Parallel-DDE Documentation

Easy to use delay differential equation solver library written in C++

`version 1.0`

Author: Dániel Nagy

Supervisor: Dr. Ferenc Hegedűs

September 3, 2020

`Budapest University of Technology`

Contents

1 Capabilities **3**

2 Installation **3**
2.1 Download libraries . . . 3
2.2 Usage on Windows, Visual Studio . . . 3

3 Delay Differential Equations **4**

4 The RK4 Solver **4**
4.1 Parallelism . . . 4
4.2 Example 1 – Basics . . . 5
4.3 Example 2 – Discontinuities . . . 7
4.4 Example 3 – Parallel solution of many similar DDE . . . 9
4.5 RK4 Solver object . . . 12
4.5.1 Template parameters . . . 12
4.5.2 Constructor . . . 12
4.5.3 Integrate . . . 12
4.5.4 Time range, step size and steps . . . 13
4.5.5 Delays . . . 13
4.5.6 Discontinuities . . . 13
4.5.7 Dense variables . . . 13
4.5.8 Mesh . . . 14
4.5.9 Initial conditions . . . 15
4.5.10 Parameters . . . 15
4.5.11 Events . . . 15
4.5.12 Save functions . . . 16

1 Capabilities

The solver is capable of handling,

- Arbitrary number of variables
- Arbitrary number of constant delays
- Arbitrary number of events
- Arbitrary number of parameters
- Discontinuities in the initial function and in the solution

Available methods are,

1. Traditional 4. order Runge-Kutta, with 3. order Hermite Interpolation and constant stepsize.
2. 5. order Dormand Prince, with 4. order continuous extension and adaptive stepsize.

`ParallelDDE` uses Single Instruction Multiply Data (SIMD) parallelism, based on the VCL library. `ParallelDDE` makes use of out-of-order execution explained in Section 4.1, by unrolling the inner loops in parameter sweeps, this may increase the speed by 3×.

2 Installation

2.1 Download libraries

Download `VCL` library from here. The solver is originally implemented using `VCL version2` however it should work with `VCL version1` as well.
Download the latest version of `ParallelDDE` from here.

2.2 Usage on Windows, Visual Studio

1. Open a new command line C++ project
2. Open the Property Pages tab at `Project > example Properties`
3. Select the active Configuration at the top
4. Navigate to `C/C++ > General > Additional Include Directories > Edit` than add both the `VCL` and `ParallelDDE` library to the Additional Include Directories.
5. Navigate to `C/C++ > Language > C++ Language Standard` and select `ISO C++17 Standard`
6. Navigate to `C/C++ > Code Generation > Enable Enhanced Instruction Set` and select `Advenced Vector Extensions` or `Advenced Vector Extensions 2`
7. It is recommended turn on optimizations by the compiler so go to `C/C++ > Optimization` and

(a) set `Optimization` to `Maximum Optimization (Favor Speed)`

(b) set `Inline Function Expansion` to `Any Suitable`

(c) set `Whole Program Optimization` to `Yes`

Now everything is set so you can use the `ParallelDDE` library.

3 Delay Differential Equations

The general form of a delay differential equations:

$$x'(t) = f\big(t, x(t), x(t-\tau_1), y(t-\tau_2)\ldots, y(t-\tau_n)\big) \tag{1}$$

Where n is the number of delays and $x(t < t_0) = \eta(t)$ where $\eta(t)$ is the initial function. In this case τ_i = constant. Here x may be a vector with an arbitrary number of dimensions.

4 The RK4 Solver

This method is the easier one. You should do the following steps when setting up the solver object

1. Initialize the solver object
2. Setup integration range and number of steps and initial points
3. Add delays to the solver object
4. Initialize the variables with dense output points. There are methods to fill it up from point to point as well as to calculate these points using an initial function and its derivative.
5. Calculate the integration mesh
6. Setup initial conditions
7. Setup parameters
8. Integrate the function
9. Save the output or the end values

There may be some variations in these steps, but it is recommended to keep this order.

4.1 Parallelism

`ParallelDDE` uses two different methods to parallelize your code, and make it run faster.

- Explicit CPU Vectorization using `256 bit ymm` registers. This means that 4 equations with double precision can be solved in parallel, each with different parameters or initial conditions. This vectorization is based on the `VCL` library and uses variables of type `Vec4d` which packs 4 double variable in a physical vector. This type can be treated as a regular variable by the user. Basic operators are functions are defined on these types, but please see the `VCL` documentation for more.

