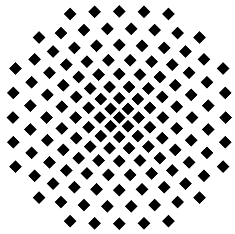


Institute of Parallel and Distributed
High-Performance Systems (IPVR)
University of Stuttgart, Germany

A tool for simulating and investigating dynamical systems

Robert Lammert

E-mail: rtlammer@informatik.uni-stuttgart.de

CONTENTS

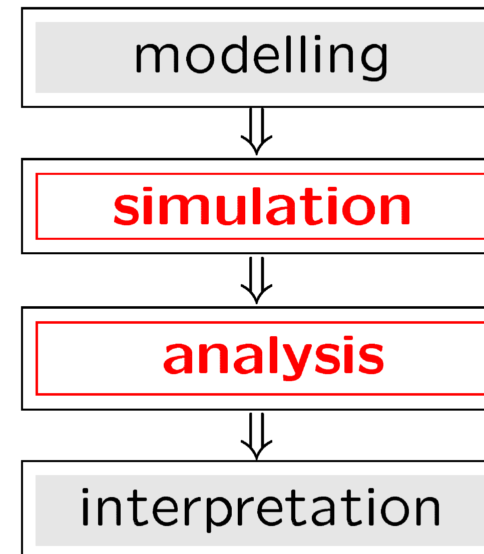
- Introduction (what, who, what for, etc.)
- Important aspects
  - simulator, static vs. dynamic programming, general iterator
- Dynamical system types (or classes)
- Relevant architectural and implementational issues
  - proxy, iteration, scan
- Input, output, parallel scan execution
- Outlook

WHAT IS AnT 4.667 ?

AnT 4.667 – a simulation and analysis tool for dynamical systems

- main application areas:
  science and education

- some properties:
  - support of several classes of dynamical systems
  - several investigation methods
  - scan-capabilities
  - open software architecture



INTRODUCTION

AnT 4.667 is developed by the Non–Linear Dynamics Group of the Department of Image Understanding (Head: Prof. Dr. P. Levi) at the Institute for Parallel and Distributed High-Performance Systems of the University of Stuttgart.

Members of the group:

- Viktor Avrutin,
- Robert Lammert,
- Dr. Michael Schanz,
- Heiko Schäfer,
- Michael Schulze,
- Sascha Riexinger,
- Georg Wackenhut.

History of the project:

1998: first prototypes (FORTRAN, C)

2000: **AnT 4.66** (C)

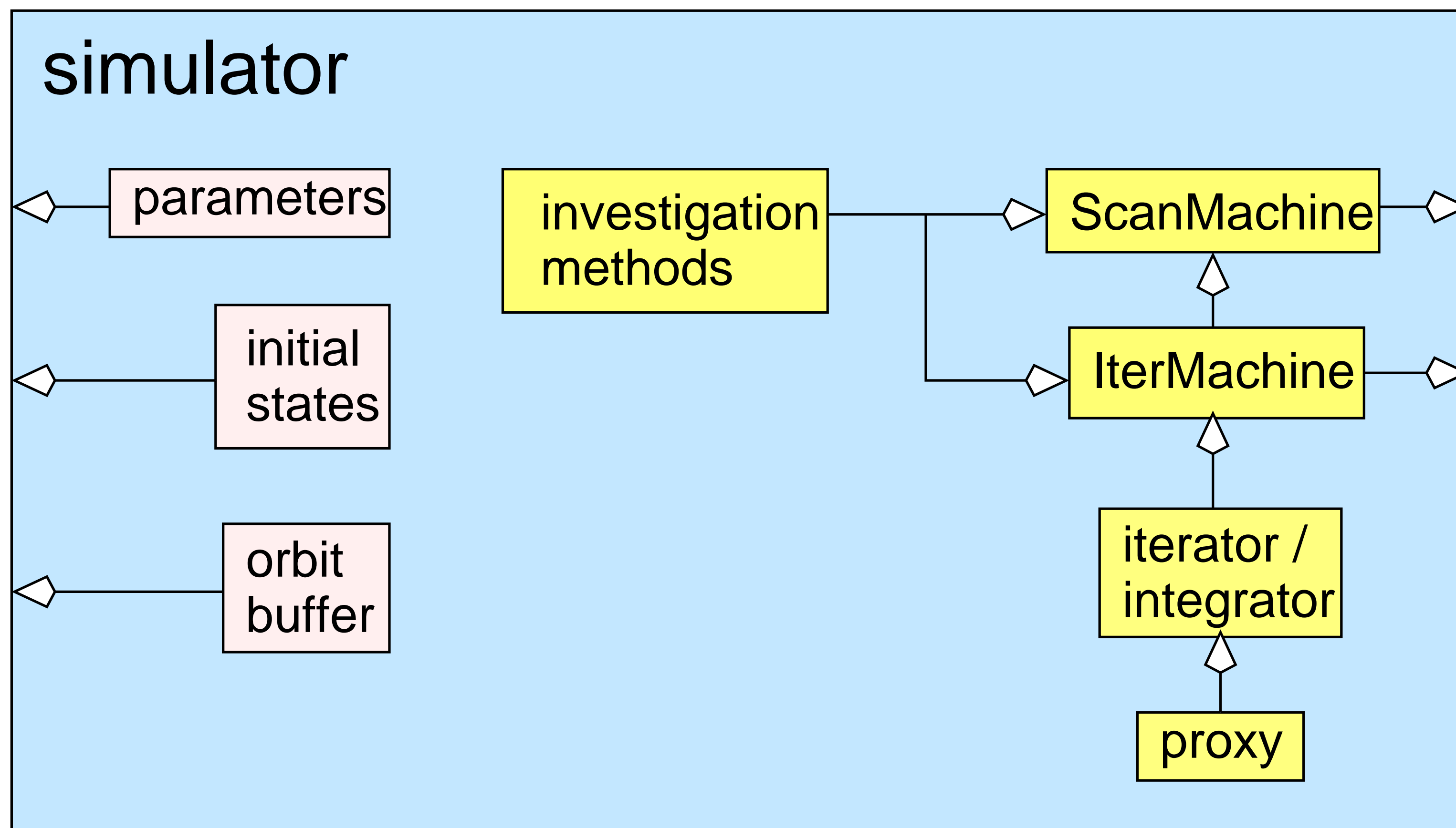
2001: **AnT 4.667** (C++)

