Modeling with Markov chains: The MARMOTE Software

Labex UCN seminar
Sophia Antipolis 3 september 2015
1 Markov modeling
Markov modeling

What is a Markov Chain
What is a Markov modeler
Markov chains

Markov chains/processes are a class of stochastic processes $X(t)$, $t \in \mathbb{Z}$ or $\mathbb{R}$

- evolving on a state space $\mathcal{E}$
- described by a transition rule

\[ i \rightarrow j \quad \text{with probability } p_{ij} \]

\[ i \rightarrow j \quad \text{with rate } \lambda_{ij} \]

- from some initial state
Markov chains, ctd

Many properties of Markov chains can be obtained

- by analyzing the graph of transitions
- by solving problems of linear algebra
Markov modeling consists in

- constructing Markov models
- analyzing them:
  - determine qualitative properties: structure, ergodicity, stability ...
  - compute metrics related with probabilities, frequencies, times, durations ...

Two main “profiles” + mixed ones
Markov modelers, type 1

**Theoretician:** Aims at developing MC solution methods

- as generic as possible
- yet taking into account the structure of the model

Involves:

- invent new formulas/algorithms
- program new methods
- test them on examples/benchmarks
- compare with previous methods (exec. time, accuracy)
Markov modelers, type 2

Practician: Develops Markov models for specific applications

Involves:

- describe/represent model (parameters, structure, ...)
- test model with simulation
- solve model (analytic, numerical), loop until model passes tests
- execute experimental plans
- compare different models (e.g. simplifications)
The MARMOTE project

MARMOTE: MARkovian MOdeling Tools and Environments
ANR-12-MONU-0019
2013 – 2016

Objectives:

- Develop Markov Environment
- New solution and simulation techniques
- Application test-cases
Objectives of the MARMOTE Software platform:

**Initial specifications**

To provide to the general scientist a “modeling environment” which must give access to algorithms developed by specialists. Principal characteristics: be as open as possible, be component-oriented, contributive. Will be populated with a modeling language, a minimal user interface, minimal solution algorithms.
MARMOTE Software base

Secondary objective: unify existing software
The partners bring existing software packages
  - Psi, Psi2, Psi3 (Perfect Simulation, INRIA/MESCAL)
  - Xborne (Solutions with bounds for MC-related distributions, UVSQ/PRiSM)
  - ERS (Basic MC solution)

Other software around
  - PEPS
  - GreatSPN
  - Tangram II
  - ...

Markov modeling
Requirements

Modeling abstraction

- possibility to handle formal models, infinite state spaces, ...

Other requirements came along the way:

- possibility to communicate with other modeling/solution systems: matlab, R, ...
- possibility to be used in workflow management systems
- possibility to implement multiple solution methods for different types of Markov chains
Architecture

Target Architecture in three layers:

- Bottom: solution methods
- Middle: Marmote API, construction of models, handling of data, algorithms, results
- Top: GUI and workflow management

Choice of an object-oriented language: C++
The MARMOTE Core
Main Objects

Classes currently implemented

▶ Markov chains: markovChain and derived classes
▶ Transitions: transitionStructure
▶ State Spaces: marmoteSet
▶ Distribution and derived classes
The Distribution object

Methods

virtual double mean() = 0;
virtual double rate() = 0; // inverse of the mean
virtual double moment(int n) = 0;
double variance();
virtual double laplace(double s) = 0;
virtual double dLaplace(double s) = 0;
virtual double cdf(double x) = 0;
double ccdf(double x) return 1.0 - cdf(x); ;
virtual bool hasMoment(int n) = 0;
virtual Distribution* rescale(double factor) = 0;
virtual Distribution* copy() = 0;
virtual double sample() = 0;
virtual void iidSample(int n, double* s) = 0;
virtual double distanceL1(Distribution*);
virtual bool hasProperty(std::string);
Distributions

Distributions implemented:

- Distribution/discreteDistribution
  - Distribution/diracDistribution
  - Distribution/bernoulliDistribution
  - Distribution/uniformDiscreteDistribution
  - Distribution/geometricDistribution
- Distribution/exponentialDistribution
- Distribution/uniformDistribution
The MARMOTE core

Transition Structures

Methods

timeType getType() return _type;
virtual double getEntry(int,int) = 0;
virtual int getNbElts(int) = 0;
virtual int getCol(int,int) = 0;
virtual double getEntryByCol(int,int) = 0;
virtual DiscreteDistribution* getTransDistrib(int) = 0;
// transitions from some state and their probas
bool readEntry(FILE*);
virtual double rowSum(int) = 0;
virtual transitionStructure* uniformize() = 0;
virtual void evaluateMeasure(double*,double*);
virtual void
evaluateMeasure(DiscreteDistribution*,DiscreteDistribution*);
virtual void evaluateValue(double*,double*) = 0;
Transition structures

