Documentation

These notes are made using R Markdown.

Verification Study

Two remote repositories:

```
git remote set-url origin https://github.com/fireofearth/2019s-verification
git remote set-url origin https://bitbucket.org/ian_mitchell/julia-intervals
git push origin master
```

20190617

We have the software that was used for Goubault+Putot's last 3 years of research: Robust INner and Outer Approximated Reachability (RINO) here as well as the dependenciesFILIB++ for interval computations, and FADBAD

Additionally for Julia (all required components to reproduce RINO for Julia): - automatic diff https://github.com/JuliaDiff/ - intervals https://github.com/JuliaIntervals/IntervalArithmetic.jl - affine arithmetic https://github.com/JuliaIntervals/AffineArithmetic.jl - taylor models https://github.com/JuliaIntervals/TaylorModels.jl

It was used in the latest paper for HSCC 2019 "Inner and Outer Reachability for the Verification of Control Systems"; and the paper we're reading that is HSCC 2017 "Forward inner-approximated reachability of non-linear continuous systems"; and the paper CAV 2018 "Inner and Outer Approximating Flowpipes for Delay Differential Equations".

20190626

Affine arithmetic does not seem to be supported (well) in Julia. Interval arithmetic does not have modal interval extensions in Julia. Ian Mitchell (IM) asked me to check with JuliaIntervals to ask whether developers (Dr David P. Sanders, Chris Rackauckas) would be open to contributions from for libraries.

20190702

Step 2: for each subdivision

call set_initialconditions (ode_def.cpp) to set x to inputs which are intervals, xCenter which are midpoints of intervals (as floats), and J = Id. Recall x, xCenter and J are all vectors and matrices containing AAF.

Affine Arithmetic

All non-affine operations (*,/,inv,^,sin,cos) default to Chebyshev approximation.

Specification for univariate Chebyshev approximation of bounded, twice differentiable $f : \mathbb{R} \to \mathbb{R}$ and affine form $x = x_0 + \sum_i^N x_i \epsilon_i$

1. let $a = x_0 - \sum_i^N |x_i|$, and $b = x_0 + \sum_i^N |x_i|$. We require $f''(u) \neq 0$ for $u \in (a, b)$
2. let $\alpha = (f(b) - f(a))/(b - a)$ be the slope of the line $l(x)$ that interpolates the points $(a, f(a))$ and $(b, f(b))$. Then $l(x) = \alpha x + (f(a) - \alpha a)$.
3. solve for $u \in (a, b)$ such that $f'(u) = \alpha$. By Mean-value theorem u must exists.
4. $\zeta = \frac{1}{2}(f(u) + l(u)) - \alpha u$
5. $\delta = \frac{1}{2}|f(u) - l(u)|$

Specification for bivariate Chebyshev approximation of ??? $f : \mathbb{R}^2 \to \mathbb{R}$ and affine

Automatic Differentiation

The Julia package ForwardDiff uses dual numbers for automatic differentiation. The dual number $x + \epsilon 1$ can be used to simultaneously evaluate a function at x and find the derivative of the function at x. Similarly hyperdual numbers can obtain higher derivatives at x.

Dual numbers have the form $x = a + \epsilon b \in \mathbb{R}[\epsilon]$. It has two coordinates similar to complex numbers except $\epsilon^2 = 0$ (ϵ is nilpotent). It's inverse is ${}^{-1}x = \frac{1}{a} + \epsilon \frac{-b}{a^2}$. If we want to simultaneously find the derivative $f'(a)$ of a function $f : \mathbb{R} \to \mathbb{R}$ at $a \in \mathbb{R}$ and $f(a)$, we just need to pass $x = a + \epsilon 1$.

$f(x) = x + b$: $f(a + \epsilon 1) = (a + b) + \epsilon 1$

$f(x) = x^n$: $f(a+\epsilon 1) = (a+\epsilon 1)^n$: using the rule $(a+b)^n = a^n + \binom{n}{1} a^{n-1} b + \binom{n}{2} a^{n-2} b^2 + \cdots + \binom{n}{n-1} a b^{n-1} + b^n = \sum_{i=0}^{n} a^{n-i} b^i$ we obtain $a^n + \epsilon \binom{n}{1} a^{n-1} + \epsilon^2 \binom{n}{2} a^{n-2} + \epsilon^3 \binom{n}{3} a^{n-3} + \cdots + \epsilon^n$ and since $\epsilon^2 = \epsilon^3 = \epsilon^4 = 0$ we get $a^n + \epsilon n a^{n-1}$.

