PRCpy: A PYTHON PACKAGE FOR PROCESSING OF PHYSICAL RESERVOIR COMPUTING

MANUAL

Harry Youel[1,2,*] **Daniel Prestwood**[1,3,4,**] Oscar Lee[1] Tianyi Wei[1,3]
Kilian D. Stenning[4] Jack C. Gartside[4] Will R. Branford[4] Karin Everschor-Sitte[5]
Hidekazu Kurebayashi[1,3,6,***]

[1]London Centre for Nanotechnology, University College London,
London, WC1H 0AH, United Kingdom
[2]Department of Physics and Astronomy, University College London,
London, WC1E 6BT, United Kingdom
[3]Department of Electronic and Electrical Engineering, University College London,
London, WC1E 7JE, United Kingdom
[4]Blackett Laboratory, Imperial College London,
London SW7 2AZ, United Kingdom
[5]Faculty of Physics and Center for Nanointegration Duisburg-Essen (CENIDE), University of Duisburg-Essen,
Lotharstraße 1, 47057 Duisburg, Germany
[6]WPI Advanced Institute for Materials Research, Tohoku University, Sendai, Japan

[*]harry.youel.19@ucl.ac.uk,[**]daniel.prestwood.22@ucl.ac.uk,[***]h.kurebayashi@ucl.ac.uk

ABSTRACT

Physical reservoir computing (PRC) is a computing framework that harnesses the intrinsic dynamics of physical systems for computation. It offers a promising energy-efficient alternative to traditional von Neumann computing for certain tasks, particularly those demanding both memory and nonlinearity. As PRC is implemented across a broad variety of physical systems, the need increases for standardised tools for data processing and model training. In this manuscript, we introduce *PRCpy*, an open-source Python library designed to simplify the implementation and assessment of PRC for researchers. The package provides a high-level interface for data handling, preprocessing, model training, and evaluation. Key concepts are described and accompanied by experimental data on two benchmark problems: nonlinear transformation and future forecasting of chaotic signals. Throughout this manuscript, which will be updated as a rolling release, we aim to facilitate researchers from diverse disciplines to prioritise evaluating the computational benefits of the physical properties of their systems by simplifying data processing, model training and evaluation.

Keywords Reservoir computing · RC · Physical reservoir computing · PRC

1 Introduction

As machine learning and artificial intelligence (AI) develop in complexity, so does the demands for vast quantities of data and accompanying processing power [1–3]. Conventional computer hardware increasingly struggles to keep pace with the growing requirements of AI, with huge energy footprints due to an intrinsic mismatch between the hardware architecture and the demands of the software algorithms. The root causes of such energy inefficiencies arise in large part from the discrete memory-processor architecture, known as the von Neumann bottleneck. Consequently, its environmental carbon impact has come under increased scrutiny, directly impacting the cost and scaling of AI developments [4–9].

The field of unconventional computing is broad and includes different frameworks that solve specific problems. Examples include neuromorphic computing [10–12], probabilistic computing [13–15], and in-memory computing [16, 17]. A particular advantage is that physical systems from diverse disciplines can offer distinct computational functionality via their intrinsic physical dynamics [10, 18, 19]. Among these, a neuromorphic computing framework termed *physical reservoir computing* (PRC) has received attention thanks to its relatively simple implementation and potential for efficiently processing dynamic data. This is achieved by offloading nonlinear and memory-dependent processing to complex physical systems, leaving only a computationally-light linear final layer to train in software [20–23]. As with other unconventional computing frameworks, examples of PRC have been demonstrated across a wide range of disciplines such as, magnetism [24–39], optics [40–45], bio-materials [46, 47], analogue electronics [48–50], and others [51–53].

While many publications provide an excellent introduction to the general background theory and the working mechanisms of the PRC scheme, its detailed practical implementation methods are often not discussed at length. Therefore, providing such a technical discussion with practical examples may help circumvent barriers to entry in the field. Such practical details include processing the physical readout data, creating a reservoir matrix, training the linear readout layer of the model, and quantitatively evaluating system performance. While software-based reservoir computing (RC) solutions exist [54–56], to our knowledge none currently focuses specifically on PRC. Here, we provide an open-source Python library, *PRCpy* to address this need.

This manuscript is organised into four parts. Section 2 outlines the fundamental background of PRC. Section 3 provides a hands-on description of the setup and use of *PRCpy* for implementing PRC. In Sec. 4 we demonstrate nonlinear signal transformation and forecasting chaotic time-series data in two simple electronic circuits and in a complex magnetic system [24, 57]. Finally, Sec. 5 provides the conclusion and outlook.

2 Reservoir computing

2.1 Working principles of reservoir computing

Reservoir computing is an unconventional computing methodology that implements recurrent neural networks (RNNs). The scheme originated from a combination of two approaches, termed 'echo state networks' (ESN) by Jaeger [58] and 'liquid-state machines' by Maass et al. [59]. The core concept of RC involves constructing a network by initialising a set of nonlinear nodes with randomised internal recurrent connectivity and randomised weights at each node. This internal structure is typically termed the 'reservoir', and in physical reservoir computing, this is represented by the complex physical system. During training, the internal structure is left fixed in its initial randomised state. Only a simple linear layer is trained (typically linear, ridge or logistic regression) - posing an alternative to conventional neural network architectures where every network weight is optimised during training, often through backpropagation [23, 58–61]. This simple linear training is especially attractive for time-domain problems demanding recurrency such as future prediction, where the backpropagation training procedure for non-reservoir recurrent neural networks is highly computationally expensive.

The complex internal structure of the reservoir nonlinearly transforms and 'projects' data into a high-dimensional output space. Due to the internal recurrency/memory, nonlinear activation and dimensional expansion provided by the reservoir, tasks that were linearly inseparable in the input space now become linearly solvable in the high-dimensional output space. As only the final linear output layer is trained, RC typically offers computationally light training. In some cases, the linear output layer of the reservoir is replaced by a multi-layer perceptron style network of smaller size and complexity than the larger internal reservoir structure.

Figure 1 depicts a typical (P)RC scheme involving three layers: the input, the reservoir, and the output. The reservoir computing framework only requires training model weights at the output layer, as mentioned above. The following provides a breakdown of each layer:

1. **Input Layer**: A time series $\mathbf{u}(t)$ is input to the reservoir via a weight matrix $\mathbf{W}_{\text{in}}$ [23, 58–60]. The result of this transformation, $\mathbf{x}_{\text{in}}(t)$ can be expressed as:

$$\mathbf{x}_{\text{in}}(t) = \mathbf{W}_{\text{in}} \cdot \mathbf{u}(t) \tag{1}$$

2. **Reservoir**: The reservoir is typically a dynamic RNN that stores information about both current and past inputs in its internal states $\mathbf{x}(t)$, which evolve over time using fixed, randomly initialized internal weights $\mathbf{W}_{\text{res}}$. The state $\mathbf{x}(t)$ at time t is a function of the previous states and the current input, described by:

$$\mathbf{x}(t) = f(\mathbf{W}_{\text{res}}(t-n) + \mathbf{x}_{\text{in}}(t)) \tag{2}$$

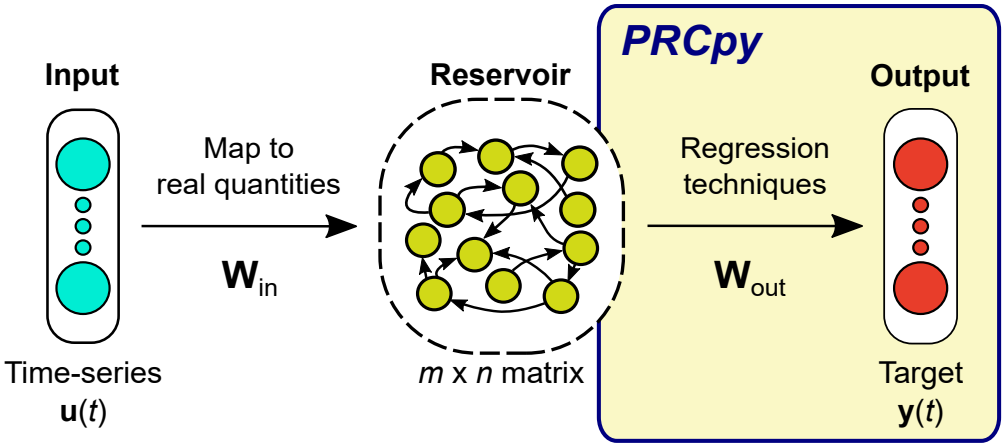

Figure 1: Typical PRC schematic. The input layer consists of a time series signal. Each value is mapped onto a controllable quantity and applied to the physical system. The reservoir is a multidimensional matrix constructed by combining all system measurement responses. The elements of the reservoir are then used to predict the desired target output via regression techniques, often computed by an external computing device and program code. *PRCpy* streamlines such complex coding required for the output layer.

where f is a nonlinear function, such as a hyperbolic tangent or sigmoid function in software-based reservoirs and n represents an arbitrary number of timesteps into the past. In PRC, the internal physical dynamics of a material system serve as the reservoir. Crucially, while the nonlinear activation functions in software reservoirs are typically uniform across all internal reservoir nodes, in physical reservoirs the internal reservoir structure can contain a broad variety of history-dependent nonlinear activations provided by the range of physical dynamics active in the system.

