# simlandr: Simulation-Based Landscape Construction for Dynamical Systems

**Jingmeng Cui**
University of Groningen
Radboud University

**Merlijn Olthof**
Radboud University

**Anna Lichtwarck-Aschoff**
University of Groningen
Radboud University

**Tiejun Li**
Peking University

**Fred Hasselman**
Radboud University

**Abstract**

We present the **simlandr** package for R, which provides a set of tools for constructing potential landscapes for dynamic systems using Monte Carlo simulation. Potential landscapes can be used to quantify the stability of system states. While the canonical form of a potential function is defined for gradient systems, generalized potential functions can also be defined for non-gradient dynamical systems. Our method is based on the potential landscape definition by Wang, Xu, and Wang (2008), and can be used for a large variety of models. Using two multistable dynamical systems as examples, we illustrate how **simlandr** can be used for model simulation, landscape construction, and barrier height calculation.

*Keywords*: dynamical systems, potential landscape, Monte Carlo simulation, R.

**Note: this is a preprint submitted for publication and also serves as a tutorial/vignette for simlandr. The manuscript has not yet been peer-reviewed.**

## 1. Introduction

To better understand a dynamical system, it is often important to know the stability of different states. The metaphor of a potential landscape consisting of hills and valleys has been used to illustrate differences in stability in many fields, including biology (Wang, Zhang, Xu, and Wang 2011; Waddington 1966), ecology (Lamothe, Somers, and Jackson 2019), and psychology (Olthof, Hasselman, Oude Maatman, Bosman, and Lichtwarck-Aschoff 2020). In such a landscape, the stable states of the system correspond to the lowest points (minima) in the valleys of the landscape. Just like a ball that is thrown in such a landscape will eventually gravitate towards such a minimum, the dynamical system is conceptually more likely to visit its stable states in which the system is also more resilient to noise. Yet, formally quantifying the stability of states is a nontrivial question.

Here we present an R package, **simlandr**, which is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=simlandr`. **simlandr** can quantify the stability of various kinds of systems without many mathematical restrictions.

It was originally developed for psychological formal models (Cui, Lichtwarck-Aschoff, Olthof, Li, and Hasselman 2021), but the workflow can be generalized to many other fields wherever quantifying the stability is needed.

Dynamical systems are usually modeled by stochastic differential equations, which take the form of

$$\frac{\mathrm{d}\mathbf{X}}{\mathrm{d}t} = \mathbf{b}(\mathbf{X}) + \sigma(\mathbf{X})\frac{\mathrm{d}\mathbf{W}}{\mathrm{d}t}, \tag{1}$$

where $\mathbf{X}$ is the random variable representing the state of the system.[1] The first term on the right-hand side of Equation 1 represents the deterministic part of the dynamics, which is a function of the system state $\mathbf{b}(\mathbf{X})$. The second term represents the stochastic part, which is standard white noise $\mathrm{d}\mathbf{W}/\mathrm{d}t$ multiplied by the noise strength $\sigma(\mathbf{X})$.

If the dynamical equation 1 can be written in the following form

$$\frac{\mathrm{d}\mathbf{X}}{\mathrm{d}t} = -\nabla U + \sqrt{2}\frac{\mathrm{d}\mathbf{W}}{\mathrm{d}t}, \tag{2}$$

then $U$ is the potential function of the system.[2]. However, this is not possible for general dynamical systems. The trajectory of such system may contain loops which are not possible to be represented by a gradient system (this issue was compared to Escher's stairs by Rodríguez-Sánchez, van Nes, and Scheffer 2020). In this case, further generalization is needed. The theoretical background of **simlandr** is the generalized potential landscape by Wang *et al.* (2008), which is based on the Boltzmann distribution and the steady-state distribution of the system. The Boltzmann distribution is a distribution law in physics, which states the distribution of classical particles depends on the energy level they occupy. When the energy is higher, the particle is exponentially less likely to be in such states

$$P(\mathbf{x}) \propto \exp(-U). \tag{3}$$

This is then linked to dynamical systems by the steady-state distribution. The steady-state distribution of stochastic differential equations is the distribution that does not change over time, denoted by $P_{\mathrm{SS}}$ which satisfies

$$\frac{\partial P_{\mathrm{SS}}(\mathbf{x}, t)}{\partial t} = 0. \tag{4}$$

The steady-state distribution is important because it extracts time-invariant information from a set of stochastic differential equations. Substituting the steady-state distribution to Equation 3 gives Wang's generalized potential landscape function (Wang *et al.* 2008)

$$U(\mathbf{x}) = -\ln P_{\mathrm{SS}}(\mathbf{x}). \tag{5}$$

If the system has ergodicity (i.e., after sufficient time it can travel to all possible states in the state space), the long-term sample distribution can be used to estimate the steady-state distribution, and the generalized potential function can be calculated.

Our approach is definitely not the only possible way for constructing potential landscapes. Many other theoretical approaches are available, including the SDE decomposition method

[1] The corresponding variable representing positions in the state space is not a random variable, so we use lowercase $\mathbf{x}$ for it as in Equation 3. This convention will be followed throughout this article.

[2] Under zero inertia approximation.

by Ao (2004) and the quasi-potential by Zhou, Aliyu, Aurell, and Huang (2012) and Lv, Li, Li, and Li (2014), and various strategies to numerically compute these landscapes have been proposed (see Zhou and Li 2016, for a review). However, available realizations are still rare. To our knowledge, the only existing software specifically for computing potential landscapes is **rolldown** by Rodríguez-Sánchez (2020). This package is based on the decomposition of the Jacobian over a grid of points. It, therefore, requires the function to be deterministic and differentiable in the whole state space. Additionally, **rolldown** currently only supports one- or two-dimensional systems. Expanding the method to a higher dimensional case is principally possible, but because the method requires grid points in all dimensions, many difficulties in computation and visualization are to be expected. **simlandr**, on the contrary, is based on Monte-Carlo simulation and the steady-state distribution. It does not have specific requirements for the model. Even for the models that are not globally differentiable and have history-dependence, **simlandr** is still applicable by assuming ergodicity (e.g., Cui *et al.* 2021). The required simulation length only depends on the time scale on which ergodicity is achieved, which is not related to the dimensionality of the system. Therefore, it is also suitable for high-dimensional systems. Finally, we remark that we do not consider rare events sampling in which the noise strength $\sigma(\mathbf{X})$ is extremely small.

In the following part of this article, we will illustrate how to compute the potential landscape for two example models using **simlandr**. The workflow is divided into three parts: model simulation, landscape construction, and barrier calculation. We first load **simlandr** before going into the illustration.

```
R> library(simlandr)
```

# 2. Model simulation

## 2.1. Single simulation

As mentioned above, dynamic models often take the form of stochastic differential equations. For numerical computation, these models are often simulated using numerical approximations, for example, the Euler-Maruyama method. An R function is usually needed to simulate how the model variables change over time. It takes some parameters as input and returns a recording of the variables during the simulation process. We will refer to these kinds of functions as *simulation functions*.

The `sim_fun_grad()` and `sim_fun_nongrad()` functions (available in **simlandr** from 0.1.3 version) will be used for illustration. The `sim_fun_grad()` function represents a two-dimensional stochastic gradient system, whose potential function is specified as

$$U(x, y) = x^4 + y^4 + axy + bx + cy. \tag{6}$$

Therefore, the actual potential function of the system is known and can be compared with the results generated by **simlandr**. The dynamic functions of the system are

$$\frac{\mathrm{d}X}{\mathrm{d}t} = -\frac{\partial U}{\partial x} + \sigma\frac{\mathrm{d}V}{\mathrm{d}t} = -4x^3 - ay - b + \sigma\frac{\mathrm{d}V}{\mathrm{d}t}, \tag{7}$$

$$\frac{\mathrm{d}Y}{\mathrm{d}t} = -\frac{\partial U}{\partial y} + \sigma\frac{\mathrm{d}W}{\mathrm{d}t} = -4y^3 - ax - c + \sigma\frac{\mathrm{d}W}{\mathrm{d}t}, \tag{8}$$

where $V$ and $W$ are independent standard Brownian motions.

