ACS Electrochemistry

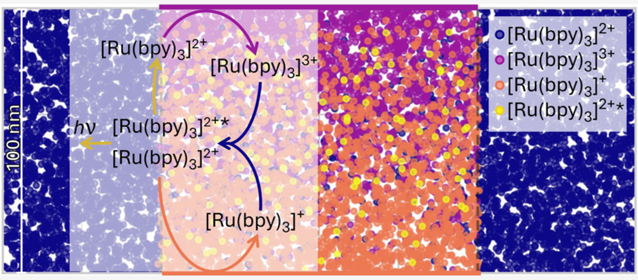
pubs.acs.org/electrochem

Communication

AI-Driven Simulation of Stochastic Electrochemiluminescence

Klaus Mathwig*

Cite This: https://doi.org/10.1021/acselectrochem.4c00166



ABSTRACT: Electrochemiluminescence (ECL) is a vital analytical technique widely used in immunosensing and emerging applications in biological imaging. Traditional ECL simulations rely on finite element methods, which provide valuable insights into reaction dynamics and spatial distribution of species. However, such methods are limited in mesoscopic environments where stochastic effects become significant. Here, I present a novel approach using ChatGPT o1 to generate a Python-based stochastic simulation for ECL reactions in a nanofluidic channel, incorporating diffusion, electrochemical and chemical reactions, and photon emission. The simulation successfully replicates results from finite element models while offering additional insights into time-dependent behaviors and enabling noise analysis for simulated luminescence traces. The iterative development of this simulation using ChatGPT was rapid, requiring minimal coding expertise while leveraging the model's "reasoning" capabilities to implement physical principles, verify calculations, and optimize performance. This work demonstrates that large language models (LLMs) can serve as effective co-intelligence tools, facilitating the development of complex simulations in electrochemistry. AI-driven tools/LLMs have a promising role in advancing electrochemistry research, though careful validation remains essential to ensure scientific accuracy.

KEYWORDS: *Electrochemiluminescence, ChatGPT, Artificial Intelligence, Stochastic Chemistry, Nanofluidics*

■ INTRODUCTION

Electrochemiluminescence (ECL) describes the generation of luminescent light in solution, initiated by electrochemical reactions.[1−3] ECL serves as an important tool in analytical chemistry, especially in clinical immunosensing applications.[4,5] Additionally, ECL is being explored for innovative biological imaging techniques[6,7] and utilized in fundamental studies of reactions at mesoscopic concentrations[8,9] and even at the single-particle level.[10,11] Numerical simulation tools, such as finite element methods (e.g., COMSOL Multiphysics), are commonly employed in ECL research.[7,9,12−17] These tools are highly effective for elucidating the underlying physical chemistry of ECL processes, as they incorporate the complex interactions between electrochemical and chemical reactions, mass transport of reactants, and the lifetimes of luminescent ECL radicals. Through simulations, researchers can obtain cyclic voltammograms, light intensity profiles, and spatial distributions of the molecules involved.

However, traditional simulation methods, including finite difference and finite element approaches, inherently rely on continuous differential equations. These methods are limited when the continuum approximation no longer holds—such as in mesoscopic or stochastic environments where only a small number of particles are involved and the fluctuation of their number density or stochasticity of reaction events becomes significant. As a result, these tools are unsuitable for simulating mesoscopic or single-particle ECL. Moreover, professional finite element software is very costly, and building custom tools requires extensive programming expertise and development time.

In this work, I present a novel approach to ECL simulation using ChatGPT[18] to generate a Python script that models annihilation ECL reactions in a nanofluidic channel. This script is based on a a two-dimensional random walk algorithm and accounts for both electrochemical and chemical reactions as well as light emission. The results of this simulation align fully with those obtained from a prior finite element analysis[9] of the same system, and they reveal dynamic, time-dependent behavior. Furthermore, because this approach evaluates the stochastic behavior of diffusing and reacting particles, it enables noise analysis—such as determining the power spectral density of luminescence-intensity time traces.

Using a large language model[19] (LLM) like ChatGPT for code generation facilitates rapid simulation development without the need for advanced coding skills. However, a strong understanding of chemistry and physics is essential to effectively work with ChatGPT as a "co-intelligence."[20] This means that an

Received: November 12, 2024
Revised: December 12, 2024
Accepted: December 13, 2024

iterative development process is used that involves progressively increasing model complexity and verifying results at each step.

Beyond being a mere tool for code development, ChatGPT o1-preview demonstrates a remarkable grasp of physics and chemistry. The model is offering insightful, accurate suggestions for implementing chemical reactions, and it is using its "reasoning" capability to perform performs logical checks, such as dimensional analysis, to automatically identify and correct potential errors within the simulation. While nanofluidic annihilation ECL is simulated here, this simulation script can be adapted, or new scripts can be developed by prompting LLMs. Adaptions could include, e.g., ECL-coreactant reaction pathways and bioassays, simulations of redox mediators, or new functionalities such as varying electrode potentials.

■ RESULTS AND DISCUSSION

Nanofluidic Annihilation Electrochemiluminescence. The electrochemiluminescence (ECL) reaction sequence employed here is illustrated in Figure 1a, following a scheme we had previously investigated.[9] In this framework, ECL-active tris(bipyridine)ruthenium(II), or $[Ru(bpy)_3]^{2+}$, is suspended in solution within a nanofluidic channel with a height of 100 nm. Along a 20 μm channel segment, electrodes are embedded in the channel's ceiling and floor; each electrode is individually addressed. $[Ru(bpy)_3]^{2+}$ undergoes oxidation at the top electrode and reduction at the bottom electrode:

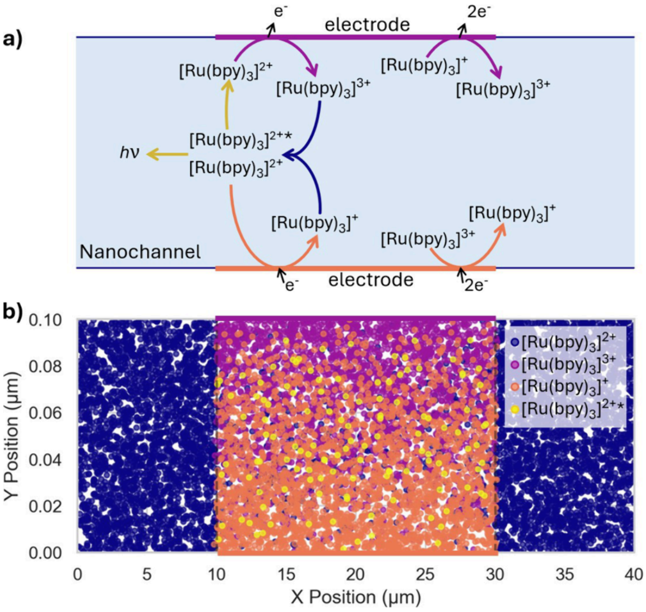


Figure 1. a) Schematic of electrochemiluminescence annihilation reaction in a nanofluidic channel. **b)** Corresponding stochastic simulation of ions undergoing a random walk as well as reacting in a two-dimensional channel geometry.

$$[Ru(bpy)_3]^{2+} + e^- \rightarrow [Ru(bpy)_3]^{+} \quad (1)$$

$$[Ru(bpy)_3]^{2+} - e^- \rightarrow [Ru(bpy)_3]^{3+} \quad (2)$$

The resulting $[Ru(bpy)_3]^{+}$ and $[Ru(bpy)_3]^{3+}$ ions diffuse across the channel, eventually meeting in the center to participate in the annihilation reaction:

$$[Ru(bpy)_3]^{+} + [Ru(bpy)_3]^{3+} \xrightarrow{k_{ann}} [Ru(bpy)_3]^{2+} + [Ru(bpy)_3]^{2+*} \quad (3)$$

This reaction occurs at a rapid rate, $k_{ann} = 10^{10}$ M^{-1} s^{-1}. The excited radical $[Ru(bpy)_3]^{2+*}$ then returns into its ground state, emitting light

$$[Ru(bpy)_3]^{2+*} \xrightarrow{k_{decay}} [Ru(bpy)_3]^{2+} + h\nu \quad (4)$$

after a characteristic lifetime of 1 μs ($k_{decay} = 10^6$ s^{-1}).

Through this sequence, $[Ru(bpy)_3]^{2+}$ completes a redox cycle, repeating the oxidation, reduction, annihilation, and light emission processes. The confined nanofluidic environment promotes rapid cycling due to the short diffusion distances, resulting in enhanced light emission compared to macroscale systems.

Some $[Ru(bpy)_3]^{+}$ and $[Ru(bpy)_3]^{3+}$, however, may evade annihilation and instead diffuse across the channel to reach the opposite electrode, where they undergo the following reactions:

$$[Ru(bpy)_3]^{3+} + 2e^- \rightarrow [Ru(bpy)_3]^{+}, \text{ and} \quad (5)$$

$$[Ru(bpy)_3]^{+} - 2e^- \rightarrow [Ru(bpy)_3]^{3+} \quad (6)$$

Due to the nanochannel's confined dimensions, the ECL process operates within a mesoscopic or stochastic regime. For example, at a 1 μM concentration, on average only 3600 ions are present in a volume of 100 nm x 20 μm × 3 μm (equating to 6 fL). This system cannot be fully described by conventional concentrations. Instead, it exhibits fluctuations inherent to the discrete nature of molecular populations.

Simulation-Script Generation with ChatGPT o1. Figure 1b illustrates the end point of a simulation, showing the distribution of four different types of ECL compounds in a two-dimensional geometry with dimensions of 40 μm in length and 100 nm in height. The simulation begins with a random distribution of $[Ru(bpy)_3]^{2+}$ particles undergoing a random walk. In the segment spanning from 10 μm to 30 μm, electrochemical reactions 1 and (5) occur at the bottom of the geometry, while reactions 2 and (6) take place at the top. The annihilation reaction 3 and subsequent decay and luminescence (4) proceed throughout the entire geometry. The Python script for this simulation is provided in the Supporting Information and available at: 10.6084/m9.figshare.27620118.v1.

