Chapter 1

Introduction

1.1 Solver theory – TODO

Euler

Take derivative, multiply with timestep, add to initial value, repeat until end.

Runge-Kutta methods

Take derivatives at some time points, multiply with proper coefficients, add all to intial value, repeat until end.

Stability

1.2 What is FP – TODO

Should I add a section on this to make everything understandable to all readers?

1.3 Using Haskell for numerical mathematics

Functional languages have several properties which make them suitable for the purpose of solving problems in numerical mathematics. First and foremost, Haskell, being based on λ-calculus is very close to mathematics. The useful mathematical properties here are *referential transparency*, easy *partial function application* and being a *declarative language*. Referential transparency implies that a variable only has a constant value which is the same everywhere in the program. This prevents that changing a variable might have influence on another computation as a side effect and it corresponds to mathematical notation. For instance, in an imperative programming like C you could write `i = i + 1`, which is a mathematical impossibility and therefore not allowed in Haskell. Partial function application is another very useful concept. Often in numerical mathematics, you want to create or process a function. You need a function that has another function as return value. For instance, take a function which requires two arguments. After only applying a single argument, the object returned still needs the second argument in order to compute the final value. This is exactly according to the definition of a function: An object that still needs arguments before being able to return its final value. Being a *declarative language* means that you write code that specifies what you want to accomplish, not how to get there. This concept is again borrowed from mathematics. You put in a set of function definitions and Haskell will figure out how to actually compute the value you request according to those definitions. This property of declarativity also has the result that Haskell is a very terse language whilst remaining easy to understand. Secondly, Haskell has a very strong type system. The type system has three main advantages. It becomes very easy to swap out and replace functions as long as you make sure that the types are the same. The Haskell compiler will start to assert errors immediately whenever you feed it something which does not make sense or could be ambiguous which is very useful when writing programs. By having a look at the types of a Haskell program it becomes very straightforward to see what the program does and how it works, which is very useful when attempting to understand your own or someone else's code. Lastly, a property which is often very important for numerical mathematics: Haskell is fast. According to the *Computer Language Benchmarks Game* [1], Haskell is almost on par with Java and Fortran but significantly faster than Python and Matlab (not shown), two languages which are often used for numerical mathematics nowadays. There is still a performance gap of around a factor 3 between Haskell and C (the reference), hence if speed is of the absolute highest concern C is still a valid option.

1.4 Numerical solutions of ODEs in Haskell

As mentioned before, the types in Haskell reveal lots of information about the structure and functionality of the program. The three main types constituting the numerical solver for ordinary differential equations are listed above.

Listing 1.1 Main types for the ODE solver

```
type Equation    = ODEState -> D_ODEState
type SolveMethod = TimeSettings  -> Equation -> ODEState -> ODEState
type Solver      = SolveMethod -> TimeSettings -> Equation -> ODEState -> [ODEState]
```

Equation

In essence, a differential equation is a mapping (function) from a certain state of the system to the change of this system. This is also what the type signature of Equation signifies, a mapping from an ODEState to a D_ODEState. This generic set up allows the specification of any ODE for solving. The implementation in pure Haskell of a simple ODE is given in listing 1.2, which corresponds to the equation $x' = -x$. However, this representation is not very elegant and a lot of the code is performing unboxing of the types. Using property that this equation is linear, it is possible to use an utility method which takes as input a matrix and returns the Haskell differential equation function belonging to that matrix. The same can be done for heterogeneous linear systems using a different utility function, which does not only takes a matrix as input but also a list of functions representing the heterogeneous part of the equation. The example code for this can be seen in appendix A.

Listing 1.2 Example equation for exponential decay

```
eq_exponential  :: Equation
eq_exponential  state     = [−x !! 0]
  where
    x = xs  state
```

SolveMethod

The SolveMethod performs the actual computations on what the next value of the solution should be: the integration scheme. In order to obtain this next state, the scheme needs three input values: It needs information on the timing constraints of the solution, in this case it needs the time step. Furthermore, it needs the equation itself and it requires the state of the system at t_n in order to be able to determine the state of the system at $t_{n+1} = t_n + \Delta t$.

The most straightforward integration scheme is called forward Euler, given in equation 1.1. Listing 1.3 depicts the translation of the mathematical expression 1.1 to Haskell. Even though some list operations have been inserted (**zipWith** and **map**), the structure is still recognizable. It computes the change in state, multiplies this with the time step obtained in line 6 and adds the initial state in line 4. Lastly, the integration scheme returns the new state of the equation, consisting of a list of x-values and a corresponding time value. Implementations of different solvers (eg. 4th order Runge-Kutta) can be found in appendix A.

Listing 1.3 Example code for the forward Euler integration scheme

```
1  euler  :: SolveMethod
2  euler  time  equation   initState     = ODEState newX newT
3    where
4      newX      = zipWith (+)  (xs   initState  )  dX
5      dX        = map (timestep  ∗)  ( equation   initState  )
6      newT      = ( t   initState  )  +  timestep
7      timestep  = dt  time
```

$$\vec{\mathbf{x}}(t+\Delta t) \approx \vec{\mathbf{x}}(t) + \frac{\mathrm{d}\vec{\mathbf{x}}(t)}{\mathrm{d}t}\Delta t \tag{1.1}$$

Solver

The Solver function in listing 1.4 acts as the main interface to the program. You specify a SolveMethod, the TimeSettings (containing the time step and the time at which to stop solving), the equation itself and an initial condition. The Solver will then return a list of states of the system. As is very common in functional programming, the Solver has been defined recursively. Line 4 is where the magic happens: the solution list is defined to be the initial condition, followed by the solution list with the new state (computed by the integration scheme on line 6) as initial condition. Additionally, there is a comparison in line 7 which ends the recursion whenever the time of the solution exceeds the maximum time value, set in the TimeSettings.