With the default parameter setting $a = -4, b = c = 0, \sigma = 1$, we generate the following simulation data.

```
R> single_output_grad <- sim_fun_grad(length = 1e5, seed = 1614)
```

The function `sim_fun_nongrad()` represents a two-dimensional stochastic non-gradient model by Wang *et al.* (2011). This model is based on the mutual regulations of gene expressions. If $X_1$ and $X_2$ represent the expression levels of two genes which activate themselves and inhibit each other, their dynamic functions can be written as

$$\frac{\mathrm{d}X_1}{\mathrm{d}t} = \frac{ax_1^n}{S^n + x_1^n} + \frac{bS^n}{S^n + x_2^n} - kx_1 + \sigma_1 \frac{\mathrm{d}W_1}{\mathrm{d}t}, \tag{9}$$

$$\frac{\mathrm{d}X_2}{\mathrm{d}t} = \frac{ax_2^n}{S^n + x_2^n} + \frac{bS^n}{S^n + x_1^n} - kx_2 + \sigma_2 \frac{\mathrm{d}W_2}{\mathrm{d}t}, \tag{10}$$

$$\frac{\mathrm{d}a}{\mathrm{d}t} = -\lambda a + \sigma_3 \frac{\mathrm{d}W_3}{\mathrm{d}t}, \tag{11}$$

where $a$ represents the strength of self-activation, $b$ represents the strength of mutual-inhibition, and $k$ represents the speed of degradation. The development of an organism is modeled as $a$ decreasing at a certain speed $\lambda$. This model cannot be written in the form of Equation 2, thus no strictly-defined potential function exists. The generalized potential function obtained by an analytical method is provided by Wang *et al.* (2011). This landscape will be compared with the result generated by **simlandr**.

With the default parameter setting $b = 1, k = 1, S = 0.5, n = 4, \lambda = 0.01, \sigma_1^2 = \sigma_2^2 = 8, \sigma_3^2 = 2$, and $a$ bounded within [-0.3,1.8], we generate the following simulation data.

```
R> single_output_nongrad <- sim_fun_nongrad(
R>   length = 1e6,
R>   stepsize = 0.1,
R>   seed = 1614
R> )
```

If you already have a function of a dynamic model, you can just run it to produce output without having to use **simlandr**. However, for Monte-Carlo methods, it is important that the simulation *converges*. In our case, it means the distribution of the system is roughly stable. Only when the distribution estimation is good enough can we construct reasonable landscapes based on that. **simlandr** provides a function `check_conv()` to check if the simulation converges. It takes the initial, middle, and final parts of the simulation result and calculates distributions based on that. If these distributions look similar, we can say the simulation converges. The convergence check result for the gradient example can then be obtained by

```
R> check_conv(single_output_grad, var = c("x"))[[1]]
```

and the resulting plot is shown in Figure 1.

An exception in which this method does not work well is when the simulation is so short or the noise of the system is so small that during the whole simulation the system is around one

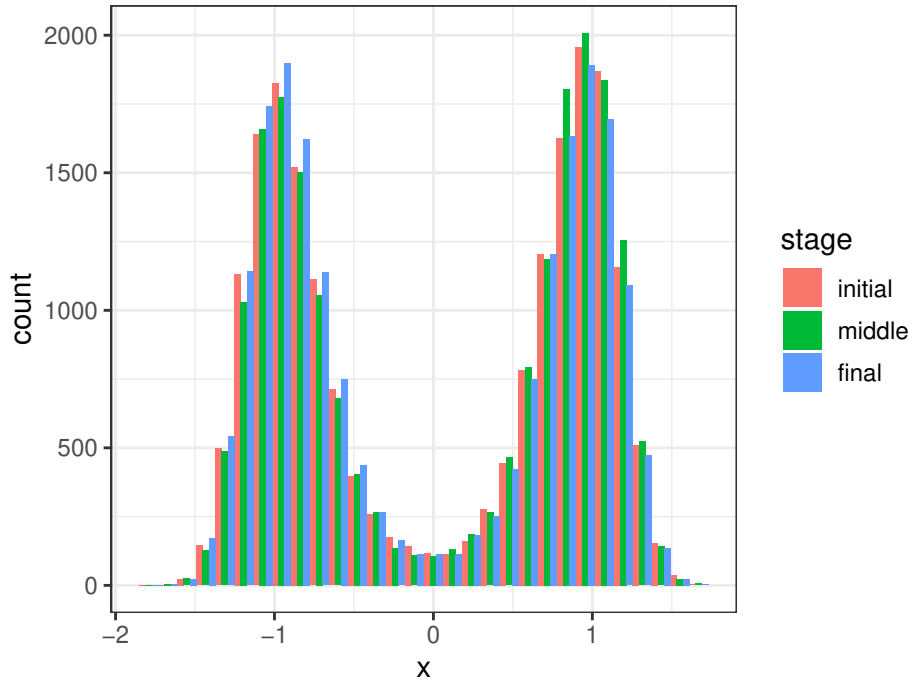


Figure 1: Convergence check result for `x`.

local stable state. Rough knowledge of the system and its stable states would be needed to rule out this situation.

## 2.2. Out-of-memory computation

Sometimes the output of the simulation is so large that it cannot be handled properly in the following computation. The size of `single_output_nongrad`, for example, is 30.5 MB. If the simulation length and the number of variables are extended, the object size can easily grow into GBs.[3] In this case, you can use the **bigmemory** package (Kane, Emerson, and Weston 2013). **bigmemory** only preserves a pointer in the RAM memory, so it can reduce memory load significantly. In most cases, you can treat it as a normal matrix. The functions in **simlandr** are fully compatible with **bigmemory**.[4] The following are the example codes for `single_output_nongrad`.

```
R> single_output_nongrad <-
R>   bigmemory::as.big.matrix(single_output_nongrad,
R>     backingfile = "single_output_nongrad.bin",
```

[3]Rule of thumb: retain matrices of $> 1$ GB in memory is likely to produce future problems in a computer with 8 GB memory.

[4] At the moment, there is an issue associated with the variable inspector of RStudio that can cause crashes when loading the workspace image with previous `big.matrix` or `hash_big.matrix` objects. If you are using R with RStudio, switching the variable inspector to "Manual Refresh Only" before attaching the external data files can prevent this problem. See https://github.com/rstudio/rstudio/issues/8923 and https://github.com/kaneplusplus/bigmemory/issues/106 for discussions.

```
R>     descriptorfile = "single_output_nongrad.desc"
R>   )
```

To reuse these images on a hard drive, `big.matrix` class in `bigmemory` requires an explicit file name for each matrix. This can be cumbersome if you need to handle a lot of matrices. (And this is even a bigger problem for batch simulation; see Section 2.3.) Therefore, `simlandr` provides a S4 `hash_big.matrix` class to solve this problem. The `hash_big.matrix` class is a modification of the `big.matrix` class from the **bigmemory** package, but it automatically generates the file names using the md5 values of the matrices, calculated by the **digest** package (Eddelbuettel, Lucas, Tuszynski, Bengtsson, Urbanek, Frasca, Lewis, Stokely, Muehleisen, Murdoch, Hester, Wu, Kou, Onkelinx, Lang, Simko, Hornik, Neal, Bell, de Queljoe, Suruceanu, Denney, Schumacher, and Chang 2021). The md5 value is also stored in the `md5` slot of `hash_big.matrix` objects. Therefore, the file link can also be restored automatically without having to specify a file name. By default, all the backing files of the `hash_big.matrix` objects are in the `\bp` directory. After this manipulation, the size of `single_output_nongrad` is reduced to only 0.9 KB.

```
R> single_output_nongrad <- as.hash_big.matrix(single_output_nongrad)
R> single_output_nongrad <- attach.hash_big.matrix(single_output_nongrad)
```

## 2.3. Batch simulation

Sometimes you need to run the models with different conditions. **simlandr** provides several tools to do this easily. Because **simlandr** needs to pass those condition settings into the simulation function and record their results properly, it requires the simulation function to have a certain format. The *arguments* of the simulation function should have a nested structure, which means they are lists that contain several *elements*. For example, in `sim_fun_grad()`, there is an argument `initial` that contains two elements `x` and `y`, which specify the initial value of $x$ and $y$. There is also an argument `parameter` that contains elements `a`, `b`, `c`, and `sigmasq`, which specify model parameters $a$, $b$, $c$, and $\sigma^2$. The simulation function should also give a matrix as the output. If your original simulation function is in another format, maybe you need to modify it or wrap it in another function.