Transition structures implemented:

- transitionStructure/sparseMatrix
- transitionStructure/multiDimHomTransition (generalized birth-death)
- transitionStructure/eventMixture

Projected:

- transitionStructure/matrix
- transitionStructure/QBD
State spaces

Attributes of marmoteSet

```c
enum opType UNION, PRODUCT, SIMPLE;
bool _isSimple, _isUnion, _isProduct;
int _nbDimensions;
int _nbZones;
long int _cardinal;
marmoteSet** _zone;
marmoteSet** _dimension;
int* _stateBuffer;
int* _dimOffset;
int* _idxOffset;
int _totNbDims;
itnt* _zeroState;
```
The MARMOTE core

Marmote sets, ctd

Methods

// constructors
marmoteSet();
marmoteSet( marmoteSet **list, int nb, opType t );
// accessors
virtual long int cardinal();
bool isFinite(), bool isSimple(), bool isUnion(), bool isProduct();
int totNbDims();
// state-index conversions
virtual void decodeState(int index, int* buffer);
int index(int* buffer);
// state space exploration
virtual void firstState(int* buffer);
virtual void nextState(int* buffer);
virtual bool isZero(int* buffer);
// utilities
virtual void enumerate();
virtual void printState(FILE* out, int index);
The Markov chains
Markov Chains

Markov Zoo

The MARMOTE Software, Labex UCN seminar, Sophia Antipolis

3 september 2015
Markov Chains

Markov Zoo

- GeneralMarkov
  - CPED
  - Urn
  - QBD
    - HomogeneousQBD
  - GraphRW
  - RW1D
- DiscreteIID
- MCD
- GlauberIsing
- MultiTypeGaltonWatson
- GaltonWatson
- TwoState

Biology

Physics (Statistical Mechanics)
The Markov Chain object

Principles:

- **“concrete” chains**
  - read/create from files in several formats Ers, Marca...
    ➞ being implemented: HBF, Xborne transition spec, PSI spec
  - create from a transitionStructure (e.g. matrix)
  - write to file in Ers format
    ➞ being implemented: Marca, HBF, R, scilab/matlab, Maple, ....

- **“virtual” chains specified by**
  - format
  - name of model
  - optional liste of extensions and/or file names

“lazy” evaluation
Attributes

Attributes of MarkovChain

timeType _type;
int _stateSpaceSize;
transitionStructure* _generator;
DiscreteDistribution* _initDistribution;

// fields related to abstractness and file names
bool _isAbstract;
int _abstractNbre;
string* _abstract;
string _format;
string _modelName;
4
Programming Markov Chains
Creation

Three classes of possibilities:

1. read the generator (and the state space) in a file
2. use a predefined class
3. create the generator “by hand”
Creation/Read

Creation code

```cpp
markovChain* c4 = new markovChain( "Xborne", NULL, 0, "rw1d", true );
markovChain* c5 = new markovChain( "PSI", NULL, 0, "rw1d", true );
markovChain* c6 = new markovChain( "Ers", NULL, 0, "rw1d", false );
```

Virtual chains:

⇒ formats PSI1/MARCA, ERS, Xborne/C, Xborne/Rii

Concrete chains:

⇒ formats ERS, MARCA
Creating chains

Interface read

/**
 * @brief Constructor for Markov chains from files in various formats.
 * In the abstract form: just stores the name(s) of the files that define the mode. In the non-abstract (concrete) form: the chain is instantiated in the memory with a concrete transition structure.
 * @param format the format or language in which the model is specified
 * @param param[] is the list of parameters
 * @param nbreParam the size of param
 * @param modelName name of the model, usually the prefix of various files
 * @param isAbstract specifies if the chain is abstract or not
 * @return Markov Chain
 */
markovChain(string format, string param[], int nbreParam, string modelName, bool isAbstract);
Creating chains

Creation/Use

Direct use of one of the pre-programmed classes. Currently implemented: small part of the Markov Zoo

- felsenstein81, a model for BioInformatics
- homogeneous1DRandomWalk
- homogeneousMultiDRandomWalk
- homogeneous1DBirthDeath
- ...

