar state vector,

$$p_i = [p_{x,i}, p_{y,i}, \theta_i]^T \tag{S24}$$

where $p_{x,i}$ and $p_{y,i}$ denote the horizontal translation components and θ_i denotes the yaw angle. The out-of-plane motion components were removed. The active optimization state in a local window was then defined as

$$x = [p_1^T, p_2^T, \ldots, p_m^T]^T \tag{S25}$$

where m is the number of active poses in the sliding window. Relative-pose constraints between temporally adjacent states were used as odometry edges in the local pose graph. For an edge connecting poses i and j, the residual $e_{ij}(x)$ measures the mismatch between the relative transformation predicted from the current pose estimates and the measured relative pose obtained from the KITTI trajectory. The backend objective was formulated as a weighted nonlinear least-squares problem,

$$\min_x \sum_{(i,j)\in\mathcal{E}} e_{ij}(x)^T \Omega_{ij} e_{ij}(x) \tag{S26}$$

where ε is the set of relative-pose constraints and Ω_{ij} is the corresponding information matrix.

At each Gauss–Newton step, the residuals were first-order linearized around the current pose estimate x,

$$e(x+\Delta x) \approx e(x) + J\Delta x \tag{S27}$$

where J is the residual Jacobian and Δx is the stacked pose increment to be solved. Substituting this linearized residual into the least-squares objective gives the matrix equation, $H\Delta x = g$, with $H = J^T\Omega J$ and $\mathrm{g} = -J^T\Omega e$,

Here, Ω is the block-diagonal information matrix, H is the approximate Hessian matrix, and g is the residual-dependent gradient vector. The solved increment Δx was then applied to update the active pose states.

To map this matrix equation to the mAIC system, the matrix equation was reformulated into the Jacobi fixed-point form. By decomposing $H=D+R$, where D is the diagonal component of H and R contains the off-diagonal entries, the Jacobi iteration for solving Δx is $\Delta x = -D^{-1}R\Delta x + D^{-1}g$.

This formulation matches the hardware iteration form $x_{i+1}=Bx_i+f$ used in the mAIC system, with the hardware voltage state representing the current estimate of the Gauss–Newton increment Δx.

To maintain compatibility with the implemented 32×32 solver macro, backend optimization was performed on a sliding local window rather than on the full sequence-scale system. Because each planar pose contributes three state variables, a 10-pose window yields a 30-dimensional active system. This window size maximizes array utilization while avoiding truncation of the local state vector. As the window advanced, matrix construction and remapping were confined to the active local subproblem, so the update remained local instead of requiring reprogramming of an entire trajectory-scale matrix. When a pose left the active window, its effect on the remaining states was preserved by marginalization and absorbed into a prior factor on the retained variables. This allowed the local problem to maintain continuity with the previous trajectory estimate without carrying all past states in the active system. The mapped matrix and vector were further scaled to the operating ranges of the hardware, with conductance values constrained to 10 to 100 μS and input voltages constrained to 0 to 0.5 V.

The digital reference followed the same task construction and linearization procedure, while the equation $H\Delta x=g$ was solved numerically. For the trajectory comparison in **Fig. 3k**, an early 20-frame segment of the evaluated sequence was reconstructed and compared with both the digital reference and the KITTI-derived ground-truth trajectory. The reported relative translation error was defined as the mean aligned two-dimensional position error normalized by the path length of the corresponding ground-truth segment.

Supplementary Fig. 26 shows the original SLAM backend matrix H and the corresponding Jacobi iteration matrix B. The matrix was then mapped onto the mAIC system using AC-PoP with different cycle numbers N. As N increased from 1 to 3, the relative conductance mapping error decreased from 6.51% to 0.09%, as shown in **Fig. 3f**. This improved matrix mapping directly enhanced the MVM accuracy and the final solution accuracy. **Supplementary Fig. 27** reports the MVM NRMSEs for N=1 and N=2, with values of 12.80% and 5.83%, respectively. Together with the N=3 result in **Fig. 3g**, the MVM NRMSE decreased to 0.59% after three AC-PoP cycles. **Supplementary Fig. 28** presents the voltage-convergence traces and corresponding absolute solution errors for N=1 and N=2, with errors of 3.67×10^{-2} and 3.02×10^{-3}, respectively. At N=3, the mean solution error further decreased to 4.57×10^{-4}, and the hardware solver converged within 331 μs, as shown in **Fig. 3h** and **Fig. 3i**. Consequently, the relative solution error at N=3 was reduced by 75.8-fold compared with that at N=1. The trajectory reconstructed from the mAIC solution achieved a relative translation error of 0.49%, comparable to the 0.39% obtained with the CPU direct-solve reference.

(2) Latency and energy benchmark

We further evaluated the solve-stage latency and energy consumption of the SLAM backend problem, as summarized in **Supplementary Fig. 29**. The benchmark focused on the matrix equation solving stage after matrix and vector construction. Image acquisition, SLAM frontend processing, matrix construction, data communication, and robot-control execution were excluded, so that the comparison reflected the computational cost of solving the same SLAM backend matrix equation. The digital baselines solved the same linearized system derived from the KITTI sequence. The NVIDIA Jetson power was obtained from tegrastats during execution, whereas the Intel i7-13700K power was estimated from CPU utilization sampled by psutil after subtracting an idle-utilization baseline.

The average solution latency of the mAIC system was measured from the continuous-time voltage settling process, T_{mAIC}=0.332 ms. The average software solving latency was T_{Jetson}=8.55 ms for the NVIDIA Jetson baseline and T_{i7}=4.04 ms for the Intel i7-13700K baseline. Thus, the mAIC system achieved 25.8-fold and 12.2-fold reductions in SLAM backend solution latency relative to the NVIDIA Jetson and Intel i7-13700K baselines, respectively.

For the SLAM backend task, the active local system contained 30 state variables, corresponding to a 10-pose sliding window with three variables per pose. The hardware implementation used three differential array pairs, corresponding to six memristor macros, to implement the signed iteration matrix. The high-resistance state and low-resistance state of the memristor devices were R_{HRS}=100 kΩ, R_{LRS}=10 kΩ. So, the average resistance of an active device was approximated as

$$R_{\mathrm{avg}} = \frac{R_{\mathrm{HRS}} + R_{\mathrm{LRS}}}{2} = 55\ \mathrm{k\Omega} \tag{S28}$$

The average power of one active memristor is

$$P_{\mathrm{device}} = \frac{V_{\mathrm{avg}}^2}{R_{\mathrm{avg}}} = \frac{0.2^2}{55000} \approx 7.27\times10^{-7}\ \mathrm{W} = 0.727\ \mu\mathrm{W} \tag{S29}$$

For the 30-dimensional SLAM backend system, each macro contained 30×30 active devices, and six macros were used. Therefore, the total number of active memristor devices was N_{active}=30×30×6=5400. The total power of the active memristor macros was then

$$P_{\mathrm{macro}} = N_{\mathrm{active}} P_{\mathrm{device}} = 5400\times0.727\ \mu\mathrm{W} \approx 3.93\ \mathrm{mW} \tag{S30}$$

The power consumption of the peripheral operational amplifiers was estimated from their output load-driving power under the worst-case computing condition. In this estimate, the memristor load was assumed to be in the low-resistance state, R_{LRS}=10 kΩ, and the maximum computing voltage was taken as V_{max}= 2V_{avg}= 0.4 V. Thus, the peak output current required from one amplifier channel is

$$I_{\mathrm{amp}} = \frac{V_{\mathrm{max}}}{R_{\mathrm{LRS}}} = \frac{0.4}{10000} = 40\ \mu\mathrm{A} \tag{S31}$$

Using an effective supply voltage of V_{supply}=12 V, the power of one operational amplifier channel is estimated as

$$P_{\mathrm{amp}} = I_{\mathrm{amp}} V_{\mathrm{supply}} = 40\ \mu\mathrm{A}\times12\ \mathrm{V} = 0.48\ \mathrm{mW} \tag{S32}$$