INTRODUCTION

Requirements for simulating and analyzing
a specific dynamical system:

- initial states and hence a description of the state space they are defined in
- a list of parameters affecting the orbit beeing computed ($\rightarrow$ scannable items)
- an iteration scheme, totally or partially given by the user of the simulator
- dynamically plugable investigation methods for data generation (computation + output)

Simplified structure of a simulator for dynamical systems:



STATIC VS. DYNAMIC PROGRAMMING

- Example of a static structure:

```
void doSomething ()
{
    doAction1 ();
    doAction2 ();
    doAction3 ();
}
```

- Example of a dynamic structure:

```
void doSomething ()
{
    CmdList* prog;
    create (prog);
    evaluate (prog);
}
```

```
void create (CmdList* prog)
{
    add (prog, doAction1);
    add (prog, doAction2);
    add (prog, doAction3);
}
```

DYNAMIC PROGRAMMING STRUCTURES

+ The creation of the dynamic flow may be distributed all over the program

+ Evaluation occurs lazily, i.e. **after** the structure has been constructed

+ **if**-cascades depending on initial constraints must be evaluated only once

+ The structure can be modified accordingly before a subsequent run (by operations like **add**, **remove**, etc.)

– Less readable programs, hence more difficult to understand and maintain

INVESTIGATION ISSUES

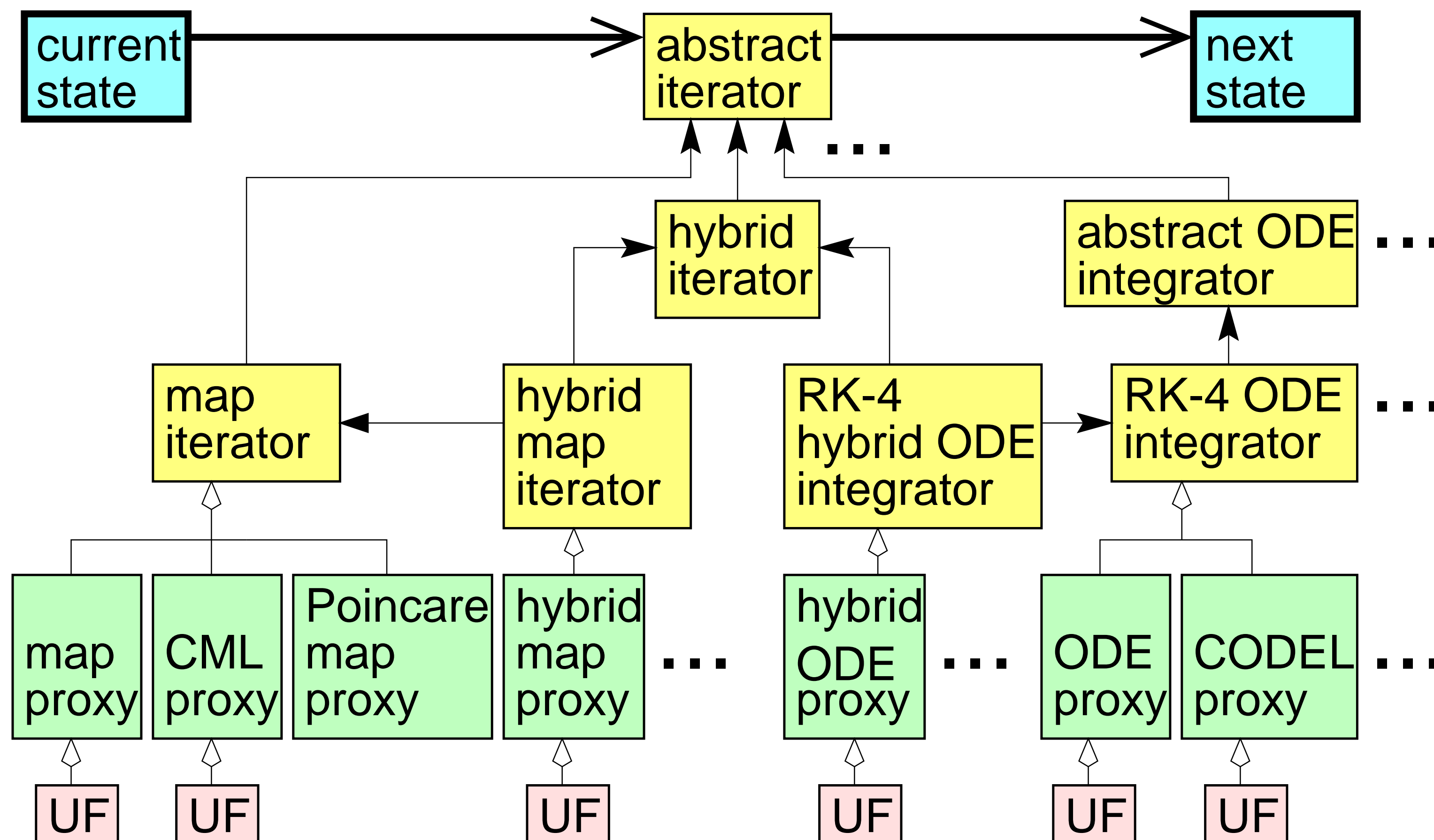
- Dynamical system analysis is based on investigation methods (the results must be interpreted by humans)
- Method activation depends on conditions which may be evaluated in the **initialization** part of the program
- Methods affect the normal program flow at several locations (e.g. initialization, computation, data output)
- Activation or omission of a method should not affect other entities (or inforce major program changes)
- Certain program (method) entities are likely to be evaluated repeatedly during scan execution

BASIC CONCEPTS

- Each dynamically executable entity is a kind (subclass) of **AbstractTransition**
- State machines are executable entities consisting of several transitions and a specific execution scheme
- States (subclasses of **AbstractState**) hold the data needed by transitions during their execution
- Common execution schemes are implemented in an abstract way, encouraging and facilitating code reuse
  - (abstract) state machines implementing e.g. sequences of transitions or loops

The **iterator/integrator**:

- computes the next state of the orbit



BASIC DYNAMICAL SYSTEM TYPES

- **maps:** $\vec{x}_{n+1} = \vec{f}(\vec{x}_n, \{\sigma\})$
- **recurrent maps:** $\vec{x}_{n+1} = \vec{f}(\vec{x}_n, \vec{x}_{n-1}, ..., \vec{x}_{n-\tau}, \{\sigma\})$
- **ODEs:** $\dot{\vec{x}}(t) = \vec{f}(\vec{x}(t), \{\sigma\})$
- **DDEs:** $\dot{\vec{x}}(t) = \vec{f}(\vec{x}(t), \vec{x}(t-\tau), \{\sigma\})$
- **FDEs:** $\dot{\vec{x}}(t) = \vec{f}\,[\vec{x}_t, \{\sigma\}]$
  with $\vec{x}_t(\theta) = \vec{x}(t+\theta)$, $\theta \in [-\tau, 0])$
- **PDEs:** $\frac{\partial}{\partial t}\vec{x}(q,t) = \vec{f}\left(\vec{x}(q,t), \frac{\partial}{\partial q}\vec{x}(q,t), \ldots, \{\sigma\}\right)$
- **external data input** $\vec{x}_{n+1} =$ next input vector

COMPOSITE (CELLULAR) DYNAMICAL SYSTEMS

- CMLs

$$\vec{x}_{n+1}^{(i)} = \vec{f}\left(\vec{x}_n^{(i-r)}, \ldots, \vec{x}_n^{(i)}, \vec{x}_n^{(i+r)}, \{\sigma\}\right)$$

- CODELs

$$\dot{\vec{x}}^{(i)}(t) = \vec{f}\left(\vec{x}^{(i-r)}(t), \ldots, \vec{x}^{(i)}(t), \ldots, \vec{x}^{(i+r)}(t), \{\sigma\}\right)$$

- CDDELs

$$\begin{aligned}\dot{\vec{x}}^{(i)}(t) \;=\; & \vec{f}\Big(\vec{x}^{(i-r)}(t), \ldots, \vec{x}^{(i)}(t), \ldots, \vec{x}^{(i+r)}(t) \\ & \vec{x}^{(i-r)}(t-\tau), \ldots, \vec{x}^{(i)}(t-\tau), \ldots, \\ & \vec{x}^{(i+r)}(t-\tau), \{\sigma\}\Big)\end{aligned}$$

i is the cell index, $\quad i = 1, \ldots, N$

HYBRID DYNAMICAL SYSTEMS

- hybrid maps

$$\begin{aligned} \vec{x}_{n+1} &= \vec{f}(\vec{x}_n, \vec{m}_n, \{\sigma\}) \\ \vec{m}_{n+1} &= \vec{\phi}(\vec{x}_n, \vec{m}_n, \{\sigma\}) \end{aligned}$$

- hybrid ODEs

$$\begin{aligned} \dot{\vec{x}}(t) &= \vec{f}(\vec{x}(t), \vec{m}(t), \{\sigma\}) \\ \vec{m}(t^+) &= \vec{\phi}(\vec{x}(t), \vec{m}(t), \{\sigma\}) \end{aligned}$$

- hybrid DDEs

$$\begin{aligned} \dot{\vec{x}}(t) &= \vec{f}(\vec{x}(t), \vec{x}(t-\tau), \vec{m}(t), \{\sigma\}) \\ \vec{m}(t^+) &= \vec{\phi}(\vec{x}(t), \vec{m}(t), \{\sigma\}) \end{aligned}$$

$\vec{x}_n \rightarrow$ real valued state vector
$\vec{m}_n \rightarrow$ integer valued state vector

STOCHASTICAL DYNAMICAL SYSTEM

- stochastical systems, based on:
  - . . . maps: $\vec{x}_{n+1} = \vec{f}(\vec{x}_n, \{\sigma\}) + \vec{\eta}$
  - . . . ODEs: $\dot{\vec{x}}(t) = \vec{f}(\vec{x}(t), \{\sigma\}) + \vec{\eta}$
  - . . . DDEs: $\dot{\vec{x}}(t) = \vec{f}(\vec{x}(t), \vec{x}(t-\tau), \{\sigma\}) + \vec{\eta}$

- averaged stochastical systems (for instance, maps)

$$\vec{x}_{n+1} = \frac{1}{N} \sum_{i=1}^{N} \left(\vec{f}(\vec{x}_n^{(i)}, \{\sigma\}) + \vec{\eta}^{(i)} \right)$$

Note:

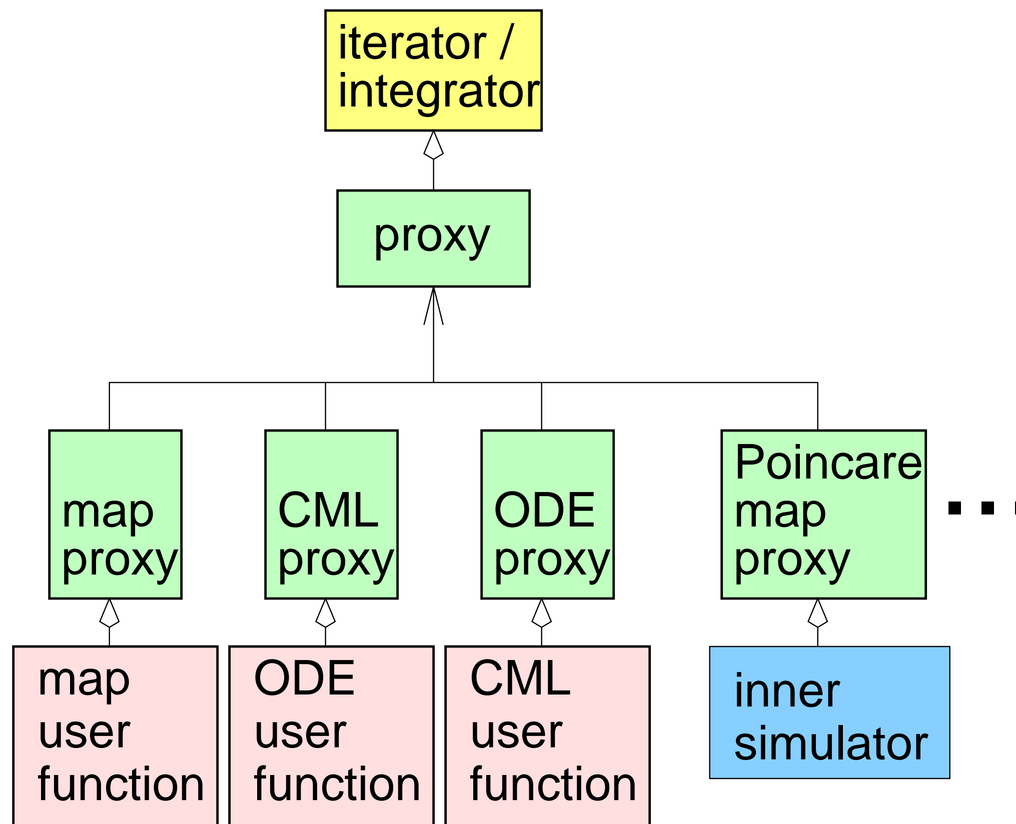
From a dynamical system viewpoint – an investigation method.