The script was developed using ChatGPT o1[21] (OpenAI, USA. Model "o1-preview" was used, the full version of model "o1" was released in the meantime in December 2024) with an iterative approach: I began with the prompt "generate 2-dimensional random walk Python script," reviewed the script and its output, and progressively added complexity and functionality through successive prompts. Prompts to implement main characteristics of the simulation of boundary and starting condition of the random walk, of electrochemical reactions, and of the annihilation reaction are

- "Limit random walk to 2D box, add starting parameters of box dimension, diffusion coefficient in m2/s and distribute particles randomly."
- "Change the boundary condition at the top boundary: particle that are being reflected there change and become a different type: Particle 2."
- "all the particles are molecules. can you also implement a chemical reaction of the particles within the box: when particle 1 and particle 3 collide with one another in the

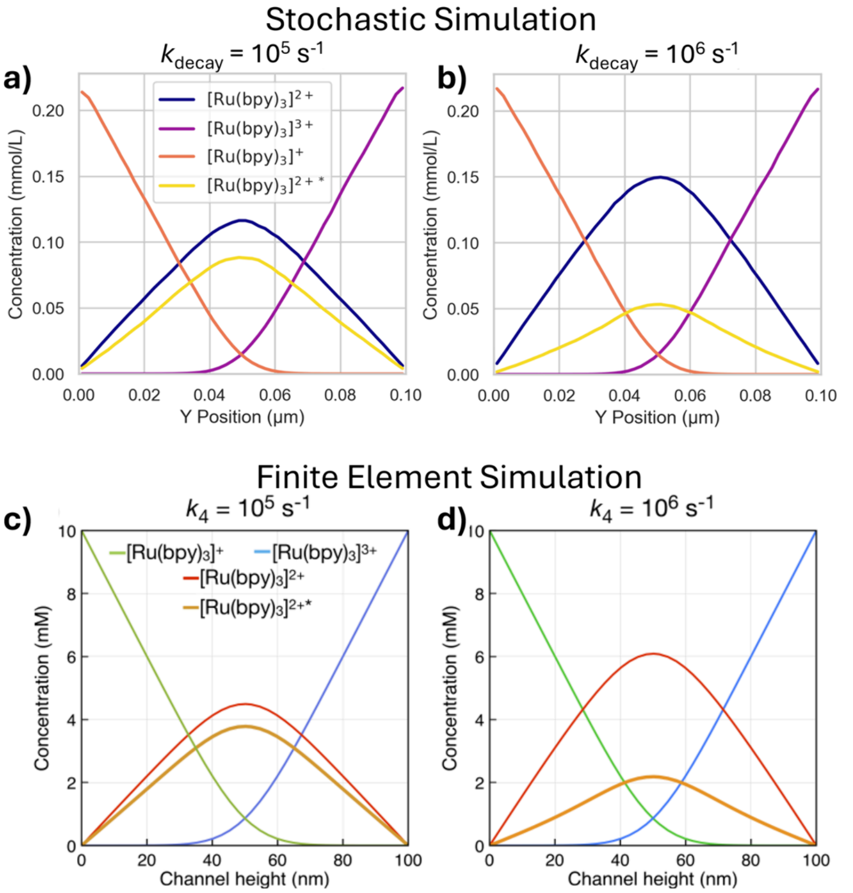


Figure 2. Comparison of concentration profile across the nanochannel obtained by stochastic vs. finite element simulation **a)** Stochastic random walk simulation using a decay rate constant of $k_{decay} = 10^5\ s^{-1}$, and **b)** $k_{decay} = 10^6\ s^{-1}$. **c, d)** Corresponding finite element (COMSOL) calculations for the same two radical lifetimes/rates for identical conditions (and higher concentrations). Panels c) and d) are reproduced from Ref 9. with permission from the Royal Society of Chemistry.

> box, they react to particle 2 and particle 4. Make suggestions how to implement the collision and reaction probability in the code. Is it possible to have a rate constant as starting parameters (in M-1 s-1)?"

Prior to this work, I had no experience with stochastic simulations of chemical reactions or Python coding (though I have recently used ChatGPT for random walk modeling[22]). Consequently, initially I did not know how to calculate and implement a collision radius for $[Ru(bpy)_3]^+$ and $[Ru(bpy)_3]^{3+}$ ions—an essential factor for simulating annihilation reactions (unlike in finite element simulations, where the reaction rate constant k_{ann} suffices without explicit collisions). To address this challenge, I asked ChatGPT to explain the concept of collision radius and integrate it into the code. The resulting explanation and code are provided in full in the Supporting Information. This interaction highlights ChatGPT o1's utility beyond programming: the model demonstrates a solid understanding of the underlying physics and chemistry and directly implements it in the script.

While large language models (LLMs) can sometimes produce "hallucinations"[23] (i.e., confident but incorrect outputs), ChatGPT o1-preview showed minimal errors in this project. As an example (see Supporting Information), the initial calculation of the collision radius was incorrect; however, the model identified the issue through a "sanity check," recognizing that the computed radius exceeded the simulation environment dimensions, and subsequently applied a dimensional analysis to correct the calculation. This level of "thinking along" suggests reasoning at the competence level of a skilled MSc student.

An additional example of the model's impressive response was its unprompted optimization of the script (see Supporting Information). When I increased the number of particles to 10,000, ChatGPT o1 automatically incorporated a spatial grid to optimize collision detection, significantly reducing computation time.

Overall, setting up the simulation, including plotting and luminescence generation along with noise analysis (see below), required approximately two working days. The final Python script serves as a robust starting point for running simulations under various conditions (e.g., altering geometry, particle numbers, random walk step length, and reaction rates) to explore parameter space. This subsequent exploration was conducted independently, without further assistance from ChatGPT.

Simulation Results. The initial validation of the random walk simulation is presented in Figure 2, which shows the steady-state concentration profiles of all four ECL compounds across the nanochannel. These profiles were obtained by averaging over a central segment (in the x-direction) of the channel. Profiles for two different lifetimes of the radical $[Ru(bpy)_3]^{2+*}$ are shown in Figures 2a and 2b and are compared to profiles from a previous finite element calculation[9] under identical conditions in Figures 2c and 2d. For both the stochastic and COMSOL simulations, diffusion coefficients $D = 10^{-9}\ m^2/s$

were chosen for all compounds. The time step Δt was set to 0.001 μs to ensure that the Brownian step length $\Delta x = \sqrt{(2D\Delta t)}$ = 1.4 nm is small compared to the 100 nm channel diameter. In the stochastic simulation, computational limits restrict the number of particles and concentration. Here, the simulation was scaled to over 100,000 particles (~0.1 mM), approximately two orders of magnitude lower than in experiments and finite element simulations, but sufficient to ensure that $[\mathrm{Ru(bpy)_3}]^{3+}$ and $[\mathrm{Ru(bpy)_3}]^{+}$ densities allowed reaction 3 to occur in the channel center and prevented electrochemically generated particles from reaching the opposite electrode. When comparing results between stochastic and finite element methods for decay rates $k_{\mathrm{decay}} = 10^5\ \mathrm{s}^{-1}$ (Figures 2a and 2c) and $10^6\ \mathrm{s}^{-1}$ (Figures 2b and 2d), both approaches produced fully consistent results, affirming the accuracy of ChatGPT's implementation of the physical chemistry in the Python script, without errors or "hallucinations." A quantitative comparison of these stochastic and finite element results is given in the Supporting Information.

The random walk simulation, being inherently time-dependent, enables visualization of concentration evolution upon biasing both electrodes (i.e., starting with only $[\mathrm{Ru(bpy)_3}]^{2+}$). This evolution is shown in Figure 3, where Figure 3a presents particle distribution snapshots at different time points, and Figure 3b shows concentration−time traces averaged over the area between the two electrodes. Upon initiating the electrochemical reactions, the $[\mathrm{Ru(bpy)_3}]^{2+}$ concentration declines as ions are reduced (reaction 1) and oxidized (reaction 2), forming equal amounts of $[\mathrm{Ru(bpy)_3}]^{+}$ and $[\mathrm{Ru(bpy)_3}]^{3+}$. The generation of the radical $[\mathrm{Ru(bpy)_3}]^{2+*}$ is delayed by approximately 0.5 μs, as $[\mathrm{Ru(bpy)_3}]^{+}$ and $[\mathrm{Ru(bpy)_3}]^{3+}$ must first diffuse to the center of the nanochannel for reaction 3 to proceed. This delay aligns well with the estimated diffusion time across half the channel, $(50\ \mathrm{nm})^2/(2D) = 1.25\ \mu\mathrm{s}$. Concentrations reach steady state after approximately 2 μs, corresponding to the diffusion time across the full channel width, $(100\ \mathrm{nm})^2/(2D) = 5\ \mu\mathrm{s}$. The concentration of $[\mathrm{Ru(bpy)_3}]^{2+*}$ remains consistently lower than that of other ions due to its decay into $[\mathrm{Ru(bpy)_3}]^{2+}$, as described in eq 4.

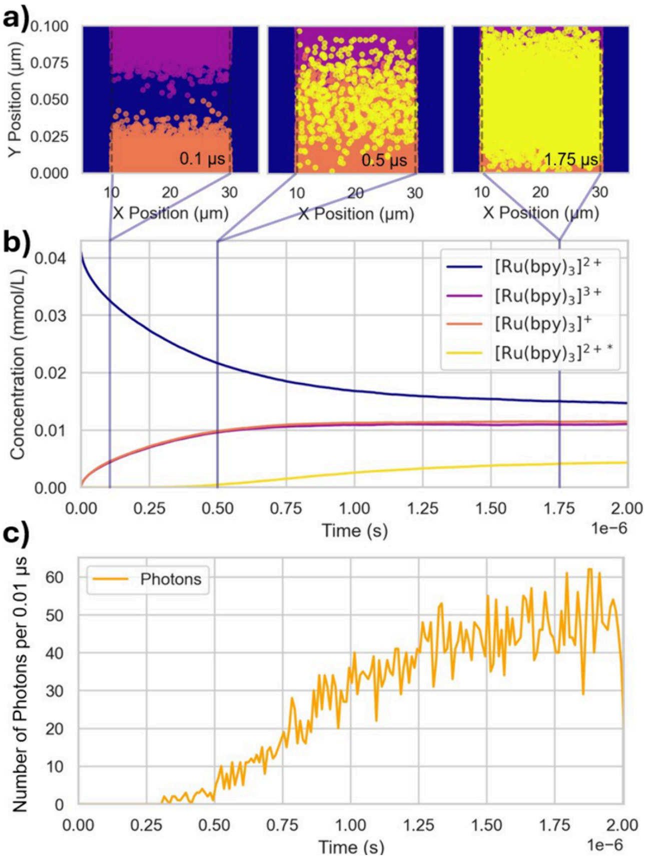


Figure 3. Time dependent simulations. **a**) Snapshot of molecular distributions at 0.1 μs, 0.5 μs and 1.75 μs. **b**) Time evolution of concentration in the area between electrodes. **b**) Corresponding time evolution of luminescence/number of photons. (100,000 particle overall are simulated at 0.001 μs time steps, binned in 0.01 μs intervals in panel c.).

In Figure 3b, the number of photons generated over time is shown. Photon intensity begins to stabilize after about 1.5 μs, combining the diffusion time (~0.5 μs) and the radical's lifetime (~1 μs). Since photons escape the channel instantaneously upon generation, their numbers remain low, and the intensity-time trace exhibits considerable noise, reflecting the stochastic behavior of a small number of molecules in nanoscale confinement.