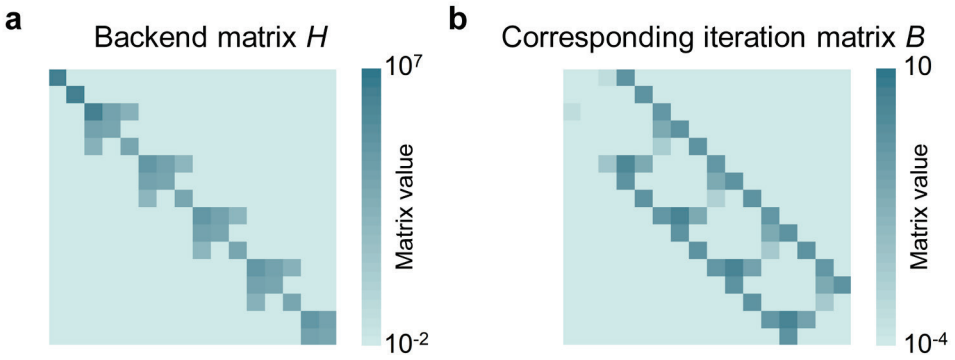
With 90 operational amplifier channels, the total amplifier power is P_{amps}=43.2 mW. Therefore, the total dynamic solving power of the mAIC system is

$$P_{\mathrm{mAIC}} = P_{\mathrm{macro}} + P_{\mathrm{amps}} = 3.93\ \mathrm{mW} + 43.2\ \mathrm{mW} = 47.13\ \mathrm{mW} \tag{S33}$$

With a measured SLAM backend solving latency of 0.332 ms, the mAIC solution energy was

$$E_{\mathrm{mAIC}} = P_{\mathrm{mAIC}} T_{\mathrm{mAIC}} = 47.13 \times 10^{-3}\ \mathrm{W} \times 0.332\ \mathrm{ms} = 1.56 \times 10^{-5}\ \mathrm{J} \tag{S34}$$

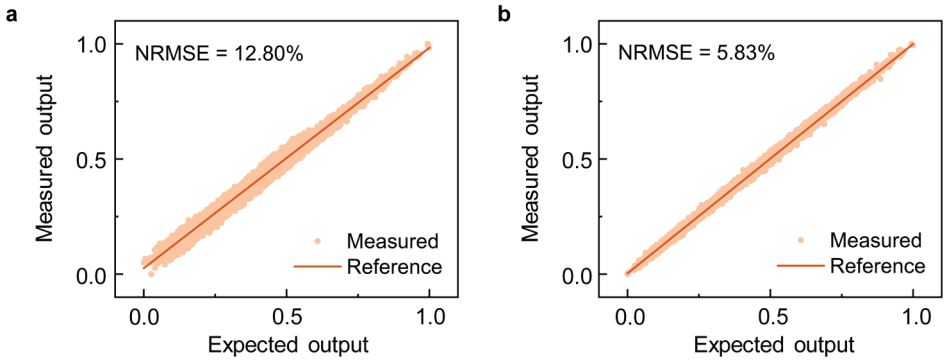
For the digital baselines, the solution energy was calculated using the measured average active power during software solving. The NVIDIA Jetson baseline had an average solving latency of 8.54 ms, and an average power of 1.87 W. Thus, its solution energy was 1.60×10^{-2} J. The Intel i7-13700K baseline had an average solving latency of 4.04 ms, and an average active power of 7.67 W. Thus, its solution energy was 3.19×10^{-2} J. Accordingly, the energy reduction of the mAIC system relative to the NVIDIA Jetson baseline was 1023.3×. These latency and energy values were used for the benchmark comparison in **Supplementary Fig. 29**. The results show that the mAIC system solves the KITTI-derived SLAM backend problem with digital-comparable trajectory accuracy while reducing both solution latency and solution energy relative to edge digital processors.



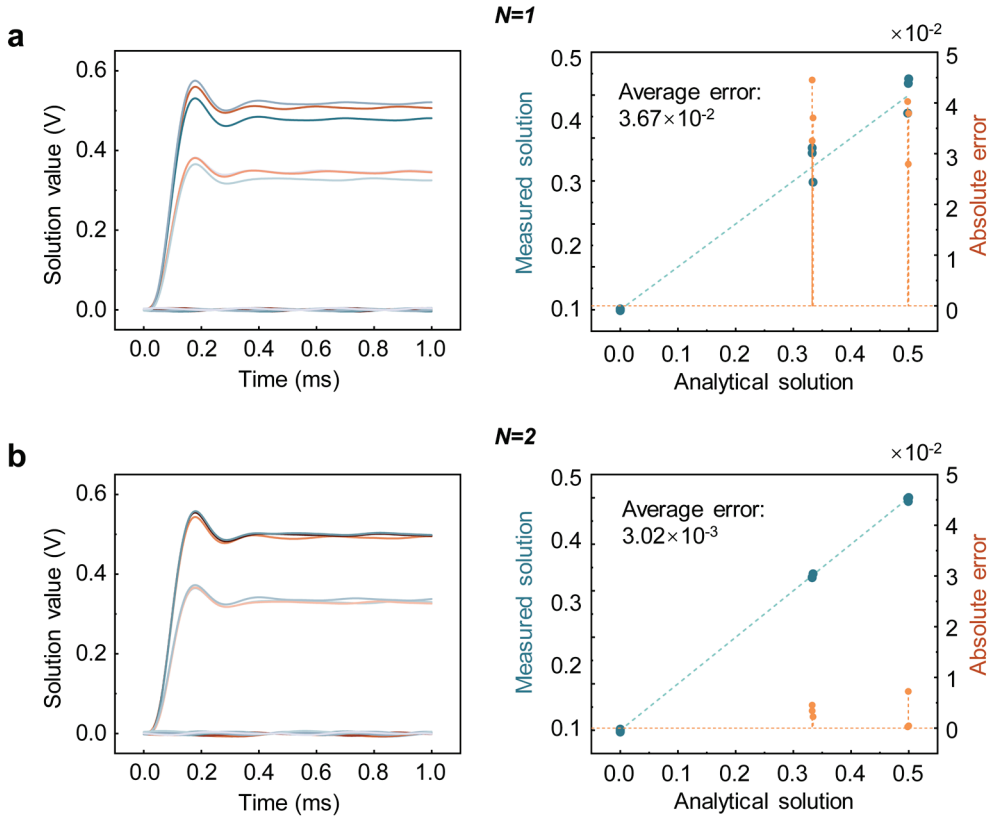
697

698 **Supplementary Fig. 26. SLAM backend matrix and iteration matrix. a,** Original backend
699 matrix *H* from the local 2D SLAM problem. **b,** Corresponding Jacobi iteration matrix *B* for mAIC
700 solving.

701



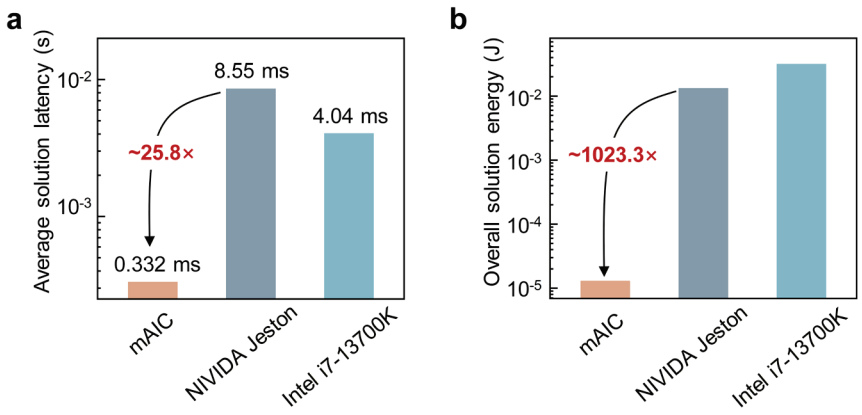
Supplementary Fig. 27. MVM accuracy for the SLAM iteration matrix. MVM results for **(a)** N=1 and **(b)** N=2, with NRMSEs of 12.80% and 5.83%, respectively.



706

707 **Supplementary Fig. 28. Hardware solution of the SLAM backend problem.** Voltage
708 convergence traces and absolute errors for **(a)** N=1 and **(b)** N=2, with errors of 3.67×10^{-2} and
709 3.02×10^{-3}, respectively.