From the simulation viewpoint – a class of dynamical systems.

We call it a meta dynamical system.

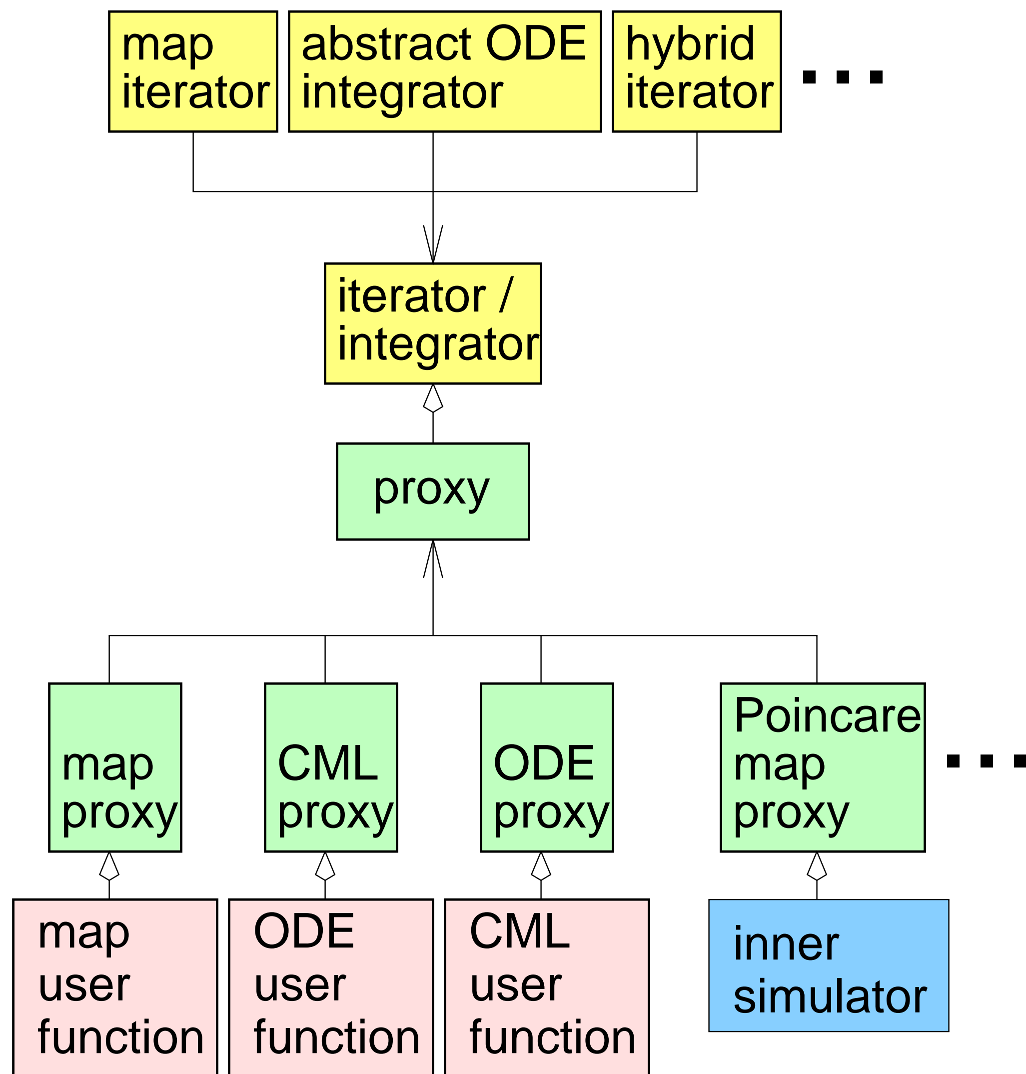
PROXIES

Proxies translate one given interface into other interfaces
$\Rightarrow$ iterators/integrators can cope with different
types of user functions (interfaces)