Unlike continuous finite-element methods, stochastic simulations allow for noise analysis on time traces such as the determination of power spectral density and autocorrelation functions in the frequency and time domains, respectively. In nanofluidic electrochemistry, such spectroscopy analyses[24−26] have been used to determine diffusion coefficients,[24,27] adsorption,[28] and flow.[29] Power spectral density analysis of the simulation data is shown in Figure 4. Figure 4a displays the power spectrum of particle numbers present in the area between the electrodes, revealing the $1/f^{3/2}$ noise characteristic of Brownian motion in nanochannels.[24] In Figure 4b, the power spectrum of a luminescence/photon-time trace is presented, showing white noise—consistent with the random and uncorrelated nature of radical decay described in eq 4 (see Supporting Information of ref 10 for exemplary experimental ECL spectra).

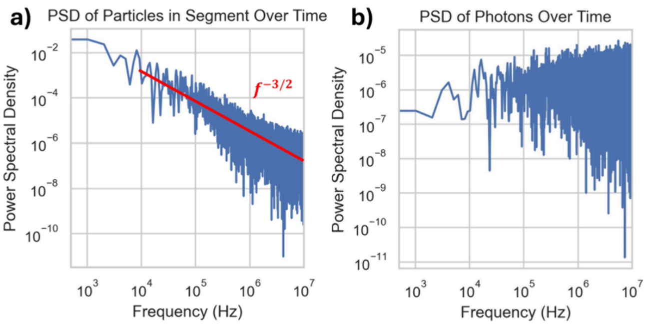


Figure 4. Power spectral density analysis. **a**) Power spectrum of all particles positioned over time in the area between the electrodes. **b**) Power spectrum of a photon-time-trace. (Overall, 30,000 particles were simulation with 0.05 μs long time steps for 20,000 steps. The last 19,500 steps after reaching the steady state were used for analysis.)

■ CONCLUSION

This work presents a novel approach to simulating electrochemiluminescence (ECL) reactions by using ChatGPT-generated Python scripts to model diffusion, electrochemical and chemical reactions, and photon generation. The simulation results closely match those obtained from finite element calculations, and the stochastic simulation provides additional insights into the time-dependent behavior of ECL reactions, and it enables noise and fluctuation analysis of simulated time traces.

Employing AI for simulation development significantly accelerates the process, requiring minimal coding experience. Beyond basic implementation, the ChatGPT o1-preview model demonstrates valuable “reasoning” capabilities, functioning as a co-intelligence. The model not only assists with coding but also aids in understanding and explaining the underlying physical chemistry, verifying the accuracy of the code, chemical principles, equations, and calculations. Thus, I encourage exploring ChatGPT or other LLMs for modelling (electro-)chemical systems with broad applicability such as sensors and batteries—in stochastic as well as continuous regimes.

In summary, this study highlights the potential of AI and large language models (LLMs) as powerful tools in electrochemistry research. I anticipate broader adoption of AI in this field, anticipating its growing impact. However, given the current limitations of LLMs, it is essential to view them as supportive tools rather than definitive sources. Careful review and validation of AI-generated output are crucial to ensure accurate and reliable scientific results.

ASSOCIATED CONTENT

Supporting Information

The Supporting Information is available free of charge at https://pubs.acs.org/doi/10.1021/acselectrochem.4c00166.

Quote of “reasoning” response of ChatGPT o1; Quote of response of ChatGPT o1 on optimizing computational performance; Quantitative comparison of COMSOL and Phyton results; Python script (PDF)

AUTHOR INFORMATION

Corresponding Author

Klaus Mathwig − *imec within OnePlanet Research Center, 6708 WH Wageningen, the Netherlands;* orcid.org/0000-0002-8532-8173; Email: klaus.mathwig@imec.nl

Complete contact information is available at:
https://pubs.acs.org/10.1021/acselectrochem.4c00166

Funding

I acknowledge financial support by Provincie Gelderland. This work was supported by the ECLectic project that has received funding from the European Union’s MSCA Doctoral Network Horizon Europe programme Grant Agreement Number 101119951.

Notes

The author declares no competing financial interest.

REFERENCES

(1) *Analytical Electrogenerated Chemiluminescence: From Fundamentals to Bioassays*; Sojic, N., Ed.; The Royal Society of Chemistry, 2019.

(2) Forster, R. J.; Bertoncello, P.; Keyes, T. E. Electrogenerated Chemiluminescence. *Annual Review of Analytical Chemistry* **2009**, *2* (1), 359−385.

(3) Rassaei, L.; Xu, G.; Ding, Z.; Mathwig, K. Electrochemiluminescence: Fundamentals to Applications. *ChemElectroChem.* **2017**, *4* (7), 1571.

(4) Yu, J.; Stankovic, D.; Vidic, J.; Sojic, N. Recent Advances in Electrochemiluminescence Immunosensing. *Sensors & Diagnostics* **2024**, *3*, 1887.

(5) Faatz, E.; Finke, A.; Josel, H.-P.; Prencipe, G.; Quint, S.; Windfuhr, M. Automated Immunoassays for the Detection of Biomarkers in Body Fluids. In *Analytical Electrogenerated Chemiluminescence: From Fundamentals to Bioassays*; RSC, 2019; pp 443−470. DOI: 10.1039/9781788015776-0044

(6) Voci, S.; Goudeau, B.; Valenti, G.; Lesch, A.; Jović, M.; Rapino, S.; Paolucci, F.; Arbault, S.; Sojic, N. Surface-Confined Electrochemiluminescence Microscopy of Cell Membranes. *J. Am. Chem. Soc.* **2018**, *140* (44), 14753−14760.

(7) Valenti, G.; Scarabino, S.; Goudeau, B.; Lesch, A.; Jović, M.; Villani, E.; Sentic, M.; Rapino, S.; Arbault, S.; Paolucci, F.; Sojic, N. Single Cell Electrochemiluminescence Imaging: From the Proof-of-Concept to Disposable Device-Based Analysis. *J. Am. Chem. Soc.* **2017**, *139* (46), 16830−16837.

(8) Collinson, M. M.; Wightman, R. M. Observation of Individual Chemical Reactions in Solution. *Science* **1995**, *268*, 1883−1885, DOI: 10.1126/science.268.5219.1883.

(9) Al-Kutubi, H.; Voci, S.; Rassaei, L.; Sojic, N.; Mathwig, K. Enhanced Annihilation Electrochemiluminescence by Nanofluidic Confinement. *Chem. Sci.* **2018**, *9* (48), 8946−8950.

(10) Zhang, Z.; Dong, J.; Yang, Y.; Zhou, Y.; Chen, Y.; Xu, Y.; Feng, J. Direct Probing of Single-Molecule Chemiluminescent Reaction Dynamics under Catalytic Conditions in Solution. *Nat. Commun.* **2023**, *14* (1) DOI: 10.1038/s41467-023-43640-1.

(11) Chang, Y.-L.; Palacios, R. E.; Fan, F.-R. F.; Bard, A. J.; Barbara, P. F. Electrogenerated Chemiluminescence of Single Conjugated Polymer Nanoparticles. *J. Am. Chem. Soc.* **2008**, *130* (28), 8906−8907.

(12) Huang, X.; Li, B.; Lu, Y.; Liu, Y.; Wang, S.; Sojic, N.; Jiang, D.; Liu, B. Direct Visualization of Nanoconfinement Effect on Nanoreactor via Electrochemiluminescence Microscopy. *Angew. Chem., Int. Ed.* **2023**, *62* (6), No. e202215078, DOI: 10.1002/anie.202215078.

(13) Klymenko, O. V.; Svir, I.; Amatore, C. A New Approach for the Simulation of Electrochemiluminescence (ECL). *ChemPhysChem* **2013**, *14* (10), 2237−2250.

(14) Mathwig, K.; Sojic, N. Towards Determining Kinetics of Annihilation Electrogenerated Chemiluminescence by Concentration-Dependent Luminescent Intensity. *J. Anal Test* **2019**, *3* (2), 160−165.

(15) Voci, S.; Al-Kutubi, H.; Rassaei, L.; Mathwig, K.; Sojic, N. Electrochemiluminescence Reaction Pathways in Nanofluidic Devices. *Anal Bioanal Chem.* **2020**, *412* (17), 4067−4075.

(16) Douman, S. F.; Collins, D.; Cumba, L. R.; Beirne, S.; Wallace, G. G.; Yue, Z.; Iwuoha, E. I.; Melinato, F.; Pellegrin, Y.; Forster, R. J. Wireless Electrochemiluminescence at Functionalised Gold Microparticles Using 3D Titanium Electrode Arrays. *Chem. Commun.* **2021**, *57* (38), 4642−4645.

(17) Svir, I. B.; Oleinick, A. I.; Klimenko, A. V. ‘ECL-PACKAGE’-Software for Electrochemiluminescence Simulation at Microelectrodes. *J. Electroanal. Chem.* **2001**, *513* (2), 119−125, DOI: 10.1016/S0022-0728(01)00608-8.

(18) Jones, N. ‘In Awe’: Scientists Impressed by Latest ChatGPT Model O1. *Nature* **2024**, *634*, 275.

(19) Bishop, C. M.; Bishop, H. Transformers. In *Deep Learning*; Springer International Publishing: Cham, Switzerland, 2024; pp 357−406 DOI: 10.1007/978-3-031-45468-4.

(20) Mollick, E. *Co-Intelligence: Living and Working with AI*; Random House, 2024.

(21) Open AI o1-preview, https://openai.com/index/introducing-openai-o1-preview/.

(22) Mathwig, K. AI-Driven Random Walk Simulations of Viscophoresis and Visco-Diffusiophoretic Particle Trapping. *arXiv* 2024. DOI: 10.48550/arXiv.2410.11481.

(23) Hicks, M. T.; Humphries, J.; Slater, J. ChatGPT Is Bullshit. *Ethics Inf. Technol.* **2024**, *26* (2) DOI: 10.1007/s10676-024-09775-5.

(24) Zevenbergen, M. A. G.; Singh, P. S.; Goluch, E. D.; Wolfrum, B. L.; Lemay, S. G. Electrochemical Correlation Spectroscopy in Nanofluidic Cavities. *Anal Chem.* **2009**, *81* (19), 8203−8212.

(25) Kätelhön, E.; Krause, K. J.; Mathwig, K.; Lemay, S. G.; Wolfrum, B. Noise Phenomena Caused by Reversible Adsorption in Nanoscale Electrochemical Devices. *ACS Nano* **2014**, *8* (5), 4924−4930.

(26) Krause, K. J.; Mathwig, K.; Wolfrum, B.; Lemay, S. G. Brownian Motion in Electrochemical Nanodevices. *European Physical Journal:*

Special Topics **2014**, *223*, 3165−3178, DOI: 10.1140/epjst/e2014-02325-5.