To run a batch simulation, you need to first make a `arg_set` that describes the conditions of these simulations and then make an `arg_grid` out of it. An `arg_grid` represents a grid that contains all possible combinations of conditions. The following is an example that creates an `arg_grid` for `sim_fun_grad()`. It initializes a `new_arg_set()`, adds an element of an argument and the starting, end, and increment values of the element with `add_arg_ele()`, and `make_arg_grid()` from the `arg_set`.

```
R> batch_arg_set_grad <- new_arg_set()
R> batch_arg_set_grad <- batch_arg_set_grad %>%
+   add_arg_ele(
+     arg_name = "parameter", ele_name = "a",
+     start = -6, end = -1, by = 1
+   )
R> batch_grid_grad <- make_arg_grid(batch_arg_set_grad)
```

We are now ready to run the batch simulation for the example gradient model. **simlandr** uses out-of-memory storage for batch simulations by default because most times batch simulation will result in very large data objects. In this case, each simulation result is stored in a separate file with the `hash_big.matrix` class. **simlandr** also provides a `attach_all_matrices()` function to help you attach all the out-of-memory `hash_big.matrix`s related to a batch simulation. Use this if you want to load the previous workspace image with (out-of-memory) batch simulation results (but see Footnote 4).

```
R> batch_output_grad <- batch_simulation(batch_grid_grad, sim_fun_grad,
R>   default_list = list(
R>     initial = list(x = 0, y = 0),
R>     parameter = list(a = -4, b = 0, c = 0, sigmasq = 1)
R>   ),
R>   length = 1e5,
R>   seed = 1614
R> )
R> batch_output_grad <- attach_all_matrices(batch_output_grad)
```

If you want to keep all the original data in the memory, you can use `bigmemory = FALSE`.

The output of the `batch_simulation()` function is a `batch_simulation` object, which is, basically, a complex `data.frame` with simulation outputs and corresponding parameter values. You can manipulate it as a `data.frame` for your purpose. `batch_simulation` objects are also the base for constructing landscapes from multiple simulations.

## 3. Landscape construction

**simlandr** provides a set of tools to construct 2D, 3D, and 4D[5] landscapes from single or multiple simulation results. The steady-state distribution for selected variables of the system is first estimated using the kernel density estimates. The `density()` function in **stats** is used for 2D landscapes. The `kde()`function from the **ks** package (Duong 2021) is used by default for 3D and 4D landscapes. For 3D landscapes, you can also choose to use the smooth function from **MASS** (Venables and Ripley 2002) package by using `kde_fun = "MASS"`. The distribution information can be accessed using `get_dist()` function. Then potential function $U$ is calculated from Equation 5. The landscape plots without a z-axis are based on **ggplot2** (Wickham 2016), and those with a z-axis are based on **plotly** (Sievert 2020). These plots can be further refined using the standard **ggplot2** or **plotly** methods. See Table 1 for an overview for the family of landscape functions.

### 3.1. Single simulation landscape

This family of functions takes the output of a single simulation, a `matrix` object, as input. We use the data generated in Section 2.1 for illustration.

[5]In this package, we use the number of dimensions in landscape plots (including $U$) to define the dimension of landscapes. The x-, y-, z-, and color- axes can all be regarded as a dimention. Therefore, the dimension of a landscape can be one more than the dimension of the kernel smooth function.

| Type of Input | Function | Dimensions |
|---|---|---|
| Single simulation data | `make_2d_static()` | x, **y** |
| | `make_3d_static()` | (1) x, y, **z+color**;<br>(2) x, y, **color** |
| | `make_4d_static()` | x, y, z, **color** |
| Multiple simulation data | `make_2d_matrix()` | x, **y**, *cols*, *(rows)* |
| | `make_3d_matrix()` | x, y, **z+color**, *cols*, *(rows)* |
| | `make_3d_animation()` | (1) x, y, **z+color**, *fr*;<br>(2) x, y, **color**, *fr*;<br>(3) x, y, **z+color**, *cols* |

Table 1: Overview of various landscape functions provided by **simlandr**. Dimensions in bold represent the potential $U$ calculated by the function. Dimensions in italic represent model parameters. Dimensions in parentheses are optional.

The first landscape function is `make_2d_static()`. In the following example, the variable name of the `x` axis in the plot is `"x"`. `from` and `to` are used to specify the range of the variable, and `adjust` is used to specify the smooth level in the kernel estimate. `from`, `to`, and `adjust` will be passed to `stats::density()`. The example plot for the gradient model is shown in Figure 2.[6]

```
R> l_single_grad_2d <- make_2d_static(single_output_grad,
+   x = "x",
+   from = -2, to = 2, adjust = 2
+ )
R> plot(l_single_grad_2d)
```

The second landscape function is `make_3d_static()`. The parameter structure is similar to the previous function, but two variables need to be specified. The range of them is specified at `lims` and the smooth level is specified at `h`, as the parameter format in `MASS::kde2d()`. These two parameters will be passed to `ks::kde()` in the following example. The landscape can produce two kinds of plots, shown in Figure 3 (the upper panel) and 4, respectively. Comparing the landscape generated by **simlandr** (Figure 3, the upper panel) and the true landscape based on Equation 6, (Figure 3, the lower panel), we can see they have different values on the z-axis. This is because the potential function generated by **simlandr** is unitless. The absolute value of the potential function is not meaningful, but only the relative height among different states is. The shape of the reconstructed potential landscape by **simlandr** resembles the true landscape nicely, indicating the methods in **simlandr** are accurate.

```
R> l_single_grad_3d <- make_3d_static(single_output_grad,
R>   x = "x", y = "y",
R>   lims = c(-2, 2, -2, 2), h = 0.05,
R>   kde_fun = "ks"
```

[6]The plot is slightly asymmetric because the steady-state distribution is estimated with possible errors. It is not yet possible to quantify the magnitude of this error, but as long as the simulation converges well, the error of landscape construction should be generally acceptable.

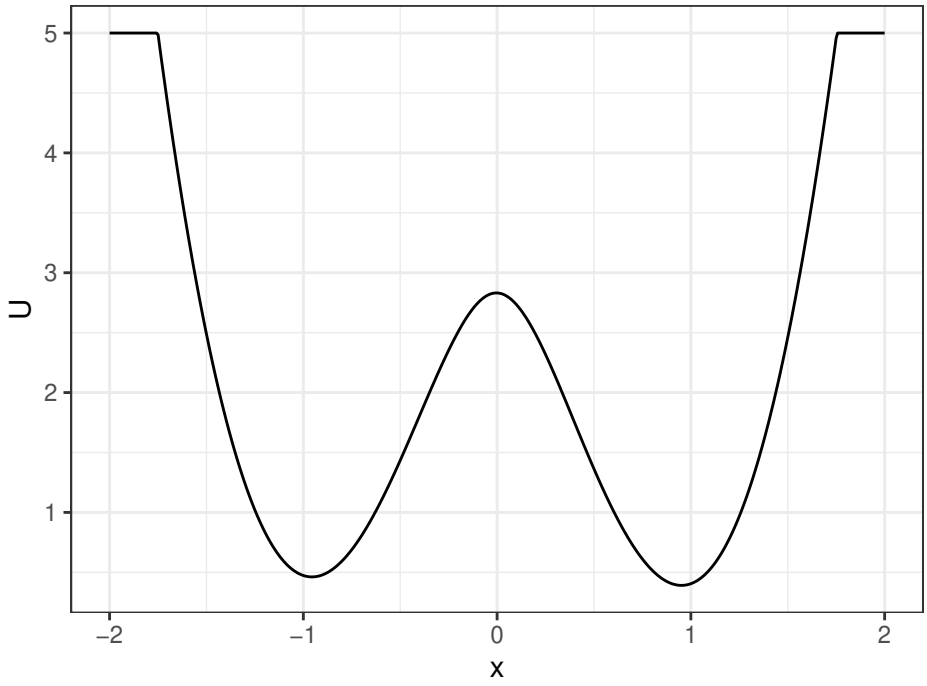


Figure 2: The 2D landscape for the gradient example.

```
R> )
R> plot(l_single_grad_3d)
R> plot(l_single_grad_3d, 2)
```

