Technical Report
Advanced Practical in Optimal Control

Horea-Alexandru Cărămizaru
MSc. Scientific Computing
Heidelberg University
Heidelberg, Germany
horea.caramizaru@stud.uni-heidelberg.de

24-July-2021

Abstract

The main scope of this practical is to expose the interfaces of **IDAS/CVODES** integrators from the **SUNDIALS suite** [4] into **Matlab**. To this end, the implementation provides the means to define a **dynamical system**, to compute **forward integration** as well as **first and second order sensitivity** in a **parallelized** way. All of these are done by **automatically C/C++ code generation** of the **integrator** and of the **sensitivity** by using a **high level language** to define a **dynamical system** on top of **CasADi**, which closely resembles a **symbolical framework** without having the corresponding disadvantages.

Alongside this report, a complete implementation can be found here: SUNDIALS2Matlab.

Index Terms: **IDAS, CVODES, SUNDIALS suite, dynamical system, forward integration, sensitivity, parallelization, code generation, CasADi, symbolical framework.**

1 Introduction

The main task of this practical was to research 2 ways of integrating **SUNDIALS suite** as part of **MLI** project and also to provide the necessary **backend framework**.

The 2 possible libraries considered for this task were: **CasADi** [2] and **AMICI** [3] both of them providing an automatic way of exposing **SUNDIALS** C/C++ code integrators to dynamical languages like **Python** and **Matlab**.

The final decision of using **CasADi** was taken after multiple testing on each of them and was based on the fact that is a better-integrated project with comprehensive documentation and a wider community.

Besides providing access to **SUNDIALS suite**, **CasADi** offers a way of defining the **dynamical systems** using a **high level language** as well as **C/C++ code generation** of the **integrator** and **sensitivity** at run time (**JIT**) starting from a **symbolical representation** of the problem and also, general means to **parallelize** the computation.

2 Structure of the report

Section 3, starts by defining the **Optimal Control problem** by splitting it using a **hierarchical layer architecture**. At the end of this section, it will be clear how the **backend framework** provided can be integrated into a general **OCP** framework.

Section 4, starts by describing the general **backend framework design** 4.1. In subsection 4.2, the general **workflow** is introduced. It continues then by introducing **CasADi's high level language** in subsection 4.3. Based on that, it is afterwords described how a **dynamical system** can be defined using **CasADi's framework** 4.4 and how **CVODES integrator, first and second order sensitivity Code Generation** is done 4.5.

Section 5 is meant to underline the advantages of using **CVODES integrator** over the default ones offered by **Matlab** by comparing the computation time required in multiple dynamical systems scenarios.

Section 6 ends this report with a section of **Conclusions and Recommendations** and possible feature extensions of the provided **backend framework**.

3 Problem Definition

The problem that needs to be solved regards the following OCP (Optimal Control problem):

$$
\begin{aligned}
\min_{x(\cdot),u(\cdot)} \quad & \Phi(x(t_f)) && \text{(1a)}\\
\text{s.t.} \quad & \dot{x}(t) = f(x(t), u(t), p), && \text{(1b)}\\
& x(t_0) = x_0 && \text{(1c)}\\
& x^{lo} \leq x(t) \leq x^{up}, && \text{(1d)}\\
& u^{lo} \leq u(t) \leq u^{up} \qquad \forall t \in [t_0, t_f] && \text{(1e)}
\end{aligned}
$$

To solve Eq:1 numerically, we are using a discretized version by introducing the following multiple shooting variables: $s_0, \cdots, s_N$ $q_0, \cdots, q_N$ for Eq:2.

$$
\begin{aligned}
\min_{x(\cdot),u(\cdot)} \quad & \Phi(S_N) && && \text{(2a)}\\
\text{s.t.} \quad & s_{i+1} = x(t_{i+1}; t_i, s_i, q_i, p) && i = 0, ..., N-1 && \text{(2b)}\\
& s_0 = x_0 && && \text{(2c)}\\
& x^{lo} \leq s_i \leq x^{up}, && i = 0, ..., N && \text{(2d)}\\
& u^{lo} \leq q_i \leq u^{up} && i = 0, ..., N && \text{(2e)}
\end{aligned}
$$

where $x(t; t_0, s, q, p)$ is the solution of Eq:3.

$$
\begin{aligned}
\dot{x}(t) &= f(x(t), q, p) && \text{(3a)}\\
x(t_0) &= s && \text{(3b)}
\end{aligned}
$$

