

**AnT 4.669**

Release 2a

Reference Manual  
Part II: User functions

University of Stuttgart  
Nonlinear Dynamics Group  
©2000 – 2003

May 9, 2003

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Implementation of system functions</b>	<b>5</b>
2.1	Basic system types . . . . .	6
2.1.1	Ordinary differential equations . . . . .	6
2.1.2	Delay differential equations . . . . .	8
2.1.3	Functional differential equations . . . . .	10
2.1.4	Maps . . . . .	11
2.1.5	Recurrent maps (with recurrence level greater than one) . .	13
2.2	Spatial inhomogeneous systems . . . . .	15
2.2.1	One-dimensional partial differential equations . . . . .	15
2.2.2	Coupled ordinary differential equation lattices . . . . .	19
2.2.3	Coupled map lattices . . . . .	20
2.3	Hybrid systems . . . . .	23
2.3.1	Hybrid maps . . . . .	24
2.3.2	Hybrid ordinary differential equations . . . . .	27
2.3.3	Hybrid delay differential equations . . . . .	32
2.4	Other types of dynamical systems . . . . .	34
2.4.1	Stochastical systems . . . . .	34
2.4.2	Poincaré maps . . . . .	35
2.4.3	External data input . . . . .	35
<b>3</b>	<b>Implementation of the linearized system functions</b>	<b>36</b>
3.1	Calculation of Lyapunov exponents . . . . .	36
3.2	Basic system types . . . . .	37
3.2.1	Ordinary differential equations . . . . .	37
3.2.2	Maps . . . . .	40
3.3	Spatial inhomogeneous systems . . . . .	42
3.3.1	Coupled ordinary differential equation lattices . . . . .	42
3.3.2	Coupled map lattices . . . . .	43

3.4	Hybrid systems . . . . .	44
3.4.1	Hybrid maps . . . . .	44
3.4.2	Hybrid ordinary differential equations . . . . .	45
<b>4</b>	<b>User defined symbolic partitions of the state space</b>	<b>46</b>
4.1	Symbolic dynamics . . . . .	46
4.2	Basic system types . . . . .	47
4.2.1	Ordinary differential equations . . . . .	47
4.2.2	Delay differential equations . . . . .	48
4.2.3	Functional differential equations . . . . .	49
4.2.4	Maps . . . . .	50
4.2.5	Recurrent maps (with recurrence level greater than one) . .	51
4.3	Spatial inhomogeneous dynamical systems . . . . .	52
4.3.1	One-dimensional partial differential equations . . . . .	52
4.3.2	Coupled ordinary differential equation lattices . . . . .	53
4.3.3	Coupled map lattices . . . . .	54
4.4	Hybrid systems . . . . .	55
4.4.1	Hybrid maps . . . . .	55
4.4.2	Hybrid ordinary differential equations . . . . .	56
4.4.3	Hybrid delay differential equations . . . . .	57
<b>5</b>	<b>User defined conditions for Poincaré maps</b>	<b>58</b>

# Chapter 1

## Introduction

For the application of the **AnT** package for simulation and investigation of a specific dynamical system one has to implement (at least) the system function of this system and to make this function callable from the **AnT** kernel. There are different possibilities known, how this task can be realized. The first one is to develop some description language for dynamical systems. In this case it is possible to support users with an editor (environment, framework, etc.), which make the input comfortable and which is directly connected to the kernel. This solution is often suitable for many reasons, but it has also some disadvantages. If the description language is planned to be simple, it is difficult to make it powerful. Hence, the input language can be a neck of the bottle, which restricts the power of the simulation software. A language, which must be able to describe an arbitrary system function, must have the complexity of any common programming language. However, it doesn't seem to be necessary to develop a new programming language for the implementation of system functions, because the existing languages can be used for this reason as well. Due to this fact, the implementation of system functions as well as of the other user-defined functions for the **AnT** package is based on the following concepts:

A user has to implement its functions as a standard **C++** routine. For all system types, which are supported by the **AnT** package, the corresponding interfaces are implemented and described in this document.

- Note that the choice of the interface is important! For instance, if one has to implement the system function of a partial differential equation,

---

it won't work with the interface for ordinary differential equations. However, such mistakes are easy to detect and are carefully handled by AnT.

- Note also, that the subset of the C++ language, which is really needed for the implementation of standard equations, contains statements, common for the most procedural programming languages. For instance, the term  $\alpha x(1 - x)$  must be written as `alpha * x * (1 - x)`. This C++ statement is identical to the corresponding statements in FORTRAN, C and Pascal.

The next step, which a user has to perform, is the connection of the system function to the simulator. For this purpose the routine `connectSystem()` is developed, and the only action, which has to be done by user, is an assignment in this routine. The left hand side of this assignment is a connection variable for the corresponding system type and the right hand side is the name of the user function. For instance, if the system function `lorenz63` is implemented, then the following line is to be written:

```
ODE_Proxy::systemFunction = lorenz63;
```

Note, that the connection variables are different for different system classes. The existing connection variables are described in this document.

The last step, which a user has to perform, is to compile the file, containing the system function and the `connectSystem()` routine with an C++ compiler, like `g++`. A standard `Makefile` for this purpose is a part of the AnT distribution. This `Makefile` create a shared object (a library) and then AnT can be started with this library.

## Chapter 2

# Implementation of system functions

In this chapter the interfaces of system function for several types of dynamical systems are described.

## 2.1 Basic system types

### 2.1.1 Ordinary differential equations

$$\frac{d}{dt}\underline{x}(t) = \underline{f}(\underline{x}(t), \underline{\sigma}) \quad (2.1)$$

The interface of the system function for an ordinary differential equation is defined by:

```
bool SystemFunction (  const Array<real_t> & currentState,
                      const Array<real_t> & parameters,
                      Array<real_t> & rhs);
```

with

<code>currentState</code>	the state $\underline{x}(t)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>rhs</code>	the right hand side $\frac{d}{dt}\underline{x}(t)$	(output)

The system function returns `true`, if the calculation of the right hand side is successfully.

The system function will be connected to the simulator using the variable `ODE_Proxy::systemFunction`.