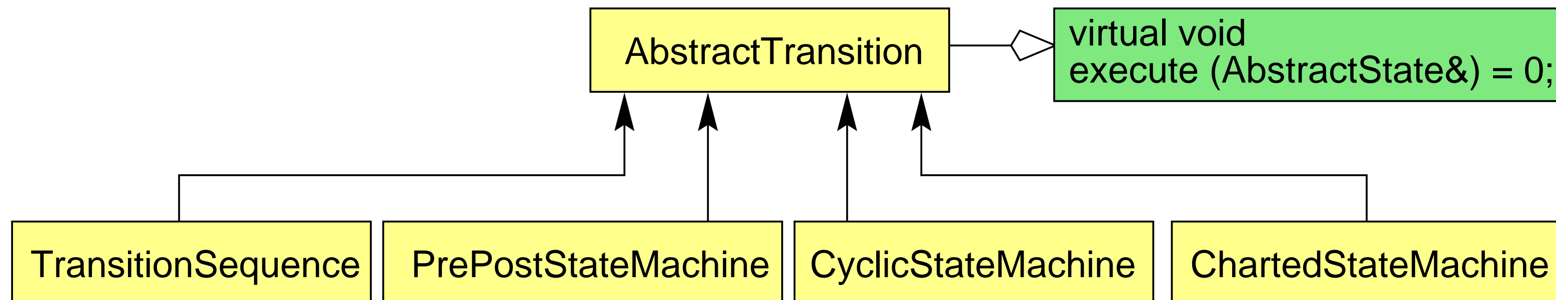


PROXIES

Iterator/integrator types have their own sets of proxies



EXECUTABLE ENTITIES

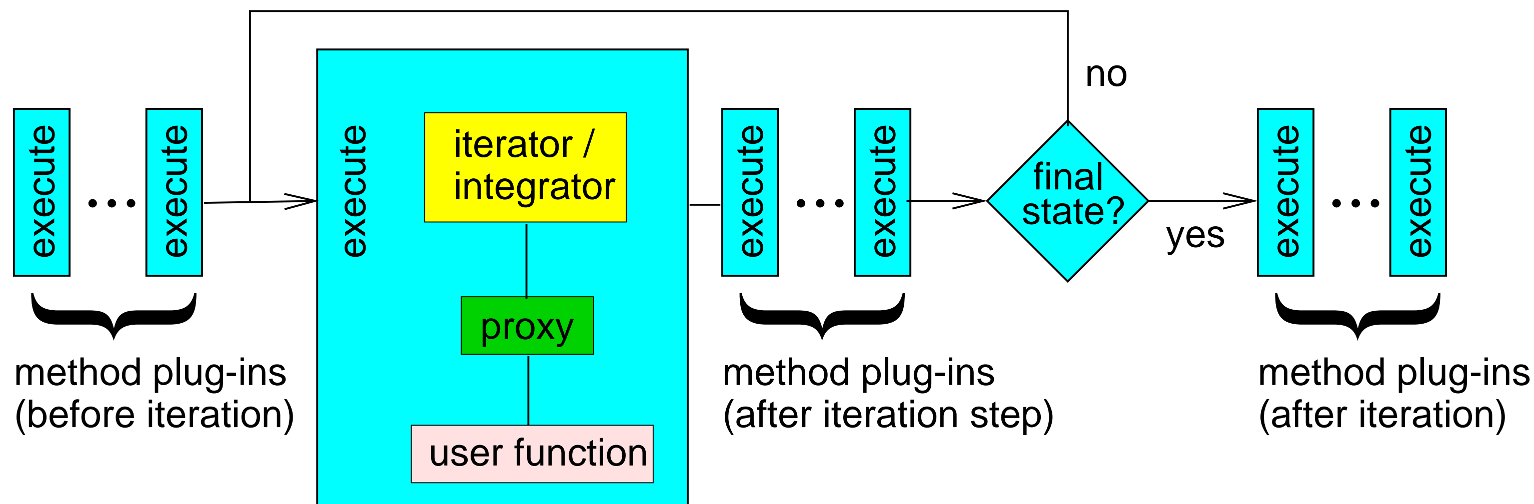


- **TransitionSequence**: a sequential flow of execution
- **PrePostStateMachine**: the pre-, main- and post-transition are executed in this order
- **CyclicStateMachine**: allows the execution of a loop based on a given transition (state machine)
- **ChartedStateMachine**: the execution scheme is implemented as a state chart $\rightarrow$ transition to be executed

ITERATION MACHINE

The **IterMachine**:

- embeds the iterator in a **CyclicStateMachine**
- includes method plug-ins as transitions
- allows pre-, post- and after-step-processing



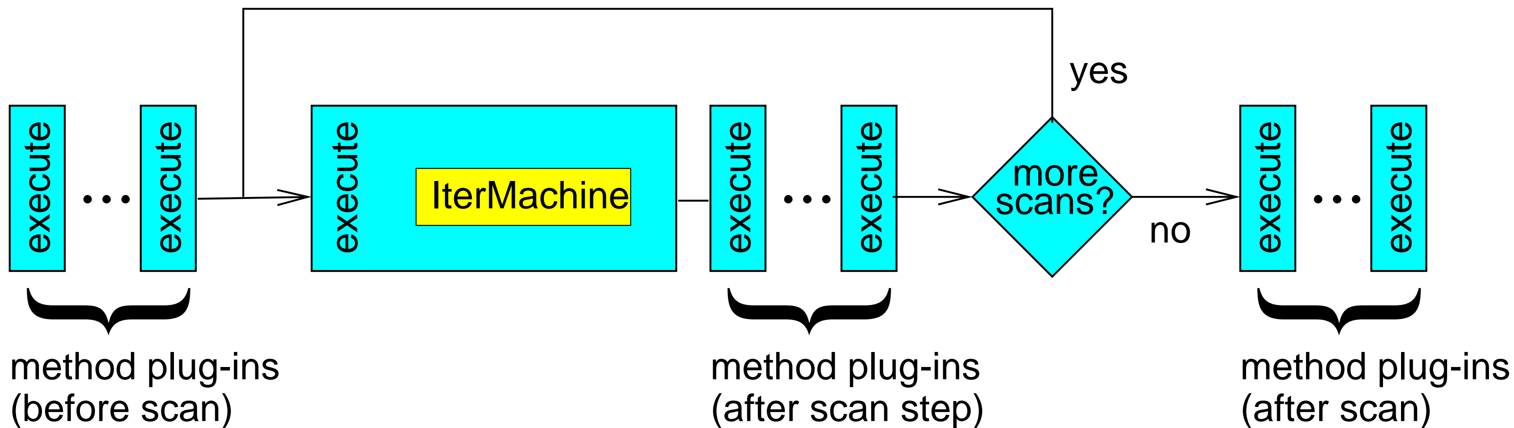
- Investigation method ≈ calculation of several characteristical quantities based on orbit(s).

INVESTIGATION METHODS

- Calculation of Lyapunov Exponents
- Spectral analysis (ODEs)
- Principal component analysis
- Period analysis (maps)
- Symbolic sequence analysis
- Computation of metric entropy and fractal dimensions
- General trajectory evaluations