Here we also show the 3D landscape for the nongradient example. The following codes are used for generating the plot in Figure 5 (the upper panel). In this plot, the x-axis represents $\Delta x (= x_1 - x_2)$, and the y-axis represents $a$. To compare with, the potential landscape obtained analytically by Wang *et al.* (2011) is shown in Figure 5 (the lower panel). The result of **simlandr** appears to be very close to the result based on the analytical derivation.

```
R> l_single_nongrad_3d <- make_3d_static(single_output_nongrad,
R>   x = "delta_x", y = "a",
R>   lims = c(-3, 3, -0.5, 2), h = 0.005,
R>   Umax = 5
R> )
R> plot(l_single_nongrad_3d)
```

Finally, there is also a `make_4d_static()` function which can construct the potential landscape for a combination of three variables. In this case, the potential function is shown as the color of dots. The states with deeper colors have higher stability. Here we construct a potential landscape for the nongradient example, with the x-, y-, and z-axes representing $x_1$, $x_2$, and $a$, respectively. The resulting plot is shown in Figure 6.

```
R> l_single_nongrad_4d <- make_4d_static(single_output_nongrad,
R>   x = "x1", y = "x2", z = "a",
```

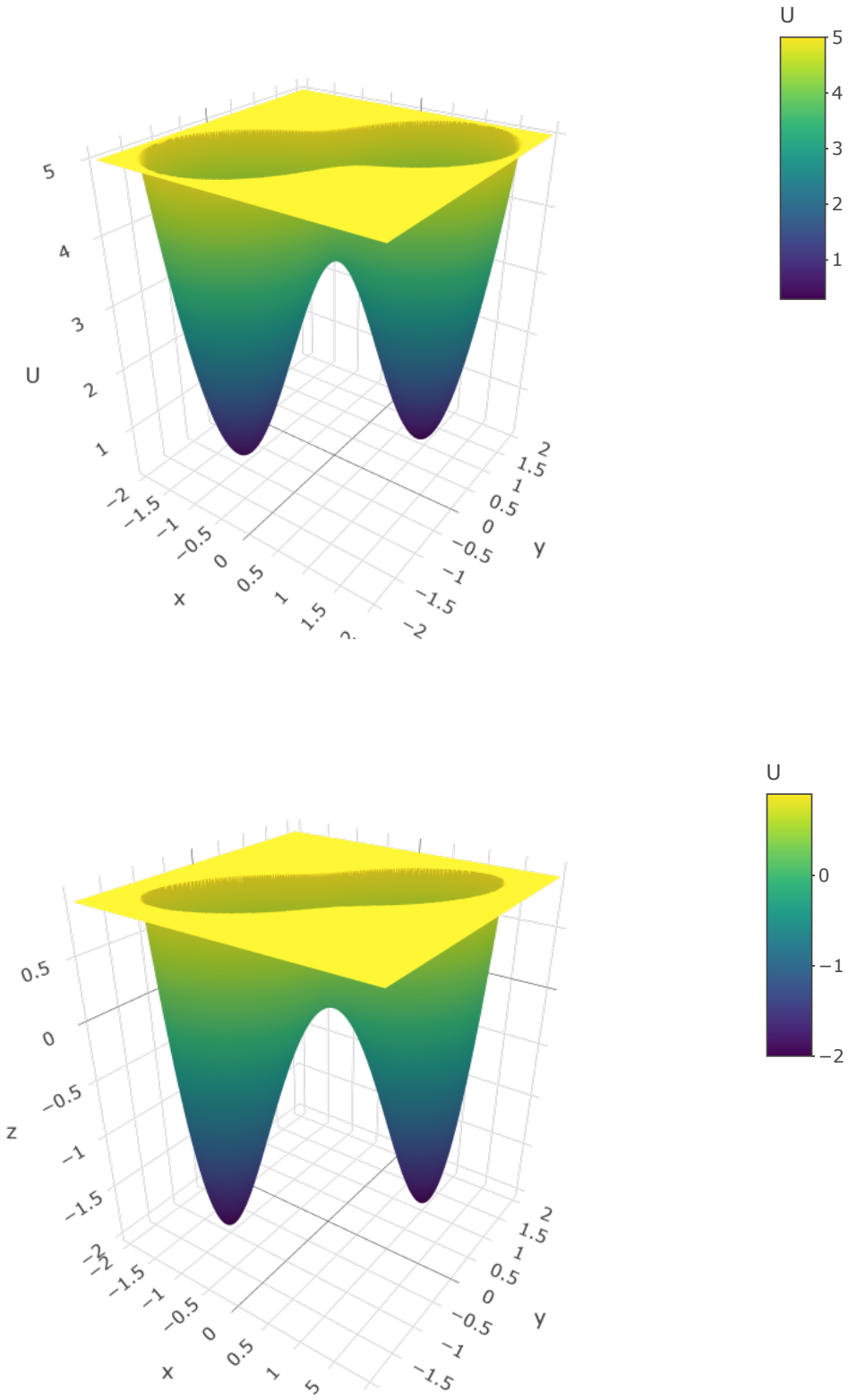


Figure 3: The 3D landscape (potential value as z-axis) for the gradient example. The upper panel is the plot produced by **simlandr**; the lower panel is the true potential function as in Equation 6.

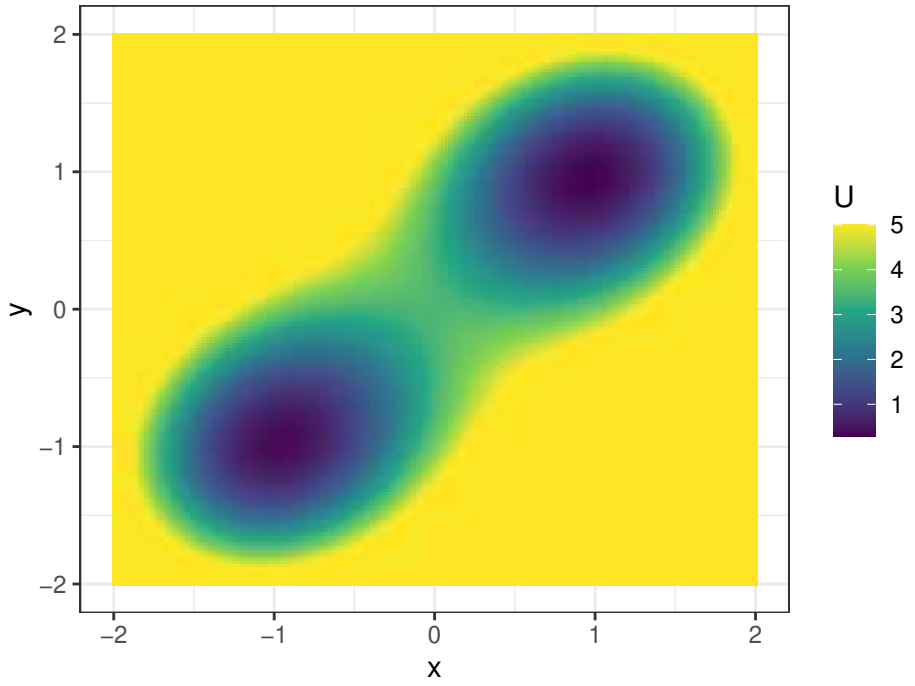


Figure 4: The 3D landscape (potential value as color) for the gradient example.

```
R>   lims = c(-2, 2, -2, 2, -0.5, 2), h = 0.005,
R>   Umax = 3
R> )
R> plot(l_single_nongrad_4d)
```

### 3.2. Multiple simulation landscape

The functions in this family construct landscapes simultaneously for multiple simulation results. The conditions and output for each simulation need to be recorded in detail. Hence, they take a `batch_simulation` object as input. We use the data generated in Section 2.3 for illustration.