## Chapter 2

### Implementation of system functions

---

#### Example: Lorenz-63 System

$$\begin{aligned}\dot{x} &= \sigma(y - x) \\ \dot{y} &= x(r - z) - y \\ \dot{z} &= xy - bz\end{aligned}\tag{2.2}$$

#### Implementation

```
#include "AnT.hpp"

#define sigma parameters[0]
#define r      parameters[1]
#define b      parameters[2]
#define X      currentState[0]
#define Y      currentState[1]
#define Z      currentState[2]

bool lorenz63 (const Array<real_t>& currentState,
               const Array<real_t>& parameters,
               Array<real_t>& rhs)
{
    rhs[0] = sigma * (Y - X);
    rhs[1] = X * (r - Z) - Y;
    rhs[2] = X * Y - b * Z;

    return true;
}

#undef sigma
#undef r
#undef b
#undef X
#undef Y
#undef Z

extern "C" {
    void connectSystem ()
    {
        ODE_Proxy::systemFunction = lorenz63;
    }
}
```

### 2.1.2 Delay differential equations

$$\frac{d}{dt}\underline{x}(t) = \underline{f}(\underline{x}(t), \underline{x}(t - \tau), \underline{\sigma}) \quad (2.3)$$

The interface of the system function for a delay differential equation is defined by:

```
bool SystemFunction (  const Array<real_t> & currentState,
                      const Array<real_t> & delayState,
                      const Array<real_t> & parameters,
                      Array<real_t> & rhs);
```

with

<b>currentState</b>	the state $\underline{x}(t)$	(input)
<b>delayState</b>	the state $\underline{x}(t - \tau)$	(input)
<b>parameters</b>	the set of parameters $\underline{\sigma}$	(input)
<b>rhs</b>	the right hand side $\frac{d}{dt}\underline{x}(t)$	(output)

The system function returns **true**, if the calculation of the right hand side is successfully.

The system function will be connected to the simulator using the variable `DDE_Proxy::systemFunction`.

## Chapter 2

### Implementation of system functions

---

#### Example: Mackey–Glass system

$$\dot{x}(t) = a \frac{x(t - \tau)}{1 + x^{10}(t - \tau)} - bx(t) \quad (2.4)$$

#### Implementation

```
#include "AnT.hpp"

#define a    parameters[0]
#define b    parameters[1]
#define X    currentState[0]
#define Xt   delayState[0]

bool mg (const Array<real_t>& currentState,
         const Array<real_t>& delayState,
         const Array<real_t>& parameters,
         Array<real_t>& rhs)
{
    rhs[0] = a * Xt/(1+pow(Xt,10.0)) - b * X;
    return true;
}

#undef a
#undef b
#undef X
#undef Xt

extern "C" {
    void connectSystem ()
    {
        DDE_Proxy::systemFunction = mg;
    }
}
```

### 2.1.3 Functional differential equations

$$\frac{d}{dt}\underline{x}(t) = \underline{f}[\underline{x}_t, \underline{\sigma}] \quad (2.5)$$

with

$$\underline{x}_t(\theta) = \underline{x}(t + \theta), \quad \theta \in [-\tau, 0] \quad (2.6)$$

and  $\tau$  - the maximal delay

The interface of the system function for a functional differential equation is defined by:

```
bool SystemFunction (
    const Subarray<CyclicArray<Array<real_t>>> &
                                                currentState,
    const Array<real_t> & parameters,
    const real_t deltaT,
    Array<real_t> & rhs);
```

with

<code>currentState</code>	the states $\underline{x}(t - \tau) \dots \underline{x}(t)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>deltaT</code>	the maximal delay $\tau$	(input)
<code>rhs</code>	the right hand side $\frac{d}{dt}\underline{x}(t)$	(output)

The system function returns `true`, if the calculation of the right hand side is successfully.

The system function will be connected to the simulator using the variable `FDE.Proxy::systemFunction`.

### 2.1.4 Maps

$$\underline{x}(n+1) = \underline{f}(\underline{x}(n), \underline{\sigma}) \quad (2.7)$$

The interface of the system function for a map is defined by:

```
bool SystemFunction (  const Array<real_t> & currentState,
                      const Array<real_t> & parameters,
                      Array<real_t> & rhs);
```

with

<b>currentState</b>	the state $\underline{x}(n)$	(input)
<b>parameters</b>	the set of parameters $\underline{\sigma}$	(input)
<b>rhs</b>	the right hand side $\underline{x}(n+1)$	(output)

The system function returns **true**, if the calculation of the right hand side is successfully.

The system function will be connected to the simulator using the variable `MapProxy::systemFunction`.

**Example: Hénon map**

$$\begin{aligned}x(n+1) &= 1 - \alpha x^2(n) + y(n) \\y(n+1) &= \beta x(n)\end{aligned}\tag{2.8}$$

**Implementation**

```
#include "AnT.hpp"

#define alpha parameters[0]
#define beta  parameters[1]
#define X    currentState[0]
#define Y    currentState[1]

bool henon (const Array<real_t>&currentState,
            const Array<real_t>& parameters,
            Array<real_t>& rhs)
{
    rhs[0] = 1 - alpha * X * X + Y;
    rhs[1] = beta * X;

    return true;
}

#undef alpha
#undef beta
#undef X
#undef Y

extern "C"
{
    void connectSystem ()
    {
        MapProxy::systemFunction = henon;
    }
}
```

### 2.1.5 Recurrent map (with recurrence level greater than one)

$$\underline{x}(n+1) = \underline{f}(\underline{x}(n-N) \dots \underline{x}(n), \underline{\sigma}) \quad (2.9)$$

with  $N > 0$  – the recurrence level of the map. It is known, that each map with a  $n$ -dimensional state vector and recurrence level  $N$  can be transformed in a map with a  $nN$ -dimensional state vector and recurrence level one. However, AnT supports the both types of maps.

The interface of the system function for a recurrent map is defined by:

```
bool SystemFunction (  
    const CyclicArray<Array<real_t>> & currentState,  
    const Array<real_t> & parameters,  
    Array<real_t> & rhs);
```

with

```
currentState  the states  $\underline{x}(n-N) \dots \underline{x}(n)$  (input)  
parameters    the set of parameters  $\underline{\sigma}$  (input)  
rhs           the right hand side  $\underline{x}(n+1)$  (output)
```

The system function returns `true`, if the calculation of the right hand side is successfully.