3. **Readout Layer**: A subset of the internal reservoir nodes are defined as output nodes $\mathbf{x}_{\text{out}}(t)$. The readout layer linearly multiplies each reservoir output node by a weight value $\mathbf{W}_{\text{out}}$ optimised via a linear regression process, then sums these values to produce the output $\mathbf{y}(t)$. The reservoir output is given by:

$$\mathbf{y}(t) = \mathbf{W}_{\text{out}} \cdot \mathbf{x}_{\text{out}}(t) \tag{3}$$

This mapping technique enables spatio-temporal feature selection at the readout layer and only requires training $\mathbf{W}_{\text{out}}$. As mentioned above, the main advantage of this concept is the low training cost, as it only necessitates computationally inexpensive linear regression techniques [23, 27, 57–60, 62, 63]. Training in RC schemes involves adjusting $\mathbf{W}_{\text{out}}$ using the reservoir states for a given set of input-output pairs, typically performed using regularised linear regression methods such as ridge regression [64]. This efficient training process contrasts with conventional deep neural networks (DNNs), which require fine-tuning interconnected node weights across multiple layers through iterative training processes such as backpropagation, often in the order of millions [62].

2.2 Design considerations of the reservoir

When designing the RC network, one must consider the following key properties of a reservoir: nonlinearity, dimensional expansion, and the fading memory/echo state property [23, 63, 65–68].

The nonlinearity and dimensional expansion (e.g. more output nodes than input nodes) of the reservoir facilitate the mapping of input features to a high-dimensional feature space, which enables one to solve nonlinear problems via linear training methods. Fading memory or the echo-state property describes the reservoir's ability to respond strongly to recent past inputs while remaining unaffected by those from the distant past. This property makes RC particularly attractive for processing temporal data with transient dependencies, such as forecasting chaotic signals [58, 59, 66, 69, 70].

It should be noted that there is typically a trade-off between these properties of nonlinearity (NL) and memory capacity (MC) [68]. Although it has been seen that including nonlinear and linear elements in a reservoir can improve performance, it is important to be aware that in a single reservoir, this often comes at the cost of reduced memory, and this trade-off should be kept in mind when designing a reservoir [11, 71]. Recent approaches have shown that designing networks of interconnected reservoirs can overcome this trade-off, a particularly promising route to enhancing the computational power of reservoir computing [72–75].

The nonlinearity and linear memory capacity of a given reservoir in *PRCpy* are both calculated using the same methodology as Ref. [65], which the following details. A linear estimator can be represented by:

$$\hat{y}(t) = \sum_i w_i u(t) \tag{4}$$

where the weights, w, have been fitted using the training data inputs, $u(t)$ and outputs $y(t)$. The R^2 coefficient can be used to compare the estimated outputs, $\hat{y}(t)$ to the true outputs by evaluating:

$$R^2[\hat{y}, y] = \frac{\text{cov}^2(\hat{y}(t), y(t))}{\sigma^2(\hat{y}(t))\sigma^2(y(t))} \tag{5}$$

The variance is represented by σ^2 and the covariance is expressed as "cov". The nonlinearity can then be interpreted as:

$$\Phi_n^{\text{NL}} = 1 - R^2[\hat{y}_n, y_n] \tag{6}$$

While the memory capacity of the reservoir can be retrived by summing the R^2 coefficients for all delayed estimators, $\hat{u}_n(t-\tau)$, and inputs, $u(t-\tau)$, up to a threshold k_{max}:

$$\Phi_n^{\text{MC}} = \sum_{\tau=1}^{k_{max}} R^2[\hat{u}_n(t-\tau), u(t-\tau)] \tag{7}$$

where τ is a time delay.

3 Using *PRCpy*

In this section, we introduce *PRCpy*, a Python library that provides a high-level interface for implementing PRC. The program is designed to be flexible and extensible, allowing researchers to easily integrate it into their existing workflows with much room for customisability. It includes modules for data handling, preprocessing, model training, and evaluation. Moreover, two reservoir metric functions are provided, which can be used to analyse the intrinsic properties of a given PRC system.

We begin by installing the library and describing its key features through an example of using *PRCpy* to solve a benchmark problem from real experimental data. The section concludes with a discussion of some of the library's advanced features and provides guidance on how to use them effectively.

3.1 License and codebase

The *PRCpy* package is open source with MIT license. The Git Repository is available at:
https://github.com/OSJL-py/PRCpy.

3.2 Installation

PRCpy requires Python 3.9 or later and has been tested on a Windows device. It can be installed using pip or Poetry.

To install *PRCpy* using pip, run:

```
pip install prcpy
```

To install *PRCpy* using Poetry, run:

```
poetry add prcpy
```

It is recommended to use the latest release of *PRCpy*, which can be obtained by running:

```
PIP: pip install prcpy --upgrade
POERTY: poetry update prcpy
```

3.3 General usage overview

The general workflow for using *PRCpy* consists of the following steps:

1. Define the data path.
2. Define the preprocessing parameters.
3. Create the RC pipeline.
4. Define the target and add it to the pipeline.
5. Define the model for training.
6. Define the RC parameters.
7. Run the RC pipeline.

Here, we provide a brief overview of each step. More detailed explanations and examples are provided in Sec. 4.

3.3.1 Defining the data path

The first step is to specify the directory containing the raw data files. The data file prefix parameter must also be specified (in all examples the prefix is set to "scan"). Example data files can be found in the `examples/data_full` directory of the *PRCpy* repository.

3.3.2 Defining the preprocessing parameters

Next, a dictionary containing the preprocessing parameters must be defined. These parameters specify how the raw data should be processed before being fed into the reservoir and a description of each parameter can be found in Table 1.

3.3.3 Creating the RC pipeline

The RC pipeline encapsulates the entire workflow, from data loading to model evaluation. To create the pipeline object, pass the data directory path, file prefix and preprocessing parameters to the `Pipeline` constructor, as shown in the example below:

```
from prcpy.RC import Pipeline

data_dir_path = "your/data/path"
prefix = "scan"
process_params = {
    "Xs": "independent_column": str,
    "Readouts": "readout_column": str,
    "delimiter": "\t": str,
    "remove_bg": True: bool,
    "bg_fname": "background_data.txt": str,
    "smooth": False: bool,
    "smooth_win": poly_window: int,
    "smooth_rank": poly_rank: int,
    "cut_xs": False: bool,
    "x1": x_limit_lower: int/float,
    "x2": x_limit_upper: int/float,
    "normalize_local": False: bool,
    "normalize_global": False: bool,
    "sample": True: bool,
    "sample_rate": adjacent_separation: int ,
    "transpose": False: bool }

rc_pipeline = Pipeline(data_dir_path, prefix, process_params)
```

Table 1: Description of available *PRCpy* parameters for creating the reservoir matrix.

