`dynamic-neural-field-composer`: A C++ open-source library and application for building and simulating Dynamic Neural Field architectures

João G. Cunha[a], Wolfram Erlhagen[b], Raymond H. Cuijpers[c], Estela Bicho[a]

[a] *Universidade do Minho, Center Algoritmi, Guimarães, Portugal*
[b] *Universidade do Minho, Centre of Mathematics, Guimarães, Portugal*
[c] *Eindhoven University of Technology, Human-Technology Interaction Group, Eindhoven, Netherlands*

Abstract

Dynamic Neural Fields (DNFs) provide a mathematical framework for modelling continuous neural population dynamics that underlie perception, working memory, decision-making, and embodied cognition. This work presents `dynamic-neural-field-composer`, an open-source C++20 library and interactive application for composing, simulating, and visualising one- and two-dimensional DNF architectures. Architectures can be created either programmatically through the API or interactively through the Graphical User Interface, and their full topology and parameters can be saved or reloaded via a single human-readable JSON file. The application offers real-time visualisation, live parameter tuning, and fine-grained simulation control in either a fixed layout or a fully dockable multi-window mode. The dual library/application design serves both researchers embedding DNFs into larger real-time systems, such as robot control architectures, and users who prefer to build and explore models interactively without writing code. Pre-compiled binaries for Windows, Linux, and macOS support immediate use, while developers can build from source through one-step scripts. Benchmarking against existing DNF frameworks shows that `dynamic-neural-field-composer` achieves the highest simulation throughput (≈15–41% faster than that of the next-fastest framework) while maintaining algebraic equivalence with established implementations. The software has been used in published studies across neuroevolution of DNF architectures, robustness analysis under simulated neural degeneration, and anticipatory action selection for human-robot collaboration, reflecting its applicability across cognitive modelling, evolutionary computation, and robotics.

Keywords: Dynamic Neural Fields, Dynamic Field Theory, Computational Neuroscience, Neural field simulation, C++

Nr.	Code metadata description	Please fill in this column
C1	Current code version	v2.7.1
C2	Permanent GitHub link to code/repository used for this code version	`https://github.com/Jgocunha/dynamic-neural-field-composer`
C3	Permanent link to Reproducible Capsule	Not applicable
C4	Legal Code License	GNU General Public License v3 (GPLv3)
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	C++20, CMake 3.20+, vcpkg, Dear ImGui, ImPlot, imgui-node-editor, nlohmann-json, imgui-platform-kit, GitHub Actions (CI, static analysis), Codecov, Doxygen
C7	Compilation requirements, operating environments & dependencies	CMake 3.20+; MSVC 2022 / GCC 11+ / Apple Clang 13+; Windows x64, Linux, macOS arm64/x64; vcpkg for dependency management
C8	If available Link to developer documentation/manual	`https://github.com/Jgocunha/dynamic-neural-field-composer/wiki`, `https://jgocunha.github.io/dynamic-neural-field-composer/`
C9	Support email for questions	id10787@uminho.pt

Table 1: Code metadata

1. Motivation and significance

Dynamic Field Theory (DFT) is a mathematical and conceptual framework for understanding how neural populations give rise to cognition and behaviour [1, 2]. Its central element is the Dynamic Neural Field (DNF): a continuous activation field defined over a feature dimension (such as space, orientation, or colour) whose evolution is governed by the Amari neural field equation [3]. Through local excitation and surround inhibition, DNFs exhibit attractor dynamics that implement elementary cognitive operations such as detection, selection, and sustained working memory, from which more complex behaviour can be composed [4, 5, 6, 7]. Because these dynamics are continuous, stable, and grounded in neurophysiology, DFT has become an established tool across cognitive science [8, 9], embodied cognition [10, 11, 12], and autonomous and cognitive robotics [13, 14, 15, 16].

Building DFT models in practice requires more than implementing the underlying integro-differential equations. A typical architecture combines several fields of differing dimensions, multiple kernel types, stimuli, and inter-field couplings. Researchers iterate over parameters and topologies, examine field activations as they evolve, and compare alternative architectures side by side. Interactive dedicated software is therefore central to the day-to-day practice of DFT research.