(27) Mathwig, K.; Zafarani, H. R.; Speck, J. M.; Sarkar, S.; Lang, H.; Lemay, S. G.; Rassaei, L.; Schmidt, O. G. Potential-Dependent Stochastic Amperometry of Multiferrocenylthiophenes in an Electrochemical Nanogap Transducer. *J. Phys. Chem. C* **2016**, *120* (40), 23262−23267.

(28) Mampallil, D.; Mathwig, K.; Kang, S.; Lemay, S. G. Reversible Adsorption of Outer-Sphere Redox Molecules at Pt Electrodes. *J. Phys. Chem. Lett.* **2014**, *5* (3), 636−640.

(29) Mathwig, K.; Mampallil, D.; Kang, S.; Lemay, S. G. Electrical Cross-Correlation Spectroscopy: Measuring Picoliter-per-Minute Flows in Nanochannels. *Phys. Rev. Lett.* **2012**, *109* (11), No. 118302, DOI: 10.1103/PhysRevLett.109.118302.

Supporting Information for:

AI-driven simulation of stochastic electrochemiluminescence

Klaus Mathwig
imec within OnePlanet Research Center, Bronland 10, 6708 WH Wageningen, the Netherlands
klaus.mathwig@imec.nl

Contents

1. Quote of "reasoning" response of ChatGPT o1

A prompt and response are quoted. Previously, had simulated nanofluidic ECL using finite element calculation (COMSOL) based on solving continuous differential equation. Therefore, I was not familiar with the method to calculate the collision of two ions (prior to then annihilation reaction eqn. (3)). Thus, I ask ChatGPT to explain this concept to me and to implement in the Python scritp.

With the answer and reasoning response quoted here, in several instances ChatGPT recognizes that results are nonphysical and corrected the calculation approach. These recognitions and double checking are highlighted in green.

Prompt:

"can you explain to me: how to choose best the value for a collision radius. In a continuous simulation such as finite elements, the reaction between two particles in solution is simply characterized by the rate constant. In this random walk simulation, a collision radius needs to be defined though. What is a realistic values for this for small molecules in solution?"

Response ChatGPT o1-preview:

Thought for 6 seconds

> OK, let me see. The assistant is addressing how to select a collision radius in a finite elements simulation, focusing on realistic values for small molecules in solution within a random walk simulation context.
>
> **Choosing collision radius**
>
> I'm thinking through how to select a collision radius for random walk simulations. This involves defining when particles can react, essential for accurately modeling chemical reactions in simulations.
>
> **Connecting concepts**
>
> I'm piecing together the collision radius, diffusion coefficients, and reaction rate constants in particle-based simulations, linking them through the Smoluchowski equation. This involves understanding how these elements interact within diffusion-controlled reactions.

Certainly! I'd be happy to explain how to choose an appropriate collision radius for your random walk simulation and discuss realistic values for small molecules in solution.

Understanding the Collision Radius in Particle-Based Simulations

In particle-based simulations like random walks or molecular dynamics, the **collision radius** (also known as the **reaction radius**) is a critical parameter that defines the distance within which two particles are considered close enough to undergo a reaction.

In continuous models (e.g., finite element methods), reactions are characterized by rate constants without explicitly modeling individual particle interactions. However, in discrete simulations, we need to define the conditions under which reactions occur at the particle level.

Relating Reaction Rate Constants to Collision Radius

To bridge the gap between macroscopic reaction rate constants and microscopic particle interactions, we can use the **Smoluchowski theory of diffusion-controlled reactions**. This theory relates the diffusion coefficients of the reacting species, the reaction (collision) radius, and the macroscopic reaction rate constant.

The Smoluchowski Equation

For a diffusion-controlled bimolecular reaction between two spherical particles in three dimensions:

$$\mathrm{A} + \mathrm{B} \rightarrow \mathrm{Products}$$

The **Smoluchowski rate constant** $k_{\mathrm{Smoluchowski}}$ is given by:

$$k_{\mathrm{Smoluchowski}} = 4\pi(D_{\mathrm{A}} + D_{\mathrm{B}})R$$

- D_{A} and D_{B} are the diffusion coefficients A and B, respectively.
- R is the sum of the radii of particles A and B (the collision radius).
- $k_{\mathrm{Smoluchowski}}$ has units of m^3/s, to $\mathrm{M}^{-1}\mathrm{s}^{-1}$ when considering molar concentrations.

Calculating the Collision Radius

If you have a desired reaction rate constant k and know the diffusion coefficients of the particles, you can rearrange the Smoluchowski equation to solve for R:

$$R = \frac{k}{4\pi(D_{\mathrm{A}} + D_{\mathrm{B}})}$$

This equation allows you to calculate the collision radius that corresponds to a given macroscopic reaction rate constant.

Example Calculation for Small Molecules in Solution

Let's perform an example calculation to find a realistic collision radius for small molecules in aqueous solution.

Given Parameters

- **Diffusion Coefficients**:
  - Small organic molecules in water at room temperature typically have diffusion coefficients D_{A} and D_{B} around 1×10^{-9} m^2/s.
- **Reaction Rate Constant**:
  - For diffusion-limited reactions between small molecules, the reaction rate constant k is often on the order of 10^9 $\mathrm{M}^{-1}\mathrm{s}^{-1}$ to 10^{10} $\mathrm{M}^{-1}\mathrm{s}^{-1}$.

Converting Units

First, ensure all units are consistent:

- $D_{\mathrm{A}} + D_{\mathrm{B}} = 2 \times 1 \times 10^{-9}\ \mathrm{m}^2/\mathrm{s} = 2 \times 10^{-9}\ \mathrm{m}^2/\mathrm{s}$
- $k = 1 \times 10^{10}\ \mathrm{M}^{-1}\mathrm{s}^{-1}$.

Note that 1 M = 1 mol/L = 1 × 10^3 $\mathrm{mol/m}^3$.

Therefore, k in units of $\mathrm{m}^3/\mathrm{mol} \cdot \mathrm{s}$ is:

$$k = 1 \times 10^{10}\ \mathrm{M}^{-1}\mathrm{s}^{-1} = 1 \times 10^{10} \times \frac{1}{10^3}\ \mathrm{m}^3/\mathrm{mol} \cdot \mathrm{s} = 1 \times 10^{7}\ \mathrm{m}^3/\mathrm{mol} \cdot \mathrm{s}$$

Calculating the Collision Radius R

Plugging the values into the rearranged Smoluchowski equation:

$$R = \frac{k}{4\pi(D_{\mathrm{A}} + D_{\mathrm{B}})} = \frac{1 \times 10^7}{4\pi \times 2 \times 10^{-9}} = \frac{1 \times 10^7}{8\pi \times 10^{-9}}$$

Simplify:

$$R = \frac{1 \times 10^{7}}{8\pi \times 10^{-9}} = \frac{1 \times 10^{7}}{25.1327 \times 10^{-9}} = \frac{1 \times 10^{7}}{25.1327 \times 10^{-9}}$$

Compute the denominator:

$$8\pi \times 10^{-9} = 25.1327 \times 10^{-9}$$

Now compute R:

$$R = \frac{1 \times 10^{7}}{25.1327 \times 10^{-9}} = \left(\frac{1 \times 10^{7}}{25.1327}\right) \times 10^{-9}\ \mathrm{m}$$

Calculate the value:

$$\frac{1 \times 10^{7}}{25.1327} \approx 397887.4$$

Therefore:

$$R \approx 397887.4 \times 10^{9}\ \mathrm{m}\ [\mathrm{Sic}] = 3.978874 \times 10^{-4}\ \mathrm{m}$$

But this result doesn't make physical sense because the radius is larger than the simulation box.

Identifying the Issue

The discrepancy arises because we made an error in unit conversion. Let's correct the calculation.

Correct Unit Conversion for k

We need to convert k from M^{-1}s^{-1} to m^{3}/particle · s because in the Smoluchowski equation, k should be in units of m^{3}/s when dealing with particle densities (number of particles per unit volume).

Alternatively, we can adjust the calculation by considering that M = mol/L =10^{3} mol/m^{3}, so:

$$\mathrm{M}^{-1}\mathrm{s}^{-1} = \frac{1}{1\ \mathrm{mol/L} \cdot \mathrm{s}} = \frac{1}{1\ \mathrm{mol/m}^{3} \cdot \mathrm{s}} = 10^{-3}\ \mathrm{m}^{3}/\mathrm{mol} \cdot \mathrm{s}$$

So a rate constant $k = 1 \times 10^{10}\ \mathrm{M}^{-1}\mathrm{s}^{-1}$ is:

$$k = 1 \times 10^{10} \times 10^{-3} = 1 \times 10^{7}\ \mathrm{M}^{-1}\ \mathrm{m}^{3}/\mathrm{mol} \cdot \mathrm{s}$$

But this still leads to unphysical results because the calculated collision radius is too large.

Re-expressing the Smoluchowski Rate Constant in Correct Units

Actually, the Smoluchowski rate constant k in SI units is:

$$k_{\mathrm{SI}} = 4\pi N_{\mathrm{A}}(D_{\mathrm{A}} + D_{\mathrm{B}})R$$

Where:

- N_{A} is Avogadro's number (6.022 × 10^{23} mol^{-1})
- D_{A} and D_{B} are in $\mathrm{m}^{2}/\mathrm{s}$
- R is in meters
- k_{SI} is in $\mathrm{m}^{3}/\mathrm{mol} \cdot \mathrm{s}$

So the correct formula is:

$$k = 4\pi N_{\mathrm{A}}(D_{\mathrm{A}} + D_{\mathrm{B}})R$$

Correct Calculation

Let's plug in the correct values:

- $N_{\mathrm{A}} = 6.022 \times 10^{23}\ \mathrm{mol}^{-1}$
- $D_{\mathrm{A}} + D_{\mathrm{A}} = 2 \times 1 \times 10^{-9} = 2 \times 10^{-9}\ \mathrm{m}^{2}/\mathrm{s}$
- Desired $k = 1 \times 10^{10}\ \mathrm{M}^{-1}\mathrm{s}^{-1} = 1 \times 10^{7}\ \mathrm{m}^{3}/\mathrm{mol} \cdot \mathrm{s}$ (as before)

Now, rearranged equation:

$$R = \frac{k}{4\pi N_{\mathrm{A}}(D_{\mathrm{A}} + D_{\mathrm{B}})}$$

Plug in the values:

$$R = \frac{1 \times 10^{7}}{4\pi \times 6.022 \times 10^{23} \times 2 \times 10^{-9}}$$

sSimplify the denominator:

$$\text{Denominator} = 4\pi \times 6.022 \times 10^{23} \times 2 \times 10^{-9} = 4\pi \times 6.022 \times 2 \times 10^{14}$$

Calculate the constants:

$$4\pi \times 6.022 \times 2 = 4 \times 3.1416 \times 6.022 \times 2 \approx 4 \times 3.1416 \times 12.044 \approx 4 \times 3.1416 \times 12.044 \approx 4 \times 37.699 \approx 150.796$$

But this is not the correct approach.