Parameter	Description	Notes
Xs	The name of the column containing the x-values (e.g., frequency).	Independent variable name.
Readouts	The name of the column containing the y-values (e.g., spectra).	Measured variable name.
delimiter	Specify measured data delimiter.	-
remove _ bg	Whether to remove the background from the data.	If True, will remove data from bg _ fname.
bg _ fname	Name of the background data file.	Note that the columns names must match Xs and Readouts.
smooth	Whether to apply smoothing to the data.	Applies a savgol _ filter. See savgol _ filter for details.
smooth _ win	Smoothing filter window length.	Ensure smooth _ win is less than the number of measured data.
smooth _ rank	Smoothing filter polynomial order.	-
cut _ xs	Whether to slice the data array.	If True, the measure data (readouts) below x1 and above x2 will be removed when creating the reservoir.
x1	Initial slicing data point.	-
x2	Ending slicing data point.	-
normalize_local	Whether to normalize each data channel separately.	If True, the measured data (readouts) will be normalized between 0 and 1.
normalize_global	Whether to normalize all data channels.	-
sample	Whether to sample data.	If True, every 'sample _ rate' of the measured data (readouts) will be used when creating the reservoir.
sample _ rate	Frequency of sampling.	-
transpose	Transposes the RC dataframe.	This cannot be used simultaneously with other optional preprocessing parameters (as of v0.1.14).

3.3.4 Defining the target

The next step is to define the target signal. *PRCpy* provides utility functions for generating common target signals, such as square waves and Mackey Glass time series.

For example, to generate a square wave target signal, one can use the `generate_square_wave()` function:

```
from prcpy.Maths.Target_functions import generate_square_wave

length = rc_pipeline.get_df_length()
num_periods = 10

target_values = generate_square_wave(length, num_periods)
```

The target signal is then added to the RC pipeline using the `define_target` method:

```
rc_pipeline.define_target(target_values)
```

3.3.5 Defining the model

PRCpy provides a set of predefined models for training the reservoir, such as ridge regression and logistic regression. The model is defined by passing a dictionary which specifies its parameters.

For example, to define a ridge regression model, the `define_Ridge` method can be called:

```
from prcpy.TrainingModels.RegressionModels import define_Ridge

model_params = {
    "alpha": 1e-3: int,
    "fit_intercept": True: bool,
    "copy_X": True: bool,
    "max_iter": None: int,
    "tol": 0.0001: float,
    "solver": "auto": str,
    "positive": False: bool,
    "random_state": None: int }

model = define_Ridge(model_params)
```

3.3.6 Defining the RC parameters

The RC parameters specify how the reservoir is trained and evaluated. The key parameters include:

- `model`: The model to use for training.
- `tau`: The delay between the input and target signals. This value can be specified to predict `tau` steps in to the future for forecasting tasks while `tau` must be set to zero for transformation tasks.
- `test_size`: The fraction of the data to use for testing.
- `error_type`: The type of error to use for evaluation (e.g., MSE).

Relevant RC parameters for training and performance evaluations are specified in a dictionary format:

```
rc_params = {
    "model": model,
    "tau": 0: int,
    "test_size": 0.3: float,
    "error_type": "MSE": {"MSE", "MAE"} }
```

3.3.7 Running the RC pipeline

Finally, the RC pipeline can be executed by calling the `run` method and passing the `rc_params` dictionary:

```
rc_pipeline.run(rc_params)
```

This will automatically load the data, perform preprocessing, train the model, and evaluate its performance on the test set.

3.3.8 Getting the results

After running the RC pipeline, the results can be obtained by using the `get_rc_results` method:

```
results = rc_pipeline.get_rc_results()
```

The results are returned as a dictionary containing:

```
results_df = {
    "train": {
        "x_train": x_train,
        "y_train": y_train,
        "train_pred": train_predictions},

    "test": {
        "x_test": x_test,
        "y_test": y_test,
```

```
        "test_pred": test_predictions},

    "error": {
        "train_error": train_error: float,
        "test_error": test_error: float}   }
```

The nonlinearity and memory capacity can also be evaluated using the following procedure. First define the input signal used to generate the readouts being evaluated:

```
rc_pipeline.define_input(input_values)
```

Then use the following functions to calculate either nonlinearity (NL) or linear memory capacity (LMC):

```
nl = rc_pipeline.get_non_linearity()

total_mc, mc_list = rc_pipeline.get_linear_memory_capacity(kmax=25,
    remove_auto_correlation=True)
```

The first command produces the average NL score for the entire reservoir ranging from 0 for completely linear to 1 for completely nonlinear by utilising Eqn. 6. The second provides two outputs: total LMC and the LMC list. The latter provides a score between 0 and 1 for the ability of a set of readouts to predict a previous input k steps ago up to the user defined k_{max} value (k_{max} is set to 25 by default). The total LMC is the summation of the LMC list (Eqn. 7). The "remove_auto_correlation" parameter subtracts the correlation at each k value of the input signal with itself which is set to False by default. This parameter can aid the user in removing the intrinsic memory imparted from a quasi-periodic input signal, such as Mackey Glass, but can lead to unmeaningful negative LMC scores if an inappropriate k_{max} value is used.

4 Tutorial examples

Here, we provide a step-by-step guide for using *PRCpy* on three different reservoir systems (the code files can be found in the `examples/tutorials` directory of the *PRCpy* repository). In Sec. 4.1, we construct a reservoir using a resistor and a red LED by time multiplexing to perform transformation tasks. In Sec. 4.2 we add a capacitor to the aforementioned circuit to demonstrate the importance of memory in a number of tasks, and utilise the reservoir metrics included in the *PRCpy* package to compare these two electronic reservoirs. Section 4.3 demonstrates PRC with a more complex system involving a nontrivial magnetic material with frequency-multiplexing.

4.1 Diode PRC

Figure 2a depicts the experiment schematic for constructing a diode reservoir. Note that such implementation is constructed for the sole purpose of demonstrating *PRCpy*. Here, a standard red LED is connected in series with a resistor, and the voltage, V, is measured as a function of the input current, I. The *I-V* characteristics of an LED are nonlinear as shown in Fig. 2b(i). The input to the reservoir, $\mathbf{x}_{\text{in}}(t)$ is a 10-period sinewave consisting of 50 points per period.

One method to increase the dimensionality of the reservoir is to utilise the time-multiplexing or 'virtual node' technique as introduced by Appeltant et al. [76]. This scheme involves randomly choosing N pairs of bounds on the *I-V* curve, which represent N windows. Three examples of these windows are shown by the dashed vertical lines in Fig. 2b(i). Subsequently, $\mathbf{x}_{\text{in}}(t)$ is mapped onto I for each window, n_i at a given time, t (Fig. 2b(ii)). This approach allows the efficient utilisation of a single-input-single-output system in creating an N-dimensional reservoir. At each applied value of I, a separate file is saved with the "scan" prefix indicating the measurement number.