Next, we define the primal variables as $w = (s, q)$ and we introduce the following functions for equality and inequality constraints:

$$a(w) = \begin{bmatrix} x_0 - s_0 \\ x(t_1; t_0, s_0, q_0, p) - s_1 \\ \vdots \\ x(t_N; t_{N-1}, s_{N-1}, q_{N-1}, p) - s_N \end{bmatrix} \tag{4}$$

$$b(w) = \begin{bmatrix} x^{lo} - s \\ s - x^{up} \\ q^{lo} - q \\ q - q^{up} \end{bmatrix} \tag{5}$$

Based on Eq:4 and Eq:5 one can write the OCP in a more compact form:

$$\min_{w} \quad \Phi(w) \tag{6a}$$

$$\text{s.t.} \quad a(w) = 0 \tag{6b}$$

$$b(w) \leq 0 \tag{6c}$$

For Eq:6 the Lagrange function and its derivatives at point (w, λ, μ) are defined as follows:

$$\mathcal{L}(w, \lambda, \mu) = \Phi(w) - \lambda^\top a(w) - \mu^\top b(w) \tag{7}$$

$$\nabla \mathcal{L}(w, \lambda, \mu) = \begin{bmatrix} \nabla_w \Phi(w) - \nabla_w a(w)\lambda - \nabla_w b(w)\mu \\ a(w) \\ b(w) \end{bmatrix} \tag{8}$$

$$\nabla^2 \mathcal{L}(w, \lambda, \mu) = \begin{bmatrix} \nabla_w^2 \Phi(w) - \nabla_w^2 a(w)\lambda & \nabla_w a(w) & \nabla_w b(w) \\ \nabla_w a(w)^\top & & \\ \nabla_w b(w)^\top & & \end{bmatrix} \tag{9}$$

We want to be able to solve Eq:10.

$$\nabla \mathcal{L}(w, \lambda, \mu) = 0. \tag{10}$$

We apply Newton's method and we have to solve for (w_i, λ_i, μ_i) :

$$\nabla^2 \mathcal{L}(w_i, \lambda_i, \mu_i)\Delta w + \nabla \mathcal{L}(w_i, \lambda_i, \mu_i) = 0. \tag{11}$$

Equivalently, the following QP (Quadratic Programming) needs to be solved:

$$\min_{\Delta w} \quad \Delta w^\top \nabla_w^2 \mathcal{L}(w_i, \lambda_i, \mu_i)\Delta w + \nabla\Phi(w_i)\Delta w \tag{12a}$$
$$\text{s.t.} \quad a(w_i) + \nabla a(w_i)\Delta w = 0 \tag{12b}$$
$$b(w_i) + \nabla b(w_i)\Delta w \leq 0 \tag{12c}$$

To solve Eq:12, the following terms, which include the evaluation of the dynamical system, must be evaluated: $a(w_i), \nabla_w a(w_i), \nabla_w^2 a(w) \cdot \lambda$

The process of evaluation of $\boldsymbol{a(w_i), \nabla_w a(w_i), \nabla_w^2 a(w) \cdot \lambda}$ requires the implementation of the following functions, as part of integration of **SUNDIALS**:

S.1 $\boldsymbol{x(t; t, s, q, p)}$ Standard forward integration.

S.2 $\boldsymbol{\nabla_w x(t; t, s, q, p) \cdot d}$ This is the directional derivative of $\boldsymbol{x(t; t, s, q, p)}$ in the direction $\boldsymbol{d}$. Multiple directions can be evaluated at the same time. The complete Jacobian can be computed by computing directional derivatives in all unit directions.

S.3 $\boldsymbol{\nabla_w^2 x(t; t, s, q, p) \cdot \lambda}$ Hessian of $\boldsymbol{\lambda}^\top \cdot \boldsymbol{x(t; t, s, q, p)}$ with respect $\boldsymbol{w}$.

Where the dimensions are:

$$\boldsymbol{t} \in \mathbb{R} \tag{13a}$$
$$\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{n_x}} \tag{13b}$$
$$\boldsymbol{q} \in \mathbb{R}^{\boldsymbol{n_q}} \tag{13c}$$
$$\boldsymbol{p} \in \mathbb{R}^{\boldsymbol{n_p}} \tag{13d}$$
$$\tag{13e}$$

A simplified, comprehensive way to visualize this problem can be seen using Figure:1 which introduces **a 3 layer architecture** where the first 2 (the **OCP** and **QP**) are provided by **MLI** whereas, the 3rd layer, introduces the **backend framework** which is build on top of **CasADi** and represents the *main contribution of this practical*.