The system function will be connected to the simulator using the variable `RecurrentMapProxy::systemFunction`.

**Example: Recurrent implementation of the Hénon map** The Hénon map (Eq 2.8) is a two-dimensional map with recurrence level one. It can be written also as an one-dimensional map with recurrence level two:

$$x(n + 1) = 1 - \alpha x^2(n) + \beta x(n - 1) \quad (2.10)$$

### Implementation

```
#include "AnT.hpp"

#define alpha parameters[0]
#define beta  parameters[1]
#define X    currentState[0][0]
#define Y    currentState[-1][0]

bool henon (const CyclicArray<Array<real_t> >& currentState,
            const Array<real_t>& parameters,
            Array<real_t>& rhs)
{
    rhs[0] = 1 - alpha * X * X + beta * Y;

    return true;
}

#undef alpha
#undef beta
#undef X
#undef Y

extern "C"
{
    void connectSystem ()
    {
        RecurrentMapProxy::systemFunction = henon;
    }
}
```

## 2.2 Spatial inhomogeneous systems

### 2.2.1 One-dimensional partial differential equations

$$\frac{\partial}{\partial t} \underline{x}(q, t) = \underline{f} \left( \underline{x}(q, t), \frac{\partial}{\partial t} \underline{x}(q, t), \dots, \underline{\sigma} \right) \quad (2.11)$$

with  $q \in [q_{\min}, q_{\max}]$ .

The domain  $[q_{\min}, q_{\max}]$  will be represented by  $N$  points

$$q_j = q_{\min} + j\Delta q, \quad \Delta q = \frac{q_{\max} - q_{\min}}{N - 1}, \quad j = 0 \dots N - 1 \quad (2.12)$$

Hence, the function  $\underline{x}(q, t)$  will be approximated by  $N$  vectors  $\underline{x}(q_j, t)$  with  $j = 0 \dots N - 1$ . Hereby the vector  $\underline{x}(q_0, t)$  corresponds to the left boundary  $\underline{x}(q_{\min}, t)$  and the vector  $\underline{x}(q_{N-1}, t)$  to the right one  $\underline{x}(q_{\max}, t)$

The interface of the system function for an one-dimensional partial differential equation is defined by:

```
bool SystemFunction (  const CellularState & currentState,
                      const Array<real_t>& parameters,
                      int cellIndex,
                      real_t deltaX,
                      StateCell & rhs);
```

with

<b>currentState</b>	the states $\underline{x}(q_0, t) \dots \underline{x}(q_{N-1}, t)$	(input)
<b>parameters</b>	the set of parameters $\underline{\sigma}$	(input)
<b>cellIndex</b>	the spatial index $j$	(input)
<b>deltaX</b>	the spatial grid step $\Delta q$	(input)
<b>rhs</b>	the right hand side $\frac{d}{dt} \underline{x}(t, j)$	(output)

The system function returns **true**, if the calculation of the right hand side is successfully.

---

## 2.2. SPATIAL INHOMOGENEOUS SYSTEMS

---

The system function will be connected to the simulator using the variable `PDE_1d_Proxy::systemFunction`.

For the implementation of the system function of a partial differential equation several differential operators are provided. Table 2.1 shows the implemented differential operators and how they can be called by user in the system function:

operator	call
$\frac{\partial}{\partial q}$	D< 0 >
$\frac{\partial}{\partial q^2}$	D< 0, 0 >
$\frac{\partial}{\partial q^3}$	D< 0, 0, 0 >
$\frac{\partial}{\partial q^4}$	D< 0, 0, 0, 0 >
$\frac{\partial}{\partial q^5}$	D< 0, 0, 0, 0, 0 >
$\frac{\partial}{\partial q^n}$	D< 0 > (n)

**Table 2.1:** Differential operators, supported by AnT

The implemented general operator D< 0 > (n) can be called for an arbitrary value of  $n > 0$ . Additionally, for the cases  $n = 1..5$  the operators D< 0 > dots D< 0, 0, 0, 0, 0 > can be called instead. These operators are preimplemented using some optimizations techniques and work faster.

## Chapter 2

### Implementation of system functions

---

The differential operators presented above have to be called using the following interface:

```
real_t operator (  const CellularState & currentState,
                  int  cellIndex,
                  int  variableIndex,
                  real_t deltaX);
```

The input parameters `currentState` `cellIndex` and `deltaX` are described above and the input parameter `variableIndex` is the index of the state variable, for which the operator must be applied. For instance, if the state vector of a system is  $(x, y, z)^T$ , then one has to use `variableIndex = 0` for the term  $\frac{\partial}{\partial q}x(q, t)$  and `variableIndex = 2` for the term  $\frac{\partial}{\partial q}z(q, t)$ .

The return value of the differential operators is the value of the corresponding partial derivative calculated at the spatial point  $q_j$ .

### Example: Korteweg – de Vries Equation

$$\frac{\partial}{\partial t}u(q,t) = au(q,t)\frac{\partial}{\partial q}u(q,t) - \frac{\partial^3}{\partial q^3}u(q,t) \quad (2.13)$$

### Implementation

```
#include "AnT.hpp"

#define a parameters[0]
#define u uState[cellIndex][0]

bool KdV ( const CellularState& uState,
           const Array<real_t>& parameters,
           int cellIndex,
           real_t deltaX,
           StateCell& rhs)
{
  rhs[0] = a * u * D<0>(uState,cellIndex,0,deltaX) - D<0>(3)(uState,cellIndex,0,deltaX);
  return true;
}

#undef a
#undef u

extern "C" {
  void connectSystem ()
  {
    PDE_1d_Proxy::systemFunction = KdV;
  }
}
```

## 2.2.2 Coupled ordinary differential equation lattices

A coupled ordinary differential equation lattice consists on  $N$  cells. Each of them contains an ordinary differential equation, which can be coupled with the differential equations of other cells. For the  $i$ -th cell the equation of motion is defined by

$$\frac{d}{dt}\underline{x}_i(t) = \underline{f}\left(\{\underline{x}_j(t)\}_{j \in 1..N-1}, \underline{\sigma}\right) \quad (2.14)$$

If the cells are coupled only with cells from the next neighborhood, the coupling of the lattice is called local, otherwise it is called global. **AnT** supported the both types of the coupling.