710



711

712 **Supplementary Fig. 29. Latency and energy comparison for SLAM backend solving. a,**
713 Average solution latency of the mAIC system, NVIDIA Jetson, and Intel i7-13700K when solving
714 the same KITTI-derived SLAM backend matrix equation. The mAIC system achieves 25.8-fold
715 and 12.2-fold latency reductions relative to the NVIDIA Jetson and Intel i7-13700K baselines,
716 respectively. **b,** Overall solution energy calculated from the measured active power and solution
717 latency of each platform. The mAIC system reduces the solution energy by 1023.3-fold relative to
718 the NVIDIA Jetson baseline.

719

Note 8: Validation and performance benchmark for real-world AMR path planning

(1) Problem formulation

The real-world mobile robotics task in **Fig. 4** was formulated as a two-dimensional path-planning problem based on a scalar potential field. Let *u(x,y)* denote the planning potential over the free space. In the obstacle-free interior region, the potential satisfies the Laplace equation:

$$\nabla^2 u(x, y) = 0 \tag{S35}$$

subject to boundary conditions imposed by the target, obstacle boundaries, and the outer boundary of the local planning region. The navigation path was obtained by following the descending direction of the resulting potential field. To enable execution on the mAIC system, the continuous domain was discretized on a finite grid, yielding a sparse matrix equation $Ax=b$, where x collects the unknown node potentials within the planning window, matrix A is the sparse coefficient matrix determined by the local finite-difference stencil, and vector b contains the contributions from the predefined boundary conditions.

For a standard five-point discretization, each interior free-space node is coupled only to its nearest-neighbour nodes. The resulting coefficient matrix therefore has a sparse local structure, with the diagonal term determined by the self-node coefficient and the off-diagonal terms determined by the adjacent grid nodes. Target, obstacle, and outer-boundary constraints were imposed as boundary conditions and incorporated into the right-hand-side term b. In the experimental implementation, the target boundary was assigned a low potential, whereas obstacle and outer-boundary nodes were assigned high potentials, so that the planned path could be extracted by following the local descending direction of the solved potential field. Under this formulation, each local planning problem was reduced to a sparse matrix equation over the active sliding window, making it compatible with analogue matrix-vector multiplication in the mAIC system.

To deploy this planning problem on the mAIC system, the discretized system was directly expressed in the damped Jacobi form:

$$x_{i+1} = \tilde{B}x_i + \tilde{f} \tag{S36}$$

with $\tilde{B} = (1-\omega)I + \omega B$, $\tilde{f} = \omega f$. Here, $B = -D^{-1}R$ and $f = D^{-1}b$ are the standard Jacobi iteration matrix and vector obtained by decomposing $A = D + R$, and ω is the relaxation factor. In the mAIC implementation, the memristor macros store the effective damped iteration matrix $\tilde{B}$, whereas the external voltage-input path supplies the effective vector term $\tilde{f}$. Thus, damping is incorporated directly into the mapped conductance weights and the input voltage vector, rather than being applied as a separate post-processing step. The relaxation factor was selected to keep the damped iteration matrix within the stable convergence regime while maintaining a short settling time.

To match the physical array size and analogue operating range, path planning was performed within a sliding local window instead of over the full environment. At each replanning step, the local map region around the robot and nearby obstacles was projected onto the solver. The active planning window was chosen so that the number of unknown node potentials remained compatible with the implemented mAIC solver dimension. The corresponding $\tilde{B}$ and $\tilde{f}$ were then scaled to the conductance and voltage ranges supported by the mAIC system. This local-window formulation preserves the boundary information required for replanning while keeping the matrix dimension within the hardware-compatible range.

Environmental changes did not always require matrix reprogramming. When the free-space topology and local discretization stencil within the sliding window remained unchanged, the effective matrix $\tilde{B}$ stored on the memristor macros was retained. In this case, replanning only required refreshing $\tilde{f}$ through the voltage-input path, without modifying the programmed conductance states. When obstacle entry or boundary changes altered the local free-space topology or discretization stencil, the updated system produced new iterative quantities $\tilde{B}(t_1)$ and $\tilde{f}(t_1)$. For a fixed relaxation factor ω, the corresponding updates are

$$\Delta\tilde{B} = \tilde{B}(t_1) - \tilde{B}(t_0) = \omega\left[B(t_1) - B(t_0)\right] \tag{S37}$$

and

$$\Delta\tilde{f} = \tilde{f}(t_1) - \tilde{f}(t_0) = \omega\left[f(t_1) - f(t_0)\right] \tag{S38}$$

Accordingly, vector-only replanning steps were implemented by refreshing the external voltage inputs, whereas matrix-changing replanning steps triggered in situ AC-PoP updates of the affected memristor weights together with the corresponding update of $\tilde{f}$. This update rule restricts conductance reprogramming to local changes in the planning matrix and avoids unnecessary matrix updates when the local stencil is unchanged.

In the real-world demonstration, this formulation embedded AC-PoP into the dynamic replanning loop. Most replanning steps required only voltage-level updates of $\tilde{f}$, whereas conductance-level updates of $\tilde{B}$ were invoked only when the local planning matrix changed. Across the 40-step replanning sequence, only four steps involved matrix-update events, while the remaining steps were handled by vector-term refresh. The experimental platform is shown in **Supplementary Fig. 30**, including the obstacle-navigation arena, host-control computer, mAIC hardware, and signal-generation/testing modules. Representative voltage-convergence traces and solution-error comparisons at selected replanning steps are shown in **Supplementary Fig. 31**, showing stable voltage convergence and low solution error during dynamic path planning.

(2) Performance evaluation

To evaluate the system performance of the proposed mAIC system in the real-world path planning task, we compared its end-to-end latency and energy consumption with two digital computing platforms: an Intel i7-13700K CPU and an NVIDIA Jetson Orin Nano 8GB. All platforms were evaluated under the same convergence criterion, where the optimization error reaches 10^{-3} after 40 solving iterations.

The end-to-end latency is defined as the total time required to complete all solving iterations. For digital platforms, it is calculated as $T_{total}=N_{solve}t_{solve}$, where N_{solve}=40 and t_{solve} is the average latency of a single solve. The corresponding energy consumption is estimated by $E_{total}=P_{avg}T_{total}$, where P_{avg} is the measured average active power during computation. For the mAIC system, the end-to-end latency includes both the mAIC solving latency and the memristor programming latency: $T_{mAIC}=T_{solve}+T_{program}$. The total energy consumption is estimated as $E_{mAIC}=E_{solve}+E_{program}$.

The solving-stage power of the mAIC system was estimated using the same active-macro and peripheral-amplifier power model introduced in Note S7. For the path-planning task, the active

local system contained 25 unknown node potentials. The implementation used three differential array pairs, corresponding to six memristor macros. Therefore, the number of active memristor devices was N_{active}= 25×25×6=3750. Using Ravg=55 kΩ and Vavg=0.2 V, as defined in Note S7, the active macro power was

$$P_{\text{macro}} = N_{\text{active}} P_{\text{device}} = 3750 \times 0.727\ \mu\text{W} \approx 2.73\ \text{mW} \tag{S39}$$

Each differential array pair produced a 25-dimensional output vector and therefore required 25 output amplification channels. For three differential array pairs, N_{amps}=25×3=75.

Using the same worst-case load-driving estimate in Note S7, the load-driving power of one operational-amplifier channel was 0.48 mW. Thus,

$$P_{\text{amps}} = 75 \times 0.48\ \text{mW} = 36.0\ \text{mW} \tag{S40}$$

The total solving-stage power of the mAIC system for the path-planning task was therefore

$$P_{\text{mAIC}} = P_{\text{macro}} + P_{\text{amps}} = 2.73\ \text{mW} + 36.00\ \text{mW} = 38.73\ \text{mW} \tag{S41}$$

The average analogue solving latency is approximately 552 μs per solve (**Supplementary Fig. 31**). Thus, for 40 solving iterations,

$$T_{\text{solve}} = 40 \times 552\ \mu\text{s} = 22080\ \mu\text{s} \tag{S42}$$

The corresponding analogue solving energy is

$$E_{\text{solve}} = 38.73\ \text{mW} \times 22.08\ \text{ms} \approx 8.55 \times 10^{-4}\ \text{J} \tag{S43}$$

For AC-PoP-based programming, the number of updated weights is 84×3=252. With a programming pulse width of 7 μs, the total programming latency is

$$T_{\text{program,AC-PoP}} = 252 \times 7\ \mu\text{s} = 1764\ \mu\text{s} \tag{S44}$$