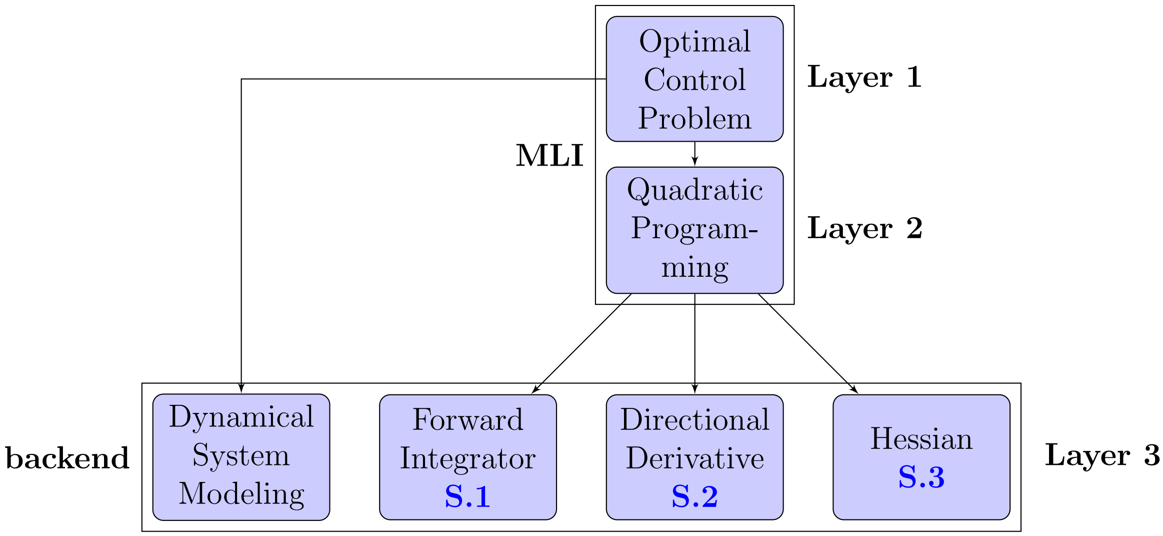

Figure 1: Problem architecture

Long story short, solving the **OCP** (defined by the **Layer 1**) requires multiple **QP** queries (introduced by the **Layer 2**) which in turn, requires a way to define the **dynamical system** and to compute **S.1**, **S.2** and **S.3** (exposed by **Layer 3**).

4 Framework

4.1 Framework Structure

This project requires the latest version of **CasADi** framework (which can be obtained from [1]) as part of the main structure of the project under a folder called **casadi**.

Alongside, the structure introduced by Figure: 2 defines the the components of the framework where:

- *DynamicalSystems* – Is the folder containing all the **dynamical systems** defined as separated files using **CasADi's high level language**.

- *functions* – Is the folder containing the main functionalities of the project: **One time code generation** and the **binding functors** for calling **forward integration** as well as **first** and **second order sensitivity**.

- *MatlabFunc* – Is the folder where the corresponding **Matlab dynamical systems** are defined used for performance comparisons with **CVODES**.

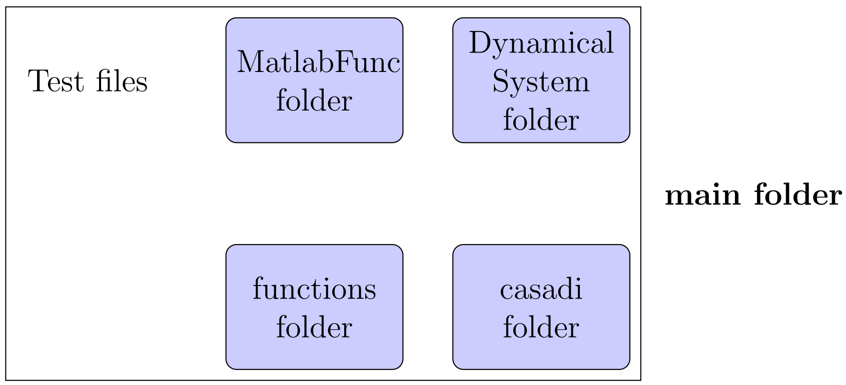

Figure 2: Framework folder structure

4.2 Framework use case workflow

In a nutshell, the **workflow** of the project is determined by 3 steps, operated in the following order:

1. *Dynamical System definition:* Definition of the problem using a high level language.