The interface of the system function for a coupled ordinary differential equation lattice is defined by:

```
bool SystemFunction (  const CellularState & currentState,
                      const Array<real_t> & parameters,
                      int cellIndex,
                      StateCell & rhs);
```

with

<b>currentState</b>	the states $\underline{x}_0(t) \dots \underline{x}_{N-1}(t)$	(input)
<b>parameters</b>	the set of parameters $\underline{\sigma}$	(input)
<b>cellIndex</b>	spatial index $i$	(input)
<b>rhs</b>	the right hand side $\frac{d}{dt}\underline{x}_i(t)$	(output)

The system function returns **true**, if the calculation of the right hand side is successfully.

The system function will be connected to the simulator using the variable `CODEL_Proxy::systemFunction`.

### 2.2.3 Coupled map lattices

A coupled map lattice consists on  $N$  cells. Each of them contains a map, which can be coupled with the maps of other cells. For the  $i$ -th cell the equation of motion is defined by

$$\underline{x}_i(n+1) = \underline{f} \left( \{ \underline{x}_j(n) \}_{j \in 1..N-1}, \underline{\sigma} \right) \quad (2.15)$$

If the cells are coupled only with cells from the next neighborhood, the coupling of the lattice is called local, otherwise it is called global. AnT supported the both types of the coupling.

The interface of the system function for a coupled map lattice is defined by:

```
bool SystemFunction (  const CellularState & currentState,
                      const Array<real_t> & parameters,
                      int cellIndex,
                      StateCell & rhs);
```

with

<code>currentState</code>	the states $\underline{x}_0(n) \dots \underline{x}_{N-1}(n)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>cellIndex</code>	spatial index $j$	(input)
<code>rhs</code>	the right hand side $\underline{x}_i(n+1)$	(output)

The system function returns `true`, if the calculation of the right hand side is successfully.

The system function will be connected to the simulator using the variable `CML_Proxy::systemFunction`.

**Example: coupled logistic maps**

The system consists of  $N$  coupled logistic maps. The equation of motion for the  $i$ -th map ( $i = 0..N - 1$ ) is defined by

$$x_i(n+1) = \begin{cases} (1 - \beta_r)\alpha x_i(n)(1 - x_i(n)) + \\ \quad \beta_r x_{i+1}(n) & \text{if } i = 0 \\ (1 - \beta_r - \beta_l)\alpha x_i(n)(1 - x_i(n)) + \beta_r x_{i+1}(n) + \\ \quad \beta_l x_{i-1}(n) & \text{if } 0 < i < N - 1 \\ (1 - \beta_l)\alpha x_i(n)(1 - x_i(n)) + \\ \quad \beta_l x_{i-1}(n) & \text{if } i = N - 1 \end{cases} \quad (2.16)$$

### Implementation

```
#include "AnT.hpp"

#define alpha parameters[0]
#define betaL parameters[1]
#define betaR parameters[2]
#define XL    currentState[cellIndex-1][0]
#define X     currentState[cellIndex][0]
#define XR    currentState[cellIndex+1][0]

bool CML_logistic (const CellularState& currentState,
                  const Array<real_t>& parameters,
                  int cellIndex,
                  StateCell& rhs)
{
    int numberOfCells = currentState.numberOfCells;

    if (cellIndex == 0)
        rhs[0] = (1-betaR) * alpha * X * (1 - X) + betaR * XR;
    else if (cellIndex == (numberOfCells-1))
        rhs[0] = (1-betaL) * alpha * X * (1 - X) + betaL * XL;
    else
        rhs[0] = (1-betaR-betaL) * alpha * X * (1 - X) + betaL * XL + betaR * XR;
    return true;
}

#undef alpha
#undef betaL
#undef betaR
#undef XL
#undef X
#undef XR

extern "C"
{
    void connectSystem ()
    {
        CML_Proxy::systemFunction = CML_logistic;
    }
}
```

## 2.3 Hybrid systems

Hybrid systems are dynamical systems, whose behavior exhibits continuous and discrete changes. AnT package support the simulation and investigation of hybrid systems using so-called hybrid equations. They are equivalent to other frameworks like hybrid automata and are based on the following idea. The state vector of a hybrid equations consists of two parts, a vector  $\underline{x}$  with real-valued components (called continuous state vector) and a vector  $\underline{m}$  with integer-valued components (called discrete state vector). The behavior of the continuous state vector will be described by a system function, which specify either its derivative (in the case of hybrid systems continuous in time) or its next value (in the case of hybrid systems discrete in time). The behavior of the discrete state vector will be described by the so-called hybrid function. This function specify the next value of the discrete state vector. In the case of hybrid systems discrete in time it is the value  $\underline{m}(n + 1)$ . In the case of hybrid systems continuous in time it is  $\underline{m}(t^+)$ , denoting the vector  $\underline{m}$  direct after the time  $t$ . By the numerical simulation this time is realized as  $t + \Delta t$  with  $\Delta t$  integration step size.

### 2.3.1 Hybrid maps

$$\underline{m}(n+1) = \underline{h}(\underline{x}(n), \underline{m}(n), \underline{\sigma}) \quad (2.17)$$

$$\underline{x}(n+1) = \underline{f}(\underline{x}(n), \underline{m}(n+1), \underline{\sigma}) \quad (2.18)$$

The interface of the system function for a hybrid map is defined by:

```

bool SystemFunction (
    const Array<real_t> & currentContinuousState,
    const Array<int_t> & currentDiscreteState,
    const Array<real_t> & parameters,
    Array<real_t> & rhs);
with
currentContinuousState  the continuous state  $\underline{x}(n)$       (input)
currentDiscreteState    the discrete state  $\underline{m}(t)$         (input)
parameters               the set of parameters  $\underline{\sigma}$     (input)
rhs                     the right hand side  $\underline{x}(n+1)$   (output)

```

The system function returns **true**, if the calculation of the right hand side is successfully.

The system function will be connected to the simulator using the variable `HybridMapProxy::systemFunction`.