$f(x) = \frac{1}{x}$: from the inverse we know $f(a + \epsilon) = \frac{1}{a+\epsilon} = \frac{1}{x} + \epsilon \frac{-1}{x^2}$.

$f(x) = \sqrt{x}$: notice $(a + \epsilon b)^2 = a^2 + \epsilon 2ab = c + \epsilon d$ gives us $a = \pm\sqrt{c}$ and $b = \pm\frac{d}{2\sqrt{c}}$ provided $c > 0$ ($d \in \mathbb{R}$). Therefore by choosing $a = \sqrt{c}$ we have $\sqrt{x + \epsilon} = \sqrt{x} + \epsilon \frac{1}{2\sqrt{x}}$.

Similarly, $f(x) = \sqrt[n]{x}$: notice $(a + \epsilon b)^n = a^n + \epsilon n a^{n-1} b = c + \epsilon d$ gives us $a := \sqrt[n]{c}$ and $b = \frac{d}{nc^{(n-1)/n}}$ provided $c > 0$ ($d \in \mathbb{R}$). Therefore $\sqrt[n]{x + \epsilon} = \sqrt[n]{x} + \frac{1}{nx^{(n-1)/n}}$.

https://blog.demofox.org/2014/12/30/dual-numbers-automatic-differentiation https://en.wikipedia.org/wiki/Dual_number

Original implementation

Good news! We have the software that was used for Goubault+Putot's last 3 years of research: Robust INner and Outer Approximated Reachability (RINO) https://github.com/cosynus-lix/RINO as well as the dependencies FILIB++ http://www2.math.uni-wuppertal.de/wrswt/software/filib.html for interval computations, and FADBAD http://www.fadbad.com/fadbad.html for automatic diff.

Gotchas

Unused for loop: the below for loop and function `init_subdiv()` is never used so we can remove it entirely. To simplify we may ignore the variables `inputs_save`, `center_inputs`.

```
for (current_subdiv=1; current_subdiv <= nb_subdiv_init; current_subdiv++) {
    if (nb_subdiv_init > 1) init_subdiv(current_subdiv, inputs_save, 0);
    // ...
}
```

Redundant variables: in all simulations, `jacdim` and `sysdim` contain identical values. We may replace these variables with `sysdim`.

Modal Arithmetic

Modal arithmetic is only used at the last step of the procedure at every time step, specifically `HybridStep_ode::TM_evalandprint_solutionstep()`. The `InnerOuter()` function obtains inner approximations from the taylor models built in `HybridStep_ode::TM_build()`. All other calculations are computed using FADBAD++ using affine forms as values.

Affine Arithmetic

Affine arithmetic is supported by the `aaflib` library.

Automatic Differentiation

RINO requires FADBAD++ for automatic differentiation. This library is much more powerful than the JuliaDiff package in that it allows for

-
- evaluating derivatives
- T is used for taylor expansion.
- F is used for forward differentiation.

Given ODE function $f : R \to R$ where $\vec{x}' = f(\vec{x})$, forward differentiation is called on f to obtain f' and then the taylor approximation $p(f')$ of f' is obtained. The Julia implementation requires this sequence of operations to occur.

Classes

`OdeVar` represents the taylor coefficients of the taylor model in `TM_Jac`. Automatic differentiation using FADBAD is called on the member variables of `OdeVar`.

Members:

x (vector<T>): independent variables
xp (vector<T>): dependent variables

`Ode` represents the taylor coefficients of the taylor model in `TM_val`.

Members:

x (vector<T>): independent variables
xp (vector<T>): dependent variables

Functions:

`Ode::Ode()` constructor
`Ode::Ode(OdeFunc f)` constructor that sets `x`, `xp` by evaluating them using the function `f` passed to it.