The first function in this family is `make_2d_matrix()`, which constructs a matrix of landscapes for different parameters. The `rows` and columns (`cols`) of the matrix correspond to the parameter values. If there is only one parameter of interest, the `rows` argument can be left blank. In the following example, we construct the landscapes for different `a` values. The plot is shown in Figure 7.

```
R> l_batch_grad_2d <- make_2d_matrix(batch_output_grad,
+   x = "x", cols = "a", Umax = 10,
+   from = -2, to = 2, adjust = 2, individual_landscape = TRUE
+ )
R> plot(l_batch_grad_2d)
```

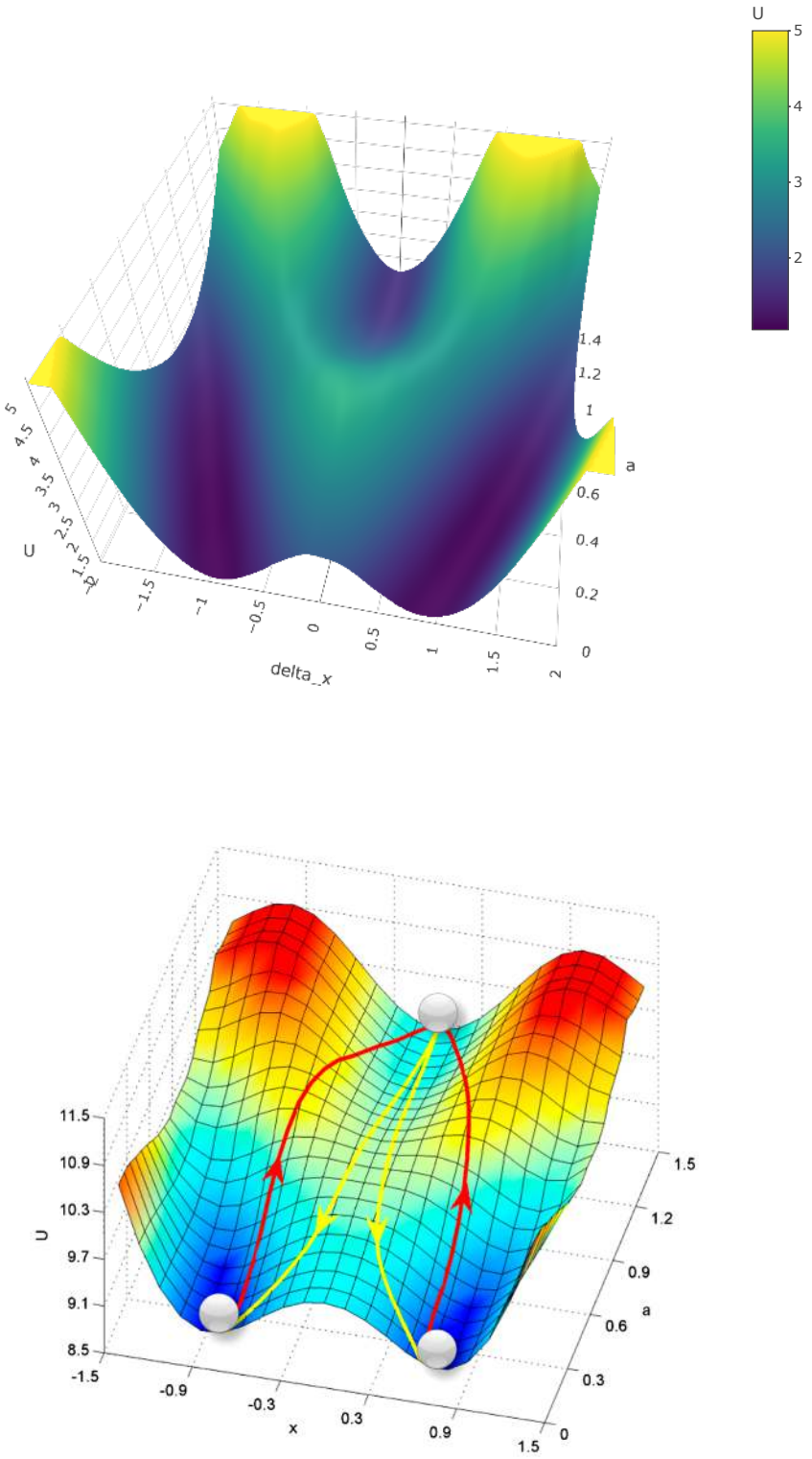


Figure 5: The 3D landscape (potential value as z-axis) for the nongradient example. The upper panel is the plot produced by **simlandr**; the lower panel is the potential landscape obtained analytically by Wang *et al.* (2011), reproduced with the permission of the authors and in accordance with the journal policy.

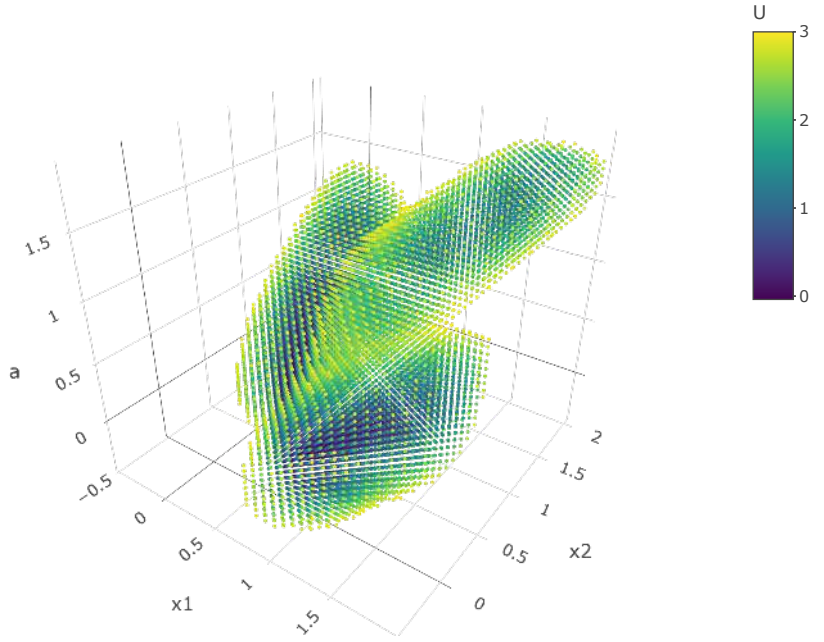


Figure 6: The 4D landscape for the nongradient example.

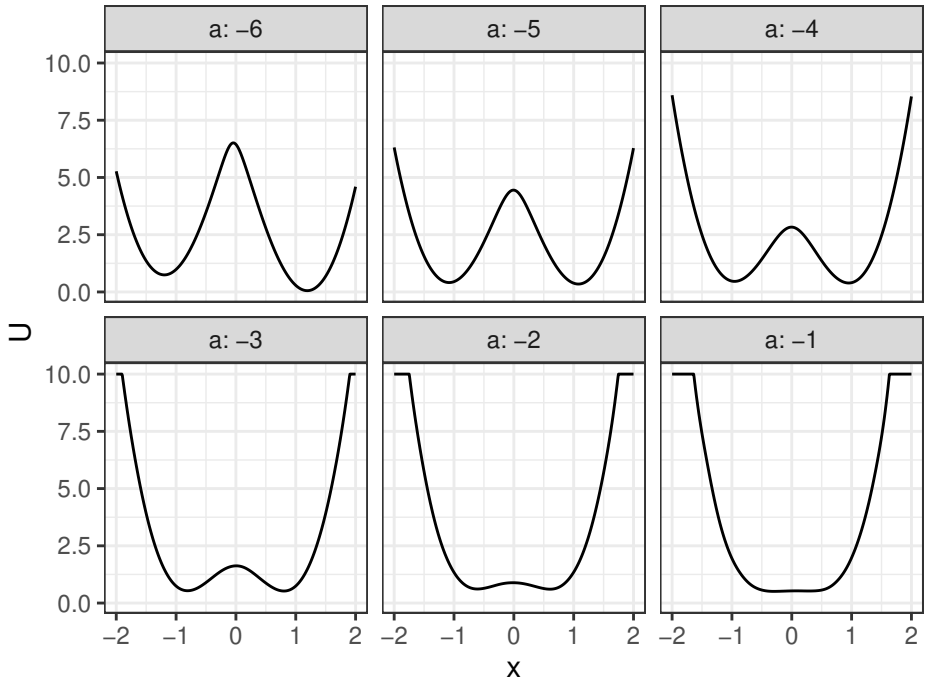


Figure 7: The 2D landscape matrix with different `a` for the gradient example.