Several established environments support this workflow. `cosivina`[1] is a mature, object-oriented MATLAB toolbox, widely used for teaching and research, that accompanies the textbook Dynamic Thinking [2]; its Python port, `cosivina-python`[2], reproduces the core simulation functionality but currently lacks a graphical interface and implements only a subset of architecture elements. `cedar`[3] [17] is a C++/Qt framework tailored to embodied cognition, with support for connecting fields to some robot sensors and actuators. Two further tools serve more specialised niches: `DynamicFieldFlow`[4] expresses field equations as TensorFlow graphs, enabling GPU-accelerated simulation and gradient-based parameter learning, but is not intended for real-time robotic deployment, while `lava-dnf`[5] maps field equivalents onto Intel Loihi 2 neuromorphic hardware and is no longer actively maintained. Of these, only `cosivina`, `cosivina-python`, and `cedar` are functionally comparable to the present work as general-purpose tools for constructing

[1] cosivina/cosivina.
[2] cosivina/cosivina_python.
[3] cedar/cedar.
[4] danielsabinasz/DynamicFieldFlow.
[5] lava-nc/lava-dnf.

and simulating DNF architectures. Each, however, tends to force a trade-off: `cosivina` is tied to a proprietary MATLAB license; `cosivina-python` offers no interactive graphical interface; and `cedar`, though powerful, is comparatively heavyweight and presents a steeper path to embedding fields into custom real-time applications outside its environment.

`dynamic-neural-field-composer` fills this gap. It is a modern, modular C++20 library that implements the core DFT elements (fields, interaction kernels, inputs, and their couplings), together with an interactive application for building and inspecting architectures. The same architecture can be assembled in a few lines of code and embedded directly into larger real-time systems, or composed visually through a node-graph editor without writing code, and saved or reloaded as a single human-readable JSON file. The software thus addresses real-time performance and embeddability, accessibility for non-programmers, and fully open-source portability across Windows, Linux, and macOS in a single tool.

The software supports several distinct usage paths. A user new to DFT can launch one of the pre-compiled applications to construct, run, and explore a model interactively, watching activation profiles evolve in real time as parameters are adjusted with sliders; the architecture can then be saved for later reuse. Researchers integrating DNFs into a larger system instead link the C++ library, instantiate elements programmatically (or load a previously designed architecture), and step the simulation from their own code. Teams can share their models by distributing the architecture's JSON files alongside a tagged release of the software, reproducing the same model and dynamics on any supported platform.

`dynamic-neural-field-composer` is already in active research use. It has been used in studies on the neuroevolution of neural-field architectures[6] [18], on the robustness of DNF models under simulated degeneration[7] [19], and on human-robot collaboration joint-action scenarios[8] [20, 21].

2. Software description

2.1. Software architecture

`dynamic-neural-field-composer` is written in C++20 and organised into loosely coupled modules: elements, simulation, visualization, user interface, application, and other supporting tools (Figure 1).

[6] Jgocunha/neat-dnfs.
[7] Jgocunha/dynamic-neural-field-degeneration.
[8] Jgocunha/vr-hr-joint-task.

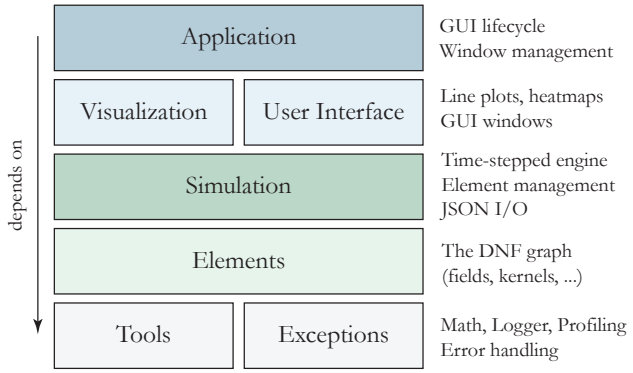

Figure 1: Layered software architecture, in which each layer depends only on those below it. The Elements layer defines the DNF primitives; the Simulation layer integrates them and handles JSON I/O; and the Visualization, User Interface, and Application layers provide rendering and interactive control, supported by shared Tools and Exceptions modules.

The central abstraction is the `Element`: an abstract base class from which every architectural primitive derives. Each element object owns a set of named data components (such as `input`, `activation`, and `output`), holds connections to its input and output elements, and implements a simple `init()`/`step()`/`close()` lifecycle; on every time step an element pulls data from its inputs and updates its components. Architectures are therefore directed graphs of elements through which data flows (Figure 2).