- The solver is capable of solving $4 \times n$ equations at a time instead of 4, by unrolling the loops by n. In this documentation n is called unroll. By increasing the unroll, more equations will be solved in parallel, increasing the amount of independent instructions, which is important when writing fast codes.

4.2 Example 1 – Basics

The full code is available under `examples/RK4/example1_basics.cpp`.
In this example we are solving the following simple delay differential equation, with 1 variable, 1 delay and 1 parameter:

$$x'(t) = -p \cdot x(t-\tau) \tag{2}$$
$$x(t<0) = 1 \tag{3}$$

Here the delay is $\tau = 1$ and the parameter is $p = 1$. We also specify that $x(0) = 1$ however it is possible to choose another value for $x(0)$ but in this case the solution is not continuous in $t = 0$. From (3) the initial derivative can be also determined:

$$x'(t<0) = 0 \tag{4}$$

We would like to integrate this system on the $t = [0, 10]$ interval. Now we have everything to build our first system. First include the header file.

```
#include "source/ParallelDDE_RK4.h"
```

Then we define the DDE function as follows:

```
void analitic1(Vec4d* dx, double t, Vec4d* x, Vec4d* delay, Vec4d* p)
{
  dx[0] = -p[0] * delay[0];
}
```

By the definition of the DDE function `Vec4d` acts like a `double` variable, operations like addition, subtraction, multiplication and division are defined, other functions such as sin, cos, log, exp are also possible, see the `VCL` documentation for more info. The library takes care of filling up the `Vec4d` vectors with the right variables, thus we don't have to deal with the underlying SIMD parallelism. Now we define the initial function and its derivative:

```
double x0(double t, int)
{
  return 1;
}

double dx0(double t, int)
{
  return 0;
}
```

For now we don't care about the second argument of this function, but it will be important in the next section. We have everything for the preamble, now we can move on to the main function. First we initialize the solver object with template parameters.

```
const unsigned Vars = 1;
const unsigned Delays = 1;
const unsigned DenseVars = 1;
const unsigned ParametersPerEquation = 1;
ParallelDDE_RK4<Vars, Delays,DenseVars,ParametersPerEquation> solver;
```

Than we set up the integration range which should be $t = [0, 10]$, and we choose an arbitrary number for the number of steps.

```
solver.setRange(0.0, 10.0);
solver.setNrOfSteps(123);
```

This way the step size will be determined automatically. Now we add the delay to the solver object.

```
solver.addDelay(0, 1.0, 0);
```

The first 0 means that the index of this delay is 0. The middle value is delay itself, here it is $\tau = 1$. The last 0 is the corresponding dense history from which the dense variable could be calculated. Here we only have 1 delay and 1 variable with dense history, so we the delays are ready. Now we add the initial function. The values of the independent variables start from $t_{start} - \tau = -1$, thus the argument of `calculateInitialTvals` is -1. The arguments of `calculateInitialXvals` are `x0` which is the initial function, `dx0` which is the derivative of the initial function, the first 0 is the variable index which means that this `x0` and `dx0` specifies the history 0. variable which happens to be the 0. dense variable.

```
solver.calculateInitialTvals(-1);
solver.calculateInitialXvals(x0, dx0, 0, 0);
```

We already have everything to calculate the integration mesh, we should also print the mesh, to check it.

```
solver.calculateIntegrationMesh();
solver.printMesh();
```

Now we set the 0. variable's initial condition to 1.0.

```
solver.setX0(1.0, 0);
```

and set the 0. parameter's value to 1.0

```
solver.setParameters(1.0, 0);
```

Now the solver object is ready. We should call the `integrate` method on the `analitic1` function.

```
solver.integrate(analitic1);
```

Finally we save the values to a `txt` file.

```
solver.save("example1_results.txt");
```

Running the program produces the following output on the console:

```
Mesh Length = 5
1        Type: 1
2        Type: 1
3        Type: 1
4        Type: 1
10       Type: 1
```

These are the mesh points, now matter how you choose the number of steps, the solver will always step exactly on those points. The last point $t = 10$ is also included in the mesh. The following image shows the results:

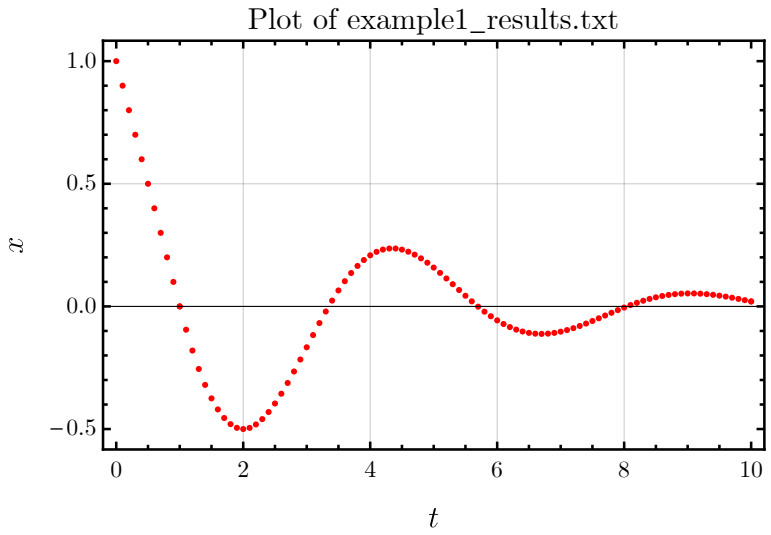

Figure 1: Solution of (2).

4.3 Example 2 – Discontinuities

The full code is available under `examples/RK4/example2_discontinuities.cpp`.
In this example we are solving the delay differential equation from before, however the initial function has some discontinuities.

$$x'(t) = -p \cdot x(t-\tau) \tag{5}$$

$$x(t \leq 0) = \begin{cases} 0 & \text{if } t \leq -2 \\ 1 & \text{if } -2 < t \leq -1 \\ 0 & \text{if } -1 < t \leq 0 \end{cases} \tag{6}$$

Here $\tau = 3$ and $p = 1$, $x(0) = 0$ is also specified. The initial derivative is $x'(t) = 0$. The integration interval is $t = [0, 25]$. We include the header and define the DDE function just like before.

```
#include "source/ParallelDDE_RK4.h"

void analitic2(Vec4d* xd, double t, Vec4d* x, Vec4d* delay, Vec4d* p)
{
  xd[0] = -p[0] * delay[0];
}
```

It's a little bit tricky, to define the initial function, since it has some discontinuities. Now we use the second argument of the initial function, which specifies the direction of the limit in the discontinuous points. We have to tell the solver that $\lim_{t\to -1^-} = 0$ however $\lim_{t\to -1^+} = 1$. If `t=-2.0` and `dir=-1` the function return `1` when `t=-2.0` and `dir=1` the function returns `1`. The derivative of the initial function may be define easily as `return 0`, because it is only for interpolation.

```
double x0(double t, int dir)
{
    if (t < -2.0 || (t == -2.0 && dir == -1)) return 0;
    if (t < -1.0 || (t == -1.0 && dir == -1)) return 1;
    return 0;
```

```
}

double dx0(double t, int)
{
    return 0;
}
```

Than we initialize the solver, integration range, and the delay.

```
//initialize solver
const unsigned Vars = 1;
const unsigned Delays = 1;
const unsigned DenseVars = 1;
const unsigned Parameters = 1;
ParallelDDE_RK4<Vars, Delays, DenseVars, Parameters> solver;

//basic setup
solver.setRange(0.0, 25.0);
solver.setNrOfSteps(235);

//delay
solver.addDelay(0, 3, 0);
```

Now we add the list of discontinuous points to the solver. All of the discontinuities are type C0, thus we initialize the type C1 fields with `0` and `NULL`. See Section 4.5.6. for further information about adding discontinuities to the solver.

```
const unsigned nrOfC0disc = 2;
double* C0disc = new double[nrOfC0disc] { -2.0, -1.0 };
solver.setInitialDisc(NULL, 0, C0disc, nrOfC0disc);
```

Finally we calculate the mesh, set the parameter and initial values, than we integrate the function `analitic2` and save the points.

```
//mesh
solver.calculateIntegrationMesh();
solver.printMesh();

//parameter
solver.setParameters(1.0, 0);

//initial function
solver.calculateInitialTvals(-3.0);
solver.calculateInitialXvals(x0, dx0, 0, 0);

//initial conditions
solver.setX0(0.0, 0);

//integrate
solver.integrate(analitic2);

//save
solver.save("example2_results.txt");
```