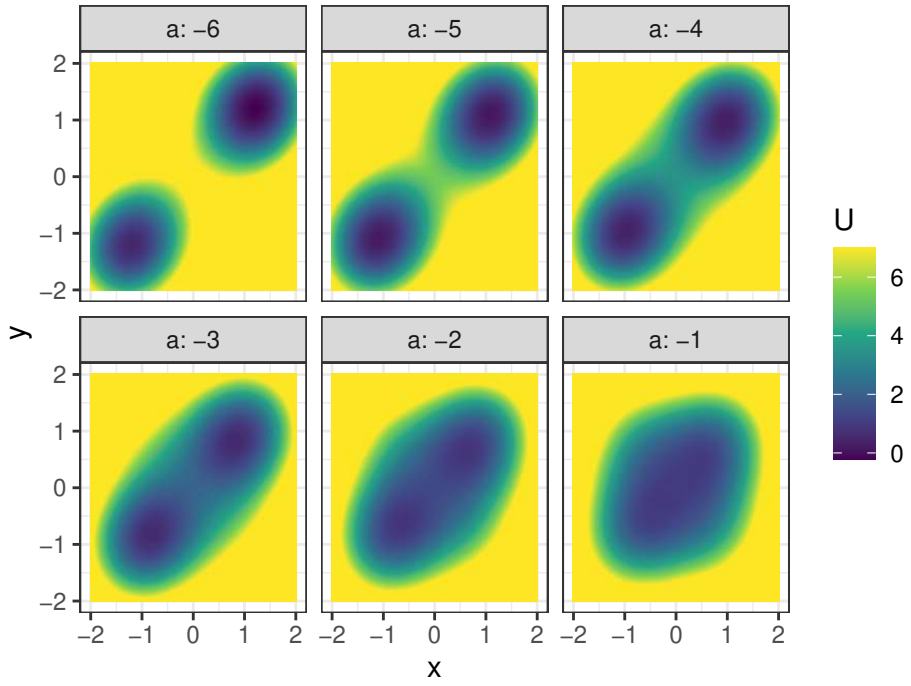


Figure 8: The 3D landscape matrix with different `a` for the gradient example.

The second function in this family is `make_3d_matrix()`. This is similar to `make_2d_matrix()`, but can show two variables. The example plot is shown in Figure 8.

```
R> l_batch_grad_3d <- make_3d_matrix(batch_output_grad,
+   x = "x", y = "y", cols = "a",
+   lims = c(-2, 2, -2, 2), h = 0.05, Umax = 7,
+   kde_fun = "ks", individual_landscape = TRUE
+ )
R> plot(l_batch_grad_3d)
```

Finally, it is also possible to show the influence of model parameters by creating animations. In the animations made by `make_3d_animation()`, the parameter will change over time, and the landscape changes accordingly. This animation cannot be shown on paper. Readers may run the following code on their devices and see the output.

```
R> l_batch_grad_3d_animation <- make_3d_animation(batch_output_grad,
R>   x = "x", y = "y", fr = "a",
R>   lims = c(-2, 2, -2, 2), h = 0.05,
R>   kde_fun = "ks"
R> )
R> plot(l_batch_grad_3d_animation)
R> plot(l_batch_grad_3d_animation, 2)
R> plot(l_batch_grad_3d_animation, 3)
```

# 4. Saddle point and barrier height

An important property of the states in a landscape is their stability, which can be indicated by the barrier height that the system has to overcome when it transitions between one stable state to another adjacent state (see Cui *et al.* 2021, for further discussions about different stability indicators). **simlandr** also provides tools to calculate the barrier heights from landscapes. You can use the S3 generic function `calculate_barrier()` to calculate the barrier for most landscapes. There are also specific `calculate_barrier_2d()`, `calculate_barrier_3d()`, `calculate_barrier_2d_batch()`, and `calculate_barrier_3d_batch()` functions available for 2D or 3D landscapes from single or batch simulation results, respectively. These functions look for the local minima in given regions and try to find the saddle point between the two[7]. Then the potential differences between the saddle point and local minima are calculated as barrier heights. The output of these functions is a `barrier` object, which contains both data and graphics. In this section, we show several examples for barrier calculations.

## 4.1. Single simulation barriers

In 2D cases, there is only one possible path connecting two points on the landscape. The `calculate_barrier()` function looks for local minima around `start_location_value` and `end_location_value` with a radius of `start_r` and `end_r`. If all the potential values in the region are equal to `Umax` (which represents effectively `Inf`), the barrier calculation functions will expand the searching area automatically.[8] Then, the point with the highest potential value on the way between two local minima is marked as the saddle point, and the potential differences between the saddle point and local minima are calculated as barrier heights. These values can be extracted using `get_barrier_height()`. The following is an example with the `l_single_grad_2d` landscape generated in Section 3.1.

```
R> b_single_grad_2d <- calculate_barrier(l_single_grad_2d,
+   start_location_value = -1, end_location_value = 1,
+   start_r = 0.3, end_r = 0.3
+ )
R> get_barrier_height(b_single_grad_2d)

delta_U_start   delta_U_end
     2.369303      2.440037
```

The `barrier` object contains a **ggplot2** `geom` object that can be added to the landscape plots to show the starting (white), end (white), and saddle (red) points. You can use `get_geom()` to access those geoms and add them to the landscape plot. The resulting plot is shown in Figure 9.

```
R> plot(l_single_grad_2d) + get_geom(b_single_grad_2d)
```

[7]For landscapes from multiple simulations, the searching regions for their starting and ending points are by default the same, but they can also be individually specified. We refer the interested readers to the `make_barrier_grid_2d()` and `make_barrier_grid_3d()` functions in **simlandr**

[8]You can use `expand = FALSE` to disable this feature.

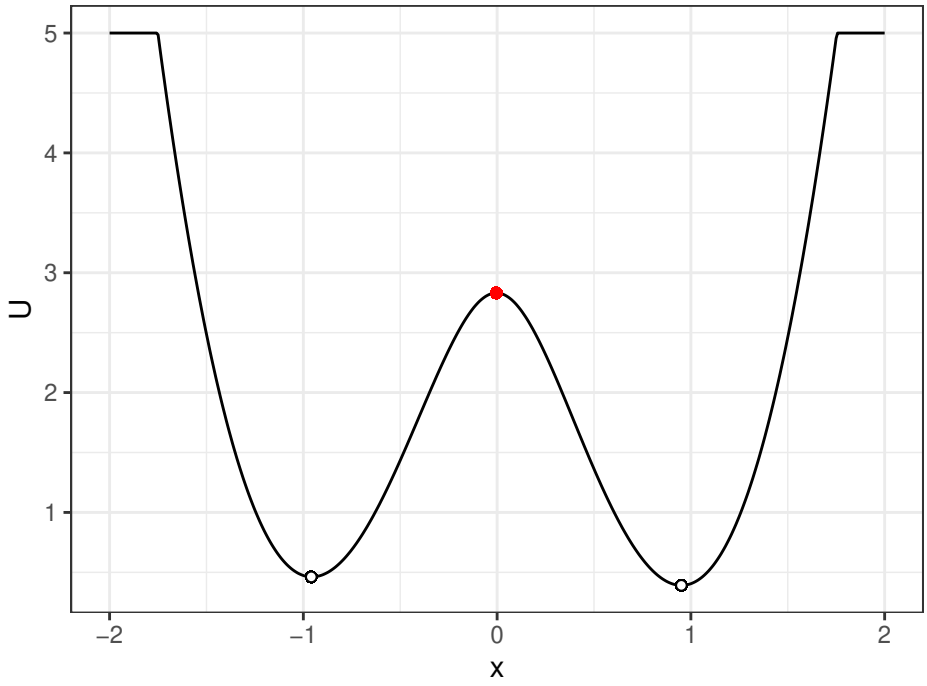


Figure 9: The example 2D landscape for the gradient model with local minima (white dots) and saddle point (red dot) marked.

It is also possible to only plot the positions of local minima and the saddle point with the S3 method for `plot()`.