TM_val

Is a vector Taylor Model

TODO: is it necessary that all points are AAF?

`TM_val` members:

ode_x (Ode): taylor coefficients from 0 to k-1 of TM
ode_g (Ode): the k-th taylor coefficient of TM

`TM_val` functions:

`TM_val::build()`

`TM_val::eval()`

HybridStep_ode

`HybridStep_ode` members:

bf (OdeFunc):
TMcenter (TM_val):
TMJac (TM_Jac):
order (int): order of the taylor model

tn (double): the time at the n-th iteration
tau (double): the time step

`HybridStep_ode` functions:

`HybridStep_ode::HybridStep_ode()` constructor
`HybridStep_ode::TM_build()` build Taylor Model using ODE function `bf` by calling `TMcenter.build()` and `TMJack.build()`.
`HybridStep_ode::TM_eval()` calls `TMcenter.eval()` and `TMJac.eval()`.
`HybridStep_ode::init_nextstep()`

Steps

Step 4: create an ODE object `HybridStep_ode` using `HybridStep_ode::init_ode()`.

Step 5.1: build Taylor Model using `HybridStep_ode::TM_build()`.

Testing

I'm writing a test suite in RINO to test their modification of the aaflib affine arithmetic library

```
g++ -ggdb -frounding-math -DMAXORDER=40 -I. -I${HOME}/lib/filib-3.0.2/include \
    -I/usr/include -I$(pwd)/aaflib-0.1 -fpermissive -std=c++11 -c testsuite.cpp

g++ -L/usr/lib -L$(pwd)/aaflib-0.1 -L${HOME}/lib/filib-3.0.2/lib -o testsuite \
    testsuite.o -laaf -lprim -lgsl -llapack -lblas -lcblas -lstdc++ \
    -lboost_unit_test_framework

./testsuite --log_level=test_suite

# in the case that the testsuite does not detect libaaf.so we need to add the
# library path for this shared resource.
LD_LIBRARY_PATH=/usr/lib:/home/fireofearth/res/mitchell-ian/programs/rino-hscc17/aaflib-0.1/
export LD_LIBRARY_PATH
```

Julia RINO implementation

Documentation

Components

Required functionality:

- Deriving taylor approximations from functions, derivatives, gradients, and jacobians.
- Automatic differentiation to compute derivatives, gradients and jacobians of functions at affine points.
- Converting affine forms to intervals.
- Modal interval arithmetic: in particular we want to evaluate the mean-value extension (multiplication, addition, and matrix vector multiplication).

A critique of existing Julia components:

Real (Base package) https://github.com/JuliaLang/julia/blob/master/base/Base.jl

The abstract type Real is a supertype of floats, integers and unsigned integers and Real has accommodating Base methods:

`iszero`, `isone`, `one`, `zero` for add./mult. identities.
`promotion_rule`
`convert`

`rounding`, `setrounding`
`isapprox`

Number (Base package) https://github.com/JuliaLang/julia/blob/master/base/number.jl

- Complex https://github.com/JuliaLang/julia/blob/master/base/complex.jl
- math https://github.com/JuliaLang/julia/blob/master/base/math.jl
- utilities https://github.com/JuliaLang/julia/blob/master/base/number.jl
- type promotion https://github.com/JuliaLang/julia/blob/master/base/promotion.jl

Any subtype of Number must support

sign() convert(::Type{T}, x::T) convert(::Type{T}, x::Number)

Outstanding:

- Complete list of methods using Real, Number
- Where is Real, Number declared in Base package?
- What are the differences between Real, Number: if a function has arguments in Real type, then can that same function evaluate with arguments in Number type?

automatic forward differentiation https://github.com/JuliaDiff/ForwardDiff.jl
documentation: http://www.juliadiff.org/ForwardDiff.jl/stable/

Points:

- `ForwardDiff.derivative()` (as well as `.gradient()`, `jacobian()`) does not return a function, but rather produces a method that can be used to evaluate the derivative. This method only accepts subtypes of Real.
- There is an outstanding concern that library ForwardDiff is unusable for functions that have inputs and outputs that are not type Real:
  - It is possible to compute the $df(x)/dx$ where x is affine, provided that the type Affine is made a subset of type Real. However, this leads to some concerns (see section on Affine Arithmetic).
- RINO obtains taylor series from derivatives (and gradients, jacobians). As of now there is no clear way to do this in Julia.
  - It seems possible to change the source code of ForwardDiff to allow for arbitrary inputs.