2. *One time code generation:* By calling one of the following functions: **initODE()**, **InitODEWSensitites()** or **InitODEWSensititesAndHessian()**

3. *Multiple function calls of:* **integrate()**, **integrateWSensitivies()** and **integrateWSensitiviesAndHessian()**

The use case **workflow** of the project is summarized by Figure:3.

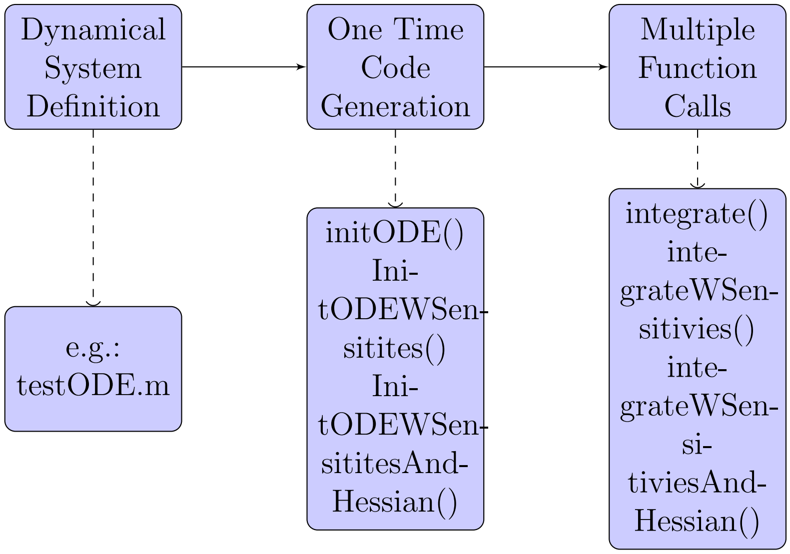

Figure 3: Use case workflow

4.3 CasADi

CasADi is an open-source software tool for **numerical optimization** in general and **optimal control** (i.e. optimization involving differential equations) in particular. [2]

The main scope of **CasADi** is **automatic differentiation**. Besides that, it has also support for **ODE/DAE integration** and **sensitivity analysis**, nonlinear programming and interfaces to other numerical tools (**SUNDIALS suite**)

At the core of **CasADi** is a self-contained **symbolic framework** that allows the user to construct symbolic expressions using a **Matlab** inspired everything-is-a-matrix syntax, i.e. vectors are treated as n-by-1 matrices and scalars as 1-by-1 matrices. Further on, the constructed symbolical expression is used by numerical means.

CasADi symbolical framework defines multiple data types but the most relevant for this project is **SX** which is used to represent **matrices** whose elements consist of symbolic expressions made up by a sequence of unary and binary operations. Below are some examples of **CasADi's API** for defining **symbolical expressions**.

```
%Defining 2 symbolical variables 'a' and 'b':
a = SX.sym('a');
b = SX.sym('b');

%Computing the Jacobian of 'sin(a)' with respect to 'a'.
J = jacobian(sin(a),a);
%Computing the Hessian
H = hessian([a;b],[a;b]);

% Function with two scalar inputs, one output. It generates a Functor.
x = a^2+b^2;
f = Function('f',{a,b},{x});
x_res = f(2,3);

% Function with one vector input, one output. It generates a Functor.
x = a^2+b^2;
f = Function('f',{[a;b]},{x});
x_res = f([2;3]);

% Solving a QP.
y = a^2+b^2;
solver = qpsol('solver','qpoases',struct('x',[a;b],'f',y));
res = solver('x0',[0.1;0.2]);
full(res.x)

% Solving NLP.
y = a^2+b^2;
solver = nlpsol('solver','ipopt',struct('x',[a;b],'f',y));
res = solver('x0',[0.1;0.2]);
full(res.x)

% Defining an ODE
%    dot(a) = 1,
%    dot(b) = a^2+b^2 = y
y = a^2+b^2;
intg = integrator('intg','cvodes',struct('x',[a;b],'ode',[1;y]));
res = intg('x0',[0.1;0.2]);
full(res.xf)
```

A comprehensive documentation for CasADi can be accessed here: CasADi.