The interface of the hybrid function for a hybrid map is defined by:

```

bool HybridFunction (
    const Array<real_t> & currentContinuousState,
    const Array<int_t> & currentDiscreteState,
    const Array<real_t> & parameters,
    Array<int> & hybridRHS);
with
currentContinuousState  the continuous state  $\underline{x}(n)$       (input)
currentDiscreteState    the discrete state  $\underline{m}(t)$         (input)
parameters              the set of parameters  $\underline{\sigma}$       (input)
hybridRHS               the right hand side  $\underline{m}(n + 1)$  (output)

```

The hybrid function returns `true`, if the calculation of the right hand side is successfully.

The system function will be connected to the simulator using the variable `HybridMapProxy::hybridFunction`.

### Example: tent map

$$x(n+1) = \begin{cases} ax(n) & \text{if } m(n+1) = M_L \\ a(1-x(n)) & \text{if } m(n+1) = M_R \end{cases} \quad (2.19)$$

$$m(n+1) = \begin{cases} m^L & \text{if } x(n) < \text{frac12} \\ m^R & \text{if } x(n) \geq \text{frac12} \end{cases} \quad (2.20)$$

## Implementation

```
#include "AnT.hpp"

#define a      parameters[0]
#define X      currentContinuousState[0]
#define mode   currentDiscreteState[0]
#define LEFT   1
#define RIGHT  2

bool tent_map_C (const Array<real_t>& currentContinuousState,
                 const Array<int>& currentDiscreteState,
                 const Array<real_t>& parameters,
                 Array<real_t>& RHS)
{
    if (mode == LEFT)
        RHS[0] = a * X;
    else
        RHS[0] = a * (1 - X);

    return true;
}

bool tent_map_D (const Array<real_t>& currentContinuousState,
                 const Array<int>& currentDiscreteState,
                 const Array<real_t>& parameters,
                 Array<int>& RHS)
{
    if (X < 0.5)
        RHS[0] = LEFT;
    else
        RHS[0] = RIGHT;

    return true;
}

#undef a
#undef X
#undef mode
#undef LEFT
#undef RIGHT

extern "C"
{
    void connectSystem ()
    {
        HybridMapProxy::systemFunction = tent_map_C;
        HybridMapProxy::hybridFunction = tent_map_D;
    }
}
```

### 2.3.2 Hybrid ordinary differential equations

$$\underline{m}(t^+) = \underline{h}(\underline{x}(t), \underline{m}(t), \underline{\sigma}) \quad (2.21)$$

$$\frac{d}{dt}\underline{x}(t) = \underline{f}(\underline{x}(t), \underline{m}(t^+), \underline{\sigma}) \quad (2.22)$$

The interface of the system function for a hybrid ordinary differential equation is defined by:

```
bool SystemFunction (
    const Array<real_t> & currentContinuousState,
    const Array<int_t> & currentDiscreteState,
    const Array<real_t> & parameters,
    Array<real_t> & rhs);
```

with

<code>currentContinuousState</code>	the continuous state $\underline{x}(t)$	(input)
<code>currentDiscreteState</code>	the discrete state $\underline{m}(t)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>rhs</code>	the right hand side $\frac{d}{dt}\underline{x}(t)$	(output)

The system function returns `true`, if the calculation of the right hand side is successfully.

The system function will be connected to the simulator using the variable `HybridODE_Proxy::systemFunction`.

The interface of the hybrid function for a hybrid ordinary differential equation is defined by:

```
bool HybridFunction (  
    const Array<real_t> & currentContinuousState,  
    const Array<int_t> & currentDiscreteState,  
    const Array<real_t> & parameters,  
    Array<int> & hybridRHS);  
with  
  
currentContinuousState  the continuous state  $\underline{x}(t)$       (input)  
currentDiscreteState    the discrete state  $\underline{m}(t)$       (input)  
parameters              the set of parameters  $\underline{\sigma}$       (input)  
hybridRHS               the right hand side  $\underline{m}(t + \Delta t)$  (output)
```

The hybrid function returns `true`, if the calculation of the right hand side is successfully.

The hybrid function will be connected to the simulator using the variable `HybridODE_Proxy::hybridFunction`.

**Example: Chua circuit**

$$\begin{aligned}C_1 \dot{U}_{C_1} &= G(U_{C_2} - U_{C_1}) - g(U_{C_1}) \\C_2 \dot{U}_{C_2} &= G(U_{C_1} - U_{C_2}) + I_L \\L \dot{I}_L &= -U_{C_2}\end{aligned}\tag{2.23}$$

$$g(U_R) = \begin{cases} m_0 U_R + B_p(m_1 - m_0) & \text{falls } U_R > B_p \\ m_1 U_R & \text{falls } |U_R| \leq B_p \\ m_0 U_R - B_p(m_1 - m_0) & \text{falls } U_R < -B_p \end{cases}\tag{2.24}$$

## Implementation

```
#include "AnT.hpp"

#define C1      parameters[0]
#define C2      parameters[1]
#define G       parameters[2]
#define L       parameters[3]
#define m0      parameters[4]
#define m1      parameters[5]
#define Bp      parameters[6]
#define mode    currentDiscreteState[0]
#define X       currentContinuousState[0]
#define Y       currentContinuousState[1]
#define Z       currentContinuousState[2]
#define Dminus  -1
#define Dnull   0
#define Dplus   1

bool chua_C (const Array<real_t>& currentContinuousState,
             const Array<int>& currentDiscreteState,
             const Array<real_t>& parameters,
             Array<real_t>& RHS)
{
    if (mode == Dminus)
        RHS[0] = (G * (Y - X) - (m0 * X + Bp * (m0 - m1)) ) * C1;
    else if (mode == Dnull)
        RHS[0] =(G * (Y - X) - m1 * X) * C1;
    else
        RHS[0] = (G * (Y - X) - (m0 * X + Bp * (m1 - m0)) ) * C1;

    RHS[1] = (G * (X - Y) + Z)*C2;
    RHS[2] = -1/L * Y;
    return true;
}
```

... continued on the next page...