For 3D landscapes, the local minima are searched in a square space with a half side length equals to `start_r` and `end_r` around a point specified by `start_location_value` and `end_location_value`. The point with the lowest potential value in the given region is set as the position of the stable state. The saddle point in this case is less obvious because there are multiple paths between two local minima. If we treat the system *as if* it is a gradient system with Brownian noise, then the most probable path (termed as the *minimum energy path*, MEP) that the system transitions is that it first goes along the steepest *ascent* path from the starting point, and then goes along the steepest *descent* path to the end point (E and Vanden-Eijnden 2010). We find this path by minimizing the following action using the Dijkstra (1959) algorithm (Heymann and Vanden-Eijnden 2008)

$$\varphi_{\text{MEP}} = \arg\min_{\varphi} \int_A^B |\nabla U||\mathrm{d}\varphi| \left(\approx \min_{\varphi} \Sigma_i |\nabla U_i||\Delta\varphi_i|\right), \tag{12}$$

where $A$ and $B$ are the starting and end points and $\varphi$ is the path starting at $A$ and ending in $B$. After that, we find the saddle point by searching for the point with the maximum potential value on the MEP. Note that while the barrier height still indicates the stability of local minima, the MEP may not be the true most probable path for a nongradient system to transition between stable states.

The MEP is shown by default in the `geom`s retrieved by `get_geom()` as a white line. You can disable showing this path by `path = FALSE`. The following are the example barrier calculations

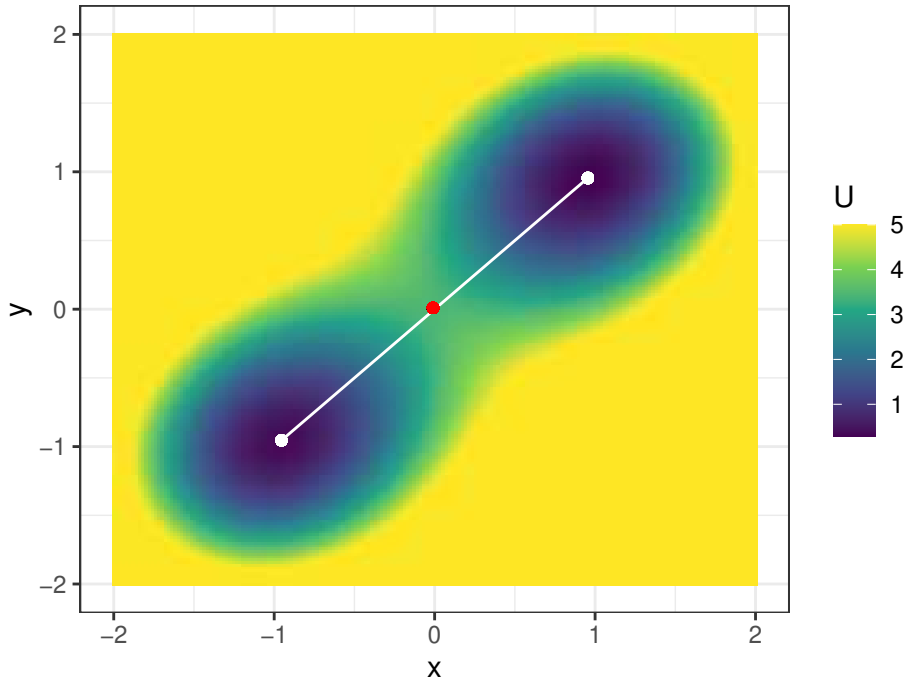


Figure 10: The example 3D static landscape for the gradient model with local minima (white dots), saddle point (red dot), and the MEP (white line) marked.

for the gradient and nongradient example models. Their landscape plots with local minima, saddle points, and MEPs are shown in Figure 10 and 11, respectively.

```
R> b_single_grad_3d <- calculate_barrier(l_single_grad_3d,
+   start_location_value = c(-1, -1), end_location_value = c(1, 1),
+   start_r = 0.3, end_r = 0.3
+ )
R> get_barrier_height(b_single_grad_3d)

delta_U_start   delta_U_end
     3.087025      3.160844

R> plot(l_single_grad_3d, 2) + get_geom(b_single_grad_3d)


R> b_single_nongrad_3d <- calculate_barrier(l_single_nongrad_3d,
+   start_location_value = c(0, 1.2), end_location_value = c(1, 0.2),
+   start_r = 0.3, end_r = 0.3
+ )
R> get_barrier_height(b_single_nongrad_3d)

delta_U_start   delta_U_end
     1.511747      2.047103
```

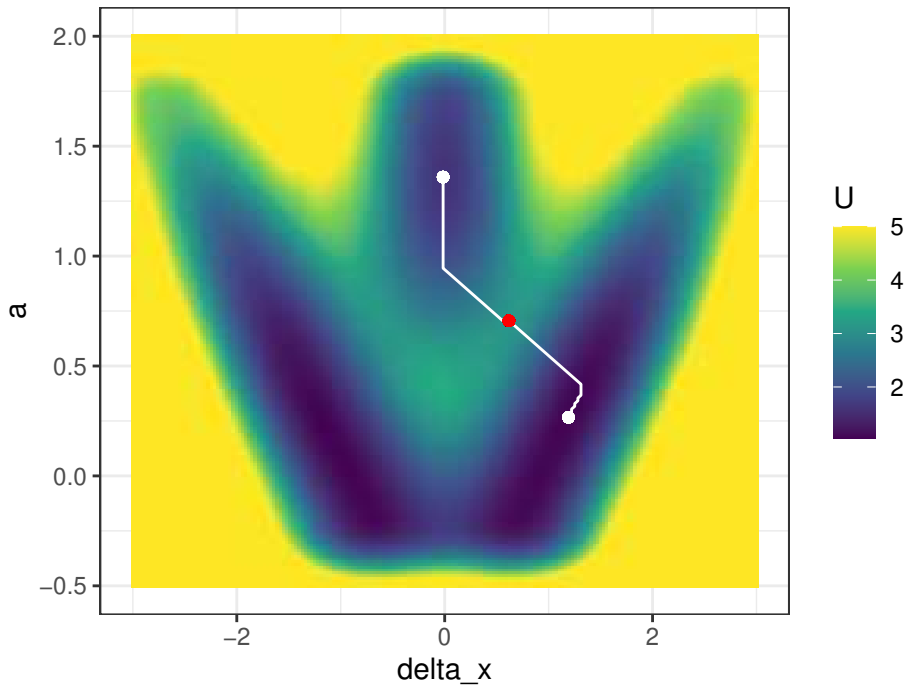


Figure 11: The example 3D static landscape for the nongradient model with local minima (white dots), saddle point (red dot), and the MEP (white line) marked. Because the system is nongradient, the MEP shown in this figure is different from the analytical result (shown in Figure 5, the lower panel).

```
R> plot(l_single_nongrad_3d, 2) + get_geom(b_single_nongrad_3d)
```

### 4.2. Multiple simulation barriers

`barriers` can also be calculated for landscapes from multiple simulations. In this case, it requires to set `individual_landscape = TRUE` in the landscape construction functions. The following are the codes for calculating the transition barriers for the landscapes in Section 3.2.

```
R> b_batch_2d <- calculate_barrier(l_batch_grad_2d,
R>   start_location_value = -1, end_location_value = 1,
R>   start_r = 1, end_r = 1
R> )
R> plot(l_batch_grad_2d) + get_geom(b_batch_2d)

R> b_batch_3d <- calculate_barrier(l_batch_grad_3d,
R>   start_location_value = c(-1, -1), end_location_value = c(1, 1),
R>   start_r = 1, end_r = 1
R> )
R> plot(l_batch_grad_3d) + get_geom(b_batch_3d)
```

## 5. Concluding comments

Potential landscapes can show the stability of states for a dynamical system in an intuitive and quantitative way. They are especially informative for multistable systems. In this article, we illustrated how to construct potential landscapes using **simlandr**. The potential landscapes generated by **simlandr** are based on the steady-state distribution of the system, which is in turn estimated using Monte Carlo simulation. Compared to analytic methods, Monte Carlo estimation is more flexible and thus more applicable for complex models. The flexibility comes together with a higher demand for time and storage, which is necessary to make the estimation precise enough. The `hash_big.matrix` class partly solved this problem by dumping the memory storage to hard disk space. We will also explore other possibilities (e.g., importance sampling, Kloek and van Dijk 1978) in future developments. Also, it is important that the simulation function itself is efficient enough. For improving the performance of the simulation function, we refer the interested readers to Wickham (2019).

