# About

The MathAct is Scala toolset for modeling, simulating and analyzing of dynamic systems. It similar to [Mathlab Simulink](https://en.wikipedia.org/wiki/Simulink) but uses the Scala instead of Mathlab language and based on the messages propagation instead of state iteration. You can use it as additional toolset in your research/testing/playing projects.

Currently it is more like research project and it contain small number of tool. However, I will continue to work on (since it used in other projects) and encourage you to contribute in.

# Getting Started

You need to have installed [JDK 1.8+](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html) and [SBT 0.13+](http://www.scala-sbt.org/download.html), also some Scala IDE will be helpful.

Download or clone this repository somewhere on your local machine, for example with Git: **git clone https://github.com/AlexCAB/MathAct.git**

Navigate to MathAct folder: **cd MathAct**

Now you can run examples with a command: **sbt mathact\_examples/run examples.Simple**

In addition, you can import SBT project to your favorite IDE and run **examples.Simple** object from there.

# Demo

Running of couple of sketches:

<video>

# Project Structure

Project comprises of next subprojects:

* Core – contains common and service definitions.
* Tools – set of tools, which can be used to compose the sketch.
* Examples – contains a set of examples that demonstrate a using of the MathAct toolset.

# Defining of Sketch

Typical developing process of whatever complicated enough include modeling stage. I.e. when you already have an idea about how to solve some issue but you not sure enough in this to start of developing of concrete solution, so you include a modeling step for making an inexpensive evaluation of your idea. Building of mathematical model of your idea and then run its simulation is good option for this.

To define and simulate an model in MathAct you can create a sketch. In fact sketch is just a Scala class that extends \*Workbench tool class. Inside this you able to place any Scala definitions.

Next simple example of the sketch (MyFirstSketch.scala <добавить ссылку>):

package manual.sketches

import mathact.tools.workbenches.SimpleWorkbench

class MyFirstSketch extends SimpleWorkbench {

//TODO Add my definitions here

}

Most likely, you will need more than one sketch in the project so it will useful to make a compact list of them. For this, you can create a list object extended from the Sketches class and register your sketches there. This object will have a **main()** method so you can start MathAct application here. After application starts, you will see list of all your sketches (unless you not set auto-start for some sketch).

Sext example of list of sketches:

package manual.sketches

import mathact.tools.Sketches

object MySketches extends Sketches{

SketchOf[MyFirstSketch](

name = "My first sketch",

description = "The first sketch that I define but not implemented.",

logger = true,

visualisation = true)

SketchOf[MySecondSketch](

name = "My second sketch",

description = "The second sketch that I wrote.")

SketchOf[MyThirdSketch](

name = "My third sketch",

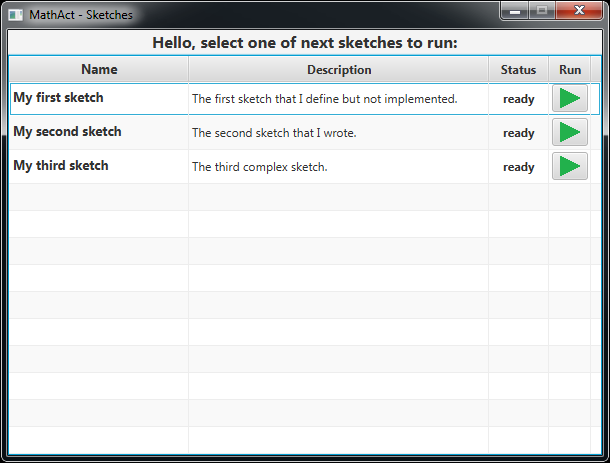
description = "The third complex sketch.",

logger = true)

//TODO Add more

}

After launch of MySketches object you will see something like next:



Now you can hit the RUN button (green triangle) and actually launch the selected sketch. After sketch starts, you will see sketch control panel (read about UI below in this document):



Nothing more interesting happens, since the sketch not contain any block.

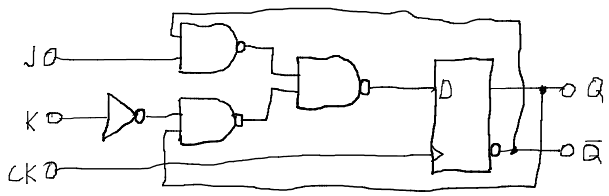
# Sketch (model) Structure

Similarly, to the Mathlab Simulink, you can compose your model from the blocks.

Unlike in the Simulink, you define a model in text format using Scala language. I am personally prefer the text format since I think it give more flexibility (especially when Scala used :) ), but also plan to implement graphical editor same like in Simulink.