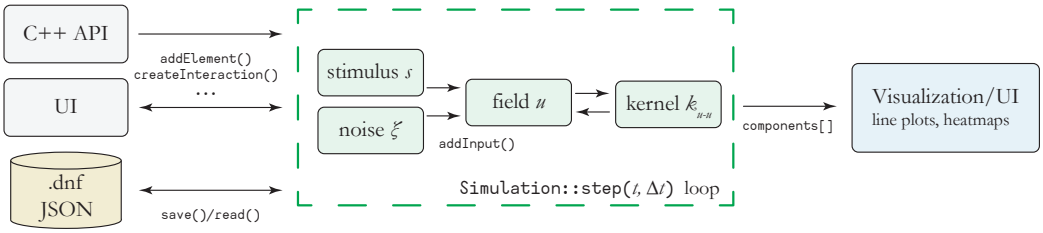

Figure 2: Build → simulate → visualize data flow. An architecture is assembled from the C++ API, the graphical interface, or a saved JSON file, with elements connected via `addInput()`. On each `Simulation::step(`$t, \Delta t$`)`, data propagates through the element graph, and the resulting components are rendered as line plots and heatmaps.

The element library provides the standard DFT primitives in both one- and two-dimensional variants: neural fields; interaction kernels (Gaussian, Mexican-hat, Oscillatory, and Asymmetric Gaussian); external stimuli (Gaussian, boost, and timed); stochastic noise (normal and correlated); inter-field couplings, including learnable Gaussian couplings (using standard Hebb, Oja, and Delta rules); and utility elements such as memory traces, resizing and dimension projection/expansion. Each element's configuration is held in a

dedicated parameter struct, separating what an element *is* from how it is *parameterised*, and an `ElementFactory` instantiates any registered element from a type label and a parameter set.
The `Simulation` class maintains a list of elements and serves as the integration engine: `init()` initialises all elements, and each `step()` advances them by a fixed time step using forward-Euler time-stepping, with kernel interactions evaluated via direct spatial convolution. It exposes batch and real-time execution methods, runtime configuration of the architecture (adding, removing, resizing, parametrising, and renaming elements), full JSON save/load, and element component recording and snapshot for offline analysis. Figure 3 shows the resulting per-step call sequence, from the application down to each element and the visualization.

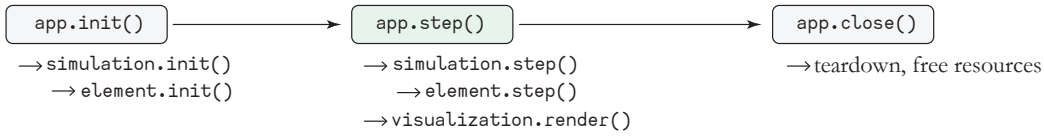

Figure 3: Execution lifecycle. The application drives the simulation through three phases: `init()`, repeated `step()` calls, and `close()`; each delegating to the corresponding Simulation, Element, and Visualization routines.

The visualization and interface layers sit on top of this core. A visualization module renders field components as line plots (one-dimensional) and heatmaps (two-dimensional) using `ImPlot`[9], while the user-interface module provides a set of composable, dockable windows: (i) a node-graph window for visual composition; (ii) plotting windows; (iii) an element inspector; (iv) simulation controls; (v) field-metrics; and (vi) a logs console. An `Application` object binds a simulation, its visualization, and a chosen set of windows; the entire GUI is built on the lightweight Dear ImGui library[10].

2.2. Software functionalities

The major functionalities are:

1. Compose DNF architectures from a library of field, kernel, stimulus, noise, and coupling elements, in one and two dimensions. Covering the standard DFT mechanisms needed to model a broad range of cognitive and robotic scenarios.
2. Two construction paths: programmatically through the C++ API, or interactively through the GUI without writing code.

