DK-Method for JUCE

A JUCE module for analog emulation using the Nodal DK-Method

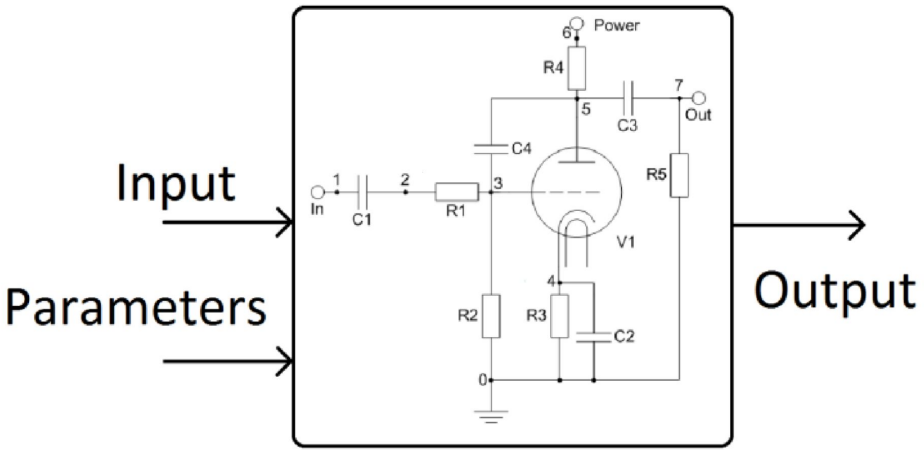
Jaromir Macak
João Rossi Filho

Virtual analog modeling

- Get the analog feel of sound into your DAW plug-in
- Various methods with different properties:
  - Level of emulation precision
  - CPU demands
  - Effort to build an emulation model
- Some of the methods:
  - Black box modeling - neural networks
  - White box modeling - Wave digital filters
  - White box modeling - DK method

White box modeling

- Electronic circuit modeled as a closed system with
  - Inputs
  - Outputs
  - Parameters
- Big advantage is to build the model in a systematic way from a given circuit



DK method

- Provides a systematic way of designing the emulation model directly from circuit schematic
- Could be used both for linear and nonlinear circuits
- Model consists of several matrices describing the whole circuit formed in state-space form:
  - $x[n] = Ax[n-1] + Bu[n] + Ci(v[n])$
  - $y[n] = Dx[n-1] + Eu[n] + Fi(v[n])$
  - $v[n] = Gx[n-1] + Hu[n] + Ki(v[n])$
- with
  - A, B, C, D, E, F, G, H, K matrices describing circuit component connections
  - i(v) nonlinear component model
  - x as a state variable, u as an input vector, y as an output vector, i and v nodes currents and voltages

Nodal DK method

- Introduced to find the state-space matrices using the component models and their connections via nodes [Holters2011]
- Example of components:
  - Linear: Resistor, Capacitor, Inductor
  - Non-linear: Triode, Diode, Transistor...
  - Input and Output
- Each component contains
  - nodes
  - value
  - model (for non-linear components)

Nodal DK method II

- Components can be inserted into a component vector - model
- This model can be processed to get state-space matrices
- Linear circuits solved directly by calculating
  - $x[n] = Ax[n-1] + Bu[n]$
  - $y[n] = Dx[n-1] + Eu[n]$

  in real time for each audio sample at time n

Nodal DK method for nonlinear circuits

- Full nonlinear state-space model must be used
- Nonlinear circuit model has to be defined for all nonlinear components
- Requires numerical algorithm to solve the nonlinearity
  - Newton-Raphson method
  - Damped Newton-Raphson method
- Real time performance can be greatly improved by using look-up tables and precomputed solution of the nonlinearity

JUCE/C++ Implementation

- Written as a JUCE module, for easy integration with audio plugins
  - GPLv3
  - Compatible with dsp::ProcessorChain
  - namespace dkm
- Uses Armadillo for matrix operations [Sanderson2018]
  - Allows fast Intel MKL, OpenBLAS and Accelerate implementations of BLAS and LAPACK
  - MatLab-like syntax
- Based on MatLab implementation " Nodal DK Framework" [Macak2016]
  - Newton-Raphson nonlinear solver

Classes

- Component - Component.h
  - Used to represent all components in the circuit
  - Holds the component's nodes position and value
  - For nonlinear cases, has a std::function for the component model
- ComponentFactory - ComponentFactory.h
  - A reliable way to instantiate components
- Model - Model.h
  - Assemble circuit and calculate matrixes
  - Nonlinear system solution for audio processing
  - Precision scaling

Classes: Component

- Holds all information regarding the component, to be used by the model.