4.4 Dynamical System definition

Given the **dynamical system** described by Eq:14, where $\boldsymbol{t}$ is the time, $\boldsymbol{x}$ is the differential states, $\boldsymbol{q}$ is the control (constant) and $\boldsymbol{p}$ is the parameter we are aiming for a way to use **CasADi** to define it.

$$\dot{\boldsymbol{x}}(\boldsymbol{t}) = \boldsymbol{f}(\boldsymbol{t}, \boldsymbol{x}(\boldsymbol{t}), \boldsymbol{q}, \boldsymbol{p}) \tag{14a}$$

$$\boldsymbol{x}(\boldsymbol{t_0}) = \boldsymbol{x_0} \tag{14b}$$

This is done in a separate file as part of the folder *DynamicalSystems* and it must follow the previously introduced **CasADi's high level language** definition convention for **ODE/DAE**.

A more comprehensive example using **Lotka-Volterra ODE** would be:

```
import casadi.*

%define the states
x = SX.sym('x',2);

%define the parameters
a = SX.sym('a',1);
b = SX.sym('b',1);
c = SX.sym('c',1);
d = SX.sym('d',1);

%define the control
u = SX.sym('u',1);

%building the dynamical system
sys = struct;
%states
sys.x = x;
%parameters
sys.p = [u;a;b;c;d];
%defining the ODE/DAE
```

```
sys.ode = [
    a * x(1) - b * x(1) * x(2) - x(1) * u ;
    c * x(1) * x(2) - d * x(2) - x(2) * u
    ];
```

4.5 Code Generation

Based on the above definition of the dynamical system, the user needs to call **one time**, a function that **generates**, **on the fly**, the customized C/C++ code for the corresponding integrator, **compiles** the newly generated code (**JIT compiling**) and returns a **Functor** that provides future access to the **forward integrator** and/or **sensitivity computation**.

The corresponding call for *Lotka-Volterra ODE* defined in the file *lotka_volterraCasADi.m* would look like as follows:

```
InitODE('lotka_volterraCasADi', tStart, tEnd);
```

The corresponding call dor DAE looks like as follows:

```
InitDAE('DinamicalSystem', tStart, tEnd);
```

The mandatory parameters of **InitODE()/InitDAE()** are as follows:

- *lotka_volterraCasADi* : Name of the file that contain the dynamical system
- *tStart* : The start time (most of the time, it's **0**)
- *tEnd* : The end time of the integration interval. The current **CasADi's code generation Matlab API** is **limited to an initial definition at compile time of the time interval** for **SUNDIALS suite**.

If one requires also access to the **first** and **second order sensitivity**, one must call one of the following functions with the same list of parameters as above:

- *InitODEWSensitites*
- *InitODEWSensititesAndHessian*

At this point, all the prerequisites for the future calls of *integrator*, *sensitivity* and *hessian* ($\boldsymbol{integrate(inp)}$, $\boldsymbol{integrateWSensitivies(inp)}$ and $\boldsymbol{integrateWSensitiviesAndHessian(inp)}$) w.r.t. the ODE are satisfied as they are the result of the automatic generation. These can be accessed using the **global variable: s2m**. Another important aspect is that one of the optional parameter **nrThreads/threads** can be used to parallelize the process by explicitly defining the **number of threads** used.

Each of the calls must contain, as parameter, an object that contains a subset of the variables defined below:

- ***inp.M*** : Number of multiple shooting intervals.
- ***inp.sd*** : Initial value for differential states for each multiple shooting interval.
- ***inp.q*** : A vector of controls used by each integrator call with size: ***inp.M***
- ***inp.p*** : The values of the parameters in the same order it was defined previously in the dynamical system.
- ***inp.nx*** : The size of $\boldsymbol{x}$.
- ***inp.nq*** : The number of control parameters.
- ***inp.np*** : The number of parameters.
- $\boldsymbol{inp.fwd\_x0}$: The sensitivity directions in form of matrix containing only the components corresponding to $\boldsymbol{x_0}$.
- $\boldsymbol{inp.fwd\_p}$: The sensitivity directions in form of matrix containing only the components corresponding to $\boldsymbol{parameters}$.
- $\boldsymbol{inp.nr\_sensdirs}$: The number of sensitivity directions.
- ***inp.lambda*** : The adjoint sensitivity direction.
- ***inp.threads*** : The number of threads for the thread pool used by the integrator.