[9] epezent/implot.
[10] ocornut/imgui

3. Simulate and inspect with play/pause/step control and real-time or batch execution, viewing activation, input, and output components live as line plots or heatmaps and editing parameters on the fly.
4. Serialize an entire architecture (topology and parameters) to a single human-readable JSON file, enabling models to be shared, version-controlled, and reproduced across platforms.
5. Embed or run standalone: link the library into any C++ application and drive the simulation from any other application, or launch one of two prebuilt applications: (i) `dnf-composer-static` (fixed layout) or `dnf-composer-dynamic` (fully rearrangeable and dockable windows, multi-monitor).
6. Built-in example library: ships with ready-to-run example programs and saved architectures across the core DFT mechanisms, e.g., detection, selection, and memory instabilities, multi-peak and travelling-bump solutions, inter-field couplings, and Hebbian learning; that serve both as a tutorial and as reusable starting points.
7. Cross-platform and tested: builds on Windows, Linux, and macOS via one-step scripts with vcpkg-managed dependencies, backed by a unit-test suite, continuous integration on all target platforms, static analysis, code-coverage reporting, and Doxygen API documentation.

3. Illustrative examples

The two construction paths (programmatically or interactively) are illustrated here with two architectures drawn from the bundled example library.

3.1. A selection field built through the API

A selection field implements a basic form of decision-making: presented with several competing inputs, lateral interactions allow only a single activation peak to survive while suppressing the alternatives (winner-take-all). The program in Listing 1 constructs one: three equally strong Gaussian stimuli are placed at different locations of a neural field whose Gaussian interaction kernel combines local excitation with global inhibition, together with a small amount of noise. Because the stimuli are of equal strength, the noise breaks the symmetry and determines which location is selected. A simulation is created with a name and integration step; a visualization and an application are instantiated to bind it to the GUI; the desired windows are added; the elements are created, registered, and wired; the field's components are plotted; and the application is initialised, stepped until the window is closed, and torn down. Figure 4 shows the resulting activation: a single peak emerges at one of the stimulated locations while the others are suppressed.

Listing 1: Constructing a selection field through the C++ API.

```cpp
using namespace dnf_composer;

// 1. A simulation, identified by name, integration step deltaT = 5
const auto simulation = std::make_shared<Simulation>("Selection instability (example)", 5.0, 0.0, 0.0);
// 2. A visualization bound to the simulation; plots are added later
const auto visualization = std::make_shared<Visualization>(simulation);
// 3. An application binding the simulation and its visualization
const Application app{ simulation, visualization };

// 4. Render only the selected windows:
app.addWindow<user_interface::MainMenuBar>();
app.addWindow<user_interface::NodeGraphWindow>();
app.addWindow<user_interface::PlotsWindow>();
// ... or render the default static layout
// app.addWindow<user_interface::StaticLayoutWindow>(simulation, visualization);

// 5. Create elements: a neural field, a Gaussian kernel with global inhibition, additive noise, and three equal, spatially separated stimuli
auto field = std::make_shared<element::NeuralField>(
    element::ElementCommonParameters{"selection field"},
    element::NeuralFieldParameters{25.0, -5.0, element::SigmoidFunction{0.0, 5.0}});
auto kernel = std::make_shared<element::GaussKernel>(
    element::ElementCommonParameters{"selection kernel"},
    element::GaussKernelParameters{3.0, 5.0, -0.25, true, true}); // local excitation + global inhibition
auto noise = std::make_shared<element::NormalNoise>(
    element::ElementCommonParameters{"normal noise"},
    element::NormalNoiseParameters{0.2});
auto stimA = std::make_shared<element::GaussStimulus>(
    element::ElementCommonParameters{"option A stimulus A"},
    element::GaussStimulusParameters{3.0, 5.0, 25.0}); // stimuli B, C added at 50, 75

// 6. Register elements and wire interactions (field <-> kernel is recurrent)
simulation->addElement(field);
simulation->addElement(kernel);
```

```
simulation->addElement(noise);
simulation->addElement(stimA); // ...and stimB, stimC
field->addInput(stimA); // ...and stimB, stimC
field->addInput(kernel);
field->addInput(noise);
kernel->addInput(field);

// 7. Plot the field's components
visualization->plot({ {field->getUniqueName(), "activation"},
                      {field->getUniqueName(), "output"},
                      {field->getUniqueName(), "input"} });

// 8. Initialise, run until the window is closed, then tear down
app.init();
while (!app.hasGUIBeenClosed())
    app.step();
app.close();
```

Figure 4: A selection field. Three equally strong Gaussian stimuli compete; the field's Gaussian kernel (local excitation with global inhibition) allows only one activation peak to survive, suppressing the others, with additive noise breaking the symmetry to determine the winner.