- Shoudn't be handled by hand (ComponentFactory)

```
struct Component
{
    const Identifier type;
    const String name;
    const double value;

    const Array<int> nodes;

    const int numOfPorts;
    const bool isNonlinear;

    const NonlinearFunc model = nullptr;

    JUCE_LEAK_DETECTOR (Component)
};
```

Classes: ComponentFactory

- Allows reliable Component instantiation

- Concentrate knowledge about components, making it easy for users to create new ones.

```
struct ComponentFactory
{
    ComponentFactory() = delete;
    ~ComponentFactory() = delete;

    static Component makeInput (const String& name,
                                int nodePositive, int nodeNegative);

    static Component makeOutput (const String& name,
                                 int nodePositive, int nodeNegative);

    static Component makePotentiometer (double maxResistance, const String& name,
                                        int nodePositive, int nodeNegative, int nodeTap);

    static Component makeOPA (double gain, const String& name,
                              int nodeInPositive, int nodeInNegative, int nodeOut);

    static Component makeResistor (double resistance, const String& name,
                                   int nodePositive, int nodeNegative);

    static Component makeCapacitor (double capacitance, const String& name,
                                    int nodePositive, int nodeNegative);

    static Component makeInductor (double inductance, const String& name,
                                   int nodePositive, int nodeNegative);

    static Component makeDiode (const String& name,
                                int nodeCathode, int nodeAnode);

    static Component makeTransistor (const String& name,
                                     int nodeBase, int nodeColector, int nodeEmiter);

    static Component makeTriode (const String& name,
                                 int nodeGrid, int nodePlate, int nodeCathode);

    JUCE_DECLARE_NON_COPYABLE (ComponentFactory)
};
```

Classes: Model

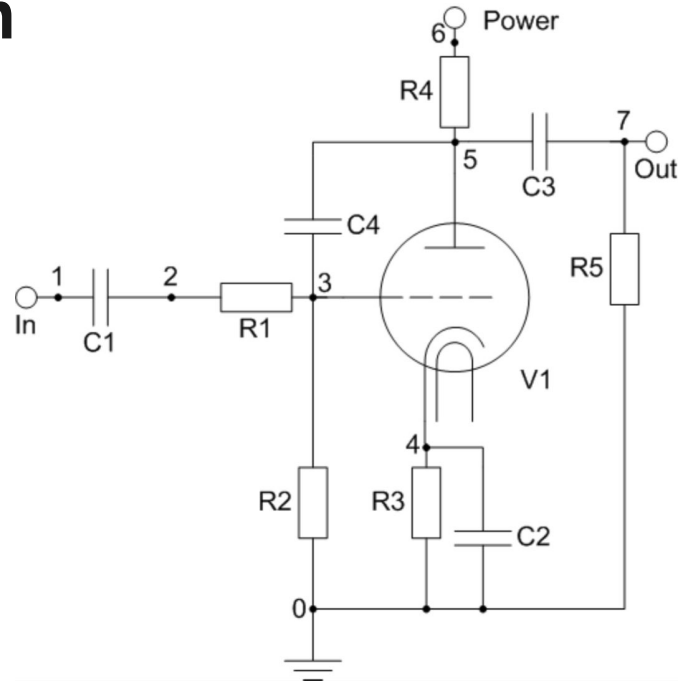
- Use addComponent method to assemble the circuit.