## Chapter 2

### Implementation of system functions

---

```
bool chua_D (const Array<real_t>& currentContinuousState,
             const Array<int>&    currentDiscreteState,
             const Array<real_t>& parameters,
             Array<int>& RHS)
{
    if (X < -Bp)
        RHS[0] = Dminus;
    else if (X < Bp)
        RHS[0] = Dnull;
    else
        RHS[0] = Dplus;
    return true;
}

#undef C1
#undef C2
#undef G
#undef L
#undef m0
#undef m1
#undef Bp
#undef mode
#undef X
#undef Y
#undef Z
#undef Dminus
#undef Dnull
#undef Dplus

extern "C" {
    void connectSystem ()
    {
        HybridODE_Proxy::systemFunction = chua_C;
        HybridODE_Proxy::hybridFunction = chua_D;
    }
}
```

### 2.3.3 Hybrid delay differential equations

$$\underline{m}(t^+) = \underline{h}(\underline{x}(t), \underline{m}(t), \underline{\sigma}) \quad (2.25)$$

$$\frac{d}{dt}\underline{x}(t) = \underline{f}(\underline{x}(t), \underline{x}(t - \tau), \underline{m}(t^+), \underline{\sigma}) \quad (2.26)$$

The interface of the system function for a hybrid delay differential equation is defined by:

```

bool SystemFunction (
    const Array<real_t> & currentContinuousState,
    const Array<int_t> & currentDiscreteState,
    const Array<real_t> & delayState,
    const Array<real_t> & parameters,
    Array<real_t> & rhs);
with
currentContinuousState  the continuous state  $\underline{x}(t)$    (input)
currentDiscreteState    the discrete state  $\underline{m}(t)$      (input)
parameters              the set of parameters  $\underline{\sigma}$    (input)
delayState              the delay state  $\underline{x}(t - \tau)$  (input)
rhs                    the right hand side  $\frac{d}{dt}\underline{x}(t)$  (output)

```

The system function returns `true`, if the calculation of the right hand side is successfully.

The system function will be connected to the simulator using the variable `HybridDDE_Proxy::systemFunction`.

## Chapter 2

### Implementation of system functions

---

The interface of the system function for a hybrid delay differential equation is defined by:

```
bool HybridFunction (  
    const Array<real_t> & currentContinuousState,  
    const Array<int_t> & currentDiscreteState,  
    const Array<real_t> & delayState,  
    const Array<real_t> & parameters,  
    Array<int> & hybridRHS);
```

with

<code>currentContinuousState</code>	the continuous state $\underline{x}(t)$	(input)
<code>currentDiscreteState</code>	the discrete state $\underline{m}(t)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>delayState</code>	the delay state $\underline{x}(t - \tau)$	(input)
<code>hybridRHS</code>	the right hand side $\underline{m}(t + \Delta t)$	(output)

The hybrid function returns `true`, if the calculation of the right hand side is successfully.

The hybrid function will be connected to the simulator using the variable `HybridDDE_Proxy::hybridFunction`.

## 2.4 Other types of dynamical systems

### 2.4.1 Stochastical systems

There are following types of dynamical systems with additive noise implemented:

#### Stochastical maps with additive noise

$$\underline{x}(n+1) = \underline{f}(\underline{x}(n), \underline{\sigma}) + \underline{\eta} \quad (2.27)$$

#### Stochastical ordinary differential equations with additive noise

$$\frac{d}{dt}\underline{x}(t) = \underline{f}(\underline{x}(t), \underline{\sigma}) + \underline{\eta} \quad (2.28)$$

#### Stochastical delay differential equations with additive noise

$$\frac{d}{dt}\underline{x}(t) = \underline{f}(\underline{x}(t), \underline{x}(t-\tau), \underline{\sigma}) + \underline{\eta} \quad (2.29)$$

The systems doesn't need special interfaces, because they can use the interfaces for the corresponding non-stochastical systems: Especially, the system function of a stochastical map with additive noise has to be implemented according to the interface for the system function of maps, see Sec. 2.1.4. The system function of a stochastical ordinary differential equation with additive noise has to be implemented according to the interface for ordinary differential equations, see Sec. 2.1.1. The system function of a stochastical delay differential equation with additive noise has to be implemented according to the interface for the system function of delay differential equations, see Sec. 2.1.2.

## 2.4.2 Poincaré maps

Poincaré maps doesn't have an own interface. Instead, the user have to implement in this case the system function for the dynamical systems, which will be simulated within the Poincaré maps. For this purpose the corresponding interface has to be used.

## 2.4.3 External data input

This in the only system type, for which the system function doesn't need be implemented.

## Chapter 3

# Implementation of the linearized system functions

### 3.1 Calculation of Lyapunov exponents

## 3.2 Basic system types

### 3.2.1 Ordinary differential equations

The interface of the linearized system function for an ordinary differential equation is defined by:

```

bool SystemFunction (  const Array<real_t> & currentState,
                      const Array<real_t> & referenceState,
                      const Array<real_t> & parameters,
                      Array<real_t> & RHS)

```

with

<code>currentState</code>	the state $\underline{x}(t)$	(input)
<code>referenceState</code>	the reference state $\underline{x}_{ref}$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>rhs</code>	the right hand side $\frac{d}{dt}\underline{x}(t)$	(output)

The linearized system function will be connected to the simulator using the variable `ODE_LinearizedProxy::systemFunction`.

**Example:**

Linearized system function of the Lorenz-63 system (Eq. 2.2).