3.2. An adaptive architecture built through the GUI

The same primitives can be assembled entirely through the graphical interface, without writing code. Figure 5 shows a Hebbian-learning architecture — in which the weights of an inter-field coupling adapt online according to a Hebbian rule — being constructed and explored interactively. In the node-graph window, elements appear as nodes with input and output pins; an

interaction is created by dragging from one element's output pin to another's input and removed by double-clicking the connection, while double-clicking a node opens a plot of any of its components. A minimap, together with fit-to-selection and fit-all controls, keeps large architectures navigable. Selecting a node brings its parameters to the top of the element-control window, where every parameter can be edited at run time while the corresponding components update live in the plots. A control bar provides the simulation name, play/pause/resume/stop controls, and execution either in real-time millisecond batches or in fixed numbers of iterations; a status bar reports the run state, elapsed iterations and time, and performance indicators; and new simulations can be created, saved, and loaded from the main menu, with keyboard shortcuts for the common actions.

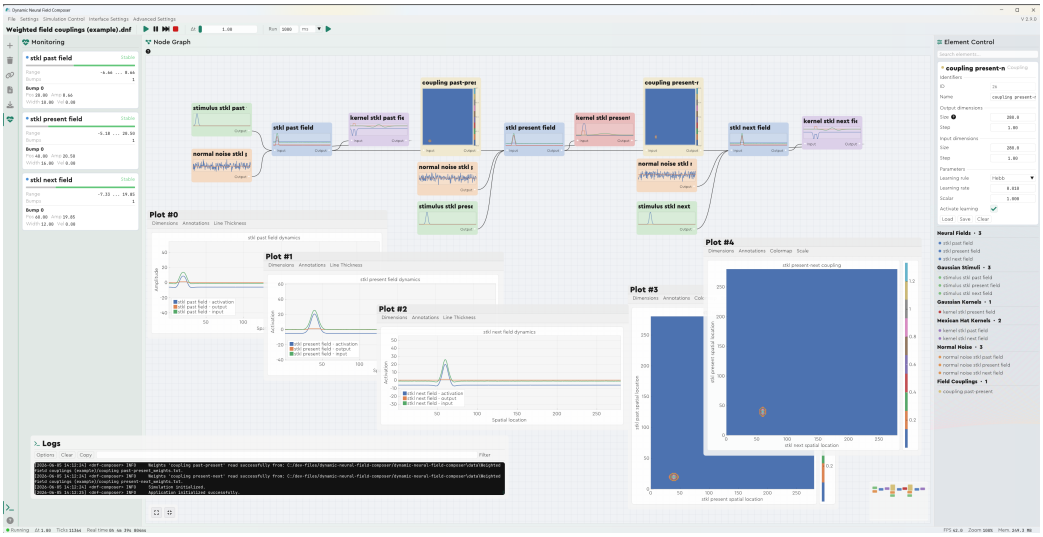

Figure 5: Interactive construction of a Hebbian-learning architecture. Elements appear as nodes in the node-graph window; connections are made by dragging between pins. Selecting a node exposes its parameters in the element-control window for live editing, with components plotted in real time. The control bar (top) and status bar (bottom) drive and monitor the simulation.

4. Impact

`dynamic-neural-field-composer` contributes to the DFT software landscape by providing a faster and more easily integrated simulator that makes certain workflows more practical to implement.

The software offers two concrete improvements: higher simulation throughput and lower integration cost. Academic simulation software is frequently cumbersome to build and to embed into other systems [] — a recurring source

of friction that the project set out to reduce by design, through a library-first architecture, standard C++ tooling (CMake with vcpkg-managed dependencies), and one-step cross-platform build scripts.
Benchmarking against `cosivina`, `cosivina-python`, and `cedar` on identical architectures shows that `dynamic-neural-field-composer` achieves the highest simulation throughput - delivering 15–41% higher throughput than Cedar, the next-fastest framework — while maintaining algebraic equivalence with established implementations (and full float64 precision).
Its higher simulation throughput suits experiments that instantiate and analyse large numbers of architectures. On this basis, the library serves as the simulation backend for `neat-dnfs`, which applies neuroevolution to evolve DNF architectures rather than design them by hand [18], and for a systematic study of artificial neural degeneration that quantifies how DNF architectures resist and adapt to progressive damage [19].
Combined with a dependency-light C++ core, this makes it straightforward to embed DNF models directly into larger real-time systems: having been used to model an architecture for anticipatory action selection for human-robot collaboration, evaluated in a virtual-reality experiment in which anticipating the human's next action improved interaction fluency [21].
We hope that a researcher who previously faced a steep path to embedding fields into a custom C++ application, or who depended on a proprietary runtime, can instead link`dynamic-neural-field-composer` directly and drive simulations from their own code with minimal setup.
The software has already been used across several research areas — evolutionary computation, robotics, and human-robot interaction — including graduate research such as MSc theses on DNFs in human-robot joint action [20]. It is openly developed and actively maintained.