The average programming voltage and resistance are 1.9 V and 55 kΩ, respectively. The programming power per active device is therefore estimated as

$$P_{\text{program}} = \frac{1.9^2}{55,000} \approx 6.56\times10^{-5}\ \text{W} \tag{S45}$$

Including all programming events, the total AC-PoP programming energy is estimated to be

$$E_{\text{program,AC-PoP}} = 1.15\times10^{-7}\ \text{J} \tag{S46}$$

Therefore, the total latency and energy of mAIC with AC-PoP are

$$T_{\text{AC-PoP}} = 22080\ \mu\text{s} + 1764\ \mu\text{s} = 23844\ \mu\text{s} \tag{S47}$$

$$E_{\text{AC-PoP}} = 8.55\times10^{-4}\ \text{J} + 1.15\times10^{-7}\ \text{J} \approx 8.55\times10^{-4}\ \text{J} \tag{S48}$$

For WVP-based programming, the average energy consumption and latency are approximately 101× and 345× compared with AC-PoP, as calculated in **Fig. 2G and 2H**. The total programming energy and latency therefore increase to

$$T_{\text{program,WVP}} = 1764\ \mu\text{s}\times345 = 608580\ \mu\text{s} \tag{S49}$$

$$E_{\text{program,WVP}} = 1.17\times10^{-5}\ \text{J} \tag{S50}$$

Thus, the total latency and energy of mAIC with WVP are

$$T_{\text{WVP}} = 22080\ \mu\text{s} + 608580\ \mu\text{s} = 630660\ \mu\text{s} \tag{S51}$$

$$E_{\text{WVP}} = 8.55\times10^{-4}\ \text{J} + 1.17\times10^{-5}\ \text{J} \approx 8.66\times10^{-4}\ \text{J} \tag{S52}$$

For the digital baselines, the Intel i7-13700K CPU achieves an average solving latency of 600.3 μs per iteration. The total latency for 40 iterations is

$$T_{\text{i7}} = 40\times600.3\ \mu\text{s} = 24012\ \mu\text{s} \tag{S53}$$

With an average active power of 3.24 W, the total energy consumption is

$$E_{\text{i7}} = 3.24\ \text{W}\times24.012\ \text{ms} = 0.0778\ \text{J} \tag{S54}$$

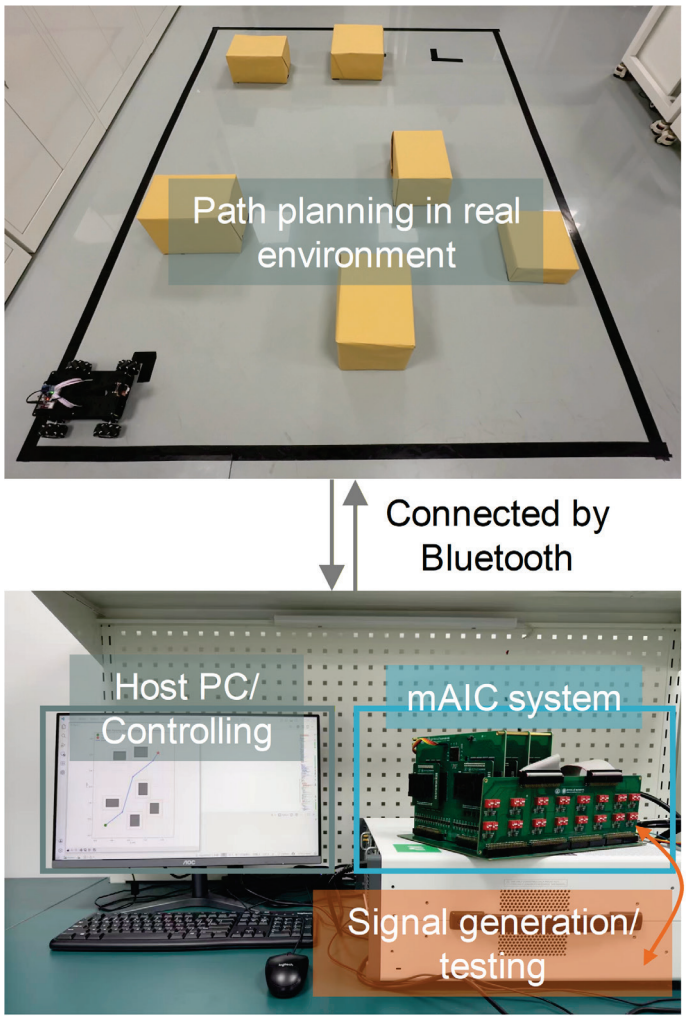
The NVIDIA Jetson Orin Nano 8GB shows an average solving latency of 2611.9 μs per iteration, leading to

$$T_{\text{Jetson}} = 40 \times 2611.9\ \mu\text{s} = 104476\ \mu\text{s} \tag{S55}$$

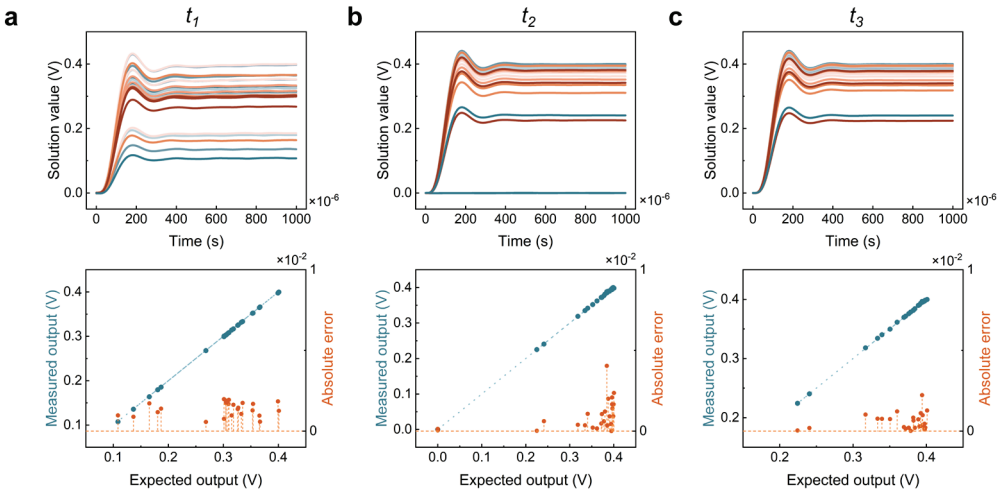
With an average active power of 5.14 W, its total energy consumption is

$$E_{\text{Jetson}} = 5.14\ \text{W} \times 104.476\ \text{ms} = 0.537\ \text{J} \tag{S56}$$

The latency and energy values calculated above were used for the benchmark comparison in **Fig. 4h** and **Fig. 4i**. In **Fig. 4h**, the total latency of mAIC with AC-PoP was compared with those of mAIC with WVP, the Intel i7-13700K CPU, and the NVIDIA Jetson baseline under the same 40-iteration convergence setting. The AC-PoP-based mAIC system achieved an average latency of approximately 0.65 ms per solve, corresponding to a 4.4× reduction relative to the NVIDIA Jetson baseline and a 26.5× reduction relative to WVP-based updating. In **Fig. 4i**, the corresponding total energy consumption was compared across the same platforms. The AC-PoP-based mAIC system consumed approximately 8.55×10^{-4} J, giving 91.0× and 627.9× lower energy than the Intel i7-13700K CPU and NVIDIA Jetson baseline, respectively. These results indicate that AC-PoP reduces the matrix-update overhead during real-world AMR path planning and enables the mAIC system to maintain low-latency and low-energy operation during dynamic obstacle-aware replanning.



Supplementary Fig. 30. Real-world AMR path-planning platform. The platform includes the obstacle-navigation arena, the host PC for environment processing and robot control, the mAIC system for analogue iterative solving, and the signal-generation/testing modules for voltage input and solution readout.



Supplementary Fig. 31. Representative convergence results in path planning. Voltage-convergence traces and solution-error comparisons at representative replanning steps. The mAIC system maintains stable convergence and low solution error during dynamic path planning.

Note 9: Scalability analysis of the row-block mAIC architecture

To support the row-block mAIC formulation used in **Fig. 5**, we evaluated the latency scaling of continuous-time analogue solving and the accuracy limitation of monolithic array scaling.