Wait, let's compute the denominator correctly.

Denominator:

$$4\pi N_\mathrm{A}(D_\mathrm{A} + D_\mathrm{B}) = 4\pi \times 6.022 \times 10^{23} \times 2 \times 10^{-9} = 4\pi \times 6.022 \times 10^{23} \times 2 \times 10^{-9}$$

Simplify the exponents:

$$\text{Denominator} = 4\pi \times 6.022 \times 2 \times 10^{23} \times 10^{-9} = 4\pi \times 12.044 \times 10^{14}$$

Now compute:

$$4\pi \times 12.044 \times 2 = 4 \times 3.1416 \times 12.044 \approx 4 \times 3.1416 \times 12.044 \approx 4 \times 378576 \approx 151.4304$$

Thus:

$$\text{Denominator} = 151.4304 \times 10^{14} = 1.515304 \times 10^{16}$$

Now compute R:

$$R = \frac{1 \times 10^{7}}{1.515304 \times 10^{16}} = 6.603 \times 10^{-10}\ \mathrm{m}$$

Which is:

$$R = 6.603 \times 10^{-10}\ \mathrm{m} = 0.6603\ \mathrm{nm}$$

This value seems more reasonable.

But let's double-check the calculation step by step.

Rewriting the Denominator:

$$\text{Denominator} = 4\pi N_\mathrm{A}(D_\mathrm{A} + D_\mathrm{B}) = 4\pi \times 6.022 \times 10^{23} \times 2 \times 10^{-9}$$

Compute 4π:

$$4\pi = 12.5664$$

Compute the denominator:

$$\text{Denominator} = 12.5664 \times 6.022 \times 10^{23} \times 2 \times 10^{-9}$$

Multiply $6.022 \times 2 = 12.044$

So:

$$\text{Denominator} = 12.5664 \times 12.044 \times 10^{23} \times 10^{-9}$$

Compute $12.5664 \times 12.044 = 151.981$

So:

$$\text{Denominator} = 151.981 \times 10^{14} = 1.51981 \times 10^{16}$$

So the denominator is approximately 1.51981×10^{16}

Now, compute R:

$$R = \frac{k}{\text{Denominator}} = \frac{1 \times 10^{7}}{1.51981 \times 10^{16}} = 6.579 \times 10^{-10}\ \mathrm{m}$$

This gives us:

$$R \approx 6.579 \times 10^{-10}\ \mathrm{m} = 0.6579\ \mathrm{nm}$$

This value is on the order of the size of small molecules, which typically have radii of around 0.1 nm to 1 nm.

Conclusion

So, for a reaction rate constant of $1 \times 10^{10}\ \mathrm{M^{-1}s^{-1}}$, and diffusion coefficients of $1 \times 10^{-9}\ \mathrm{m^2/s}$ for each particle, the collision radius R is approximately 0.66 nm.

This suggests that a realistic collision radius for small molecules in solution is on the order of **0.5 nm to 1 nm**.

Choosing the Collision Radius in Simulations

When setting up your simulation:

1. **Determine the Reaction Rate Constant k**:
   - Use experimental values if available.
   - For diffusion-controlled reactions between small molecules, k is typically 10^9 to $10^{10}\ \mathrm{M^{-1}s^{-1}}$.
2. **Use the Smoluchowski Equation to Calculate R**:
   - Rearrange the equation to solve for R:

$$R = \frac{k}{4\pi N_\mathrm{A}(D_\mathrm{A}+D_\mathrm{B})}$$

   - Plug in the values for k, N_A, and $D_\mathrm{A} + D_\mathrm{B}$.
3. **Set the Collision Radius in Your Simulation**:
   - Use the calculated R as the collision radius.
   - Ensure that R is consistent with the spatial scale of your simulation box and particle positions.

Practical Considerations

- **Spatial Resolution**:
  - The collision radius should be larger than the minimum spatial resolution of your simulation (e.g., larger than the average step size in the random walk).
  - If the collision radius is too small compared to the particle displacements per time step, you might miss collisions.
- **Time Step Size**:
  - The time step should be small enough to accurately capture the diffusion process.
  - Ensure that particles do not move distances larger than the collision radius in a single time step, or adjust the collision detection accordingly.
- **Adjusting for Simulation Efficiency**:
  - In some cases, using a slightly larger collision radius can help capture more reactions without significantly affecting the physical accuracy.
  - However, be cautious not to overestimate R, as it may lead to artificially high reaction rates.

Summary

- For small molecules in solution:
  - **Diffusion coefficients** are typically around $1 \times 10^{-9}\ \mathrm{m^2/s}$.
  - **Reaction rate constants** for diffusion-controlled reactions are around 10^9 to $10^{10}\ \mathrm{M^{-1}s^{-1}}$.
  - **Calculated collision radii** using the Smoluchowski equation are approximately 0.5 nm to 1 nm.
- **Choosing the collision radius**:
  - Use the Smoluchowski equation to calculate R based on k and D.
  - Ensure R is compatible with your simulation's spatial and temporal scales.
  - Validate your simulation by comparing results with theoretical predictions or experimental data.

Final Remarks

Selecting the appropriate collision radius is crucial for accurately simulating reaction kinetics in particle-based models. By grounding your choice in fundamental physical principles and established equations like the Smoluchowski equation, you can ensure that your simulation reflects realistic molecular behavior.

If you have further questions or need assistance with specific calculations or adjustments in your simulation code, feel free to ask!

2. Quote of response of ChatGPT o1 on optimizing computational performance

In the prompt leading to the response quoted below, I updated the number of particles in the simulation to 10,000.

Response ChatGPT o1-preview:

Considerations and Suggestions

1. Performance Optimization

With num_particles = 10,000, simulating all pairwise collisions between particles of type 1 and type 3 becomes computationally intensive. To address this, I've implemented a **spatial grid (also known as a cell list)** to optimize collision detection:

- **Grid-Based Collision Detection**: Particles are assigned to grid cells based on their positions. Only particles within the same or neighboring cells are checked for collisions.
- **Grid Size**: The grid size is based on the collision radius to ensure that all potential collisions are considered.

[...]

Conclusion

The updated script incorporates your specified parameters and includes optimizations to handle the increased computational demands. By using a spatial grid for collision detection and adjusting the reaction probability calculation, the simulation can efficiently model the particle interactions and reactions.

Please note: Due to the complexity and computational intensity of simulating 10,000 particles with collision detection, running this script may require significant computational resources and time. Consider testing with a smaller number of particles initially or using more advanced optimization techniques (e.g., parallel processing) for large-scale simulations.

3. Quantitative comparison of COMSOL and Python results.

In Figure S1, the stochastic/Python simulation of concentration profiles of the ECL compounds across the nanochannel is compared to finite element results *with matching concentrations*, i.e. a starting condition of $[[\mathrm{Ru(bpy)_3}]^{2+}] = 0.225$ mM. This simulation was implements in COMSOL Multiphysics 6.3, simulation details are identical to the one described in ref. 9 of the main text (https://doi.org/10.1039/C8SC03209B). As shown in the Fig. S1c, the nanochannel is coupled to a larger fluidic reservoir, and the entire floor of the nanochannel acts as an electrode.

Concentration profiles of $[\mathrm{Ru(bpy)_3}]^{2+}$, $[\mathrm{Ru(bpy)_3}]^{2+*}$, and $[\mathrm{Ru(bpy)_3}]^{3+}$ compounds are compared in Figure S2 for the two different types of simulations. All concentrations agree within a few percent, except for the positions of the simulation boundaries/electrodes. Here, the stochastic simulation yields unrealistic non-vanishing concentrations for the species being consumed there. I believe this is due to a limited spatial resolution, i.e., particles are crossing the electrode with a finite step length of the Brownian random walk.

Overall, both types of simulation match well considering that $[\mathrm{Ru(bpy)_3}]^{2+}$ and $[\mathrm{Ru(bpy)_3}]^{2+*}$ concentrations depend very sensitively on the rates k_{ann} and k_{decay}. Certainly, the difference between stochastic and finite elements results must be smaller than the difference between either of them and reality as neither simulation considers differences in the compounds' diffusion coefficient or migration in electrical double layers extending into the shallow nanochannel.

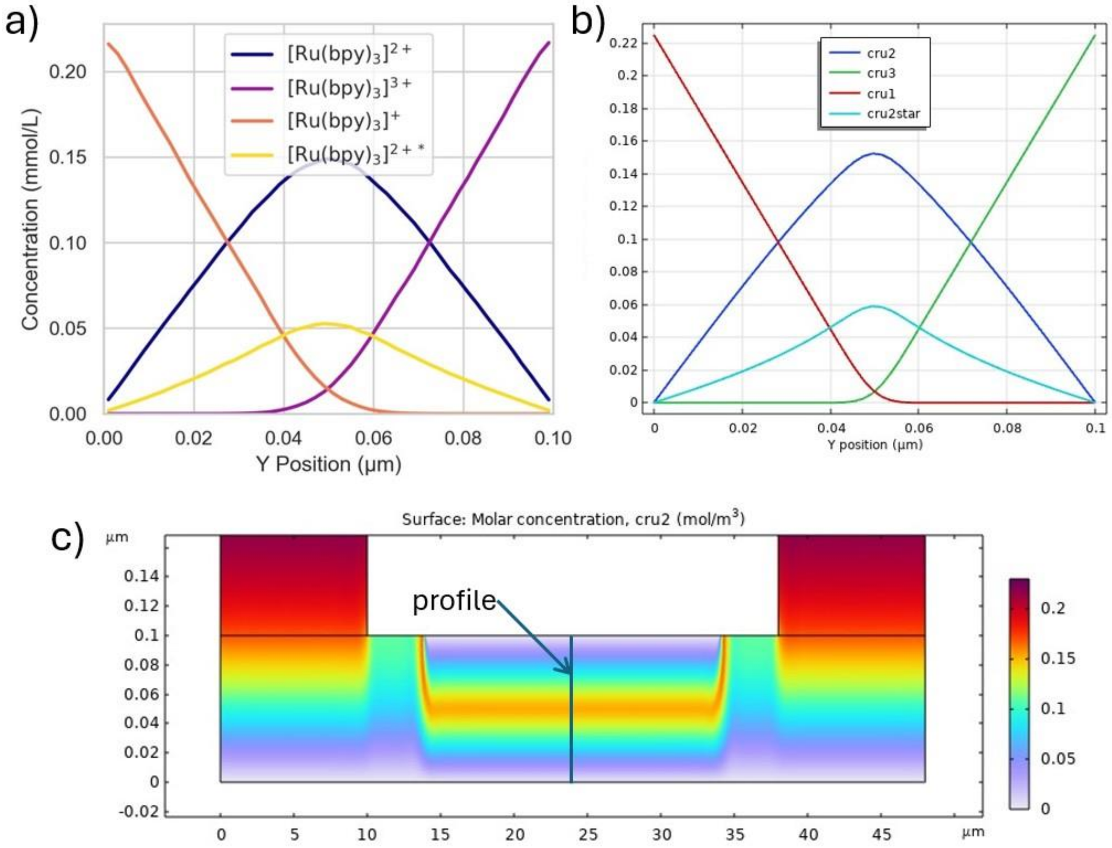


Figure S1. Concentration profiles across the nanochannel obtained by stochastic (random walk; Python) *vs.* finite element simulation (COMSOL). **a)** Stochastic simulation with k_{decay} = 10^6 s^{-1}, identical to Figure 2b in the main text. **b)** COMSOL simulation with concentrations matched to the stochastic simulation. **c)** Corresponding two-dimensional profile calculated using COMSOL Multiphysics 6.3 showing the concentration of $[\mathrm{Ru(bpy)_3}]^{2+}$. The dark blue line indicates the position of the profiles shown in panel b).