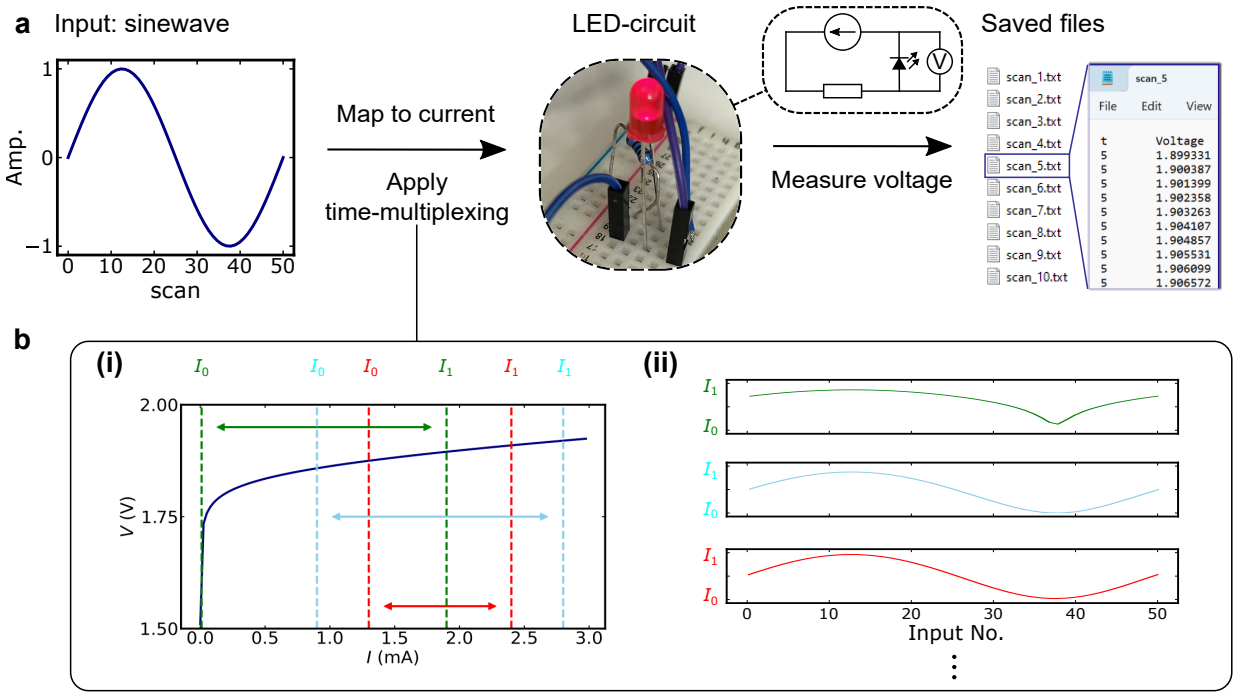

Figure 2: Creating the reservoir layer. a, A pre-defined input signal ($u(t) = \sin(t)$) is mapped onto the current and applied to a simple LED circuit as shown in the middle panel. For each current, voltage is measured and is individually saved to a file. Here, "scan" refers to the next measurement number. **b**, Time-multiplexing is employed to increase the dimensionality of the reservoir for single-input-single-output systems as such. Three example windows are shown with randomly selected bounds on the *I*-*V* curve (**b (i)**). The corresponding mapped currents (**b (ii)**) are subsequently applied.

PRCpy allows one to create a **customised reservoir matrix**. No preprocessing steps are activated by default and can be left empty unless specified. The code below shows the necessary steps for creating and viewing the reservoir matrix. Note that all file names must match the specified prefix string (for this example we have set prefix="scan", see right panel of Fig. 2a).

```
""" setting up PRCpy """

## essential module
from prcpy.RC import Pipeline

## optional modules (used for this example)
from prcpy.TrainingModels.RegressionModels import define_Ridge
from prcpy.Maths.Target_functions import generate_square_wave
import numpy as np
import matplotlib.pyplot as plt

## specify data directory path
data_dir_path = r"examples/data_100/sine_mapping/diode"
prefix = "scan"

## define processing parameters
process_params = {
    "Xs": "t",
    "Readouts": "Voltage",
    "delimiter": "\t",
    "remove_bg": False,
    "smooth": False,
    "cut_xs": False,
    "sample": False,
```

```
    "normalize_local": False,
    "normalize_global": False,
    "transpose": True
}

## create a pipeline object, encapsulating all RC features.
rc_pipeline = Pipeline(data_dir_path, prefix, process_params)

## view the reservoir matrix
reservoir_df = rc_pipeline.rc_data.rc_df
print(reservoir_df)

>>           r0        r1        r2       ...   r498      r499      r500
>> Scan1    1.787651  1.826460  1.846522  ...   1.902053  1.910276  1.918088
>> Scan2    1.791340  1.827726  1.847296  ...   1.902375  1.910583  1.918371
>> Scan3    1.794408  1.828896  1.848057  ...   1.902732  1.910921  1.918700
>> Scan4    1.797150  1.830044  1.848818  ...   1.903122  1.911282  1.919042
>> ...           ...       ...       ...  ...        ...       ...       ...
>> Scan497  1.768355  1.819889  1.841874  ...   1.899318  1.907673  1.915581
>> Scan498  1.773524  1.821212  1.842721  ...   1.899721  1.908047  1.915943
>> Scan499  1.778505  1.822643  1.843633  ...   1.900164  1.908474  1.916348
>> Scan500  1.782793  1.824015  1.844522  ...   1.900621  1.908901  1.916750
>>
>> [500 rows x 501 columns]
```

Next, the target function must be defined and added to the reservoir dataframe. Targets can be generated by calling the functions defined in `prcpy.Maths.Target_functions` or can be user-generated. To perform transformation tasks, one must ensure the period and the length of the input and the target are matched. This is automatically handled by *PRCpy*. See the code below:

```
""" target generation (transformation) """

## define number of periods and length of the input signal
length = rc_pipeline.get_df_length()
num_periods = 10

## define target
target_values = generate_square_wave(length, num_periods)

## add target to the reservoir matrix
rc_pipeline.define_target(target_values)
```

The RC training parameters must be specified before running the model. This includes the model definition, test size (percentage as a decimal), and the error type (supports: “MSE": Mean Square Error (MSE) and "MAE": Mean Absolute Error) for performance evaluation. By default, *PRCpy* provides three simple regression models: ridge, linear and logistic regression, utilising the `sklearn` module. Custom models can also be applied and specified by the RC parameters.

```
""" training parameters definition (transformation) """

## define the training model. We will choose the ridge regression.
model_params = {
    "alpha": 1e-6,
    "fit_intercept": True,
    "copy_X": True,
    "max_iter": None,
    "tol": 0.0001,
    "solver": "auto",
    "positive": False,
    "random_state": None,
}
model = define_Ridge(model_params)
```

```
## define the RC parameters
rc_params = {
    "model": model,
    "tau": 0, # transformation task
    "test_size": 0.3,
    "error_type": "MSE"
}
```

Once the RC parameters are defined, the model can be passed to the `run` function. This will split the reservoir matrix and its targets into training (learning) and testing (unseen) datasets in a ratio according to the user-specified value of "test_size". The obtained weights from the regression are then applied to the testing dataset to perform the computation as shown in Figs. 3a-e.

```
""" perform RC (transformation) """

## run the pipeline
rc_pipeline.run(rc_params)
```

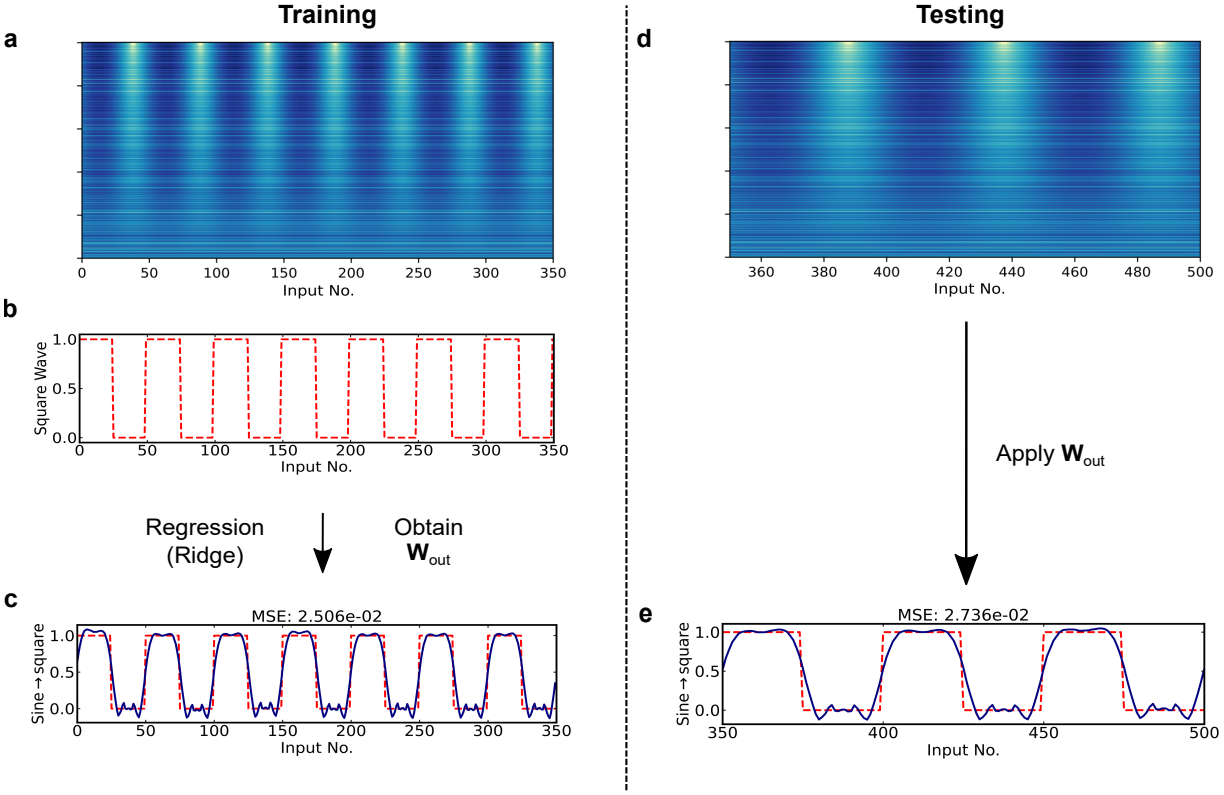

Figure 3: Transformation RC example. a(d), A visualisation of the reservoir matrix for the training (testing) data. **b**, Defined target signal ($y(t) = \text{square}(t)$). Ridge regression is applied to the training set of the reservoir matrix and their corresponding target values to obtain an optimised weight matrix. **c(e)**, Performance of training – the weights are applied to the training (testing) reservoir matrix, which is shown by a blue line, and compared with the target signal (dashed red line).