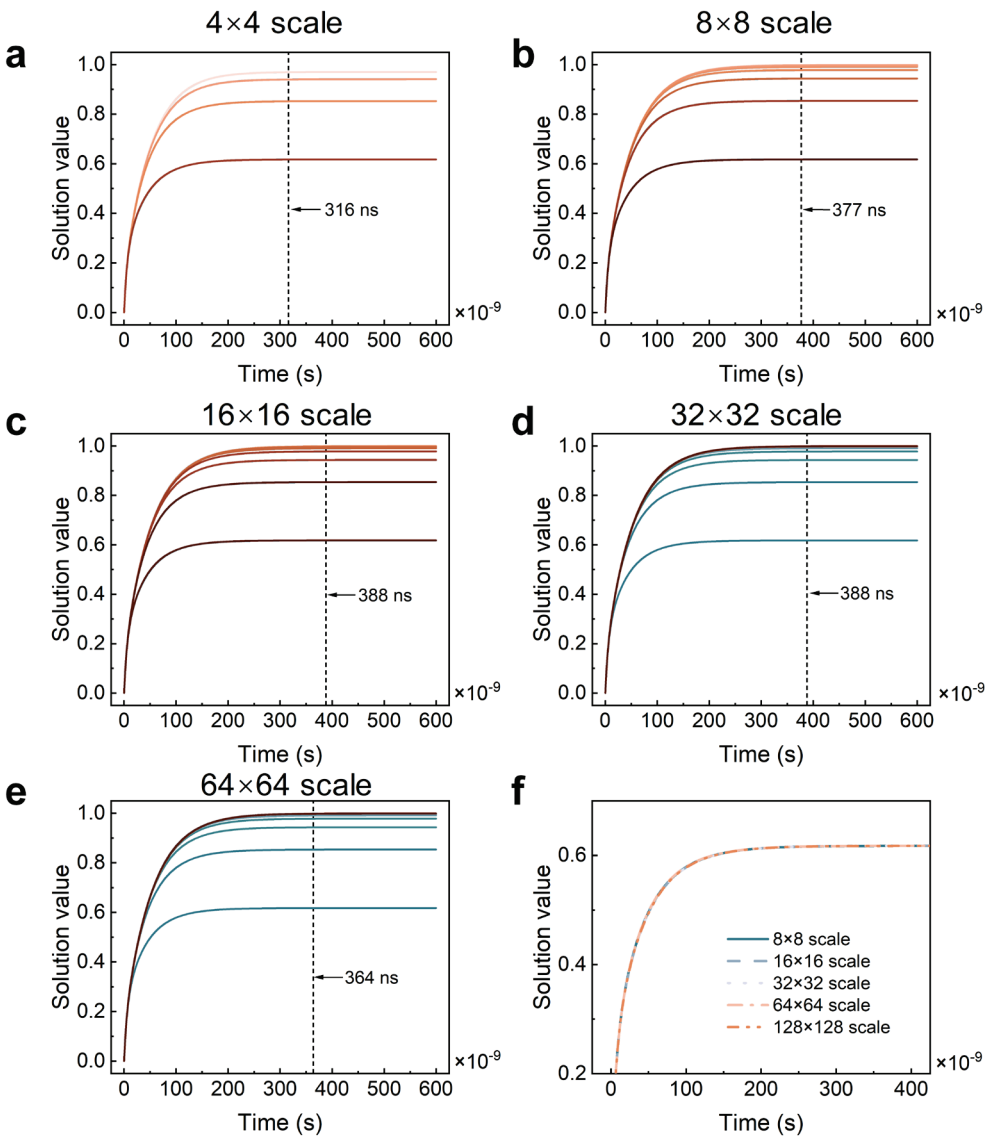
Supplementary Fig. 32 presents the solving latency for matrix equations with different matrix dimensions under comparable convergence conditions. In an ideal sufficiently large array, where wire resistance and peripheral bandwidth are not the dominant limitations, the mAIC feedback circuit converges with similar settling times across different matrix sizes, with an average settling time of approximately 388 ns. This behaviour arises because the continuous-time iterative trajectory is governed primarily by the feedback dynamics, the spectral radius of the implemented iteration matrix, and the circuit bandwidth, rather than by sequential traversal of matrix elements. Therefore, the result indicates near-O(1) latency scaling for the tested class of systems, under comparable spectral-radius and convergence-threshold conditions.

However, this ideal latency scaling does not imply that a single monolithic array can be enlarged indefinitely. **Supplementary Fig. 33** evaluates the accuracy penalty introduced by wire resistance during array-size scaling. As the array dimension increases, line resistance produces spatially dependent voltage drops along the input and output paths, causing the effective device voltages and accumulated column currents to deviate from their ideal values. This deviation perturbs the implemented MVM operation and can be represented as an equivalent matrix perturbation in the iterative solver. According to the perturbation analysis in Note S5, such hardware-induced matrix perturbations increase the residual solution error after convergence. Thus, monolithic array scaling can preserve the latency advantage in principle, but it progressively degrades solution accuracy in practice.

These two observations define the architectural requirement for scalable mAIC: the array size within each compute unit should remain bounded to limit wire-resistance-induced errors, whereas the system dimension should be increased by distributing the matrix computation across multiple units. The row-block mAIC architecture in **Supplementary Fig. 34** follows this principle. The global iteration matrix is partitioned into row blocks, and each processing element (PE) implements the local MVM associated with its assigned block using size-limited TIA-based memristor macros. The required state-vector components are supplied through the routing network,

and the local analogue outputs are combined with the corresponding vector terms to form the next-state voltages. Boundary AD/DA interfaces provide block-level and system-level signal exchange, while the internal row-block computation remains within the analogue feedback loop.

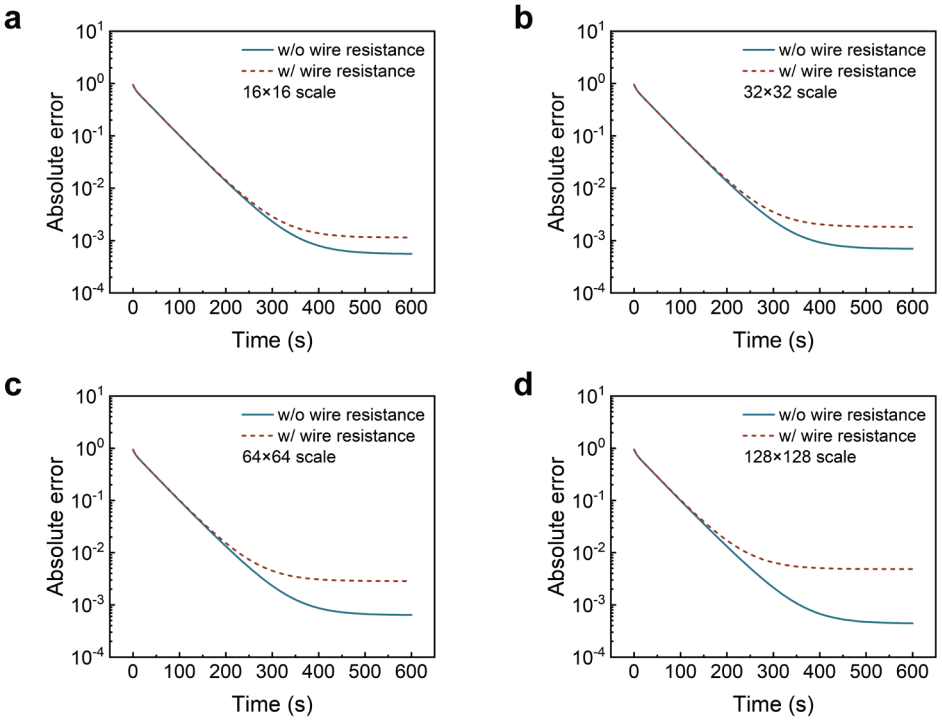
This organization also preserves the locality of matrix updates. When the robotic workload changes only in a local region, the corresponding perturbation affects a limited subset of matrix blocks. The AC-PoP controller therefore programs only the affected local macros, while the remaining blocks retain their previous conductance states. In this way, the architecture avoids both full-matrix remapping and excessive monolithic array enlargement. It therefore extends continuous-time mAIC solving to larger sparse robotic systems by scaling the number of processing elements rather than the physical size of a single memristor array.