When running the code, you should see the following output on the console:

```
Mesh Length = 15
1       Type: 2
2       Type: 2
```

```
3         Type: 1
4         Type: 1
5         Type: 1
6         Type: 1
7         Type: 1
8         Type: 1
9         Type: 1
10        Type: 1
11        Type: 1
12        Type: 1
13        Type: 1
14        Type: 1
25        Type: 1
```

Now we see that, the points $t = 1$ and $t = 2$ are type 2 mesh points, which means that the derivative of the trajectory in those points are expected to be non-continuous. The solver will save two separate points near these locations, thus the Hermite interpolation won't fail. If we look at the file `example2_results.txt`:

```
t                       x                          dx
0.8510638297872342      0                          -0
0.9574468085106385      0                          -0
0.9999999999            0                          -0
1.0000000001            -1.666666666666667e-10     -1
1.106382978823404       -0.1063829788900709        -1
1.212765957546809       -0.2127659576134752        -1
```

we see that at $t = 1$ a jump in the derivative really happens.

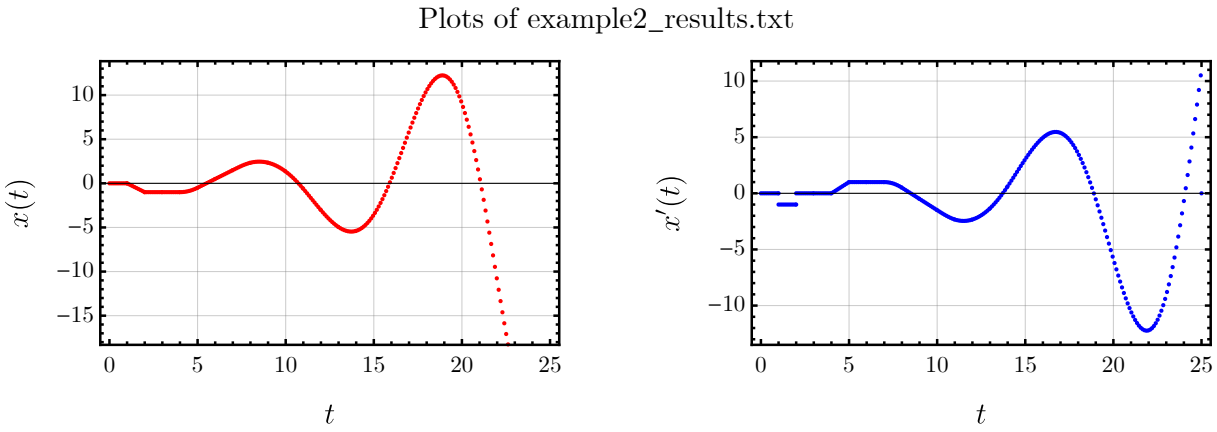

Figure 2: The plots show the solution of (5).

4.4 Example 3 – Parallel solution of many similar DDE

The full code is available under `examples/RK4/example3_parametersweep.cpp`.
We are going to solve the delayed case of the logistic equation, with a given initial function:

$$\begin{aligned} x'(t) &= x(t) \cdot (p - x(t-1)) \\ x(t \leq 0) &= 1.5 - \cos(t) \\ p &= [0, 4] \end{aligned} \tag{7}$$

We are interested in the end values $x(t = 10)$ as a function of the parameter p. We choose $N = 18144$ parameters on a linear scale between 0 and 4. Then define the initial functions and the DDE just as before:

```
void logistic1(Vec4d* xd, double t, Vec4d* x, Vec4d* delay, Vec4d* p)
{
    xd[0] = x[0] * (p[0] - delay[0]);
}

double x0(double t, int)
{
    return 1.5 - cos(t);
}

double dx0(double t, int)
{
    return sin(t);
}
```

Now we have more parameters, and `ParallelDDE` by default solves 4 equations at a time, exploiting SIMD parallelism. However we can also unroll the inner loops, and exploit the out-of-order execution of the CPU. Now we define the solver object as follows:

```
const unsigned Vars = 1;
const unsigned Delays = 1;
const unsigned DenseVars = 1;
const unsigned Parameters = 1;
const unsigned Steps = 1000;
const unsigned InitialPoints = 100;
const unsigned Unroll = 1;
ParallelDDE_RK4<Vars, Delays, DenseVars, Parameters, 0, Unroll> solver;
```