5. Conclusions

This is a mandatory section summarising your software based on the sections above.

CRediT authorship contribution statement

João G. Cunha: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Raymond H. Cuijpers**: Conceptualization, Supervision, Writing – review & editing. **Wolfram Erlhagen**: Conceptualization, Supervision, Writing – review & editing. **Estela Bicho**: Conceptualization, Project administration, Resources, Supervision, Writing – review & editing.

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this manuscript, the authors used generative AI tools (Claude by Anthropic) to assist with language editing. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the content of the article.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was financed by FCT (Portuguese Foundation for Science and Technology) through the R&D Unit Project Scope UID/00319/2025 - Centro ALGORITMI (ALGORITMI/UM) and PhD Fellowship under grant 2022.13948.BD.

References

[1] S. Coombes, P. Beim Graben, R. Potthast, J. Wright (Eds.), Neural Fields: Theory and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. doi:10.1007/978-3-642-54593-1.
URL https://link.springer.com/10.1007/978-3-642-54593-1

[2] G. Schöner, J. Spencer, D. Research Group, Dynamic Thinking: A Primer on Dynamic Field Theory, Oxford Univ. Press, 2015. doi:10.1093/acprof:oso/9780199300563.001.0001.
URL https://academic.oup.com/book/10360

[3] S.-i. Amari, Dynamics of pattern formation in lateral-inhibition type neural fields, Biol. Cybern. 27 (2) (1977) 77–87. doi:10.1007/BF00337259.
URL http://link.springer.com/10.1007/BF00337259

[4] W. Erlhagen, E. Bicho, The dynamic neural field approach to cognitive robotics, J. Neural Eng. 3 (3) (2006) R36–R54. doi:10.1088/1741-2560/3/3/R02.
URL https://iopscience.iop.org/article/10.1088/1741-2560/3/3/R02

[5] J. S. Johnson, J. P. Spencer, G. Schöner, Moving to higher ground: The dynamic field theory and the dynamics of visual cognition, New Ideas in Psychology 26 (2) (2008) 227–251. doi:10.1016/j.newideapsych.2007.07.007.
URL https://linkinghub.elsevier.com/retrieve/pii/S0732118X07000505

[6] J. P. Spencer, S. Perone, A. T. Buss, Twenty Years and Going Strong: A Dynamic Systems Revolution in Motor and Cognitive Development, Child Development Perspectives 5 (4) (2011) 260–266. doi:10.1111/j.1750-8606.2011.00194.x.
URL https://academic.oup.com/cdpers/article/5/4/260/8234836

[7] Y. Sandamirskaya, Dynamic neural fields as a step toward cognitive neuromorphic architectures, Front. Neurosci. 7 (2014). doi:10.3389/fnins.2013.00276.
URL http://journal.frontiersin.org/article/10.3389/fnins.2013.00276/abstract

[8] W. Wojtak, S. Coombes, D. Avitabile, E. Bicho, W. Erlhagen, A dynamic neural field model of continuous input integration, Biol Cybern 115 (5) (2021) 451–471. doi:10.1007/s00422-021-00893-7.
URL https://link.springer.com/10.1007/s00422-021-00893-7

[9] J.-C. Quinton, F. Gautheron, A. Smeding, Embodied sequential sampling models and dynamic neural fields for decision-making: Why hesitate between two when a continuum is the answer, Neural Networks 179 (2024) 106526. doi:10.1016/j.neunet.2024.106526.
URL https://linkinghub.elsevier.com/retrieve/pii/S0893608024004507