Finally, the results of the computation can be accessed by the `get_rc_results()` function. This returns the relevant data frames used for computation, prediction and error analysis. See Sec. 3.3.8 for details.

```
""" obtaining PRC results """

```

```
results = rc_pipeline.get_rc_results()

## it may be beneficial to attribute each result into its own variable.

# targets
train_ys = results["train"]["y_train"]
test_ys = results["test"]["y_test"]

# predictions
train_preds = results["train"]["train_pred"]
test_preds = results["test"]["test_pred"]

# errors
train_MSE = results["error"]["train_error"]
test_MSE = results["error"]["test_error"]

# x-axis
train_ts = np.arange(train_ys.shape[0])
test_ts = np.arange(test_ys.shape[0])
```

Results of PRC performance can be evaluated by viewing the magnitude of the error or plotting the training and testing data with their target values, as seen in Figs. 3c&e.

```
""" Evaluating the PRC performance """

## error
print(f"Training MSE = {train_MSE, '0.3e'}")
print(f"Testing MSE = {test_MSE, '0.3e'}")

>>> Training MSE = 2.506e-02
>>> Testing MSE = 2.736e-02
```

```
""" Evaluating the PRC performance """

## plotting
plt.plot(train_ts, train_ys, label="Train", color="red", ls="--")
plt.plot(train_ts, train_preds, label="Train predict", color="navy")
plt.show()
```

It should be noted that while we refer to this diode circuit as a reservoir, it infact does not fulfil the echo state property [58], and therefore is not conventionally defined as a reservoir. A more concise description would be that we have used a ridge regression layer on the diode's readout. Despite this technicality, the presence of nonlinearity does enable it to perform tasks with no memory requirement.

4.2 Capacitor & diode PRC

This example uses the same circuit as in Sec. 4.1 with the addition of a capacitor as shown in Fig 4b(i). The capacitor here, along with the resistor, is indented to cause voltage decay that provides memory of previous inputs. This memory effect is evident when comparing the output line profiles to the sine input for the diode only (Fig 4a(ii)) and the capacitor & diode reservoir (Fig 4b(ii)). The output depends only on the input in Fig 4a(ii) while the output depends both on the input and position on the sine wave for Fig 4b(ii). This memory is what allows the circuit to perform well for forecasting tasks, such as predicting the Mackey Glass time series.

The contrast in performance for the three tasks shown in Fig 4 between both circuits can be explained by the reservoir metrics (Table 2). The MSEs for both transformation tasks: sine to square wave (Fig 4c&d) and sine to sawtooth wave (Fig 4e&f), differ between the two circuits. While the sine to square task is purely nonlinear, the sawtooth does require some memory as the transformation is not just a one-to-one mapping from the input to the output. Information about the position on the sine wave is required. The capacitor & diode's superior nonlinearity score of 0.151 exhibits an improved MSE of 9.048×10^{-3} compared to the diode's MSE of 2.736×10^{-2}. The sine to sawtooth tasks only requires a small amount of memory capacity to effectively transform the input data, and as such the capacitor & diode far outperform the diode only reservoir. Figure 4g shows how poorly the diode performs when predicting a Mackey Glass signal. While

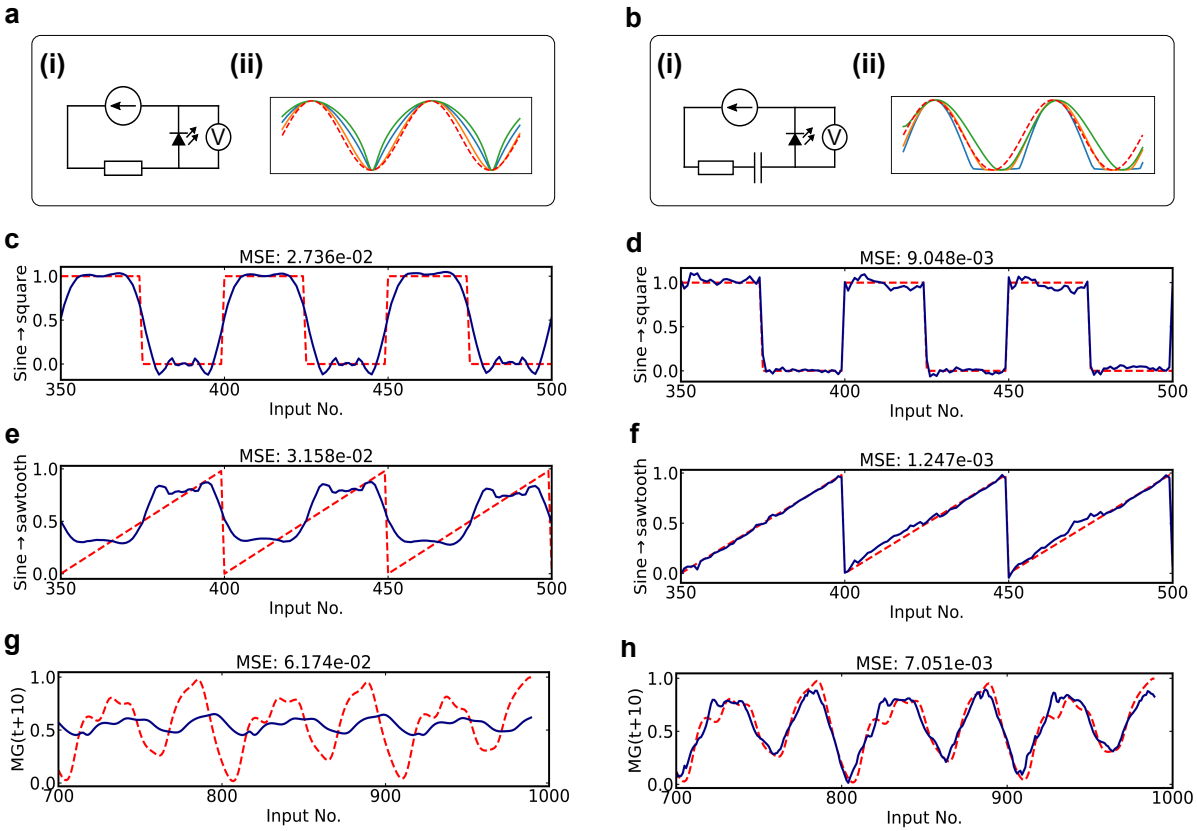

Figure 4: Comparison between diode circuit and capacitor & diode circuit RC performances. a(i), A diode circuit diagram. **a(ii)**, Three example line profiles from the diode reservoir. **b(i)**, A capacitor & diode circuit diagram. **b(ii)**, Three example line profiles from the capacitor & diode reservoir. **c & d**, Comparison between sine to square wave and (**e & f**), sine to sawtooth wave transformation testing performances for both circuits. **g & h**, Comparison between Mackey Glass forecasting testing performances for both circuits.

Table 2: Comparison between reservoir metrics for diode and capacitor & diode circuits.

Reservoir	Nonlinearity	Linear memory capacity
Diode	0.060	1.976
Capacitor & Diode	0.151	5.201

the MSE score is on the order of 10^{-2}, it should be noted that a straight line at the MG$(t+10) = 0.5$ produces an MSE of $\sim 7 \times 10^{-2}$. On the other hand, the capacitor & diode reservoir performs significantly better with an MSE of 1.247×10^{-3}. This is explained by the greater MC score of 5.201 compared to the diode only score of 1.976.

The MC score presented in this work of 1.976 appears to be rather high for the diode only reservoir that ostensibly has no memory. This reveals an issue with using MC to characterise the memory of the reservoir. For our calculation, we used the Mackey Glass input as it best demonstrated the memory that exists within our reservoir. This input, however, has some self-correlation that allows it to obtain a non-zero MC score on itself. As such, we include an option to remove this contribution ("remove_auto_correlation"=True), but this feature is not comprehensive as discussed in Sec. 3.3.8. Despite this, MC is useful when comparing reservoirs using the same input method, as we show in Table 2.

