Documentation

Types

`Main.Branch` — *Type*.

A series of `Solutions`. One can `push`, `unshift`, `pop`, and `shift` new/old `Solutions` to/from it.

`Main.Solution` — *Type*.

A `Solution` comprises a single solution vector and its parent

`Main.ContinuationMethod` — *Type*.

Continuation method implementations extend `ContinuationMethod` and the corresponding `show` and `step`. Responsible for changing the project itself

`Main.PC` — *Type*.

An "implementation" of a `ContinuationMethod` using a predictor-corrector-method with step-size adaption. Look up the theoretical documentation for more details.

`Main.SystemCore` — *Type*.

An abstract type that requires to implement the basic functions needed for path following: Homotopy `H`, its jacobian `J` (and `show` to display a GUI)

`Main.Galerkin` — *Type*.

"Implementation" of `SystemCore`; Only requires the ode `f` and its derivative `f'`, the respective homotopy and its jacobian are derived. Again, look up the theoretical documentation for more detail.

`Main.Project` — *Type*.

Contains all found `Branch`es

`Main.Session` — *Type*.

Comprises everything needed for path following: `Project`, `SystemCore`, `ContinuationMethod`, and `Visualization`. Sessions can be managed via `create`, `save`, and `load`.

Utility Functions

Differentiation

mbNewton.forwardDifference — *Function.*

```
forwardDifference(homotopy, v, epsilon)
forwardDifference(homotopy, epsilon)
```

Returns the difference quotient in `v` or an approximate jacobian using this method.

mbNewton.centralDifference — *Function.*

```
centralDifference(homotopy, v, epsilon)
centralDifference(homotopy, epsilon)
```

Same as `forwardDifference`, but with the symmetric difference quotient.

mbNewton.broyden — *Function.*

```
broyden(homotopy, jacobian)
```

Creates an approximate jacobian based on broyden-updates and maintainance of an internal state. Consider this for performance improvements.

all jacobians can be used in

mbNewton.newton — *Function.*

```
newton(homotopy, jacobian, v , pred[, init, callback, useOpt])
```

Newton's method. For example: `v1 = newton(H, J, v0, predEps(0.001))`.

Galerkin

Main.findCycle — *Function.*

```
findCycle(H, t0, y0, transientIterations, transientStepSize,
    steadyStateIterations, steadyStateStepSize)
```

Returns all points from the `transientIterations`-times numerical integration of `H` with fixed step-size `transientStepSize`, and all points from the `steadyStateIterations`-times integration with fixed step-size `steadyStateStepSize` (assuming it hit the steady state part), and the periodicity of the found cycle.

Main.findCyclePoincare — *Function.*

```
findCyclePoincare(F, y[, plane, clusterRating, nIntersections,
    maxCycles, sampleSize, transientIterations, transientStepSize,
    steadyStateStepSize])
```

extracts a single cycle of the steady state of ode `F` using poincare cuts through the `plane`.

`Main.prepareCycle` — *Function.*

```
prepareCycle(data, h, P[, fac])
```

cut single cycle of length `P*fac` from data, resample, shift s.t. `X(0) 0`, Fourier transform.

`mbInterpolate.interpolateLanczos` — *Function.*

```
interpolateLanczos(V, a::Integer)
```

simple periodic (!) Lanczos interpolation

`mbInterpolate.interpolateTrigonometric` — *Function.*

```
interpolateTrigonometric(a , a, b)
```

Returns trigonometric polynomial. Use with 2a,-2b and divide by 2m+1 to use with rfft coefficients.

GUI

`Main.mkControlGrid` — *Function.*

```
mkControlGrid(D, C)
```

Creates a grid of controls with labels, handlers and encapsulated storage (Dictionary `D`) `c` in `C` is Tuple `(name::String, ::DataType, init, v...)`

`Main.ctrl` — *Function.*

```
ctrl(D, x)
```

Dictionary `D`, Tuple `x=(name::String, ::DataType, init, v...)`; used by mkControlGrid

Session Control

`Main.create` — *Function.*

```
create(homotopy, jacobian, projection)
```

`Main.save` — *Function.*

```
save(filename, session[, overwrite])
```

`Main.load` — *Function.*

```
load(filename, homotopy, jacobian, projection)
```

ODE

`mbRK.rk` — *Function.*

`rk(butcherTableau)`

returns a runge-kutta method using the respective tableau:

`function(f, t0, y0, h, pred[, init, callback])`

e.g. `rk1`, or `rk4`. Examines the ode `f` starting from `t0`, `y0` with fixed stepsize `h` until `pred` evalutes to `false`.

Index