The MARMOTE Software, Labex UCN seminar, Sophia Antipolis
3 September 2015
Creating chains

Creation/Make

Typically in two steps:

1. create a transitionStructure object
2. create the Markov chain from this object

Example of creation code

```cpp
sparseMatrix* gen = makeGenerator( sp, N, E1, E2, M, nu );
markovChain* myMC = new markovChain( gen );
```
Creation with a state space

Objects of type marmoteSet are useful to create the generator:

```cpp
sparseMatrix* makeGenerator(layeredStateSpace* sp, ... ) {
    sparseMatrix* gen = new sparseMatrix( sp->cardinal() );

    int stateBuffer[5];
    sp->firstState(stateBuffer);
    int idx = 0;
    do {
        ...
        // destination state stored in nextBuffer
        nextBuffer[0] = MIN( stateBuffer[0] + 1, someBound );
        ...
        gen->addToEntry( idx, sp->index(nextBuffer), someRate );
        gen->addToEntry( idx, idx, -someRate );
        ...

        sp->nextState( stateBuffer );
        idx++;
    } while (!sp->isZero(stateBuffer));
}
```
5
Computing with Markov Chains
Available Solution Methods for Markov Chain

**Monte Carlo Simulation (forward)**

- virtual simulationResult* simulateChain(double, bool, bool, bool, bool, bool);
- virtual simulationResult* simulateChainDT(int, bool, bool, bool);
- virtual simulationResult* simulateChainCT(double, bool, bool, bool, bool);
- simulationResult* simulatePSI(int, bool, bool, bool);

**Exact sampling from the stationary distribution (backwards)**

- simulationResult* stationaryDistributionSample (int nbSamples);
Solution methods, ctd

**Computation of the stationary distribution**

```cpp
virtual Distribution* stationaryDistribution(bool);
virtual Distribution* stationaryDistributionCT(bool);
virtual Distribution* stationaryDistributionDT(bool);
Distribution* stationaryDistributionGhtLD();
Distribution* stationaryDistributionSOR();
```

**Entry point**

```cpp
Distribution* stationaryDistribution_iterative(
    string method,
    int tmax,
    double precision,
    string initDistribType,
    discreteDistribution* initDistrib,
    bool progress);
```
Hitting times

Distribution * hittingTimeDistribution (int iState, bool *hitSetIndicator);
int * simulateHittingTime (int iState, bool *hittingSet, int nbSamples, int tMax);
double* averageHittingTime (bool *hitSetIndicator);
double* averageHittingTimeDT (bool *hitSetIndicator);
double* averageHittingTimeDT_iterative (bool *hitSetIndicator);
Simulations

Simulating uses a variety of parameters and provides a variety of results → specific object simulationResult

**Attributes**

timeType _type;
int _stateSpaceSize;
int _trajectorySize;
bool _hasDistrib;
bool _hasTrajectory;
DiscreteDistribution* _distrib;
double* _dates;
double* _increments;
int* _states;

**Methods**
simulationResult(int size, timeType t, bool stats);
simulationResult(string format, string modelName, bool stats);
Utilities for MarkovChain

Utilities

virtual MarkovChain* copy();
virtual MarkovChain* uniformize();
virtual MarkovChain* embed();
void write( string format );
Other

Direct solution methods for specific chains:

- **homogeneous1DRandomWalk**

  ```c
  DiscreteDistribution* transientDistribution(int t, int nMax);
  GeometricDistribution* stationaryDistribution();
  DiscreteDistribution* stationaryDistribution(int nMax);
  simulationResult* simulateChain(long int tMax, bool stat, bool traj, bool trace);
  ```

- **felsenstein81**

  ```c
  DiscreteDistribution* transientDistribution(double);
  DiscreteDistribution* stationaryDistribution();
  simulationResult* simulateChain(double, bool stat, bool traj, bool incr, bool trace);
  ```
Further code examples

Comparison of computations for the stationary distribution

// specific methods for F81
felsenstein81* c1 = new felsenstein81(...);
Distribution* d1 = c1->stationaryDistribution();
Distribution* d2 = c1->simulateChain(...) ->getDistribution();

// generic methods for MCs
MarkovChain* c2 = static_cast<MarkovChain*>(c1);
Distribution* d3 = c2->stationaryDistribution_GaussSeidel();
Distribution* d4 = c2->stationaryDistribution_PowerMethod();
Distribution* d5 = c2->stationaryDistribution_Xborne_LowBound();
Distribution* d6 = c2->replicateSamples_Psi3(...);
Distribution* d7 = c2->simulateChain(...) ->getDistribution();
Distribution* d8 = c2->simulateChain2(...) ->getDistribution();

// comparison
cout << "Distance L1(d1,d2) = " << d1->distanceL1(d2) << endl;
Conclusion
As a conclusion

An ongoing development

- interfaces with R
- addition of solution methods
- more interface formats
- ...

An open development

- in need of users/testers
- in need of contributors