The nonlinearity and linear memory capacity reservoir metrics can be obtained by running:

```
""" Evaluating reservoir properties """

mg_path = "examples/data_full/chaos/mackey_glass_t17.npy"
input_values = get_npy_data(mg_path, norm=True)

## metrics
rc_pipeline.define_input(input_values[:1000])

nl = rc_pipeline.get_non_linearity()
lmc = rc_pipeline.get_linear_memory_capacity(remove_auto_correlation=True, kmax
    = 12)[0]

print(f"NL = {nl}")
print(f"LMC = {lmc}")

>>> NL = 0.1507705132191488
>>> LMC = 5.200838123059376
```

4.3 Magnetic PRC

In this example, a magnetic system (Cu_2OSeO_3) is realised as a physical reservoir as shown by prior works in Ref. [24]. This system has a large memory capacity originating from nontrivial spin-textures (magnetic skyrmions) and is suitable for future forecasting tasks. The input signal to the reservoir is defined by a Mackey Glass signal [77], tuned numerically for chaotic behaviour (see Ref. [24] for details). The target signal is the same as the input, however, with a 10-future step offset in the x-axis. This is equivalent to using the current data t_i to predict t_{i+10}.

Moreover, a different multiplexing technique to the electronic PRC examples is employed. Here, frequency-multiplexing is utilised, and for every input (external magnetic field), the power absorption from a range of frequencies is measured using specialised experimental equipment (vector network analyser). See Ref. [24] for full details. For further reading on skyrmion-based reservoir computing, see also a comprehensive review in Ref. [57] and references therein.

The code for performing the computation with *PRCpy* is presented below. The final result of the prediction is shown in Fig. 5.

```
""" Magnetic reservoir (forecasting)"""

from prcpy.RC import Pipeline
from prcpy.TrainingModels.RegressionModels import define_Ridge
from prcpy.Maths.Target_functions import get_npy_data
from prcpy.Maths.Maths_functions import normalize_list

data_dir_path = r"examples/data_full/mg_mapping/Cu2OSeO3/skyrmion"
prefix = "scan"
mg_path = "examples/data_full/chaos/mackey_glass_t17.npy"

process_params = {
    "Xs": "Frequency",
    "Readouts": "Spectra",
    "delimiter": ",",
    "remove_bg": True,
    "bg_fname": "BG_450mT_1_0to6_0GHz_4K.txt",
    "sample": True,
    "sample_rate": 13,
    "normalize_local": False,
    "normalize_global": False,
    "transpose": False
}

rc_pipeline = Pipeline(data_dir_path, prefix, process_params)
```

```
target_values = normalize_list(get_npy_data(mg_path))
rc_pipeline.define_target(target_values)

model_params = {
    "alpha": 1e-3,
    "fit_intercept": True,
    "copy_X": True,
    "max_iter": None,
    "tol": 0.0001,
    "solver": "auto",
    "positive": False,
    "random_state": None,
}

model = define_Ridge(model_params)

rc_params = {
    "model": model,
    "tau": 10, # forecasting of 10 steps into the future
    "test_size": 0.3,
    "error_type": "MSE"
}

rc_pipeline.run(rc_params)
```

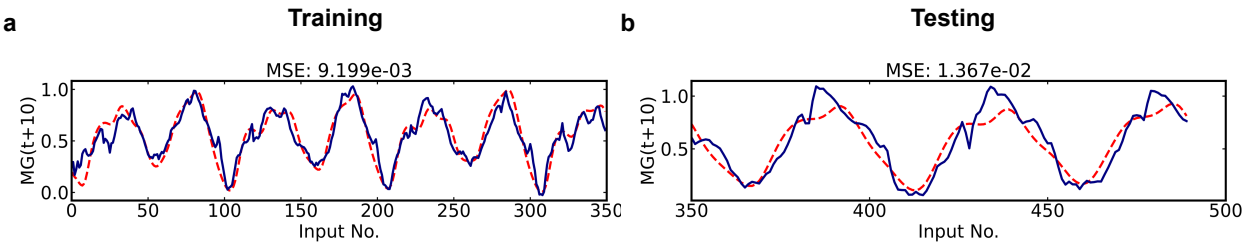

Figure 5: Forecasting RC example. Forecasting of a Mackey Glass signal for 10 future steps using a magnetic (skyrmion) reservoir.

5 Conclusion and outlook

This manuscript has covered a brief introduction to PRC and its practical use cases utilising the *PRCpy* package on three physical systems. While excellent literature and reviews have already presented thorough workings of (P)RC, its practical implementations remained nontrivial, bottlenecked by an adequate understanding of programming knowledge requirements.

Here, we provide tools to allow more researchers from diverse communities to participate in PRC research with their unique systems. Despite numerous proposals and demonstrations, the PRC field requires more steps before sparking its route for commercialisation. Future research into reservoir systems needs to consider their viability for mass production and scalability. The practical implementation in terms of the energy consumption, footprint and operating conditions also needs consideration to enable the application of PRC to edge computing.

We understand that the research is no longer a singular study. Researchers from different backgrounds are encouraged to collaborate within multidisciplinary areas not limited to physics, electrical engineering, or computer science, to explore possible paths for designing and engineering future energy-efficient computing systems. Given the range of different physical systems that can be used in the implementation of PRC, it stands out as a field particularly opportune for this kind of collaboration. As previously stated, the *PRCpy* package remains open source, and we encourage all researchers to contribute to making the project more widely available.

Acknowledgments

The authors thank Dr. Matthias Sitte for providing insightful advice to improve the quality and standards of the codebase.

References

[1] OpenAI. GPT-4 Technical Report, DOI: https://doi.org/10.48550/arXiv.2303.08774 (2023). 2303.08774.

[2] Rombach, R., Blattmann, A., Lorenz, D., Esser, P. & Ommer, B. High-resolution image synthesis with latent diffusion models. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 10674–10685, DOI: https://doi.ieeecomputersociety.org/10.1109/CVPR52688.2022.01042 (IEEE Computer Society, Los Alamitos, CA, USA, 2022).

[3] Gozalo-Brizuela, R. & Garrido-Merchan, E. C. ChatGPT is not all you need. A State of the Art Review of large Generative AI models, DOI: https://doi.org/10.48550/arXiv.2301.04655 (2023). 2301.04655.

[4] Dhar, P. The carbon impact of artificial intelligence. *Nat. Mach. Intell.* **2**, 423–425, DOI: https://doi.org/10.1038/s42256-020-0219-9 (2020).

[5] Freitag, C. *et al.* The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations. *Patterns* **2**, 100340, DOI: https://doi.org/10.1016/j.patter.2021.100340 (2021).

[6] AI and compute — openai.com. https://openai.com/research/ai-and-compute. [Accessed 18-08-2023].

[7] Strubell, E., Ganesh, A. & McCallum, A. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 3645–3650, DOI: https://aclanthology.org/P19-1355 (Association for Computational Linguistics, Florence, Italy, 2019).

[8] Strubell, E., Ganesh, A. & McCallum, A. Energy and Policy Considerations for Modern Deep Learning Research. In *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, 13693–13696, DOI: https://doi.org/10.1609/aaai.v34i09.7123 (2020).

[9] Schwartz, R., Dodge, J., Smith, N. A. & Etzioni, O. Green AI. *Commun. ACM* **63**, 54–63, DOI: https://doi.org/10.1145/3381831 (2020).

[10] Schuman, C.D., Kulkarni, S.R., Parsa, M. *et al.* Opportunities for neuromorphic computing algorithms and applications. *Nat. Comput. Sci.* **2**, 10–19, DOI: https://doi.org/10.1038/s43588-021-00184-y (2022).

[11] Stenning, K.D., Gartside, J.C., Manneschi, L. *et al.* Neuromorphic overparameterisation and few-shot learning in multilayer physical neural networks. *Nat. Commun.* **15**, 7377, DOI: https://doi.org/10.1038/s41467-024-50633-1 (2024).

[12] Marrows, C.H., Barker, J., Moore, T.A. *et al.* Neuromorphic computing with spintronics. *npj Spintron.* **2**, 12, DOI: https://doi.org/10.1038/s44306-024-00019-2 (2024).