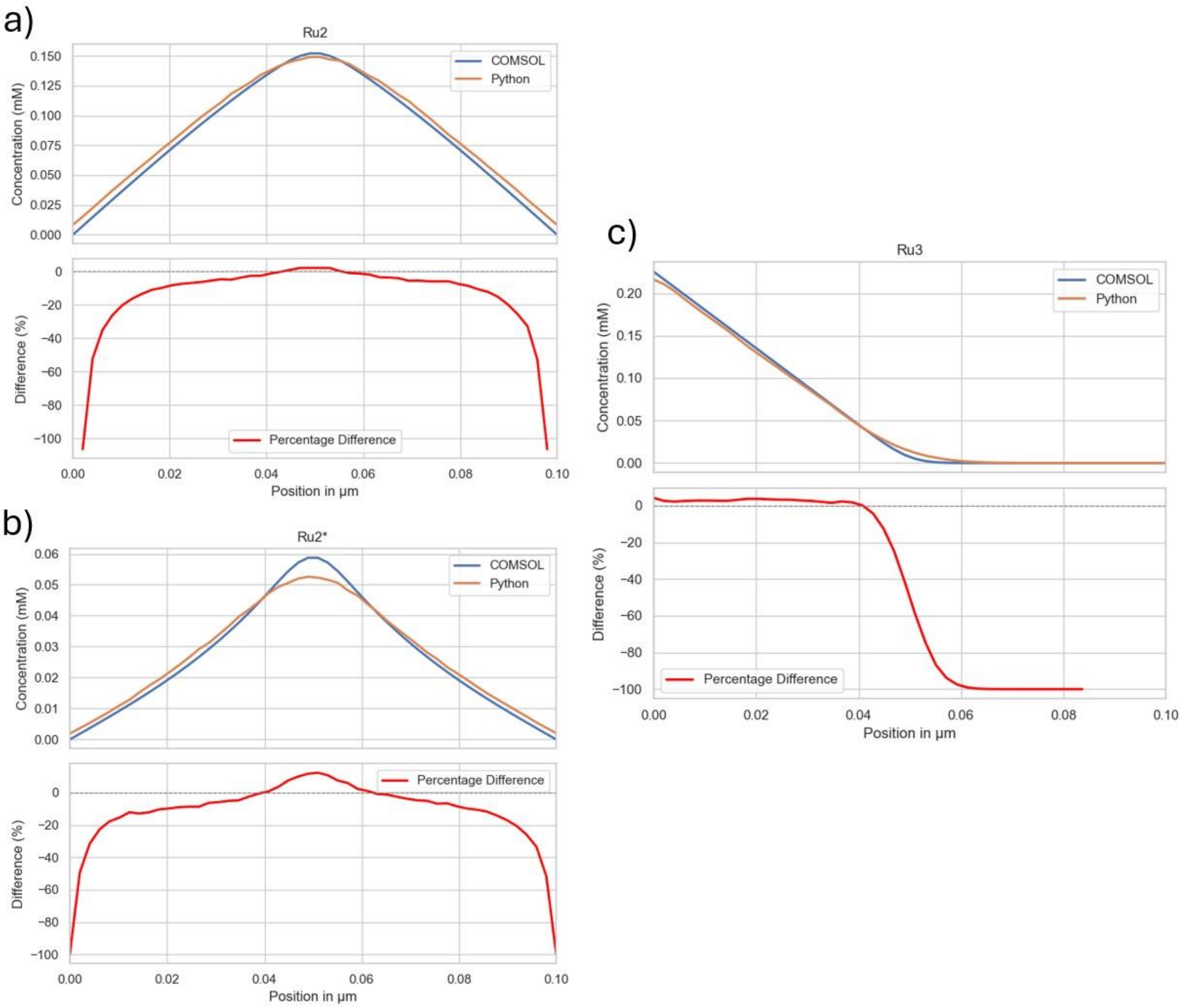
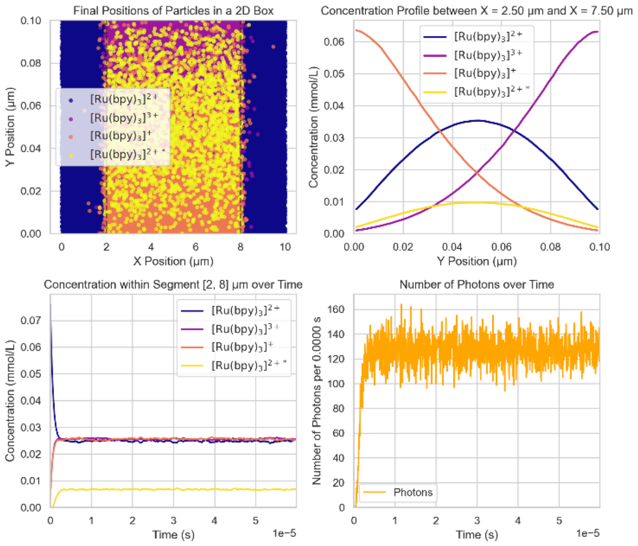
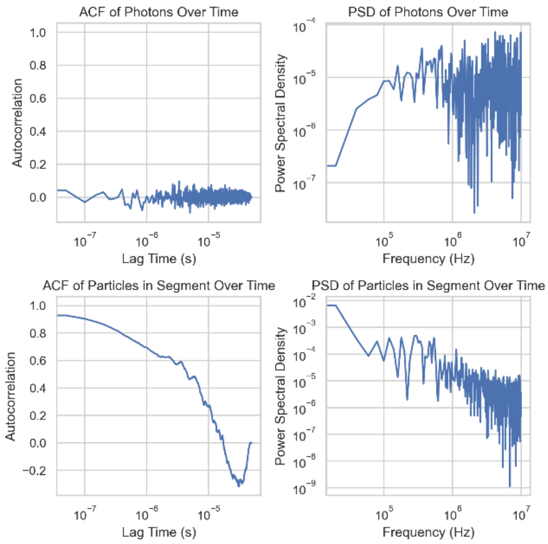


Figure S2. Comparison of profiles shown in Figure S1a, b for **a)** $[Ru(bpy)_3]^{2+}$, **b)** $[Ru(bpy)_3]^{2+}$*, and **c)** $[Ru(bpy)_3]^{3+}$*. The upper graph in each panel shows the concentration profiles obtained using COMSOL and Python, and lower graphs show the difference between both curves in percent.