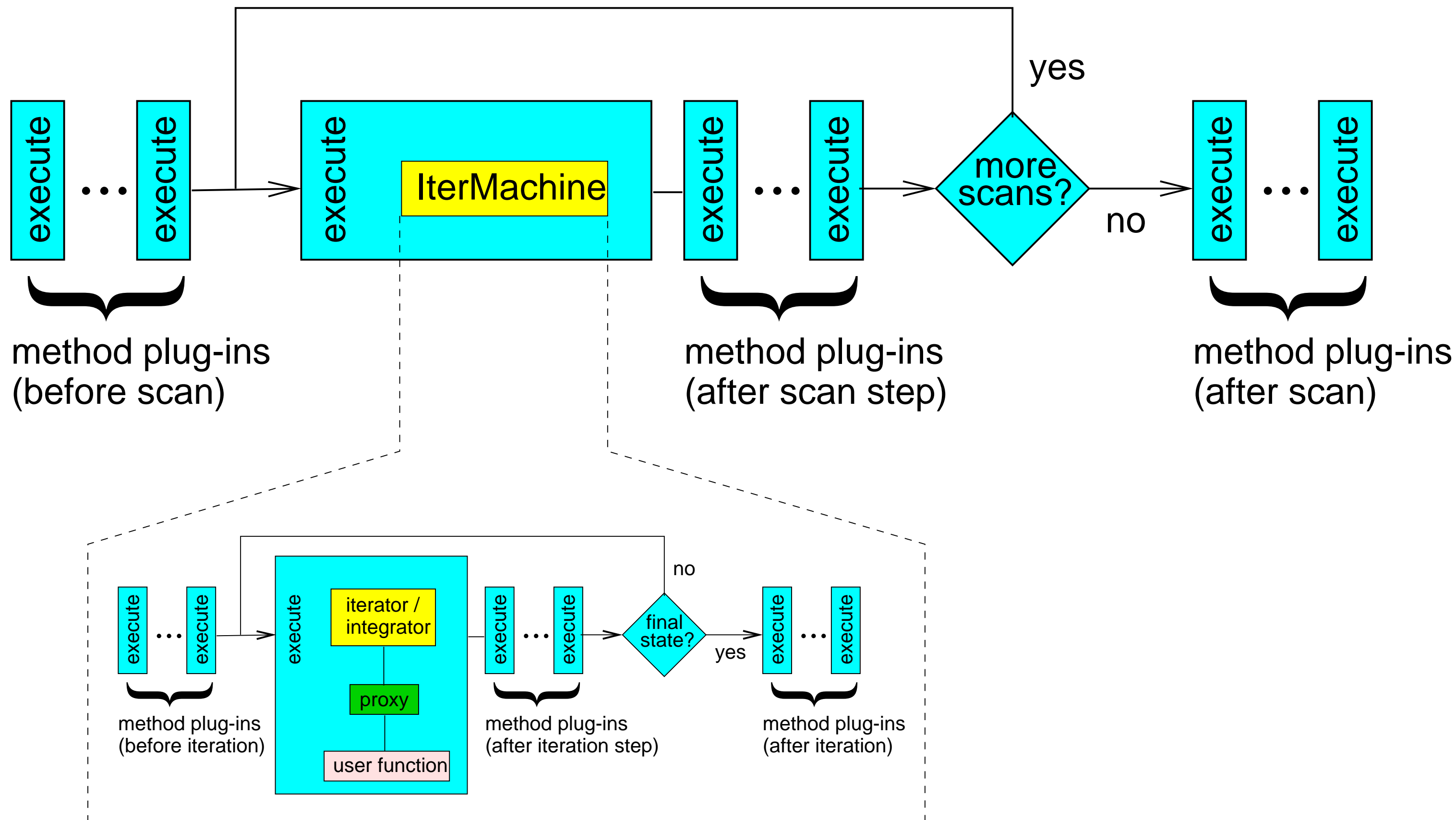
SCAN MACHINE

- structure similar to that of **IterMachine**
- certain investigation methods are only available (and reasonable) in a scan mode run
- some commonly used scan modi are available yet (linear, logarithmic)



SCAN MACHINE

- **IterMachine** is the core of all simulations
- Iterations (**IterMachine**) are executed repeatedly (**CyclicStateMachine**) until the scan mode completes



SCAN MODE

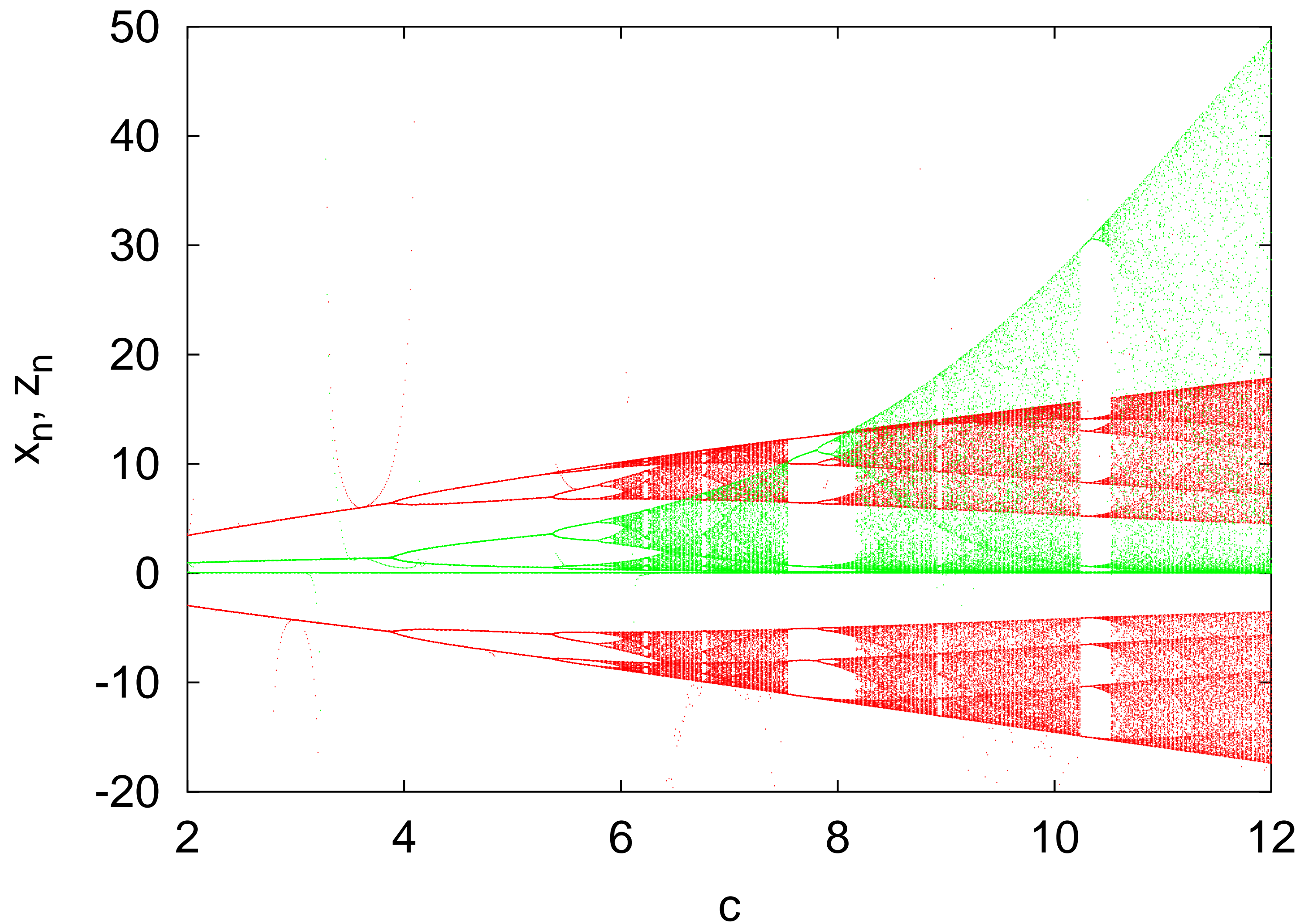
- **scan** is the procedure of updating values and restarting the execution as long as a given condition holds