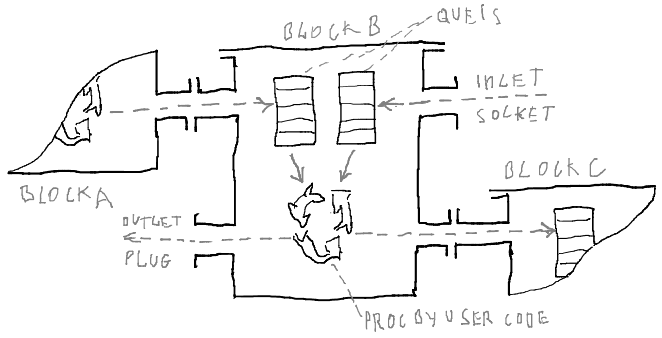
Huge deferens from the Simulink is in using of [reactive streams](https://en.wikipedia.org/wiki/Reactive_Streams) to implement blocks interactions. I.e. the model can be represented as a graph where nodes (blocks) is the message processors and edges that connect the blocks, is the paths of message distribution (message streams).

Each block have one or more connections points (inlets and outlets) that can be connected. Figuratively you can think about blocks like about electronic components (or [pneumatic automation](https://www.youtube.com/watch?v=IqIqpTwKMPI) components or hydraulic components) which could be composed to form some device and think about the messages like about electricity (or air or liquid) that circulate between components.



Internally the connections are implemented as messages queues, that reactive and buffering messages from other blocks till it will processed. During processing of income messages (received by inlet) block may send some outcome messages (via outlet) to other connected blocks.

Block structure may be visualized like next:



The code of the block B from image above may look like next:

package manual.sketches

import mathact.core.bricks.plumbing.wiring.fun.FunWiring

import mathact.tools.EmptyBlock

import mathact.tools.workbenches.SimpleWorkbench

class MySecondSketch extends SimpleWorkbench {

//Blocks

class BlockB extends EmptyBlock with FunWiring {

//Connection points

val in1 = In[Double]

val in2 = In[String]

val out1 = Out[Double]

val out2 = Out[String]

//Wiring

in1.map(\_.toString) >> out2

in1.filter(\_ != 0) >> out1

in2.map(s ⇒ "Received: " + s) >> out2

}

//Connecting

//TODO

}

Read more about composing of blocks below in this document.

For connecting of the blocks used DSL similar to [akka-streams DSL](http://doc.akka.io/docs/akka/2.4.14/scala/stream/stream-composition.html), next example show connecting of A and B and C blocks:

package manual.sketches

import mathact.core.bricks.linking.{LinkIn, LinkOut, LinkThrough}

import mathact.core.bricks.plumbing.wiring.fun.FunWiring

import mathact.core.bricks.plumbing.wiring.obj.{ObjOnStart, ObjWiring}

import mathact.tools.EmptyBlock

import mathact.tools.workbenches.SimpleWorkbench

import scala.concurrent.Future

class MyThirdSketch extends SimpleWorkbench {

//Blocks

object BlockA extends EmptyBlock with ObjWiring with ObjOnStart with LinkOut[Double]{

//Parameters

name = "BlockA"

//Wiring

private val gen = new Outflow[Double] {

def start(): Unit = Future{

(0 to 10).foreach{ i ⇒

pour(i)

Thread.sleep(500)}}}

protected def onStart(): Unit = gen.start()

//Connection points

val out = Outlet(gen) }

object BlockB extends EmptyBlock with FunWiring with LinkThrough[Double, String]{

//Parameters

name = "BlockB"

//Connection points

val in = In[Double]

val out = Out[String]

val out2 = Out[Double]

//Wiring

in.map(s ⇒ "Received: " + s) >> out

in.filter(\_ % 2 == 0) >> out2 }

object BlockC extends EmptyBlock with FunWiring with LinkIn[String]{

//Parameters

name = "BlockC"

//Connection points

val in = In[String]

val in2 = In[Double]

//Wiring

in.foreach(v ⇒ logger.info("IN: " + v))

in2.foreach(v ⇒ logger.info("IN2: " + v)) }

object BlockD extends EmptyBlock with FunWiring with LinkIn[String]{

//Parameters

name = "BlockD"

//Connection points

val in = In[String]

//Wiring

in.foreach(v ⇒ logger.info("IN: " + v))}

//Connecting

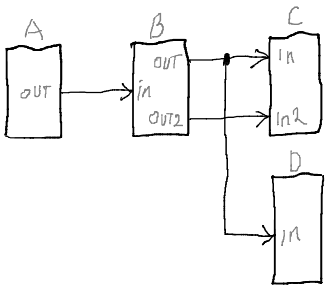
BlockA ~> BlockB ~> BlockC //<-- Shortcut DSL

BlockB.out2 ~> BlockC.in2 //<-- Standard way of connecting

BlockB ~> BlockD

}

This sketch can be represented as next graph:



Each outlet can be connected to the several inlets (for example ‘out’ block B connected to ‘in’s of block C and D), in this case, outlet will broadcast messages to all connected inlets. Also each inlet can be connected to the several outlets, in this case messages from all connected outlets will enqueued to inlet queue in the received order.

To make a connecting of blocks easier and compact, you can use shortcut-linking DSL that allow you to connect blocks like next:

BlockA ~> BlockB ~> BlockC

Instead of using standard connation DSL:

BlockA.out ~> BlockB.in

BlockB.out ~> BlockC.in

In additional I encourage you to check and play with examples<ссылка на гитхаб примеры>.

# Simulation Approach

Same as in Simulink you can simulate discrete, continues and mixed models. However, because of parallel computing and message propagation nature of the reactive streams, the simulation approach is different.