4. Python script

Python script generated by prompting ChatGPT o1-preview. This script was used all figures are results shown in the main paper. Output of the script are the two figures shown below. The identical code is also available here: https://doi.org/10.6084/m9.figshare.27620118.v1.





```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.cm as cm
import sys
from scipy.signal import welch, correlate
from scipy.fft import fftshift

# Set Seaborn style
sns.set(style="whitegrid")

# Simulation parameters
box_width_um = 10           # Width of the box in micrometers (μm)
box_height_um = 0.1         # Height of the box in micrometers (μm)
num_particles = 50000       # Number of particles
diff_coeff_type1 = 1e-9     # Diffusion coefficient for particle type 1 in m²/s
diff_coeff_type2 = 1e-9     # Diffusion coefficient for particle type 2 in m²/s
diff_coeff_type3 = 1e-9     # Diffusion coefficient for particle type 3 in m²/s
diff_coeff_type4 = 1e-9    # Diffusion coefficient for particle type 4 in m²/s
time_step_us = 0.05        # Time step in microseconds (μs)
num_steps = 1200        # Number of simulation steps
binning_interval = 1       # New parameter for binning

# Reaction parameters
collision_radius_nm = 1.0   # Collision radius in nanometers (nm)
reaction_rate_constant = 1e10  # Reaction rate constant k in M⁻¹·s⁻¹

# Decay parameters for particle 4
decay_rate_constant = 1e6   # Rate at which particle 4 turns into particle 1 (s⁻¹)

# Number of time steps to sum over for the concentration profiles
num_steps_concentration_profile = 1000  # Must be less than or equal to num_steps

# Convert time_step from microseconds to seconds
time_step = time_step_us * 1e-6  # Convert time_step to seconds

# Time array for plotting
```

```
time_array = np.arange(num_steps) * time_step  # Ensures correct length

# Boundary segment parameters (in micrometers)
x1_um = 2   # Start of the segment along x-axis in µm
x2_um = 8   # End of the segment along x-axis in µm

# New x-axis boundary values for the concentration profile window
x_left_um = 2.5  # Left boundary of the window in µm
x_right_um = 7.5 # Right boundary of the window in µm

# Convert dimensions to meters
box_width = box_width_um * 1e-6      # Convert width to meters
box_height = box_height_um * 1e-6    # Convert height to meters
x1 = x1_um * 1e-6                    # Convert x1 to meters
x2 = x2_um * 1e-6                    # Convert x2 to meters
x_left = x_left_um * 1e-6            # Convert x_left to meters
x_right = x_right_um * 1e-6          # Convert x_right to meters
collision_radius = collision_radius_nm * 1e-9  # Convert collision radius to meters

# Calculate the standard deviations of the displacement for each particle type
std_dev_type1 = np.sqrt(2 * diff_coeff_type1 * time_step)
std_dev_type2 = np.sqrt(2 * diff_coeff_type2 * time_step)
std_dev_type3 = np.sqrt(2 * diff_coeff_type3 * time_step)
std_dev_type4 = np.sqrt(2 * diff_coeff_type4 * time_step)

# Initialize particle positions randomly within the box
x_positions = np.random.uniform(0, box_width, num_particles)
y_positions = np.random.uniform(0, box_height, num_particles)

# Initialize particle types: 1 for initial particles
particle_types = np.ones(num_particles, dtype=int)

# Data storage for analysis
particles_in_segment_over_time = []
photons_over_time = []
reaction_events = []  # Initialize reaction_events list
top_boundary_changes = []     # Number of particle changes at top boundary per time step
bottom_boundary_changes = []  # Number of particle changes at bottom boundary per time step

# Data storage for concentration profiles
positions_over_time = []
types_over_time = []

# Pre-calculate the volume of the simulation box for concentration calculations
# Assuming unit depth in the z-direction
box_depth = 1e-6  # Depth in meters (1 µm)
volume = box_width * box_height * box_depth  # Volume in cubic meters

# Avogadro's number
N_A = 6.022e23  # particles/mol

# Concentration in mol/m³
concentration = num_particles / (N_A * volume)

# Reaction probability per collision
reaction_probability = reaction_rate_constant * concentration * time_step

# Decay probability for particle 4
decay_probability = 1 - np.exp(-decay_rate_constant * time_step)

# Convert collision radius to a squared distance for comparison
collision_radius_sq = (2 * collision_radius) ** 2

# Simulate the random walk
for step in range(num_steps):
   # Calculate and display progress
    percent_complete = (step + 1) / num_steps * 100  # Calculate percentage
    sys.stdout.write(f"\rProgress: {percent_complete:.2f}% complete")  # Overwrite the line
    sys.stdout.flush()  # Ensure it gets printed immediately
```

```
   # Generate random displacements based on particle types
    dx = np.zeros(num_particles)
    dy = np.zeros(num_particles)

    # For particles of type 1
    type1_indices = np.where(particle_types == 1)[0]
    dx[type1_indices] = np.random.normal(0, std_dev_type1, len(type1_indices))
    dy[type1_indices] = np.random.normal(0, std_dev_type1, len(type1_indices))

    # For particles of type 2
    type2_indices = np.where(particle_types == 2)[0]
    dx[type2_indices] = np.random.normal(0, std_dev_type2, len(type2_indices))
    dy[type2_indices] = np.random.normal(0, std_dev_type2, len(type2_indices))

    # For particles of type 3
    type3_indices = np.where(particle_types == 3)[0]
    dx[type3_indices] = np.random.normal(0, std_dev_type3, len(type3_indices))
    dy[type3_indices] = np.random.normal(0, std_dev_type3, len(type3_indices))

    # For particles of type 4
    type4_indices = np.where(particle_types == 4)[0]
    dx[type4_indices] = np.random.normal(0, std_dev_type4, len(type4_indices))
    dy[type4_indices] = np.random.normal(0, std_dev_type4, len(type4_indices))

    # Update positions
    x_positions += dx
    y_positions += dy

    # Reflective boundary conditions for x-axis (all types)
    x_positions = np.where(x_positions < 0, -x_positions, x_positions)
    x_positions = np.where(x_positions > box_width, 2 * box_width - x_positions, x_positions)

    # Identify particles that have crossed the top boundary
    crossed_top = y_positions >= box_height

    # Identify particles that have crossed the bottom boundary
    crossed_bottom = y_positions <= 0

    # Define the segment condition along x-axis
    within_segment = np.logical_and(x_positions >= x1, x_positions <= x2)

    # --- Top Boundary Conditions ---
    # Initialize counters for boundary changes
    top_boundary_change_count = 0
    bottom_boundary_change_count = 0

    # Particles of type 1, 3, and 4 crossing the top boundary within the segment
    types_top_change = [1, 3, 4]
    for t in types_top_change:
        indices_crossed_top_in_segment = np.logical_and.reduce((crossed_top, particle_types == t, 
within_segment))
        particle_types[indices_crossed_top_in_segment] = 2
        y_positions[indices_crossed_top_in_segment] = 2 * box_height - 
y_positions[indices_crossed_top_in_segment]
        top_boundary_change_count += np.sum(indices_crossed_top_in_segment)

        indices_crossed_top_outside_segment = np.logical_and.reduce((crossed_top, particle_types == 
t, ~within_segment))
        y_positions[indices_crossed_top_outside_segment] = 2 * box_height - 
y_positions[indices_crossed_top_outside_segment]

    # Other types just reflect
    other_types_crossed_top = np.logical_and(crossed_top, ~np.isin(particle_types, 
types_top_change))
    y_positions[other_types_crossed_top] = 2 * box_height - y_positions[other_types_crossed_top]

    # --- Bottom Boundary Conditions ---
    # Particles of type 1, 2, and 4 crossing the bottom boundary within the segment
    types_bottom_change = [1, 2, 4]
    for t in types_bottom_change:
```

```
        indices_crossed_bottom_in_segment = np.logical_and.reduce((crossed_bottom, particle_types ==
t, within_segment))
        particle_types[indices_crossed_bottom_in_segment] = 3
        y_positions[indices_crossed_bottom_in_segment] = -
y_positions[indices_crossed_bottom_in_segment]
        bottom_boundary_change_count += np.sum(indices_crossed_bottom_in_segment)

        indices_crossed_bottom_outside_segment = np.logical_and.reduce((crossed_bottom,
particle_types == t, ~within_segment))
        y_positions[indices_crossed_bottom_outside_segment] = -
y_positions[indices_crossed_bottom_outside_segment]

    # Other types just reflect
    other_types_crossed_bottom = np.logical_and(crossed_bottom, ~np.isin(particle_types,
types_bottom_change))
    y_positions[other_types_crossed_bottom] = -y_positions[other_types_crossed_bottom]

    # Record boundary changes
    top_boundary_changes.append(top_boundary_change_count)
    bottom_boundary_changes.append(bottom_boundary_change_count)

    # --- Collision Detection and Reactions ---
    # Only proceed if there are particles of type 2 and 3
    if len(type2_indices) > 0 and len(type3_indices) > 0:
        # Create a spatial grid to optimize collision detection
        # Define grid size based on collision radius
        grid_size = collision_radius * 2

        # Number of grid cells in x and y
        num_cells_x = int(np.ceil(box_width / grid_size))
        num_cells_y = int(np.ceil(box_height / grid_size))

        # Assign particles to grid cells
        x_indices_type2 = (x_positions[type2_indices] // grid_size).astype(int)
        y_indices_type2 = (y_positions[type2_indices] // grid_size).astype(int)
        cell_indices_type2 = x_indices_type2 + y_indices_type2 * num_cells_x

        x_indices_type3 = (x_positions[type3_indices] // grid_size).astype(int)
        y_indices_type3 = (y_positions[type3_indices] // grid_size).astype(int)
        cell_indices_type3 = x_indices_type3 + y_indices_type3 * num_cells_x

        # Create dictionaries mapping cell indices to particle indices
        cell_dict_type2 = {}
        for idx, cell_idx in zip(type2_indices, cell_indices_type2):
            cell_dict_type2.setdefault(cell_idx, []).append(idx)
        cell_dict_type3 = {}
        for idx, cell_idx in zip(type3_indices, cell_indices_type3):
            cell_dict_type3.setdefault(cell_idx, []).append(idx)

        # Find overlapping cells
        common_cells = set(cell_dict_type2.keys()) & set(cell_dict_type3.keys())

        # Loop over cells where both type 2 and type 3 particles are present
        for cell_idx in common_cells:
            indices_type2_in_cell = cell_dict_type2[cell_idx]
            indices_type3_in_cell = cell_dict_type3[cell_idx]

            # Check for collisions within the cell
            for idx2 in indices_type2_in_cell:
                for idx3 in indices_type3_in_cell:
                    dx = x_positions[idx2] - x_positions[idx3]
                    dy = y_positions[idx2] - y_positions[idx3]
                    distance_sq = dx * dx + dy * dy
                    if distance_sq < collision_radius_sq:
                        # Decide if reaction occurs
                        if np.random.rand() < reaction_probability:
                            # Change particle types
                            particle_types[idx2] = 1  # Type 2 becomes Type 1
                            particle_types[idx3] = 4  # Type 3 becomes Type 4
                            # Record reaction event
```

```
                            reaction_events.append((step * time_step, idx2, idx3))

    # --- Decay of Particle 4 into Particle 1 ---
    # For particles of type 4
    type4_indices = np.where(particle_types == 4)[0]
    if len(type4_indices) > 0:
        # Determine which particles decay
        decay_random = np.random.rand(len(type4_indices))