- a 1-dim bifurcation diagram is the result of a scan execution, where a parameter gets updated before starting the iteration

- the scan usually involves the systematic exploration of a given space, according to a specific exploration scheme

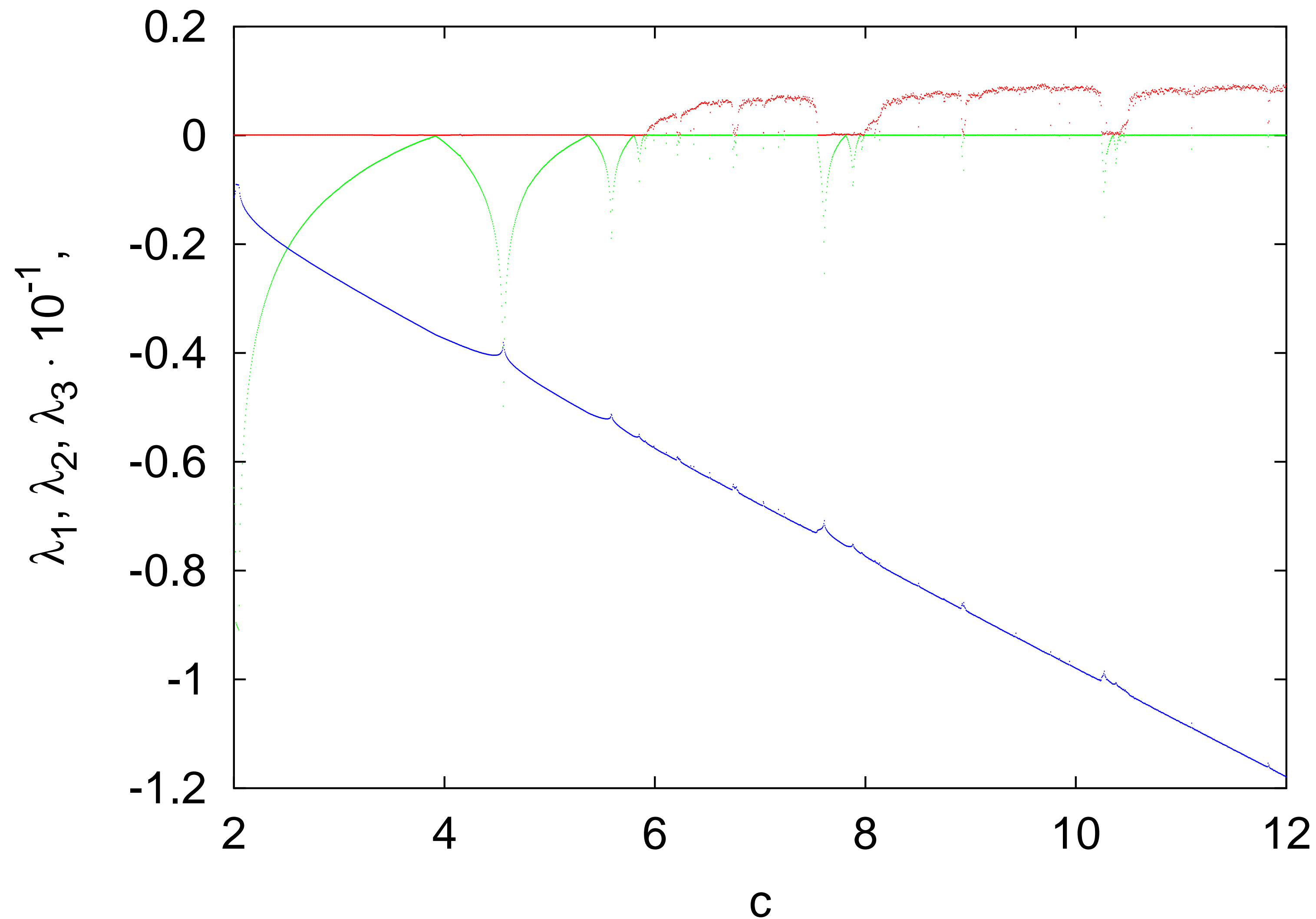
- the space of exploration consists of the so-called **scannable items**, the exploration scheme denotes the algorithm used to determine the next scan settings

Bifurcation diagram (Roessler system)



SCAN MODE

Lyapunov exponents diagram (Roessler system)