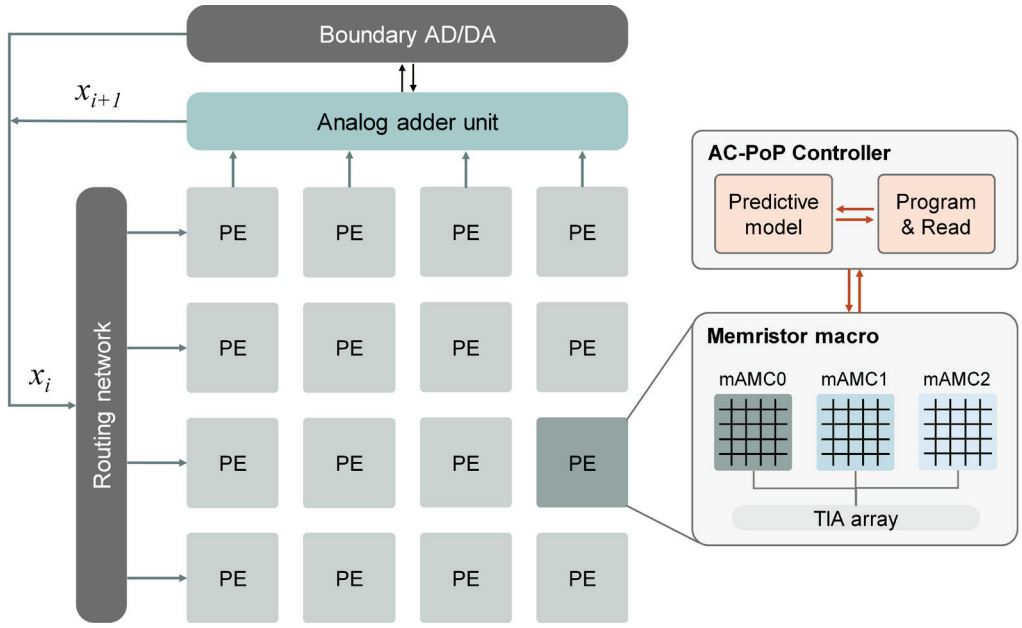
912

913 **Supplementary Fig. 32. Latency analysis for solving different matrix sizes. (a) to (e)** Voltage-
914 convergence traces of mAIC solving for matrix equations with different matrix sizes. **f,** The
915 average settling time is approximately 388 ns, indicating near-O(*1*) latency scaling under the tested
916 ideal-array conditions.

917



Supplementary Fig. 33. Impact of array wire resistance on solving accuracy. Simulated solution errors of monolithic memristor arrays with different array sizes and wire resistances. Increasing array size enhances IR-drop-induced matrix perturbations and increases the residual solution error.



924

925 **Supplementary Fig. 34. Scalable row-block mAIC architecture.** Schematic of the row-block
926 mAIC architecture. The global iteration matrix is distributed across multiple processing elements,
927 while local AC-PoP updates are confined to the affected memristor macros.

928

Note 10: Performance benchmark of scaled mAIC to 28 nm node

To evaluate the performance of a fully integrated mAIC chip at an advanced node, we performed an equivalent scaling analysis for the same real-world AMR path planning task. The algorithmic workload, the number of mAIC solving operations, and the number of matrix-element updates were kept unchanged, while the circuit-level power and delay parameters were scaled to the 28-nm node. For the mAIC solving stage, the macro power was kept at 2.73 mW, following Eq. S30. For the operational-amplifier peripheral circuits, the supply voltage at the 28-nm node was set to 1.0 V, consistent with the nominal core-supply options of 28-nm CMOS technologies[4]. According to Eq. S32, the corresponding operational-amplifier power was estimated to be 3.00 mW. Therefore, the total mAIC solving power at the 28-nm node was 5.73 mW.

The single mAIC solving latency was estimated from the benchmark using the OPA690 operational amplifier, which has a gain–bandwidth product (GBWP) of 500 MHz. Based on this high-GBWP analogue front end, one mAIC solving operation was taken as approximately 120 ns[5]. For the real-world AMR path planning task, the total solving latency was therefore

$$T_{\mathrm{solve}} = 120 \times 40\ \mathrm{ns} = 4.8\ \mu\mathrm{s} \tag{S57}$$

The corresponding solving energy was

$$E_{\mathrm{solve}} = P_{\mathrm{solve}} T_{\mathrm{solve}} = 5.73\ \mathrm{mW} \times 4.8\ \mu\mathrm{s} = 2.75 \times 10^{-8}\ \mathrm{J} \tag{S58}$$

The latency values used in this 28-nm scaling analysis differ from those in the previous board-level implementation. In the board-level experiments, 2-μs programming and 5-μs read pulses were used to ensure reliable operation under PCB parasitics, external wiring, and commercial instrument limitations. In a fully integrated 28-nm SoC, the write/read drivers can be placed close to the memristor array, reducing extrinsic RC delay. Therefore, a 150-ns program-read cycle is assumed for the scaled update benchmark. The 120-ns solving latency corresponds to analogue settling of the mAIC system, whereas the 150-ns program-read cycle corresponds to memristive conductance updating.

For the matrix-updating stage, the write power was kept the same as that in Eq. S45, namely 6.56×10^{-5} W. With the same number of matrix updates, Eqs. S44, S46, S48, and S49 give

$$T_{\text{program,AC-PoP}} = 37.8\ \mu\text{s} \tag{S59}$$

$$E_{\text{program,AC-PoP}} = 2.48\times10^{-9}\ \text{J} \tag{S60}$$

$$T_{\text{program,WVP}} = 13041\ \mu\text{s} \tag{S61}$$

$$E_{\text{program,WVP}} = 8.83\times10^{-7}\ \text{J} \tag{S62}$$

Combining the solving and matrix-updating stages gives

$$T_{\text{total,AC-PoP}} = T_{\text{solve}} + T_{\text{program,AC-PoP}} = 4.8\ \mu\text{s} + 37.8\ \mu\text{s} = 42.6\ \mu\text{s} \tag{S63}$$

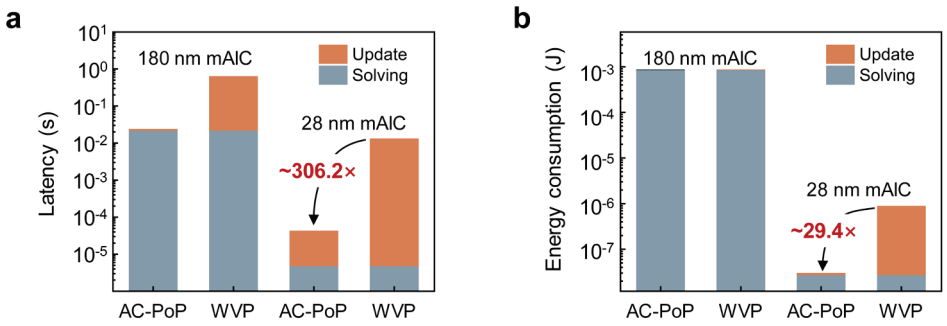
$$E_{\text{total,AC-PoP}} = E_{\text{solve}} + E_{\text{program,AC-PoP}} = 2.75\times10^{-8}\ \text{J} + 2.48\times10^{-9}\ \text{J} \approx 3.00\times10^{-8}\ \text{J} \tag{S64}$$

$$T_{\text{total,WVP}} = 4.8\ \mu\text{s} + 13041\ \mu\text{s} = 13045.8\ \mu\text{s} \tag{S65}$$

$$E_{\text{total,WVP}} = E_{\text{solve}} + E_{\text{program,WVP}} = 2.75\times10^{-8}\ \text{J} + 8.55\times10^{-7}\ \text{J} \approx 8.83\times10^{-7}\ \text{J} \tag{S66}$$

Therefore, after scaling to the 28-nm node, AC-PoP achieves a total latency of 42.6 μs and a total energy of 3.00×10^{-8} J, corresponding to 306.2× lower latency and 29.4× lower energy than WVP for the same real-world AMR path-planning task (**Supplementary Fig. 35**).