[13] Borders, W.A., Pervaiz, A.Z., Fukami, S. *et al.* Integer factorization using stochastic magnetic tunnel junctions. *Nature* **573**, 390–393, DOI: https://doi.org/10.1038/s41586-019-1557-9 (2019).

[14] Chowdhury, S. *et al.* A full-stack view of probabilistic computing with p-bits: Devices, architectures, and algorithms. *IEEE J. on Explor. Solid-State Comput. Devices Circuits* **9**, 1–11, DOI: https://doi.org/10.1109/JXCDC.2023.3256981 (2023).

[15] Camsari, K. Y., Faria, R., Sutton, B. M. & Datta, S. Stochastic p-bits for invertible logic. *Phys. Rev. X* **7**, 031014, DOI: https://link.aps.org/doi/10.1103/PhysRevX.7.031014 (2017).

[16] Bao, H., Zhou, H., Li, J. *et al.* Toward memristive in-memory computing: principles and applications. *Front. Optoelectron* **15**, 23, DOI: https://doi.org/10.1007/s12200-022-00025-4 (2022).

[17] Zhou H, Li S, Ang K-W, Zhang Y-W. Recent Advances in In-Memory Computing: Exploring Memristor and Memtransistor Arrays with 2D Materials. *Nanomicro Lett.* DOI: https://doi.org/10.1007/s40820-024-01335-2 (2024).

[18] Roy, K., Jaiswal, A. & Panda, P. Towards spike-based machine intelligence with neuromorphic computing. *Nature* **575**, 607–617, DOI: https://doi.org/10.1038/s41586-019-1677-2 (2019).

[19] Joksas, D. *et al.* Memristive, Spintronic, and 2D-Materials-Based Devices to Improve and Complement Computing Hardware. *Adv. Intell. Syst.* **4**, 2200068, DOI: https://doi.org/10.1002/aisy.202200068 (2022).

[20] Yan, M., Huang, C., Bienstman, P. *et al.* Emerging opportunities and challenges for the future of reservoir computing. *Nat. Commun.* **15**, 2056, DOI: https://doi.org/10.1038/s41467-024-45187-1 (2024).

[21] Liang, X., Tang, J., Zhong, Y. *et al.* Physical reservoir computing with emerging electronics. *Nat. Electron.* 1–14, DOI: https://doi.org/10.1038/s41928-024-01133-z (2024).

[22] Tanaka, G. *et al.* Recent advances in physical reservoir computing: A review. *Neural Networks* **115**, 100–123, DOI: https://doi.org/10.1016/j.neunet.2019.03.005 (2019).

[23] Nakajima, K. & Fischer, I. *Reservoir Computing: Theory, Physical Implementations, and Applications* (Springer, 2021). https://link.springer.com/book/10.1007/978-981-13-1687-6.

[24] Lee, O., Wei, T., Stenning, K.D. *et al.* Task-adaptive physical reservoir computing. *Nat. Mater.* **23**, 79–87, DOI: https://doi.org/10.1038/s41563-023-01698-8 (2024).

[25] Torrejon, J., Riou, M., Araujo, F. *et al.* Neuromorphic computing with nanoscale spintronic oscillators. *Nature* **547**, 428–431, DOI: https://doi.org/10.1038/nature23011 (2017).

[26] Gartside, J.C., Stenning, K.D., Vanstone, A. *et al.* Reconfigurable training and reservoir computing in an artificial spin-vortex ice via spin-wave fingerprinting. *Nat. Nanotechnol.* 460–469, DOI: https://doi.org/10.1038/s41565-022-01091-7 (2022).

[27] Allwood, D. A. *et al.* A perspective on physical reservoir computing with nanomagnetic devices. *Appl. Phys. Lett.* **122**, 040501, DOI: https://doi.org/10.1063/5.0119040 (2023).

[28] Vidamour, I.T., Swindells, C., Venkat, G. *et al.* Reconfigurable reservoir computing in a magnetic metamaterial. *Commun. Phys.* **6**, 230, DOI: https://doi.org/10.1038/s42005-023-01352-4 (2023).

[29] Yokouchi, T. *et al.* Pattern recognition with neuromorphic computing using magnetic field–induced dynamics of skyrmions. *Sci. Adv.* **8**, eabq5652, DOI: https://www.science.org/doi/abs/10.1126/sciadv.abq5652 (2022).

[30] Körber, L., Heins, C., Hula, T. *et al.* Pattern recognition in reciprocal space with a magnon-scattering reservoir. *Nat. Commun.* **14**, 3954, DOI: https://doi.org/10.1038/s41467-023-39452-y (2023).

[31] Furuta, T. *et al.* Macromagnetic simulation for reservoir computing utilizing spin dynamics in magnetic tunnel junctions. *Phys. Rev. Appl.* **10**, 034063, DOI: https://link.aps.org/doi/10.1103/PhysRevApplied.10.034063 (2018).

[32] Edwards, A.J., Bhattacharya, D., Zhou, P. *et al.* Passive frustrated nanomagnet reservoir computing. *Commun. Phys.* **6**, 215, DOI: https://doi.org/10.1038/s42005-023-01324-8 (2023).

[33] Taniguchi, T., Ogihara, A., Utsumi, Y. & Tsunegi, S. Spintronic reservoir computing without driving current or magnetic field. *Sci. Reports* **12**, 10627, DOI: https://doi.org/10.1038/s41598-022-14738-1 (2022).

[34] Papp, A., Csaba, G. & Porod, W. Characterization of nonlinear spin-wave interference by reservoir-computing metrics. *Appl. Phys. Lett.* **119**, DOI: https://doi.org/10.1063/5.0048982 (2021).

[35] Pinna, D., Bourianoff, G. & Everschor-Sitte, K. Reservoir Computing with Random Skyrmion Textures. *Phys. Rev. Appl.* **14**, 054020, DOI: https://link.aps.org/doi/10.1103/PhysRevApplied.14.054020 (2020).

[36] Watt, S. & Kostylev, M. Reservoir Computing Using a Spin-Wave Delay-Line Active-Ring Resonator Based on Yttrium-Iron-Garnet Film. *Phys. Rev. Appl.* **13**, 034057, DOI: https://link.aps.org/doi/10.1103/PhysRevApplied.13.034057 (2020).

[37] Nomura, H. *et al.* Reservoir computing with dipole-coupled nanomagnets. *Jpn. J. Appl. Phys.* **58**, 070901, DOI: https://doi.org/10.7567/1347-4065/ab2406 (2019).

[38] Marković, D. *et al.* Reservoir computing with the frequency, phase, and amplitude of spin-torque nano-oscillators. *Appl. Phys. Lett.* **114**, DOI: https://doi.org/10.1063/1.5079305 (2019).

[39] Lee, M.-K. & Mochizuki, M. Reservoir Computing with Spin Waves in a Skyrmion Crystal. *Phys. Rev. Appl.* **18**, 014074, DOI: https://link.aps.org/doi/10.1103/PhysRevApplied.18.014074 (2022).

[40] Van der Sande, G., Brunner, D. & Soriano, M. C. Advances in photonic reservoir computing. *Nanophotonics* **6**, 561–576, DOI: https://doi.org/10.1515/nanoph-2016-0132 (2017).

[41] Larger, L. *et al.* High-Speed Photonic Reservoir Computing Using a Time-Delay-Based Architecture: Million Words per Second Classification. *Phys. Rev. X* **7**, 011015, DOI: https://link.aps.org/doi/10.1103/PhysRevX.7.011015 (2017).

[42] García-Beni, J., Giorgi, G. L., Soriano, M. C. & Zambrini, R. Scalable Photonic Platform for Real-Time Quantum Reservoir Computing. *Phys. Rev. Appl.* **20**, 014051, DOI: https://link.aps.org/doi/10.1103/PhysRevApplied.20.014051 (2023).

[43] Rafayelyan, M., Dong, J., Tan, Y., Krzakala, F. & Gigan, S. Large-Scale Optical Reservoir Computing for Spatiotemporal Chaotic Systems Prediction. *Phys. Rev. X* **10**, 041037, DOI: https://link.aps.org/doi/10.1103/PhysRevX.10.041037 (2020).

[44] Vandoorne, K., Mechet, P., Van Vaerenbergh, T. *et al.* Experimental demonstration of reservoir computing on a silicon photonics chip. *Nat. Commun.* **5**, 3541, DOI: https://doi.org/10.1038/ncomms4541 (2014).