We must define the parameters for each equation in advance, we may use the function `linspace` defined in `ParallelDDE_Common.cpp`.

```
const unsigned nrOfEquations = 18144;
double* parameterList = linspace(0, 4, nrOfEquations);
```

We define the system as usual, however now the `integrate` method is placed in a loop, with the `setParameters` method. The integration time is measured using the `seconds()` function defined in `ParallelDDE_Common.cpp`. An output file stream is also defined to save the end values.

```
std::ofstream ofs("endvals.txt");
ofs << std::setprecision(16);
double iStart = seconds();
for (size_t i = 0; i < nrOfEquations; i += Unroll * vecSize)
{
    //parameter
    solver.setParameters(parameterList + i, 0);

    //integrate
    solver.integrate(logistic1);

    double* endVals = solver.getEndValues();

    for (size_t j = 0; j < Unroll * vecSize; j++)
    {
        ofs << parameterList[i + j] << "\t" << endVals[j] << std::endl;
```

```
    }

    delete endVals;
}
double iElapsed = seconds() - iStart;
```

Notice that the loop counter `i` is incremented by `Unroll*vecSize`, that is because `Unroll*vecSize` DDEs are solved in parallel. For reference we save the whole dense output of the `Unroll*vecSize -1`.th variable. Figure 3 shows the last time series as well as the end values.

(Left) Plot of last time series where $p = 4$ (Right) Plot of end values

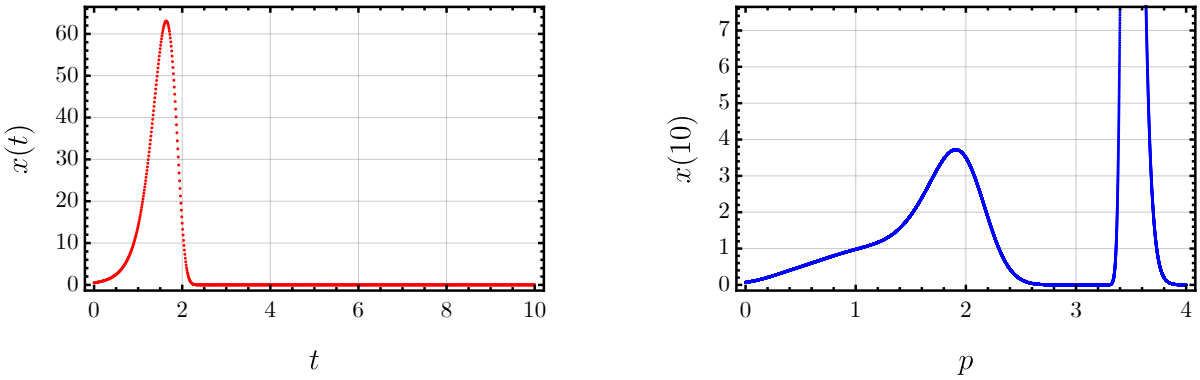

Figure 3: Solution and end values of (7)

4.5 RK4 Solver object

4.5.1 Template parameters

The solver object uses the following template parameters.

```
<unsigned int nrOfVars, unsigned int nrOfDelays = 0,
unsigned int nrOfDenseVars = 0, unsigned int nrOfParameters = 0,
unsigned int nrOfEvents = 0, unsigned int nrOfUnroll = 1>
```

- `nrOfVars`: Number of dependent variables in the DDE.
- `nrOfDelays`: Number of delays in the DDE, each delay belongs to variable and has a predefined value.
- `nrOfDenseVars`: Number of variables where a dense output should be saved it should always be $1 \leq$ nrOfDenseVars $\leq$ nrOfVars.
- `nrOfParameters`: Number of parameters *in each equation.*
- `nrOfEvents`: Number of pre defined events, where intervention in the solver may happen.
- `nrOfUnroll`: Degree of loop unrolling in the solver. When unroll $= n$ $4n$ equations are solved simultaneously. By small systems, with $2-4$ variables unroll should be between $4-32$, for larger system unroll should be $1-4$. When the system has a significant number of events it is recommended to set it to 1. You should always do some test, with just a few runs to determine the best configuration. Keep in my that unroll increases the number of independent operations thus the performance my improve due to the out-of-order execution however it also increases cache use, so you cannot increase it indefinitely.