All landscape construction and barrier calculation functions in **simlandr** contain both visual aids and numerical data that can be used for further processing. The `html` plots based on **plotly** are more suitable for interactive illustrations, while it is also possible to export them to static plots using `plotly::orca()`. The **ggplot2** plots are readily usable for flat printing. The barrier height data can also be further analyzed. For example, sometimes it is helpful to look into how the barrier height changes with varying parameters (Cui *et al.* 2021). We encourage users to explore other ways of analyzing and visualizing the various results provided by **simlandr**.

Finally, we want to note that the potential landscape generated by **simlandr** is not a complete description of all dynamics in a system. It emphasizes the stability of different states by

filtering out other dynamical information. Some behaviors are not possible in gradient systems (e.g., oscillations and loops), thus cannot be shown in a potential landscape. The method we chose for **simlandr** is not the only possible one, either. The generalized landscape by Wang *et al.* (2008), which we implemented, is more flexible and emphasizes the possibility that the system is in a specific state, while other methods may have other strengths (e.g., the method by Rodríguez-Sánchez 2020, emphasizes the gradient part of the vector field). We look forward to future theoretical and methodological developments in this direction.

## Acknowledgments

T.Li is supported by NSFC under grant No. 11825102 and Beijing Academy of Artificial Intelligence (BAAI).

## References

Ao P (2004). "Potential in Stochastic Differential Equations: Novel Construction." *Journal of Physics A: Mathematical and General*, **37**(3), L25–L30. ISSN 0305-4470. doi:10.1088/0305-4470/37/3/L01.

Cui J, Lichtwarck-Aschoff A, Olthof M, Li T, Hasselman F (2021). "From Metaphor to Computation: Constructing the Potential Landscape for Multivariate Psychological Formal Models." *Preprint*, PsyArXiv. doi:10.31234/osf.io/49xyg.

Dijkstra EW (1959). "A Note on Two Problems in Connexion with Graphs." *Numerische mathematik*, **1**(1), 269–271.

Duong T (2021). ***ks**: Kernel Smoothing*.

E W, Vanden-Eijnden E (2010). "Transition-Path Theory and Path-Finding Algorithms for the Study of Rare Events." *Annual Review of Physical Chemistry*, **61**(1), 391–420. ISSN 0066-426X, 1545-1593. doi:10.1146/annurev.physchem.040808.090412.

Eddelbuettel D, Lucas A, Tuszynski J, Bengtsson H, Urbanek S, Frasca M, Lewis B, Stokely M, Muehleisen H, Murdoch D, Hester J, Wu W, Kou Q, Onkelinx T, Lang M, Simko V, Hornik K, Neal R, Bell K, de Queljoe M, Suruceanu I, Denney B, Schumacher D, Chang W (2021). ***digest**: Create Compact Hash Digests of R Objects*.

Heymann M, Vanden-Eijnden E (2008). "Pathways of Maximum Likelihood for Rare Events in Nonequilibrium Systems: Application to Nucleation in the Presence of Shear." *Physical Review Letters*, **100**(14), 140601. doi:10.1103/PhysRevLett.100.140601.

Kane MJ, Emerson J, Weston S (2013). "Scalable Strategies for Computing with Massive Data." *Journal of Statistical Software*, **55**(14), 1–19. doi:10.18637/jss.v055.i14.

Kloek T, van Dijk HK (1978). "Bayesian Estimates of Equation System Parameters: An Application of Integration by Monte Carlo." *Econometrica*, **46**(1), 1–19. ISSN 0012-9682. doi:10.2307/1913641.

Lamothe KA, Somers KM, Jackson DA (2019). "Linking the Ball-and-Cup Analogy and Ordination Trajectories to Describe Ecosystem Stability, Resistance, and Resilience." *Ecosphere*, **10**(3), e02629. ISSN 2150-8925. doi:10.1002/ecs2.2629.

Lv C, Li X, Li F, Li T (2014). "Constructing the Energy Landscape for Genetic Switching System Driven by Intrinsic Noise." *PLOS ONE*, **9**(2), e88167. ISSN 1932-6203. doi:10.1371/journal.pone.0088167.

Olthof M, Hasselman F, Oude Maatman F, Bosman AMT, Lichtwarck-Aschoff A (2020). "Complexity Theory of Psychopathology." *Preprint*, PsyArXiv. doi:10.31234/osf.io/f68ej.

Rodríguez-Sánchez P (2020). "PabRod/**rolldown**: Post-Publication Update." Zenodo. doi:10.5281/zenodo.3763038.

Rodríguez-Sánchez P, van Nes EH, Scheffer M (2020). "Climbing Escher's Stairs: A Way to Approximate Stability Landscapes in Multidimensional Systems." *PLOS Computational Biology*, **16**(4), e1007788. ISSN 1553-7358. doi:10.1371/journal.pcbi.1007788.

Sievert C (2020). *Interactive Web-Based Data Visualization with R, **plotly**, and **shiny***. Chapman and Hall/CRC. ISBN 978-1-138-33145-7.

Venables WN, Ripley BD (2002). *Modern Applied Statistics with S*. Fourth edition. Springer, New York.

Waddington CH (1966). *Principles of Development and Differentiation*. Macmillan, New York.

Wang J, Xu L, Wang E (2008). "Potential Landscape and Flux Framework of Nonequilibrium Networks: Robustness, Dissipation, and Coherence of Biochemical Oscillations." *Proceedings of the National Academy of Sciences*, **105**(34), 12271–12276. ISSN 0027-8424, 1091-6490. doi:10.1073/pnas.0800579105.

Wang J, Zhang K, Xu L, Wang E (2011). "Quantifying the Waddington Landscape and Biological Paths for Development and Differentiation." *Proceedings of the National Academy of Sciences*, **108**(20), 8257–8262. ISSN 0027-8424, 1091-6490. doi:10.1073/pnas.1017017108.

Wickham H (2016). ***ggplot2**: Elegant Graphics for Data Analysis*. Springer-Verlag New York. ISBN 978-3-319-24277-4.

Wickham H (2019). "Improving Performance." In *Advanced R*, 2nd edition edition, pp. 531–546. Chapman and Hall/CRC, Boca Raton. ISBN 978-0-8153-8457-1.

Zhou JX, Aliyu MDS, Aurell E, Huang S (2012). "Quasi-Potential Landscape in Complex Multi-Stable Systems." *Journal of The Royal Society Interface*, **9**(77), 3539–3553. doi:10.1098/rsif.2012.0434.

Zhou P, Li T (2016). "Construction of the Landscape for Multi-Stable Systems: Potential Landscape, Quasi-Potential, A-Type Integral and Beyond." *The Journal of Chemical Physics*, **144**(9), 094109. ISSN 0021-9606. doi:10.1063/1.4943096.

**Affiliation:**

Jingmeng Cui
University of Groningen
Faculty of Behavioural and Social Sciences
Grote Rozenstraat 38, 9712TJ Groningen, the Netherlands
E-mail: jingmeng.cui@rug.nl

Merlijn Olthof
Radboud University
Behavioural Science Institute
Thomas van Aquinostraat 4, 6525GD Nijmegen, the Netherlands
E-mail: merlijn.olthof@ru.nl

Anna Lichtwarck-Aschoff
University of Groningen
Faculty of Behavioural and Social Sciences
Grote Rozenstraat 38, 9712TJ Groningen, the Netherlands
E-mail: a.lichtwarck-aschoff@rug.nl

Tiejun Li
*(corresponding author)*
Peking University
LMAM and School of Mathematical Sciences
Yiheyuan Road 5, Beijing 100871, China
E-mail: tieli@pku.edu.cn

Fred Hasselman
*(corresponding author)*
Radboud University
Behavioural Science Institute
Thomas van Aquinostraat 4, 6525GD Nijmegen, the Netherlands
E-mail: fred.hasselman@ru.nl