[10] F. Ferreira, W. Wojtak, E. Sousa, L. Louro, E. Bicho, W. Erlhagen, Rapid Learning of Complex Sequences With Time Constraints: A Dynamic Neural Field Model, IEEE Trans. Cogn. Dev. Syst. 13 (4) (2021) 853–864. doi:10.1109/TCDS.2020.2991789.
URL https://ieeexplore.ieee.org/document/9085956/

[11] M. Richter, J. Lins, G. Schöner, A Neural Dynamic Model of the Perceptual Grounding of Spatial and Movement Relations, Cognitive Science 45 (10) (2021) e13045. doi:10.1111/cogs.13045.
URL https://onlinelibrary.wiley.com/doi/10.1111/cogs.13045

[12] D. Sabinasz, M. Richter, G. Schöner, Neural dynamic foundations of a theory of higher cognition: the case of grounding nested phrases, Cogn Neurodyn 18 (2) (2024) 557–579. doi:10.1007/s11571-023-10007-7.
URL https://link.springer.com/10.1007/s11571-023-10007-7

[13] S. Reynolds, D. Fan, T. M. Taha, A. DeMange, T. Jenkins, An Implementation of Simultaneous Localization and Mapping Using Dynamic Field Theory, in: NAECON 2021 - IEEE National Aerospace and Electronics Conference, IEEE, Dayton, OH, USA, 2021, pp. 80–83. doi:10.1109/NAECON49338.2021.9696440.
URL https://ieeexplore.ieee.org/document/9696440/

[14] D. Chatziparaschis, S. Zhong, V. Christopoulos, K. Karydis, Adaptive Environment-Aware Robotic Arm Reaching Based on a Bio-Inspired Neurodynamical Computational Framework, in: 2024 33rd IEEE International Conference on Robot and Human Interactive Communication (ROMAN), IEEE, Pasadena, CA, USA, 2024, pp. 510–515. doi:10.1109/RO-MAN60168.2024.10731226.
URL https://ieeexplore.ieee.org/document/10731226/

[15] R. Grieben, S. Sehring, J. Tekülve, J. P. Spencer, G. Schöner, ROBOVERINE: A human-inspired neural robotic process model of active visual search and scene grammar in naturalistic environments, in: 2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, Abu Dhabi, United Arab Emirates, 2024, pp. 11470–11477. doi:10.1109/IROS58592.2024.10801621.
URL https://ieeexplore.ieee.org/document/10801621/

[16] Z. Qin, Q. Fu, J. Peng, A computationally efficient and robust looming perception model based on dynamic neural field, Neural Networks 179 (2024) 106502. doi:10.1016/j.neunet.2024.106502.
URL https://linkinghub.elsevier.com/retrieve/pii/S0893608024004026X

[17] O. Lomp, M. Richter, S. K. U. Zibner, G. Schöner, Developing Dynamic Field Theory Architectures for Embodied Cognitive Systems with cedar, Front. Neurorobot. 10 (Nov. 2016). doi:10.3389/fnbot.2016.00014.
URL http://journal.frontiersin.org/article/10.3389/fnbot.2016.00014/full

[18] J. G. Cunha, R. H. Cuijpers, W. Erlhagen, E. Bicho, NEAT-DNFs: A NeuroEvolutionary Framework for Evolving Dynamic Neural Field

Architectures, in: Genetic and Evolutionary Computation Conference (GECCO '26), ACM, 2026. doi:10.1145/3795095.3805169.

[19] J. G. Cunha, R. H. Cuijpers, W. Erlhagen, E. Bicho, Robustness and Adaptability in a Dynamic Neural Field Architecture Subject to Degeneration, in: Lecture Notes in Networks and Systems, Springer, 2025.

[20] T. H. Janssen, C. Zhang, D. Lakens, R. H. Cuijpers, How We Can Use Dynamic Neural Fields in Human-Robot Joint Action, Student thesis: Master, Eindhoven University of Technology, TU/e, Eindhoven, Netherlands (Feb. 2026).
URL https://research.tue.nl/en/studentTheses/how-we-can-use-dynamic-neural-fields-in-human-robot-joint-action/

[21] J. G. Cunha, E. Bicho, W. Erlhagen, R. H. Cuijpers, Dynamic Neural Field Based Anticipatory Action Selection for Human Robot Collaboration: A Virtual Reality Experiment, in: Lecture Notes in Artificial Intelligence, Springer International Publishing, 2026.