4.5.2 Constructor

The construction doesn't have any argument, by definition you should only provide the values of the template parameters.

```
ParallelDDE_RK4<unsigned int nrOfVars, unsigned int nrOfDelays = 0,
    unsigned int nrOfDenseVars = 0, unsigned int nrOfParameters = 0,
    unsigned int nrOfEvents = 0, unsigned int nrOfUnroll = 1>();
```

For example if you would like to define a solver object with 3 variable, 3 delay, 2 variable with dense history, 2 parameters in every equation and unroll 2:

```
ParallelDDE_RK4<3,3,2,2,0,2> solver;
```

4.5.3 Integrate

```
void integrate(void f(Vec4d* dx, double t, Vec4d* x, Vec4d* xDelay,
    Vec4d* p))
```

Integrates the function f. Keep in mind that everything should be already defined in the solver object, when calling the integrate method. The arguments of the f function should be:

- `*dx`: array of x', the left hand side of the DDE.
- `t`: independent variable.

- `*x`: array of dependent variables x on the right hand side of DDE.
- `*xDelay`: array of delayed values, the array index corresponds to the one given with the `void addDelay(unsigned delayId, double t0, unsigned denseVarId)` function.
- `*p`: array of parameters

4.5.4 Time range, step size and steps

`void setRange(double tStart, double tEnd)`
Sets the start and the end value of the independent integration variable t.

`void setStepSize(double dt)`
Sets the step size to the give value, and calculates the necessary step count from this information.

`void setNrOfSteps(unsigned int nrOfSteps)`
Sets the number of steps to the give value, and calculates the necessary step size from this information.
Keep in mind that the `setRange` method must be called before `setStepSize` or `setNrOfSteps`.

4.5.5 Delays

`void addDelay(unsigned delayId, double t0, unsigned denseVarId)`
Adds a delay with index `delayId` to the dense history with index `denseVarId`, where $\tau =$ `t0`

4.5.6 Discontinuities

The initial function may contain 2 kinds of discontinuities. A discontinuity is `C0` discontinuous if the initial function is *non-continuous*. A discontinuity is `C1` discontinuous if the initial function is *continuous* but its derivative is *non-continuous*.

`void setInitialDisc(double* C1disc, unsigned nrOfC1, double* C0disc, unsigned nrOfC0)`
Sets the C1 discontinuous points given by `*C1disc` and the same for C0. `*C1disc` should be an array with the location of the C1 discontinuities the length of the array is `nrOfC1`. Same for C0.

4.5.7 Dense variables

Each dense variable has a history consisting of points, which must be known by the solver object before integration. The points can be added directly, or the functions can be added and discretized by the solver. These functions should be called when the mesh and the delays are already known.

`void setNrOfInitialPoints(unsigned nrOfInitialPoints)`
Sets the number of initial points. The number should be big enough so that the interpolation does not cause any additional error. If the initial functions are polynoms with degree 3 or less, than this number might be small.

`void setInitialTvals(double* tVals)`
Copies the `tVals` array to the solver object. Be aware that this method does the allocation of the dense history memory. It is recommended to use `calculateInitialTvals` instead.

`void setInitialXvals(double* xVals, double* dxVals, unsigned varId, unsigned denseVarId)`
Copies the `xVals` ($x(t < 0)$) and `dxVals` ($x'(t < 0)$) arrays into memory. The `varId` is the index of the variable which history is given by `xVals` and `dxVals`; `denseVarId` is the corresponding dense variable index. For example if we give the history of the `x[2]` variable to the dense output with index `[1]` we should call it with `setInitialXvals(xVals, dxVals, 2, 1)`. It is recommended to use `calculateInitialXvals` instead

`void calculateInitialTvals(double t0max)`
Calculates the initial values of the independent variable t given that the discontinuous points are already known. `t0max` is the starting value of the initial function, the name is a reminder that it should be at least the value of the largest delay. Usually `t0max` = `tStart` - τ_{max}

`void calculateInitialXvals(double x0(double t, int dir), double dx0(double t, int dir), unsigned varId, unsigned denseVarId)`
Calculates the initial values of the dependent variable x given by `x0` with index `varId` and its derivative x' given by `dx0`. It is assumed that the location of the discontinuous points are already known.

`void printInitialPoints(int precision = 6)`
Prints all the initial points with a given precision.

4.5.8 Mesh

The mesh contains the points where the integrator should step (type = 1), as well as the points where the solution is non-continuous thus two derivatives must be saved (type = 2).