POINCARÉ MAPS

- Poincaré maps consider only discrete aspects of a continuous (complex) system

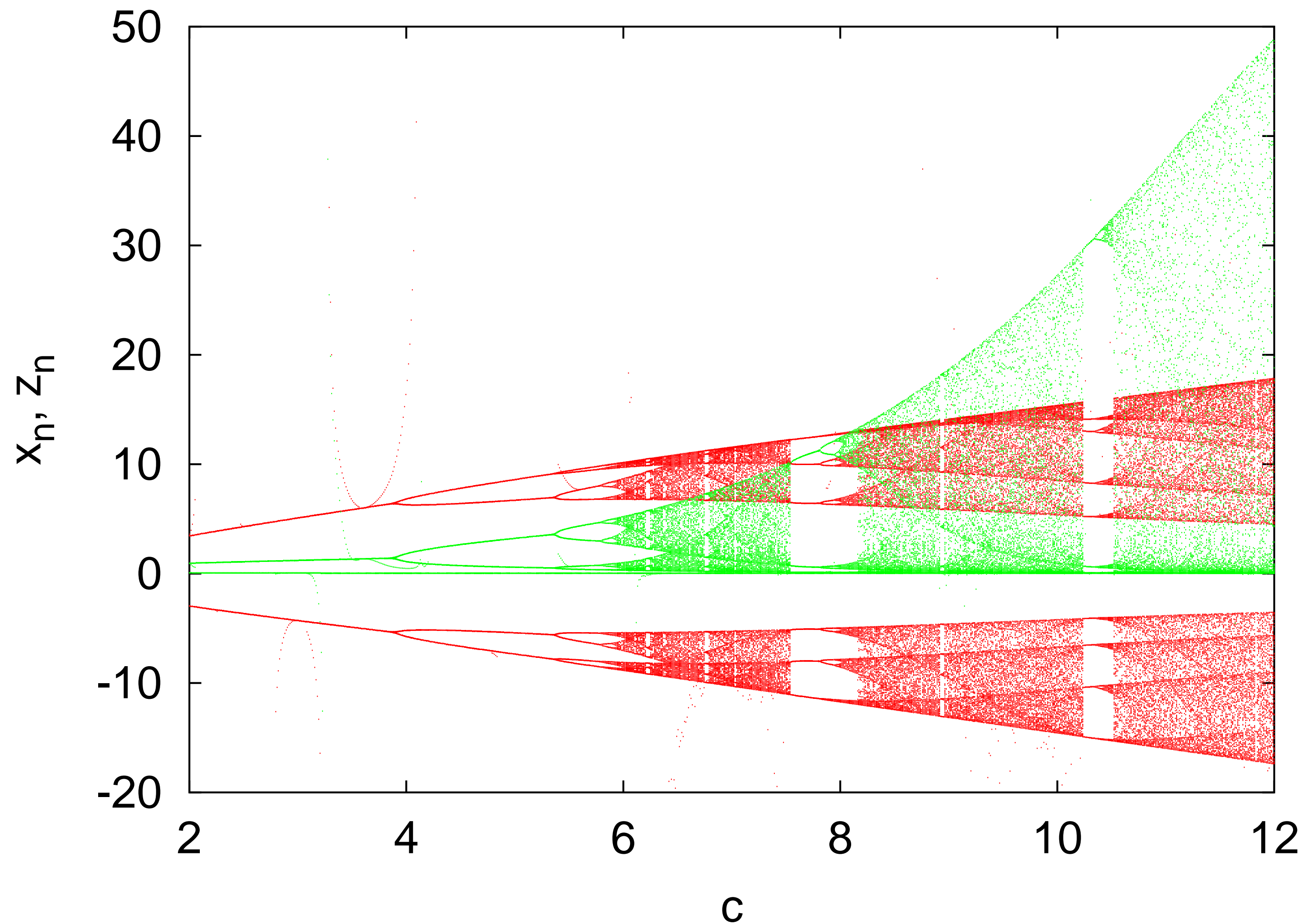
- The key concept is the **PoincareProxy**-class, which deals with the "low-level" iteration steps

- Poincaré maps

  - **Standard Poincaré sections:** Cross-section points of the orbit with a (hyper)–plane in the state space.

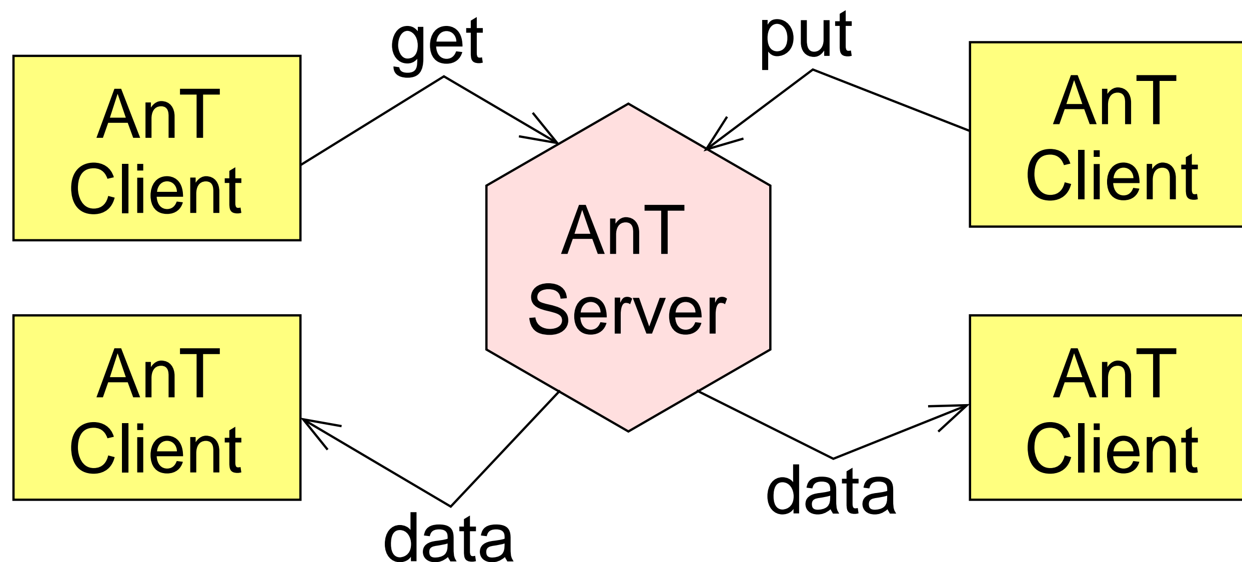
  - **Generalized Poincaré sections:** Points in the state space, where some specific condition concerning the orbit is fulfilled.

POINCARÉ MAP: EXAMPLE



PARALLEL SCAN EXECUTION

- A **scan point run** creates an orbit depending on the actual scan settings (parameters, initial values, etc.)
- scan point runs are unrelated to each other and may be processed concurrently (**parallel processing**)
- the parallel processing of scan points relies on a **server/client** architecture (one server, many clients)



USER INTERFACE

- the system function(s) must be written in C++, in accordance with the system type requirements
- the data needed for initializing the simulator is stored in a text file (the so called **ini-file**)
- one can either edit the ini-file directly or use a related graphical interface (not fully implemented yet)
- the simulator produces data files containing numbers in a tabular form (→ gnuplot ;-) . . .)
- a package for the on-line drawing of 3D-orbits is available for single mode runs (no scans)

USER INTERFACE

Example of a user provided system function:

```
#define sigma parameters[0]
#define r     parameters[1]
#define b     parameters[2]
#define alpha parameters[3]
#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;
}
```

OUTLOOK

- implementation of more commonly used methods, especially concerning **numerical integration** algorithms
- completion of the graphical user interface (input), allowing a **consistent initialization** of the simulator
- improvement of the on-line visualization module w.r.t. stability and graphical design (appearance)
- possibly altering the basic structures by introducing an **abstract number** class ($\rightarrow$ complex numbers, . . .)
- user defined scheme for updating the scan settings
- . . .