intervals https://github.com/JuliaIntervals/IntervalArithmetic.jl
documentation: https://juliaintervals.github.io/IntervalArithmetic.jl/stable/

Points:

- Originally I considered implementing modal intervals (using the quantified modal interval interpretation $([a, b], Q) \in \mathbb{IR} \times \{\forall, \exists\}$) on top of IntervalArithmetic library. However IntervalArithmetic only admits outer approximations for intervals. Outer approximations are are done automatically within each interval operation.
- There are two possible approaches:
  - Modify IntervalArithmetic directly so that it admits and inner approximations of intervals. It may be possible to utilize exists macros that set approximations in package.
  - Create a completely new IntervalArithmetic library. In this case we may only need to implement modal interval multiplication and addition (matrix vector multiplication can invoke interval multiplication).

affine https://github.com/JuliaIntervals/AffineArithmetic.jl
documentation: none

Points:

- This library is currently a thin implementation (only implemented +, -, *, /, ==, zero, one, range) so I am implementing an affine arithmetic library from scratch. This library is being actively developed. Around mid-July, the went from one to four files.

- Any Julia library for affines must correspond with C++'s `aaflib` library. In particular it should use the same approximation Chebyshev approximation method for non-affine operations, as well as all elementary functions and binary arithmetic operations.
- Any implementation of affine forms must be able to save deviation coefficients in a compact manner. Otherwise, vectors for coefficients will increase linearly by the number of (non-affine) operations (something approaching $O(n^2)$ space considering affine forms proliferate).

taylor series https://github.com/JuliaIntervals/TaylorSeries.jl
documentation: http://www.juliadiff.org/TaylorSeries.jl/stable/

Points:

- Interestingly, TaylorSeries does not provide a method that takes a function as input to evaluate a taylor approximation. Instead it provides types Taylor1 and TaylorN that once passed as variables to a function generates a taylor approximation as output. This infers that we can obtain a taylor approximation of a derivative of f when passing f and an instance of Taylor1 to `ForwardDiff.derivative()`.
- TaylorSeries supports arbitrary types as coefficients (limitations?).
- Taylor1, TaylorN are subtypes of Number, and are iterable https://github.com/JuliaDiff/TaylorSeries.jl/blob/master/src/auxiliary.jl

taylor models https://github.com/JuliaIntervals/TaylorModels.jl
documentation: https://juliaintervals.github.io/TaylorModels.jl/stable/

Points:

- Is built on top of the TaylorSeries module to provide rigorous bounds for the enclosure representing the error term of a taylor approximation.
- It may be possible to use TaylorModels to obtain an outer approximation of the Jacobian at each time step, I am hesitant on making this library a component of the Julia RINO port until I understand more on how inner approximations are computed in RINO.
- This is the leading package in Julia for validated numerics, and hence I'm interested in benchmarking RINO with TaylorModels.

Internal Julia modules:

ModalInterval
AffineArithmetic
AffineTaylorModel
ODECommon
ODEIntegration

Development Environment

OS: Linux, GNU/Linux
System package manager: pacman
Distribution Environment: Anaconda 3
Python: >3.5
Python package manager: conda, pip
Julia version: >1.x
Qt version (for plots): 5

Modal interval arithmetic

Julia IntervalArithmetic documentation.

Outstanding: operations should return inner approximations of improper integrals and outer approximations of proper integrals.

Affine arithmetic

Non-affine operations can be approximated in $O(1)$ runtime by Chebyshev approximation.

Complete discourse on interval and affine arithmetic operations are discussed in the paper Self-Validated Numerical Methods and Applications (1997) by Jorge Stolfi, Luiz Henrique De Figueiredo.