Most of the modeled systems have a state, which chances during the time. Therefore, to be precise, models need to reflect the state too. To achieve maximum performance of parallel computing the state (i.e. data) need to be distributed in some way between the computation units (i.e. blocks in our case) to minimize synchronization costs. Which is actually hard, especially when you using classical (i.e. blocking) synchronization approach, but I think using of the reactive approach make this task something simpler.

The main idea is in the taking apart of the model state and keeping of its parts inside the blocks. Moreover, in using of messages exchange to keep the spitted model state consist (i.e. to synchronize parts of model). You can imagine that as the parts of state evaluates independently from each other, and in the same time, the changes of the parts of state are propagated in between by messages exchange (sending of message by block with changed state and reaction on the changes by connected block).

…Общая идея в том что состояние модели хранится распределённо внутри блоков, а изменяя состояния модели распространяются посредством передачи сообщений между блоками (синхронизация состояния). Сами же блоки вычисляются асинхронно и независимо, так как это и должно происходить в модели реактивных потоков.

...Альтернативно вы можете думать что состояние хранится в сообщениях и блоки только выполняют трансформирование этого состояния (как это принят например в [Erlang language](https://en.wikipedia.org/wiki/Erlang_(programming_language))) и строить модель в соответствии с этим принципом. Но лично я считаю такой подход менее удобным.

## Discrete models

...Это просто симулировать дискретные модели, так как каждое измене состояния может быть представлено посылкой одного сообщением, которое распространяет это измените.

...Следующий пример демонстрирует симуляцию работы модели [D trigger](https://en.wikipedia.org/wiki/Flip-flop_(electronics)#Gated_D_latch) :

< рисунок скетча Д тригера >

...Скетч для модели:

<код скетча Д тригера>

## Continuous models

...Симулирование непрерывных моделей несколько сложнее, так как передача сообщений по свой сути дискретна. Потому непрерывные модели должны быть дискретезироаны in some way.

...Эффективный способ сделать это состоит в том чтобы отправлять сообщение каждый раз когда состояние изменяется на некоторую дельту.

...Следующий пример показывает симуляцию работы модели усилится with [negative feedback](https://en.wikipedia.org/wiki/Negative_feedback):

<анимация работы усилителя: дельта задана 0.1 на прямой вход усилителя поступает сигнал с шагом 0.025 и коэфициент усиления 2 таким образом с выходы усилителя сообщение посылется через раз или когда из-за обратной связи выход падает больше чем на 0.1 (собственно показать этот момент, кода посылаются два сообщения)>

...Скетч для модели:

<код скетча усилителя, должн быть регулятор обратной связи>

## Model time

...Для большинства реальных систем время имеет значения соответственно оно должно быть отражено в модели. В данном случае это сделать несколько сложнее так как состояние распределено и вычисляется асинхронно.

...Для симуляции времени используется концепция «виртуального времени», т.е. внутреннего времени модели на зависящего от реального времени её вычисления.

...Для реализации этой концепции каждое сообщение должно иметь дополнительное поле со значением тикущего времени и в модели должен существовать один блок являщийся источником виртуального времени т.е. собсвенно устанавливающий значение поля времени.

...Другие блоки могут изменять поле времени, например симулируя некорою задержку распространия сообщения. Но в целом при таком подходе все вычиления модели происходят мгновенно и одновременно (с точки зрения виртуального времени модели).

...Пример реализации виртуального времени описан в следующей главе.

# Build and Run Simple Model

...В качестве примера будем использовать модель простой системы подержания заданного уровня воды в ёмкости:

<схематическое изображение модели, ёмкость с водой, кран наполнения , кран слова, дачик уровня и ПИД регулятор>

...Для симуляции этой модели мы будем использовать следующие блоки:

TimeLoop

...пошаговое построени и запуск простой модели (пид регулятора)

<video>

# User Interface

The application have simple UI for a selecting of the sketch to be launch and for managing of the launched sketch.

<скрин списка скетчей>

<описание элементов>

<скрин сетча>

<описание элементов>

# Common Architecture

The toolset built on top of the AKKA actor’s library with using of ScalaFX for the UI. But for using of application you not need to have deal with actors (unless you want to) since app provide a simple OOP API for creating and composing of blocks.

Here is actors/objects diagram:

<диаграма из заметок>

<описание элемнтов>

# Compose Your Own Block

...пошагово описать создание простого блока (например сложения, но такого к которому можно добавить UI) с помощью обьектного и функционального вайринга.

...расказать о функция старта и стопа

...Расказать о режымах работы входов (все сообщения, только последнее)

# Adding Block UI

...Пошаговое описание добавления UI в два мбока опасные в перд главе (с обьектным и функиональным вайрингом).

...опимать добавления UI при поможи FXML

# Future Work

...больше блоков

...динамическая компиляция

...визуализация (отображение графа более структурированым, отображение полезной иформации как например колическа чообщений в очередях, подсветка блоков в которых проихошла ошибка)

...графический редактор для построеня скетчей и кодогенерации.

...динамическое подключение и отключение, добавление и удаление блоков врвботаюжий скетч.