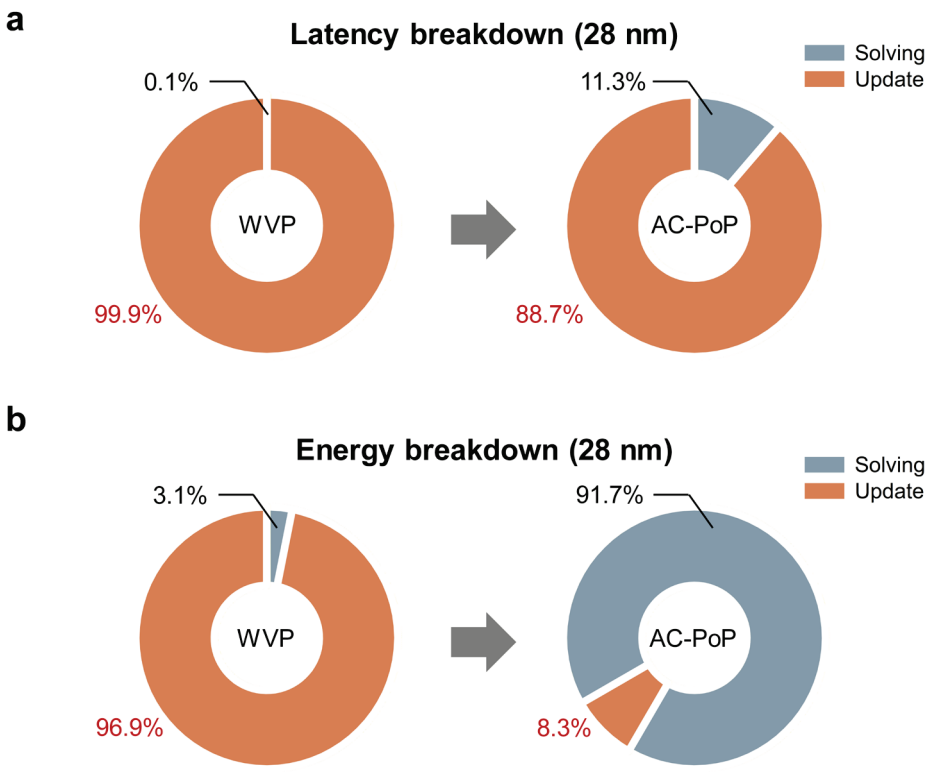
The solve-update breakdown further shows that AC-PoP primarily reduces the update overhead. The update-stage latency contribution decreases from >99.9% in WVP to 88.7% in AC-PoP, while the update-stage energy contribution decreases from 96.9% to 8.3% (**Supplementary Fig. 36**). These results indicate that AC-PoP is particularly beneficial for AMR workloads requiring frequent and accurate matrix updates.



975

976 **Supplementary Fig. 35. Latency and energy comparison between AC-PoP and WVP.** Total
977 latency and energy of the 180-nm and 28-nm mAIC system. AC-PoP reduces total latency and
978 energy by approximately 306.2-fold and 29.4-fold in the scaled mAIC system, respectively.

979



980

981 **Supplementary Fig. 36. Solve-update contribution analysis.** Latency and energy contributions
982 of solving and matrix updating in the 28-nm scaled mAIC system. AC-PoP reduces the update
983 contribution from 99.9% to 88.7% in latency and from 96.9% to 8.3% in energy.

984

Note 11: Benchmark normalization of prior CIM solvers

To compare the mAIC system with prior CIM-based linear solvers under a common robotics-oriented solve-update workload, we used the per-unknown accounting summarized in **Supplementary Table 2**. The digital processor baselines follow the task-matched measurements described in Materials and Methods. Here, we place reported CIM solvers on the same solve–update basis by normalizing problem dimension, reconstructing missing solve-stage metrics when required, and estimating matrix-update overheads that were not explicitly reported.

The problem size was denoted by N_{dim}, defined as the dimension of the solved matrix equation. For each platform, T_{solve} and E_{solve} denote the latency and energy of one complete solve. Matrix remapping or weight programming was described by a platform-specific update event with latency T_{upd} and energy E_{upd}. When the original work only reported or implied a primitive update operation, such as programming one analogue memory cell or one effective digital weight, the event-level update cost was calculated as, $T_{\mathrm{upd}} = n_{\mathrm{upd}}\tau_{\mathrm{upd}}$ and $E_{\mathrm{upd}} = n_{\mathrm{upd}}\epsilon_{\mathrm{upd}}$, where τ_{upd} and ϵ_{upd} are the latency and energy of the elementary update primitive, and n_{upd} is the effective number of updated units in one matrix-update event. For platforms whose reported update cost already corresponds to one complete update event, T_{upd} and E_{upd} were used directly.

For a workload containing N_{solve} solve events and N_{update} matrix-update events, the normalized latency and energy were calculated as

$$T_{\mathrm{norm}} = \frac{N_{\mathrm{solve}}T_{\mathrm{solve}} + N_{\mathrm{update}}T_{\mathrm{upd}}}{N_{\mathrm{dim}}} \tag{S67}$$

$$E_{\mathrm{norm}} = \frac{N_{\mathrm{solve}}E_{\mathrm{solve}} + N_{\mathrm{update}}E_{\mathrm{upd}} + E_{\mathrm{hold}}}{N_{\mathrm{dim}}} \tag{S68}$$

where E_{hold} denotes static hold energy when applicable, such as for SRAM-based systems. This formulation accumulates solve-stage and update-stage costs over the same workload and normalizes the total cost by equation dimension.

The update event is not physically identical across different platforms. Depending on the architecture, it may correspond to programmed analogue memory cells, updated digital weights, or another platform-specific remapping unit. This treatment preserves the architectural constraints of each system while expressing heterogeneous update overheads in a common workload-level metric. Values in **Supplementary Table 2** were therefore categorized as directly reported, reconstructed from reported system models or figures, or estimated for workload-level remapping overhead.

(1) PCM Hybrid solver [6]

We used the N_{dim}=5000 banded mixed-precision benchmark with K=8, 998752 devices, and 23 outer iterative refinements. Because the paper does not directly report solve-level latency and energy, we reconstructed them from the system-level baseline and dedicated-computing memory unit (CMU) constraints. The ideal-CMU system baseline gives $T_{sys,ideal}$≈11.5 ms and $P_{sys,ideal}$≈33 W, hence

$$E_{\mathrm{sys,ideal}} \approx 33\ \mathrm{W} \times 11.5\ \mathrm{ms} = 0.380\ \mathrm{J} \tag{S69}$$

Using $\tau_{10} \approx 14\ \mu\mathrm{s}$, the number of inner low-precision operations is

$$N_{\mathrm{inner}} \approx \frac{0.1 \times 11.5\ \mathrm{ms}}{14\ \mu\mathrm{s}} \approx 82 \tag{S70}$$

With 1000×1000 crossbars, 10-way parallelism, and a 1 μs cycle, the CMU latency is T_{CMU}≈0.25-0.41 ms, giving T_{solve}≈11.8 ms. The dedicated core energy is estimated from 222 mW per core and 10 parallel cores:

$$E_{\mathrm{CMU}} \approx 2.22\ \mathrm{W} \times (0.25 - 0.41\ \mathrm{ms}) \approx 0.56 - 0.91\ \mathrm{mJ} \tag{S71}$$

We used the conservative reconstructed value E_{solve}=0.416 J. The update estimate is T_{update}=998752×10 μs ≈9.99 s, E_{update}=998752×5 nJ ≈4.99×10^{-3} J.

With N_{solve}=901 and N_{update}=2026, T_{norm}≈4.05 s/unknown, E_{norm}≈7.70×10^{-2} J /unknown.

(2) RRAM Hybrid solver [7]

We used the 128×128 Poisson benchmark and set N_{dim}=128. The selected solve-stage values are T_{solve}=3.67 ms and E_{solve}=2.25×10^{-4} J. The update event programs five 36×36 preconditioner blocks, giving $N_{cell,update}$=5×36×36=6480. Using 30 write-verify cycles and a 20 ns programming pulse, the serial update latency is

$$T_{\text{update}} = 6480 \times 30 \times 20\ \text{ns} = 3.888\ \text{ms} \tag{S72}$$

The row-parallel latency lower estimate is 108 μs, but **Supplementary Table 2** uses the serial upper-bound latency. For energy, using V=2 V, R=15 kΩ, and t=10 ns,

$$E_{\text{update}} = 6480 \times 30 \times \frac{2V^2}{R} t \approx 1.04 \times 10^{-6}\ \text{J} \tag{S73}$$

Thus, under the **Supplementary Table 2** workload, T_{norm}≈8.74×10^{-2} s/unknown, E_{norm}≈1.58×10^{-3} J /unknown.