`void setMesh(double* mesh, int* meshType, unsigned meshLen)`
Sets up the mesh with user given values. `meshType` is 1 when the integratior should simply step on that point and `meshType` is 2 when a double point saving is necessary. `meshLen` gives the size of the arrays. It is recommended to use `calculateIntegrationMesh()` method instead to avoid errors.

`void calculateIntegrationMesh()`
Calculates the integration mesh given that the solver object already knows the delays and discontinuous points. *Recommended method.*

`void setMeshPrecision(double prec)`
Sets the mesh precision to a given value. If not set 10^{-10} is used. *Keep in mind* that the mesh precision should be at max 10^{-14} times as big as the magnitude of the independent variable.

`void addMeshPoint(double t, int type)`
A user given mesh point is added to the mesh at time `t` with the given `type`. The solver will step on that point exactly.

`double* getMesh()`
Returns the array of mesh points.

`int* getMeshType()`
Returns the array of mesh point types.

`unsigned getMeshLen()`
Returns the size of the mesh points array.

`void printMesh(int precision = 6)`
Prints the mesh points to console with the given precision.

4.5.9 Initial conditions

The history of the dense variables is given, but the initial condition may vary from that, thus creating a discontinuity in t_0. The initial condition of all variables should be given.

`void setX0(double x0)`
Set *all* initial conditions to `x0`

`void setX0(double x0, unsigned int varId)`
Set the initial condition of the given variable to `x0`.

`void setX0(double* x0, unsigned int varId)`
Set the initial condition of the given variable to different values given by `*x0`. The size of `*x0` must be at least $4 \times$ nrOfUnroll. It may be used when different equations have different initial conditions.

4.5.10 Parameters

Each equation has a pre-defined number of parameters, which should be initialized. In a case of a parameter sweep, you should only load new parameters, using these methods.

`void setParameters(double p, unsigned int parameterId)`
Set the value of the given parameter to `p`. `parameterId` is the index of the parameter. Use it when a parameter have the same value in all equations.

`void setParameters(double* p, unsigned int parameterId)`
Set the value of the given parameter to different values given by `*p`. The size of `*p` must be at least $4 \times$ nrOfUnroll. It may be used when different equations have different parameters.

4.5.11 Events

These functions are also vectorized, which makes using those a little bit tricky. You should see the examples on them.

`void addEventLocationFunction(void (*eventLocation)(Vec4d* lst, double t, Vec4d* x, Vec4d* xDelay , Vec4d* p))`
Add the event location function to the solver. In the event location function, if the sign of `lst[i]` changes, that we have an event with index `i`.

`void addEventInterventionFunction(void (*eventIntervention)(int eventId, int eventDir, Vec4db mask, double t, Vec4d* x, Vec4d* xDelay, Vec4d* p))`
Add the event intervention function to the solver object. If an event is detected this function will be called, where `eventId` is the index of the event, `eventDir` is the direction $+1$ or -1 depending on the direction of the sign change (The new sign of `lst[eventId]` gives the direction). A `mask` is given to only change those equations where an event is happened.

`void setEventPrecision(double prec)`
Sets the precision of the event finding. This gives the precision of the independent variable t at event. The preset value is 10^{-10}.

`void setEventFlushLookback(unsigned int setback)`
The solver always saves the last used index of the dense output memory, thus it does not have to find the necessary indices. However in the case of events, sometimes steps are not accepted, and than this saved index will be set back with the `setback` number of indexes. Than the search for the new index is implemented by linear search. You should try to keep this value low, however if it is too low it may cause the solver to break. If there are many events in a small interval set it to a higher value. Preset value is 100.

`void setNrOfExtraPoints(unsigned int nrOfExtraPoints)`
When using the event handler, the solver takes extra steps, and it requires extra place in the memory. If the code fails when using the event handler set it to a bigger value. This is recommended to call this method immediately after calling `setNrOfSteps`.

4.5.12 Save functions

`void save(std::string filename, unsigned int vecId = 0,int precision = 16)`
Saves all dense variables t, x and than x' values. `vecId` should be $0 \dots 4 \times \text{nrOfUnroll} - 1$. It should be used only for testing, because saving a lot of points to files is slow.

`double* getEndValues(int varId = 0)`
Get the end values of *all* variables with index `varId`, this means $4 \times$ `nrOfUnroll` values.