- Audio processing methods compatible with dsp::ProcessorBase

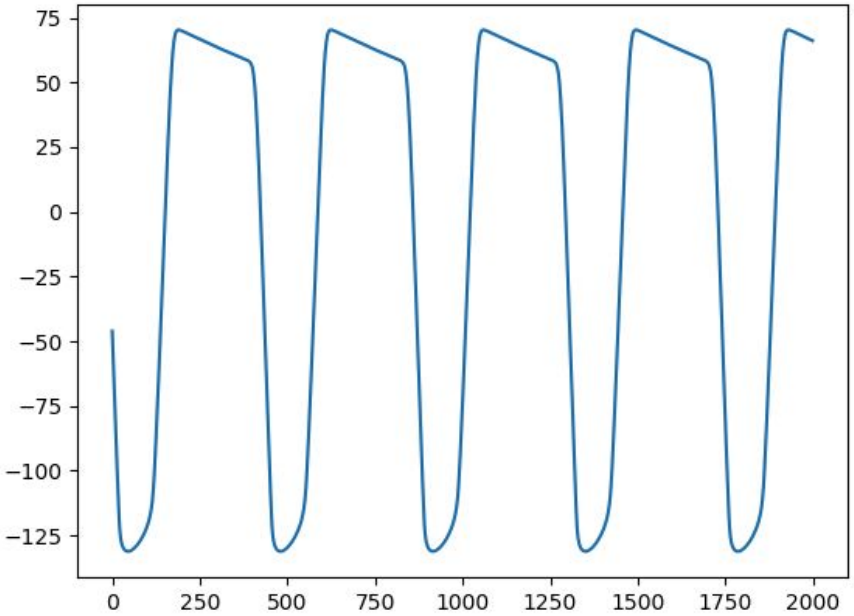
```
class Model
{
public:
    Model (double initialSampleRate = 48000.0);

    /** Add a component to the current Net List

        For safely adding components, use the ComponentFactory.
        You must have added the complete circuit before calling any
        of the methods below.
    */
    void addComponent (Component component);

    /** Process model.

        The AudioBlock should contain <numOfInputs> channels. In the
        same order the input components were added.

        For now, only double precision is supported, when a more robust
        non-linear solver is implemented float should be considered.
    */
    void process (const dsp::ProcessContextReplacing<double>&);

    /** Prepera model for playing

        Update matrixes for new sample rate and calculate system steady state.
        Should be called before starting a new audio stream.
    */
    void prepare (const dsp::ProcessSpec&);

    /** Reset model internal state.

        Sets the state matrix X and non-linear voltages vector V to zeros.
    */
    void reset();
```

Demo: Triode Amp Simulation

```
// Resistors
model.addComponent (dkm::ComponentFactory::makeResistor (68000.0,   "R1", 2, 3));
model.addComponent (dkm::ComponentFactory::makeResistor (1000000.0, "R2", 3, 0));
model.addComponent (dkm::ComponentFactory::makeResistor (2700.0,    "R3", 4, 0));
model.addComponent (dkm::ComponentFactory::makeResistor (100000.0,  "R4", 5, 6));
model.addComponent (dkm::ComponentFactory::makeResistor (1000000.0, "R5", 7, 0));

// Capacitors
model.addComponent (dkm::ComponentFactory::makeCapacitor (0.00000002,     "C1", 1, 2));
model.addComponent (dkm::ComponentFactory::makeCapacitor (0.00002,        "C2", 4, 0));
model.addComponent (dkm::ComponentFactory::makeCapacitor (0.00000002,     "C3", 5, 7));
model.addComponent (dkm::ComponentFactory::makeCapacitor (0.000000000002, "C4", 3, 5));

// Triode
model.addComponent (dkm::ComponentFactory::makeTriode ("T1", 3, 5, 4));

// Input and Output
model.addComponent (dkm::ComponentFactory::makeInput ("In", 1, 0));
model.addComponent (dkm::ComponentFactory::makeInput ("Vps", 6, 0));
model.addComponent (dkm::ComponentFactory::makeOutput ("Out", 7, 0));
```



Demo: Triode Amp Simulation - Result



Demo: Triode Preamp Plugin

Live Demo!

References

- [Holters2011] M. Holters, U. Zölzer – Physical Modelling of a Wah-wah Effect Pedal as a Case Study for Application of the Nodal DK Method to Circuits with Variable Parts, Proc. of the 14th International Conference on Digital Audio Effects (DAFx-11), Paris, France, September 19-23, 2011
- [Macak2016] https://github.com/jardamacak/NodalDKFramework
- [Sanderson2018]C. Sanderson, R. Curtin. - A User-Friendly Hybrid Sparse Matrix Class in C++. International Congress on Mathematical Software, 2018.

Thank you for the attention!

https://github.com/joaorossi/dkmethod

João Rossi Fillho: joaorossifilho@gmail.com

Jaromir Macak: jarda.macak@seznam.cz