(3) RRAM Analogue solver [5]

We benchmarked the 128×8 massive-MIMO example. The 8×8 complex-valued inverse problem was counted as a 16×16 real-valued system, so N_{dim}=16. For the HP-INV/BlockAMC solve model, we used L=3, N_0=4, α_{INV}=9, and α_{MVM}=30. With t_{INV}=120 ns, t_{MVM}=60 ns, and t_{ADC}=6.7 ns,

$$T_{\text{solve}} = 3[(120 + 6.7) \times 9 + (60 + 6.7) \times 30]\ \text{ns} \approx 9.42 \times 10^{-6}\ \text{s} \tag{S74}$$

Using the reported block-energy parameters gives

$$E_{\text{solve}} = 3(E_{\text{INV,blk}} \alpha_{\text{INV}} + E_{\text{MVM,blk}} \alpha_{\text{MVM}}) \approx 1.29 \times 10^{-7}\ \text{J} \tag{S75}$$

The update overhead was estimated by a write-verify model with c_{WV}=12.625, t_{pulse}=t_{verify}=100 ns, V_{prog}=2 V, V_{verify}=0.2 V, and R=50 kΩ. Therefore,

$$T_2 = 12.625 \times 200\ \text{ns} = 2.525 \times 10^{-6}\ \text{s} \tag{S76}$$

$$E_2 = 12.625 \left[\frac{2^2}{50\ \text{k}\Omega} \times 100\ \text{ns} + \frac{0.2^2}{50\ \text{k}\Omega} \times 100\ \text{ns} \right] \approx 1.02 \times 10^{-10}\ \text{J} \tag{S77}$$

With N_{upd}=567,

$$T_{\text{update}} = 567T_2 \approx 1.43\times10^{-3}\ \text{s} \tag{S78}$$

$$E_{\text{update}} = 567E_2 \approx 5.7\times10^{-8}\ \text{J} \tag{S79}$$

Under the **Supplementary Table 2** workload, T_{norm}≈1.82×10^{-1} s/unknown, E_{norm}≈1.46×10^{-5} J /unknown.

(4) SRAM-DCIM solver [8]

We set N_{dim} =192, corresponding to the 192×192 real-SLAM matrix equation benchmark evaluated in the original paper. The single-solve latency and energy were approximately extracted from the 192-dimensional SLAM-CIM data in Fig. 22 of the original paper, giving T_1=1.80×10^{-5} s and E_1=1.00×10^{-5} J.

For the update-stage normalization, T_2 and E_2 are effective per-coefficient SRAM update costs derived from the full 192×192 FP16 matrix-remap model. Equivalently, the resident FP16 coefficient matrix contains 192×192×16=589824 bits, equivalent to 4608 128-bit SRAM words. Assuming 12-macro parallel writes at 40 MHz, the full remap latency is 384/40 MHz≈9.62 μs, giving T_2=9.62 μs/36864=2.61×10^{-10} s. Assuming 2 pJ per 128-bit SRAM word write, the full remap energy is 4608×2 pJ=9.22 nJ, giving E_2=9.22 nJ/36864=2.50×10^{-13} J.

With N_{solve}=901 solve events and N_{update}=2026 update events, the workload-normalized latency is T_{norm}≈1.86×10^{-4} s/unknown, $E_{norm,solve+update}$≈4.70×10^{-5} J /unknown.

Because the weights are stored in SRAM, we also included a static hold term. Using P_{hold}=192×192×16×0.6 V×0.1 nA=3.54×10^{-5} W, the normalized hold energy is $E_{hold,norm}$= $3.54\times10^{-5}\times1.86\times10^{-4}$=$6.58\times10^{-9}$ J/unknown. Therefore, the total normalized energy including SRAM hold is

$$E_{\text{norm,total}} = E_{\text{norm,solve+update}} + E_{\text{hold,norm}} \approx 4.70\times10^{-5}\ \text{J/unknown} \tag{S80}$$

where the hold contribution is much smaller than the solve-plus-update energy and does not change the reported significant digits.

Supplementary Table 2 therefore compares solver architectures under a common workload-level accounting, rather than isolating device-level properties or technology-node scaling. Directly reported quantities include, depending on the paper, the matrix equation dimension, array configuration, iteration count, or measured solve cost. Reconstructed quantities include solve latency or energy inferred from reported throughput, timing, power, or solver models. Estimated quantities include update-stage programming overhead when the original paper does not explicitly report matrix-remapping cost. This distinction was maintained throughout **Supplementary Table 2** to avoid conflating directly measured system metrics with engineering estimates introduced solely for cross-platform comparison.

Platform/ solver	Problem size N	Solving latency (s)	Solving energy (J)	Update latency (s)	Update energy (J)	Norm. latency (s/unk.)	Norm. energy (J/unk.)
NVIDIA Jetson	32	4.02×10^{-3}	0.013	2.02×10^{-6}	6.53×10^{-6}	0.113	0.366
Intel i7-13700K	32	2.41×10^{-3}	0.012	2.02×10^{-6}	1.04×10^{-5}	6.80×10^{-2}	0.339
PCM Hybrid solver [6]	5000	1.18×10^{-2}	0.416	9.99	4.99×10^{-3}	4.05	7.70×10^{-2}
RRAM Hybrid solver [7]	128	3.67×10^{-3}	2.25×10^{-4}	3.89×10^{-3}	1.73×10^{-8}	8.74×10^{-2}	1.58×10^{-3}
RRAM Analogue solver [5]	16	9.42×10^{-6}	1.29×10^{-7}	1.43×10^{-3}	5.78×10^{-8}	1.82×10^{-1}	1.46×10^{-5}
SRAM-DCIM solver [8]	192	1.80×10^{-5}	1.00×10^{-5}	9.62×10^{-6}	9.22×10^{-9}	1.86×10^{-4}	4.70×10^{-5}
mAIC (180 nm)	32	8.52×10^{-6}	3.30×10^{-7}	5.04×10^{-4}	3.31×10^{-8}	3.21×10^{-2}	1.14×10^{-5}
mAIC (28 nm)	32	1.20×10^{-7}	1.34×10^{-10}	2.52×10^{-5}	1.65×10^{-9}	1.60×10^{-3}	1.08×10^{-7}

Supplementary Table 2. Benchmark comparison of representative matrix equation solvers. Single-solve cost and workload-normalized latency/energy including update overhead.

Supplementary Information References

1. Ma, Y. *et al.* Formation of the Conducting Filament in TaOx-Resistive Switching Devices by Thermal-Gradient-Induced Cation Accumulation. *ACS Appl. Mater. Interfaces* **10**, 23187–23197 (2018).
2. Sun, W. *et al.* Understanding memristive switching via in situ characterization and device modeling. *Nat. Commun.* **10**, 3453 (2019).
3. Geiger, A., Lenz, P. & Urtasun, R. Are we ready for autonomous driving? The KITTI vision benchmark suite. in *2012 IEEE Conference on Computer Vision and Pattern Recognition* 3354–3361 (2012). doi:10.1109/CVPR.2012.6248074.
4. Ni, Y., Liu, L., Zhang, Y. & Zhu, T. A 12-bit 2-GS/s Pipeline ADC in 28-nm CMOS With Linear-Error Self-Calibration. *IEEE Trans. Very Large Scale Integr. VLSI Syst.* **33**, 1561–1569 (2025).
5. Zuo, P. *et al.* Precise and scalable analogue matrix equation solving using resistive random-access memory chips. *Nat. Electron.* **8**, 1222–1233 (2025).
6. Le Gallo, M. L. *et al.* Mixed-Precision In-Memory Computing. *Nat. Electron.* **1**, 246–253 (2018).
7. Song, W. *et al.* Programming memristor arrays with arbitrarily high precision for analogue computing. *Science* **383**, 903–910 (2024).
8. Li, M. *et al.* SLAM-CIM: A Visual SLAM Backend Processor With Dynamic-Range-Driven-Skipping Linear-Solving FP-CIM Macros. *IEEE J. Solid-State Circuits* **59**, 3853–3865 (2024).