The solutions of a wide range of equations, both linear and non-linear, both homogeneous and heterogeneous and using the input matrix utility functions have been plot with suitable initial conditions to show their behavior in figure 1.1.

Listing 1.4 The main controlling function

```
solve  ::  Solver
solve  solvemethod time  equation   initState
  | end = []
  | otherwise =  initState   :  solve  solvemethod time  equation  newState
  where
    newState = solvemethod time  equation   initState
    end = ( t   initState  ) > (tMax time)
```

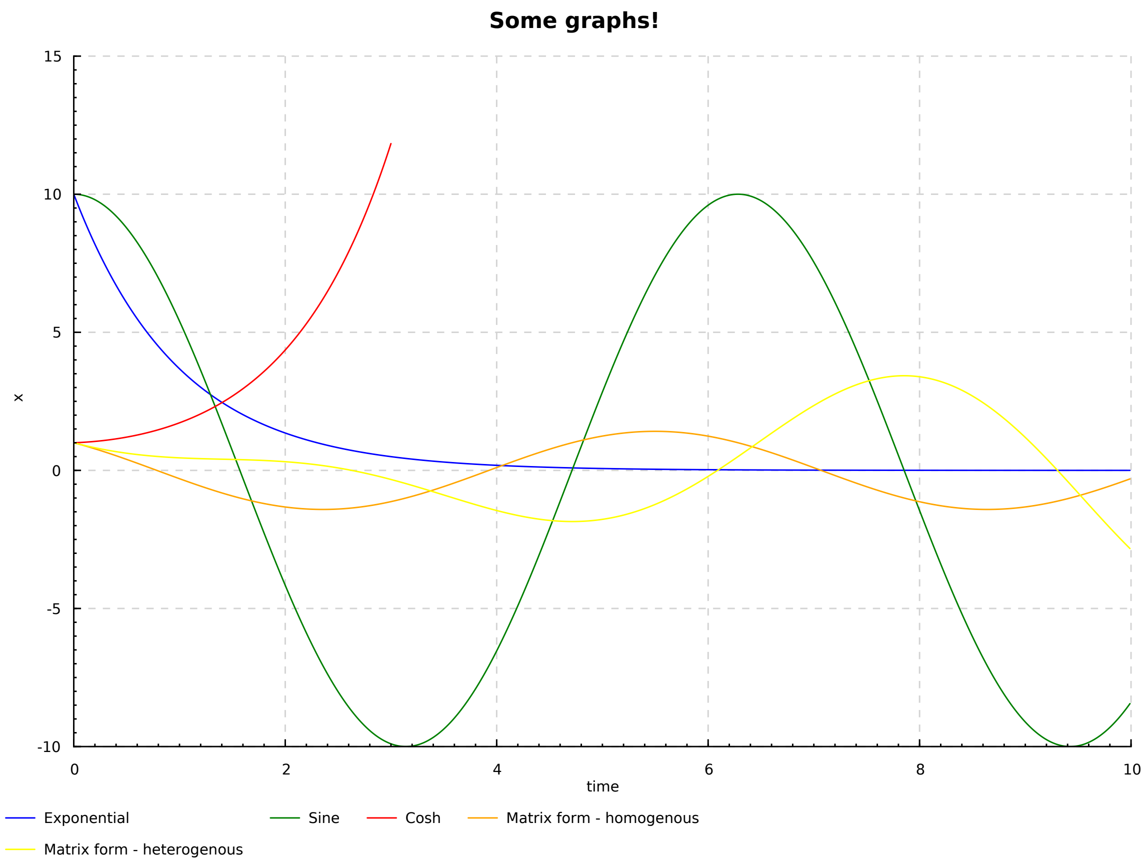

Fig. 1.1 Graphs

$$\begin{aligned}
&\text{Exponential} && x(t)' = -x(t) && (1.2)\\
&\text{Simple harmonic} && x(t)'' = -x(t) && (1.3)\\
&\text{Cosine hyperbolic} && x(t)' = \frac{\sqrt{x(t)^2 - a^2}}{a} && (1.4)\\
&\text{Simple harmonic} && \vec{\mathbf{x}}(t)' = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \vec{\mathbf{x}}(t) && (1.5)\\
&\text{Simple forced harmonic} && \vec{\mathbf{x}}(t)' = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \vec{\mathbf{x}}(t) + \begin{bmatrix} \sin(t) \\ e^{-t} \end{bmatrix} && (1.6)
\end{aligned}$$

1.5 Conversion to Mealy Machines / CλaSH – TODO

Changes to:

- Equation : numerical types (fixed point) - what operations would be supported in VHDL (hard to do nonlinear operators, sqrt(), sin(), cos(), tan()) -
- SolutionMethod - becomes transition function of the mealy machine
- Solver - keeping track of the state of the solution - responsible for loading the data from the AXI-bridge to the HPS - responsible for outputting the data to the HPS for storage/processing/streaming

1.6 FPGAs – TODO

- What are FPGAs used for?
- Why should I care about FPGAs
- What is the current workflow for programming FPGAs

1.7 Data transfer – TODO

References

[1] Computer language benchmarks game. http://benchmarksgame.alioth.debian.org/u32q/which-programs-are-fastest.html, 2015. [Accessed:].