Implemented functionality:

zero, one, iszero, isone, convert, isapprox, promote_rule, getindex, length, repr, size, firstindex, lastindex, <, <=, >, >=, ==, +, -, *, /, inv, ^, sin, cos, Affine Interval, inf, sup, rad, getMax, getMin, getAbsMax, getAbsMin, compact

Affine forms must evaluate under the following functions:

`Base.convert` -
`Base.one` - return a multiplicative identity for x: a value such that one(x)*x == x*one(x) == x. Alternatively one(T) can take a type T, in which case one returns a multiplicative identity for any x of type T.
`Base.isone` - return true if x == one(x) if x is an array, this checks whether x is an identity matrix.
`Base.zero` -
`Base.iszero`
`Base.promotion_rule` -
`Base.isapprox` -
`Base.rtoldefault` - returns the default relative tolerance value base on number type.

Automatic differentiation

Cassette was originally designed for better language support for automatic differentiation. AutoGrad is a port of Python autograd. I'm avoiding this package due to lack of documentation.

Both AutoDiffSource and ReverseDiffSource are deprecated and limited to Julia version 0.5 (and the are hence in the JuliaAttic list)

By the process of elimination ForwardDiff is the one I will be using.

We wish to do the following using forward differentiation:

- generate taylor approximations of derivatives.
- compute affine forms under derivatives of functions.

Outstanding: ForwardDiff does not give good expressions for derivatives when the original expression contains inverse (i.e. `1.0 /x`, `inv(x)`, or `x^(-1)`). For example:

I expected the expression `f(x) = x^(-2)` to have the derivative `df(x) /dx = -2x^(-3)`. However ForwardDiff gives an expressions more complicated than: `2*inv(x) * -abs2(inv(x))`. I was unable to find the exact expression despite using the DiffLogger module I created (in `module`).

This makes it pretty challenging to test ForwardDiff on affine forms. Since affine operations (*, /) only computes approximations, the expression of the formula matters and a more concise formula gives approximations that can differ by `10E-1`. Since this does not aversely affect the goal of porting RINO to Julia I'm ignoring this for now.

Procedure

Discretize $t_0 < t_1 < \cdots < t_j < t_{j+1} = t_j + \tau < \cdots < t_n$
Outer approximation $[\boldsymbol{x}^{(0)}]$ (given)
Outer approximation $[J^{(0)}] := I$
Outer approximate center of $[\boldsymbol{x}^{(0)}]$ as $[\tilde{\boldsymbol{x}}^{(0)}] := \mathrm{mid}\big[\boldsymbol{x}^{(0)}\big]$
Inner approximation $]\boldsymbol{x}^{(0)}[= [\boldsymbol{x}^{(0)}]$

For each $j = 0, \ldots, n$

1. get priori enclosure of solutions and Jacobians of solutions over $t \in [t_j, t_{j+1}]$
   - $\left[\boldsymbol{r}^{(j+1)}\right]$ of ; $\boldsymbol{x}(t, t_j, \left[\boldsymbol{x}^{(j)}\right])$
   - $\left[\tilde{\boldsymbol{r}}^{(j+1)}\right]$ of ; $\boldsymbol{x}(t, t_j, \left[\tilde{\boldsymbol{x}}^{(j)}\right])$
   - $\left[R^{(j+1)}\right]$ of ; $J(t, t_j, \left[\boldsymbol{x}^{(j)}\right]) = \mathrm{Jac}_{\boldsymbol{x}^{(j)}} \boldsymbol{x}(t, t_j, \left[\boldsymbol{x}^{(j)}\right])$
2. get Taylor approximations $[\boldsymbol{x}](t_{j+1}, t_j, \left[\boldsymbol{x}^{(j)}\right])$, $[\boldsymbol{x}](t_{j+1}, t_j, \left[\tilde{\boldsymbol{x}}^{(j)}\right])$ and $[J](t_{j+1}, t_j, \left[\boldsymbol{x}^{(j)}\right])$
3. deduce an inner approximation $]\boldsymbol{x}[(t_{j+1}, t_j, \left[\boldsymbol{x}^{(j)}\right])$
4. set $]\boldsymbol{x}^{(j+1)}[$, $\left[\boldsymbol{x}^{(j+1)}\right]$, $\left[\tilde{\boldsymbol{x}}^{(j+1)}\right]$ and $\left[J^{(j+1)}\right]$