## Chapter 3

### Implementation of the linearized system functions

---

#### Implementation

```
#include "AnT.hpp"

#define sigma parameters[0]
#define r      parameters[1]
#define b      parameters[2]
#define X      currentState[0]
#define Y      currentState[1]
#define Z      currentState[2]

bool lorenz63 (const Array<real_t>& currentState,
              const Array<real_t>& parameters,
              Array<real_t>& rhs)
{
    rhs[0] = sigma * (Y - X);
    rhs[1] = X * (r - Z) - Y;
    rhs[2] = X * Y - b * Z;
    return true;
}

#define Xr      referenceState[0]
#define Yr      referenceState[1]
#define Zr      referenceState[2]

bool lorenz63_linearized (const Array<real_t>& currentState,
                        const Array<real_t>& referenceState,
                        const Array<real_t>& parameters,
                        Array<real_t>& rhs)
{
    rhs[0] = sigma * (Y - X);
    rhs[1] = X * (r - Zr) - Y - Xr * Z;
    rhs[2] = Yr * X + Xr * Y - b * Z;
    return true;
}

#undef sigma
#undef r
#undef b
#undef X
#undef Y
#undef Z
#undef Xr
#undef Yr
#undef Zr

extern "C" {
    void connectSystem ()
    {
        ODE_Proxy::systemFunction = lorenz63;
        ODE_LinearizedProxy::systemFunction = lorenz63_linearized;
    }
}
```

### 3.2.2 Maps

The interface of the linearized system function for a map is defined by:

```
bool SystemFunction (  const Array<real_t> & currentState,
                      const Array<real_t> & referenceState,
                      const Array<real_t> & parameters,
                      Array<real_t> & rhs);
```

with

<b>currentState</b>	the state $\underline{x}(n)$	(input)
<b>referenceState</b>	the reference state $\underline{x}_{ref}$	(input)
<b>parameters</b>	the set of parameters $\underline{\sigma}$	(input)
<b>rhs</b>	the right hand side $\underline{x}(n + 1)$	(output)

The linearized system function will be connected to the simulator using the variable `MapLinearizedProxy::systemFunction`.

## Chapter 3

### Implementation of the linearized system functions

---

#### **Example:**

Linearized system function of the Hénon system (Eq. 2.8).

#### **Implementation**

```
#include "AnT.hpp"

#define alpha parameters[0]
#define beta  parameters[1]
#define X     currentState[0]
#define Y     currentState[1]

bool henon (const Array<real_t>&currentState,
            const Array<real_t>& parameters,
            Array<real_t>& rhs)
{
    rhs[0] = 1 - alpha * X * X + Y;
    rhs[1] = beta * X;
    return true;
}

#define Xr     referenceState[0]
#define Yr     referenceState[1]

bool henon_linearized (const Array<real_t>& currentState,
                      const Array<real_t>& referenceState,
                      const Array<real_t>& parameters,
                      Array<real_t>& rhs)
{
    rhs[0] = -2.0 * alpha * Xr * X + Y;
    rhs[1] = beta * X;
    return true;
}

#undef alpha
#undef beta
#undef X
#undef Y
#undef Xr
#undef Yr

extern "C"
{
    void connectSystem ()
    {
        MapProxy::systemFunction = henon;
        MapLinearizedProxy::systemFunction = henon_linearized;
    }
}
```

## 3.3 Spatial inhomogeneous systems

### 3.3.1 Coupled ordinary differential equation lattices

The interface of the linearized system function for a coupled ordinary differential equation lattice is defined by:

```
bool SystemFunction (  const CellularState & currentState,
                      const CellularState & referenceState,
                      const Array<real_t> & parameters,
                      int cellIndex,
                      StateCell & rhs);
```

with

<code>currentState</code>	the states $\underline{x}(t, 1) \dots \underline{x}(t, N)$	(input)
<code>referenceState</code>	the reference state $\underline{x}_{ref}$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>cellIndex</code>	spatial index $j$	(input)
<code>rhs</code>	the right hand side $\frac{d}{dt}\underline{x}(t, j)$	(output)

The linearized system function will be connected to the simulator using the variable `CODEL_Proxy::linearizedSystemFunction`.

### 3.3.2 Coupled map lattices

The interface of the linearized system function for a coupled map lattice is defined by:

```
bool SystemFunction (  const CellularState & currentState,
                      const CellularState & referenceState,
                      const Array<real_t> & parameters,
                      int cellIndex,
                      StateCell & rhs);
```

with

<code>currentState</code>	the state $\underline{x}(n)$	(input)
<code>referenceState</code>	the reference state $\underline{x}_{ref}$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>cellIndex</code>	spatial index $j$	(input)
<code>rhs</code>	the right hand side $\underline{x}(n + 1)$	(output)

The linearized system function will be connected to the simulator using the variable `CML_LinearizedProxy::systemFunction`.

## 3.4 Hybrid systems

### 3.4.1 Hybrid maps

The interface of the linearized system function for an hybrid map is defined by:

<pre>bool SystemFunction (     const Array&lt;real_t&gt; &amp; currentContinuousState,     const Array&lt;int_t&gt; &amp; currentDiscreteState,     const Array&lt;real_t&gt; &amp; referenceContinuousState,     const Array&lt;int&gt; &amp; referenceDiscreteState,     const Array&lt;real_t&gt; &amp; parameters,     Array&lt;real_t&gt; &amp; rhs);</pre>		
with		
currentContinuousState	the continuous state $\underline{x}(n)$	(input)
currentDiscreteState	the discrete state $\underline{m}(t)$	(input)
referenceContinuousState	$\underline{x}_{ref}$	(input)
referenceDiscreteState	$\underline{m}_{ref}$	(input)
parameters	the set of parameters $\underline{\sigma}$	(input)
rhs	the right hand side $\underline{x}(n + 1)$	(output)

The linearized system function will be connected to the simulator using the variable `HybridMapLinearizedProxy::systemFunction`.

### 3.4.2 Hybrid ordinary differential equations

The interface of the linearized system function for an hybrid ordinary differential equation is defined by:

```
bool SystemFunction (
    const Array<real_t> & currentContinuousState,
    const Array<int_t> & currentDiscreteState,
    const Array<real_t> & referenceContinuousState,
    const Array<int> & referenceDiscreteState,
    const Array<real_t> & parameters,
    Array<real_t> & rhs);
```

with

<code>currentContinuousState</code>	the continuous state $\underline{x}(n)$	(input)
<code>currentDiscreteState</code>	the discrete state $\underline{m}(t)$	(input)
<code>referenceContinuousState</code>	$\underline{x}_{ref}$	(input)
<code>referenceDiscreteState</code>	$\underline{m}_{ref}$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>rhs</code>	the right hand side $\frac{d}{dt}\underline{x}(t)$	(output)

The linearized system function will be connected to the simulator using the variable `HybridODE.LinearizedProxy::systemFunction`.