        decayed = decay_random < decay_probability
        decayed_indices = type4_indices[decayed]
        # Update particle types
        particle_types[decayed_indices] = 1
        # Record photon events (previously "decay events")
        photons_over_time.append(len(decayed_indices))
    else:
        photons_over_time.append(0)

    # Data collection: Count particles within the segment
    in_segment = np.logical_and(x_positions >= x1, x_positions <= x2)
    num_type1_in_segment = np.sum(np.logical_and(particle_types == 1, in_segment))
    num_type2_in_segment = np.sum(np.logical_and(particle_types == 2, in_segment))
    num_type3_in_segment = np.sum(np.logical_and(particle_types == 3, in_segment))
    num_type4_in_segment = np.sum(np.logical_and(particle_types == 4, in_segment))
    particles_in_segment_over_time.append((num_type1_in_segment, num_type2_in_segment,
                                           num_type3_in_segment, num_type4_in_segment))

    # Store positions and types for concentration profile calculation
    if step >= num_steps - num_steps_concentration_profile:
        positions_over_time.append((x_positions.copy(), y_positions.copy()))
        types_over_time.append(particle_types.copy())

# Convert positions back to micrometers for plotting
x_positions_um = x_positions * 1e6
y_positions_um = y_positions * 1e6

# Separate particles by type for plotting
type1_indices = particle_types == 1
type2_indices = particle_types == 2
type3_indices = particle_types == 3
type4_indices = particle_types == 4

# Prepare data for plotting
particles_in_segment_over_time = np.array(particles_in_segment_over_time)
photons_over_time = np.array(photons_over_time)
top_boundary_changes = np.array(top_boundary_changes)
bottom_boundary_changes = np.array(bottom_boundary_changes)

# Binning for photons over time
bin_edges = np.arange(0, num_steps + binning_interval, binning_interval)
binned_photons, _ = np.histogram(np.arange(len(photons_over_time)),
                                 bins=bin_edges,
                                 weights=photons_over_time)

# Calculate bin centers in time units
bin_centers = (bin_edges[:-1] + bin_edges[1:]) / 2 * time_step

# --- Plot all graphs together ---
fig, axes = plt.subplots(2, 2, figsize=(9, 8), dpi=250)  # Adjusted figsize and dpi

# --- Color Map ---
cmap = cm.get_cmap('plasma', 16)
colors = cmap(np.linspace(0, 1, 16))

# Define labels with LaTeX formatting
label_type1 = r'$[\mathrm{Ru(bpy)_3}]^{2+}$'
label_type2 = r'$[\mathrm{Ru(bpy)_3}]^{3+}$'
label_type3 = r'$[\mathrm{Ru(bpy)_3}]^{+}$'
label_type4 = r'$[\mathrm{Ru(bpy)_3}]^{2+*}$'

# --- Plot 1: Final positions of the particles (Top Left) ---
```

```
ax = axes[0, 0]
# Plot particles of type 1
ax.scatter(x_positions_um[type1_indices], y_positions_um[type1_indices],
            s=10, alpha=0.7, color=colors[0], label=label_type1)
# Plot particles of type 2
ax.scatter(x_positions_um[type2_indices], y_positions_um[type2_indices],
            s=10, alpha=0.7, color=colors[5], label=label_type2)
# Plot particles of type 3
ax.scatter(x_positions_um[type3_indices], y_positions_um[type3_indices],
            s=10, alpha=0.7, color=colors[10], label=label_type3)
# Plot particles of type 4
ax.scatter(x_positions_um[type4_indices], y_positions_um[type4_indices],
            s=10, alpha=0.7, color=colors[15], label=label_type4)

# Highlight the segment on the top and bottom boundaries
ax.plot([x1_um, x1_um], [0, box_height_um], 'k--', alpha=0.5)
ax.plot([x2_um, x2_um], [0, box_height_um], 'k--', alpha=0.5)
ax.fill_between([x1_um, x2_um], box_height_um, box_height_um + 0.005, color='red', alpha=0.1)
ax.fill_between([x1_um, x2_um], -0.005, 0, color='green', alpha=0.1)

# ax.set_xlim(0, box_width_um)
ax.set_ylim(0, box_height_um)
ax.set_title('Final Positions of Particles in a 2D Box')
ax.set_xlabel('X Position (µm)')
ax.set_ylabel('Y Position (µm)')
ax.legend()

# --- Plot 2: Concentration Profile as a function of Y-Position (Top Right) ---
ax = axes[0, 1]
# Combine positions and types over the specified time steps
x_positions_cp = np.concatenate([pos[0] for pos in positions_over_time])
y_positions_cp = np.concatenate([pos[1] for pos in positions_over_time])
particle_types_cp = np.concatenate(types_over_time)

# Convert positions to micrometers
x_positions_cp_um = x_positions_cp * 1e6
y_positions_cp_um = y_positions_cp * 1e6

# Define x-position range for the concentration profile
x_min_um = x_left_um
x_max_um = x_right_um

# Select particles within the x-window
window_indices_cp = np.where(
    (x_positions_cp_um >= x_min_um) & (x_positions_cp_um <= x_max_um)
)[0]

# Define y-bins for concentration profile
num_bins = 50
y_bins = np.linspace(0, box_height_um, num_bins + 1)
y_bin_centers = (y_bins[:-1] + y_bins[1:]) / 2
y_bin_width_um = y_bins[1] - y_bins[0]

# Initialize counts
counts_type1 = np.zeros(num_bins)
counts_type2 = np.zeros(num_bins)
counts_type3 = np.zeros(num_bins)
counts_type4 = np.zeros(num_bins)

# Bin particles along y-axis
for idx in window_indices_cp:
    y_pos = y_positions_cp_um[idx]
    bin_idx = np.searchsorted(y_bins, y_pos) - 1
    if 0 <= bin_idx < num_bins:
        if particle_types_cp[idx] == 1:
            counts_type1[bin_idx] += 1
        elif particle_types_cp[idx] == 2:
            counts_type2[bin_idx] += 1
        elif particle_types_cp[idx] == 3:
            counts_type3[bin_idx] += 1
```

```
        elif particle_types_cp[idx] == 4:
            counts_type4[bin_idx] += 1

# Calculate concentrations in mol/L (mol per liter)
# Volume per bin in µm^3
depth_um = 1.0  # Assuming unit depth in µm
volume_per_bin_um3 = (x_max_um - x_min_um) * y_bin_width_um * depth_um

# Convert volume from µm^3 to liters (1 µm^3 = 1e-15 liters)
volume_per_bin_L = volume_per_bin_um3 * 1e-15

# Total number of time steps summed
num_time_steps = num_steps_concentration_profile

# Average counts per bin per time step
avg_counts_type1 = counts_type1 / num_time_steps
avg_counts_type2 = counts_type2 / num_time_steps
avg_counts_type3 = counts_type3 / num_time_steps
avg_counts_type4 = counts_type4 / num_time_steps

# Convert counts to moles (particles / Avogadro's number)
moles_type1 = avg_counts_type1 / N_A
moles_type2 = avg_counts_type2 / N_A
moles_type3 = avg_counts_type3 / N_A
moles_type4 = avg_counts_type4 / N_A

# Calculate concentrations in mol/L
concentration_type1 = moles_type1 / volume_per_bin_L
concentration_type2 = moles_type2 / volume_per_bin_L
concentration_type3 = moles_type3 / volume_per_bin_L
concentration_type4 = moles_type4 / volume_per_bin_L

# Plot concentration profiles
ax.plot(y_bin_centers, concentration_type1 * 1e3, label=label_type1, linewidth=2, color=colors[0])
ax.plot(y_bin_centers, concentration_type2 * 1e3, label=label_type2, linewidth=2, color=colors[5])
ax.plot(y_bin_centers, concentration_type3 * 1e3, label=label_type3, linewidth=2, color=colors[10])
ax.plot(y_bin_centers, concentration_type4 * 1e3, label=label_type4, linewidth=2, color=colors[14])

ax.set_xlabel('Y Position (µm)')
ax.set_ylabel('Concentration (mmol/L)')
ax.set_xlim(0, box_height_um)  # Limit x-axis to box height
ax.set_ylim(0, None)           # Start y-axis at 0
ax.set_title('Concentration Profile between X = {:.2f} µm and X = {:.2f} µm'.format(x_left_um,
x_right_um))
ax.legend()

# --- Plot 3: Concentration in the segment over time (Bottom Left) ---
ax = axes[1, 0]
# Calculate the volume of the segment (in liters)
segment_width_um = x2_um - x1_um
segment_area_um2 = segment_width_um * box_height_um  # in µm²
segment_volume_L = segment_area_um2 * depth_um * 1e-15  # Convert to liters

# Convert number of particles to concentration (mmol/L)
particles_in_segment_over_time = np.array(particles_in_segment_over_time)
concentration_type1_segment = (particles_in_segment_over_time[:, 0] / N_A) / segment_volume_L * 1e3
# mmol/L
concentration_type2_segment = (particles_in_segment_over_time[:, 1] / N_A) / segment_volume_L * 1e3
concentration_type3_segment = (particles_in_segment_over_time[:, 2] / N_A) / segment_volume_L * 1e3
concentration_type4_segment = (particles_in_segment_over_time[:, 3] / N_A) / segment_volume_L * 1e3

sns.lineplot(ax=ax, x=time_array, y=concentration_type1_segment, label=label_type1, color=colors[0])
sns.lineplot(ax=ax, x=time_array, y=concentration_type2_segment, label=label_type2, color=colors[5])
sns.lineplot(ax=ax, x=time_array, y=concentration_type3_segment, label=label_type3,
color=colors[10])
sns.lineplot(ax=ax, x=time_array, y=concentration_type4_segment, label=label_type4,
color=colors[14])
ax.set_title('Concentration within Segment [{}, {}] µm over Time'.format(x1_um, x2_um))
ax.set_xlabel('Time (s)')
ax.set_ylabel('Concentration (mmol/L)')
```

```
ax.set_xlim(0, time_array[-1])  # x-axis ranges from 0 to max time
ax.set_ylim(0, None)            # y-axis starts at 0
ax.legend()

# --- Plot 4: Photons over time (Bottom Right) ---
ax = axes[1, 1]
sns.lineplot(ax=ax, x=bin_centers, y=binned_photons, label='Photons', color='orange')
ax.set_title('Number of Photons over Time')
ax.set_xlabel('Time (s)')
ax.set_ylabel('Number of Photons per {:.4f} s'.format(binning_interval * time_step))
ax.set_xlim(0, time_array[-1])  # x-axis ranges from 0 to max time
ax.set_ylim(0, None)            # y-axis starts at 0
ax.legend()

plt.tight_layout()
plt.show()

# --- Additional Analysis: PSD and ACF ---

# Number of steps for PSD and ACF calculation
analysis_steps = num_steps_concentration_profile

# Photons over time
last_photons = binned_photons[-analysis_steps:]
delta_t_photons = binning_interval * time_step
time_array_photons = bin_centers[-analysis_steps:]

# Calculate PSD and ACF for each time trace using only the last steps
def calculate_psd_acf(data, delta_t):
    # Remove mean
    data_zero_mean = data - np.mean(data)
    # Calculate ACF
    acf_full = correlate(data_zero_mean, data_zero_mean, mode='full')
    acf = acf_full[acf_full.size // 2:] / np.max(acf_full)
    lag_times = np.arange(len(acf)) * delta_t
    # Calculate PSD using Welch's method
    freq, psd = welch(data_zero_mean, fs=1/delta_t, nperseg=len(data_zero_mean))
    return lag_times, acf, freq, psd

# Calculate PSD and ACF for the last analysis_steps of each time trace

# Photons over time
lag_photons, acf_photons, freq_photons, psd_photons = calculate_psd_acf(last_photons,
delta_t_photons)

# Particles in segment over time (sum of all types), using last steps only
total_particles_segment = np.sum(particles_in_segment_over_time[-analysis_steps:], axis=1)
delta_t_segment = time_step
lag_segment, acf_segment, freq_segment, psd_segment = calculate_psd_acf(total_particles_segment,
delta_t_segment)


# --- Plot PSD and ACF ---
fig2, axes2 = plt.subplots(2, 2, figsize=(7, 7), dpi=250)

# Photons
axes2[0, 0].semilogx(lag_photons, acf_photons)
axes2[0, 0].set_title('ACF of Photons Over Time')
axes2[0, 0].set_xlabel('Lag Time (s)')
axes2[0, 0].set_ylabel('Autocorrelation')
axes2[0, 0].grid(True)


axes2[0, 1].loglog(freq_photons, psd_photons)
axes2[0, 1].set_title('PSD of Photons Over Time')
axes2[0, 1].set_xlabel('Frequency (Hz)')
axes2[0, 1].set_ylabel('Power Spectral Density')
axes2[0, 1].grid(True)
```

```
# Particles in Segment
axes2[1, 0].semilogx(lag_segment, acf_segment)
axes2[1, 0].set_title('ACF of Particles in Segment Over Time')
axes2[1, 0].set_xlabel('Lag Time (s)')
axes2[1, 0].set_ylabel('Autocorrelation')
axes2[1, 0].grid(True)


axes2[1, 1].loglog(freq_segment, psd_segment)
axes2[1, 1].set_title('PSD of Particles in Segment Over Time')
axes2[1, 1].set_xlabel('Frequency (Hz)')
axes2[1, 1].set_ylabel('Power Spectral Density')
axes2[1, 1].grid(True)

plt.tight_layout()
plt.show()
```