[45] Ng, W. K. *et al.* Retinomorphic Machine Vision in a Network Laser, DOI: https://arxiv.org/abs/2407.15558 (2024). 2407.15558.

[46] Jones, B., Stekel, D., Rowe, J. & Fernando, C. Is there a Liquid State Machine in the Bacterium Escherichia Coli? In *2007 IEEE Symposium on Artificial Life*, 187–191, DOI: https://doi.org/10.1109/ALIFE.2007.367795 (2007).

[47] Maass, W., Joshi, P. & Sontag, E. D. Computational Aspects of Feedback in Neural Circuits. *PLOS Comput. Biol.* **3**, 1–20, DOI: https://doi.org/10.1371/journal.pcbi.0020165 (2007).

[48] Milano, G., Pedretti, G., Montano, K. *et al.* In materia reservoir computing with a fully memristive architecture based on self-organizing nanowire networks. *Nat. Mater.* **21**, DOI: https://doi.org/10.1038/s41563-021-01099-9 (2022).

[49] Du, C., Cai, F., Zidan, M.A. *et al.* Reservoir computing using dynamic memristors for temporal information processing. *Nat. Commun.* **8**, 2204, DOI: https://doi.org/10.1038/s41467-017-02337-y (2017).

[50] Moon, J., Ma, W., Shin, J.H. *et al.* Temporal data classification and forecasting using a memristor-based reservoir computing system. *Nat. Electron.* **2**, 480–487, DOI: https://doi.org/10.1038/s41928-019-0313-3 (2019).

[51] Obst, O. *et al.* Nano-scale reservoir computing. *Nano Commun. Networks* **4**, 189–196, DOI: https://doi.org/10.1016/j.nancom.2013.08.005 (2013).

[52] Fujita, K., Yonekura, S., Nishikawa, S., Niiyama, R. & Kuniyoshi, Y. Physical reservoir computing in tensegrity with structural softness and ground collision dynamics. *J. Inst. Ind. Appl. Eng* **6**, 92–99, DOI: https://doi.org/10.12792/jiiae.6.92 (2018).

[53] Hauser, H., Ijspeert, A. J., Füchslin, R. M., Pfeifer, R. & Maass, W. The role of feedback in morphological computation with compliant bodies. *Biol. Cybern.* **106**, 595–613, DOI: https://doi.org/10.1007/s00422-012-0516-4 (2012).

[54] Trouvain, N., Pedrelli, L., Dinh, T. T. & Hinaut, X. ReservoirPy: An Efficient and User-Friendly Library to Design Echo State Networks. In *International Conference on Artificial Neural Networks*, 494–505, DOI: https://doi.org/10.1007/978-3-030-61616-8_40 (Springer, 2020).

[55] Miao, S., Kulseng, O. T., Stasik, A. & Fuchs, F. G. QuantumReservoirPy: A Software Package for Time Series Prediction. *arXiv:2401.10683* DOI: https://doi.org/10.48550/arXiv.2401.10683 (2024).

[56] Cabessa, J., Hernault, H., Lamonato, Y., Rochat, M. & Levy, Y. Z. The EsnTorch Library: Efficient Implementation of Transformer-Based Echo State Networks. In *International Conference on Neural Information Processing*, 235–246, DOI: https://doi.org/10.1007/978-981-99-1648-1_20 (Springer, 2022).

[57] Lee, O. *et al.* Perspective on unconventional computing using magnetic skyrmions. *Appl. Phys. Lett.* **122**, DOI: https://doi.org/10.1063/5.0148469 (2023).

[58] Jaeger, H. The "echo state" approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Ger. Ger. Natl. Res. Cent. for Inf. Technol. GMD Tech. Rep.* **148**, 13, DOI: https://api.semanticscholar.org/CorpusID:15467150 (2001).

[59] Maass, W., Natschläger, T. & Markram, H. Real-Time Computing Without Stable States: A New Framework for Neural Computation Based on Perturbations. *Neural Comput.* **14**, 2531–2560, DOI: https://doi.org/10.1162/089976602760407955 (2002).

[60] Steil, J. Backpropagation-decorrelation: online recurrent learning with O(N) complexity. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*, vol. 2, 843–848 vol.2, DOI: https://doi.org/10.1109/IJCNN.2004.1380039 (2004).

[61] Nakajima, K. Physical Reservoir Computing—an Introductory Perspective. *Jpn. J. Appl. Phys.* **59**, 060501, DOI: https://doi.org/10.35848/1347-4065/ab8d4f (2020).

[62] Jaeger, H. A tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network approach". *Ger. Natl. Res. Cent. for Inf. Technol. GMD Tech. Rep.* **5**, DOI: https://api.semanticscholar.org/CorpusID:192593367 (2002).

[63] Cucchi, M., Abreu, S., Ciccone, G., Brunner, D. & Kleemann, H. Hands-on reservoir computing: a tutorial for practial implementation. *Neuromorphic Comput. Eng.* **2**, 032002, DOI: https://doi.org/10.1088/2634-4386/ac7db7 (2022).

[64] Hilt, D. E. & Seegrist, D. W. *Ridge, a computer program for calculating ridge regression estimates*, vol. 236 (Upper Darby, Pa, Dept. of Agriculture, Forest Service, Northeastern Forest Experiment Station, 1977). https://research.fs.usda.gov/treesearch/19260.

[65] Love, J. *et al.* Spatial analysis of physical reservoir computers. *Phys. Rev. Appl.* **20**, 044057, DOI: https://link.aps.org/doi/10.1103/PhysRevApplied.20.044057 (2023).

[66] Jaeger, H. *et al. Short term memory in echo state networks*, vol. 5 (GMD-Forschungszentrum Informationstechnik Bremen, Germany, 2001). https://doi.org/10.24406/publica-fhg-291107.

[67] Roy, O. & Vetterli, M. The effective rank: A measure of effective dimensionality. *Eur. Signal Process. Conf. (EUSIPCO)* 606–610, DOI: https://api.semanticscholar.org/CorpusID:12184201 (2007).

[68] Dambre, J., Verstraeten, D., Schrauwen, B. & Massar, S. Information Processing Capacity of Dynamical Systems. *Sci. Reports* **2**, 514, DOI: https://doi.org/10.1038/srep00514 (2012).

[69] Jaeger, H. & Haas, H. Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication. *Science* **304**, 78–80, DOI: https://doi.org/10.1126/science.1091277 (2004).

[70] Khovanov, I. A. Stochastic approach for assessing the predictability of chaotic time series using reservoir computing. *Chaos: An Interdiscip. J. Nonlinear Sci.* **31**, 083105, DOI: https://doi.org/10.1063/5.0058439 (2021).

[71] Inubushi, M. & Yoshimura, K. Reservoir Computing Beyond Memory-Nonlinearity Trade-off. *Sci. Reports* **7**, 10199, DOI: https://doi.org/10.1038/s41598-017-10257-6 (2017).

[72] Gallicchio, C., Micheli, A. & Pedrelli, L. Deep reservoir computing: A critical experimental analysis. *Neurocomputing* **268**, 87–99, DOI: https://doi.org/10.1016/j.neucom.2016.12.089 (2017).

[73] Gallicchio, C. & Micheli, A. Echo state property of deep reservoir computing networks. *Cogn. Comput.* **9**, 337–350, DOI: https://doi.org/10.1007/s12559-017-9461-9 (2017).

[74] Manneschi, L. *et al.* Exploiting Multiple Timescales in Hierarchical Echo State Networks. *Front. Appl. Math. Stat.* **6**, 616658, DOI: https://doi.org/10.3389/fams.2020.616658 (2021).

[75] Manneschi, L. *et al.* Optimising network interactions through device agnostic models. *arXiv preprint arXiv:2401.07387* DOI: https://doi.org/10.48550/arXiv.2401.07387 (2024).

[76] Appeltant, L., Soriano, M., Van der Sande, G. *et al.* Information processing using a single dynamical node as complex system. *Nat. communications* **2**, 468, DOI: https://doi.org/10.1038/ncomms1476 (2011).

[77] Mackey, M. C. & Glass, L. Oscillation and Chaos in Physiological Control Systems. *Science* **197**, 287–289, DOI: https://doi.org/10.1126/science.267326 (1977).