# Chapter 4

## User defined symbolic partitions of the state space

### 4.1 Symbolic dynamics

## 4.2 Basic system types

### 4.2.1 Ordinary differential equations

The interface of the symbolic function for an ordinary differential equation is defined by:

```
bool SymbolicFunction (  const Array<real_t> & currentState,
                        const Array<real_t> & parameters,
                        char* symbolicRHS);
```

with

<code>currentState</code>	the state $\underline{x}(t)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>symbolicRHS</code>	the right hand side $\underline{s}$	(output)

### 4.2.2 Delay differential equations

The interface of the symbolic function for a delay differential equation is defined by:

```
bool SymbolicFunction (  const Array<real_t> & currentState,  
                        const Array<real_t> & delayState,  
                        const Array<real_t> & parameters,  
                        char* symbolicRHS);
```

with

<code>currentState</code>	the state $\underline{x}(t)$	(input)
<code>delayState</code>	$\underline{x}(t - \tau)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>symbolicRHS</code>	the right hand side $\underline{s}$	(output)

### 4.2.3 Functional differential equations

The interface of the symbolic function for a functional differential equation is defined by:

```

bool SymbolicFunction (
    const Subarray<CyclicArray<Array<real_t>>> &
                                                currentState,
    const Array<real_t> & parameters,
    const real_t deltaT,
    char* symbolicRHS);

```

with

<code>currentState</code>	the states $\underline{x}(t - \tau) \dots \underline{x}(t)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>deltaT</code>	$\tau$	(input)
<code>symbolicRHS</code>	the right hand side $\underline{s}$	(output)

### 4.2.4 Maps

The interface of the symbolic function for a map is defined by:

```
bool SymbolicFunction (  const Array<real_t> & currentState,
                        const Array<real_t> & parameters,
                        char* symbolicRHS);
```

with

<code>currentState</code>	the state $\underline{x}(n)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>symbolicRHS</code>	the right hand side $\underline{s}$	(output)

### 4.2.5 Recurrent maps (with recurrence level greater than one)

The interface of the symbolic function for a recurrent map is defined by:

```

bool SymbolicFunction (
    const CyclicArray<Array<real_t>> & currentState,
    const Array<real_t> & parameters,
    char* symbolicRHS);

```

with

<code>currentState</code>	the states $\underline{x}(n - N) \dots \underline{x}(n)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>symbolicRHS</code>	the right hand side $\underline{s}$	(output)

## 4.3 Spatial inhomogeneous dynamical systems

### 4.3.1 One-dimensional partial differential equations

The interface of the symbolic function for an one-dimensional partial differential equation is defined by:

```
bool SymbolicFunction (  const CellularState & currentState,
                        const Array<real_t> & parameters,
                        int cellIndex,
                        real_t deltaX,
                        char* symbolicRHS);
```

with

<code>currentState</code>	the states $\underline{x}(t, 1) \dots \underline{x}(t, N)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>cellIndex</code>	spatial index $j$	(input)
<code>deltaX</code>	spatial grid step	(input)
<code>symbolicRHS</code>	the right hand side $\underline{g}$	(output)

### 4.3.2 Coupled ordinary differential equation lattices

The interface of the symbolic function for a coupled ordinary differential equation lattice is defined by:

```
bool SymbolicFunction (  const CellularState & currentState,
                        const Array<real_t> & parameters,
                        int cellIndex,
                        char* symbolicRHS);
```

with

<b>currentState</b>	the states $\underline{x}(t, 1) \dots \underline{x}(t, N)$	(input)
<b>parameters</b>	the set of parameters $\underline{\sigma}$	(input)
<b>cellIndex</b>	spatial index $j$	(input)
<b>symbolicRHS</b>	the right hand side $\underline{s}$	(output)

### 4.3.3 Coupled map lattices

The interface of the symbolic function for a coupled map lattice is defined by:

```
bool SymbolicFunction (  const CellularState & currentState,
                        const Array<real_t> & parameters,
                        int cellIndex,
                        char* symbolicRHS);
```

with

<code>currentState</code>	the state $\underline{x}(n)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>cellIndex</code>	spatial index $j$	(input)
<code>symbolicRHS</code>	the right hand side $\underline{s}$	(output)

## 4.4 Hybrid systems

### 4.4.1 Hybrid maps

The interface of the symbolic function for an hybrid map is defined by:

```

bool SymbolicFunction (
    const Array<real_t> & currentContinuousState,
    const Array<int> & currentDiscreteState,
    const Array<real_t> & parameters,
    char* symbolicRHS);

```

with

<code>currentContinuousState</code>	the continuous state $\underline{x}(n)$	(input)
<code>currentDiscreteState</code>	the discrete state $\underline{m}(t)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>symbolicRHS</code>	the right hand side $\underline{s}$	(output)

### 4.4.2 Hybrid ordinary differential equations

The interface of the symbolic function for a hybrid ordinary differential equation is defined by:

```
bool SymbolicFunction (  
    const Array<real_t> & currentContinuousState,  
    const Array<int> & currentDiscreteState,  
    const Array<real_t> & parameters,  
    char* symbolicRHS);  
with  
  
currentContinuousState  the continuous state  $\underline{x}(t)$  (input)  
currentDiscreteState    the discrete state  $\underline{m}(t)$  (input)  
parameters              the set of parameters  $\underline{\sigma}$  (input)  
symbolicRHS            the right hand side  $\underline{s}$  (output)
```

### 4.4.3 Hybrid delay differential equations

The interface of the symbolic function for a hybrid delay differential equation is defined by:

```
bool SymbolicFunction (
    const Array<real_t> & currentContinuousState,
    const Array<int> & currentDiscreteState,
    const Array<real_t> & delayState,
    const Array<real_t> & parameters,
    char* symbolicRHS);
```

with

<code>currentContinuousState</code>	the continuous state $\underline{x}(t)$	(input)
<code>currentDiscreteState</code>	the discrete state $\underline{m}(t)$	(input)
<code>parameters</code>	the set of parameters $\underline{\sigma}$	(input)
<code>delayState</code>	the delay state $\underline{x}(t - \tau)$	(input)
<code>symbolicRHS</code>	the right hand side $\underline{s}$	(output)

## Chapter 5

# User defined conditions for Poincaré maps