For complete examples (*input/output*) please check the following files: $\boldsymbol{test\_integrate.m}$, $\boldsymbol{test\_integrateWSensitivies.m}$ and $\boldsymbol{test\_integrateWSensitiviesAndHessia.m}$

5 Integrators comparison

The development of this project is based on the *hypothesis* that a **native C/C++ integrator**, exposed into **Matlab**, is faster than the **Matlab** counterpart. To see how well the **CVODES integrator** works, a set of tests were developed. For comparison, the complete list can be checked in Figure:4.

Integration time (s)	Dynamical system	Matlab integrator	Matlab integration steps	Matlab computation time (s)	SUNDIALS integrator	SUNDIALS computation time (s)	Computation time ratio: (Matlab / SUNDIALS)	Test file
1	Pendulum	ode45	65	0.0027	CVODES	0.0016	1.69	main_test1.m
5	Pendulum	ode45	137	0.0048	CVODES	0.0031	1.55	main_test1.m
10	Pendulum	ode45	249	0.0055	CVODES	0.0018	3.06	main_test1.m
50	Pendulum	ode45	1153	0.0244	CVODES	0.0046	5.30	main_test1.m
100	Pendulum	ode45	2101	0.0492	CVODES	0.0078	6.31	main_test1.m
500	Pendulum	ode45	5541	0.0971	CVODES	0.0157	6.18	main_test1.m
1	Second order ODE	ode23	17	0.0029	CVODES	0.0015	1.93	main_test2.m
5	Second order ODE	ode23	32	0.0049	CVODES	0.0016	3.06	main_test2.m
10	Second order ODE	ode23	38	0.0032	CVODES	0.0014	2.29	main_test2.m
50	Second order ODE	ode23	58	0.0042	CVODES	0.0015	2.80	main_test2.m
100	Second order ODE	ode23	65	0.0052	CVODES	0.0017	3.06	main_test2.m
500	Second order ODE	ode23	79	0.0057	CVODES	0.0016	3.56	main_test2.m
1	The van der Pol equation	ode15s	40	0.004	CVODES	0.0015	2.67	main_test3.m
5	The van der Pol equation	ode15s	43	0.006	CVODES	0.0024	2.50	main_test3.m
10	The van der Pol equation	ode15s	43	0.0042	CVODES	0.0014	3.00	main_test3.m
50	The van der Pol equation	ode15s	46	0.0042	CVODES	0.0015	2.80	main_test3.m
100	The van der Pol equation	ode15s	46	0.0052	CVODES	0.0015	3.47	main_test3.m
500	The van der Pol equation	ode15s	49	0.0058	CVODES	0.0015	3.87	main_test3.m
1	Lorenz system	ode45	65	0.0029	CVODES	0.0017	1.71	main_test4.m
5	Lorenz system	ode45	309	0.006	CVODES	0.0023	2.61	main_test4.m
10	Lorenz system	ode45	601	0.0105	CVODES	0.0033	3.18	main_test4.m
50	Lorenz system	ode45	2989	0.0467	CVODES	0.0121	3.86	main_test4.m
100	Lorenz system	ode45	5989	0.0651	CVODES	Error 1	N/A	main_test4.m
500	Lorenz system	ode45	29721	0.2691	CVODES	Error 2	N/A	main_test4.m
1	Lotka Volterra	ode23	11	0.0003	CVODES	0.0014	0.21	main_test5.m
5	Lotka Volterra	ode23	15	0.0004	CVODES	0.0014	0.29	main_test5.m
10	Lotka Volterra	ode23	33	0.0011	CVODES	0.0016	0.69	main_test5.m
50	Lotka Volterra	ode23	150	0.0037	CVODES	0.0023	1.61	main_test5.m
100	Lotka Volterra	ode23	303	0.005	CVODES	0.0026	1.92	main_test5.m
500	Lotka Volterra	ode23	1503	0.022	CVODES	0.0083	2.65	main_test5.m

Error 1 : CV_TOO_MUCH_WORK (at t = 75.62)
Error 2 : CV_TOO_MUCH_WORK (at t = 74.7571)

Figure 4: **CVODES integrator** *vs.* **Matlab native integrator**, Intel i7-6500U CPU@2.5GHz 8GB RAM, Matlab R2020a

From Figure:5 one can observe that the **CVODES integrator** tend to be, on average, **2** up to **3** times faster. For longer integration time, an exponential computation improvement can be observed. This comparison doesn't take into consideration that, most of the time, multiple integration intervals are computed within a **multiple shooting approach** which can be **parallelized for free** based on the provided **backend framework** implementation which can also **improve the computation time significantly**. For **Lotka-Volterra ODE**, a time disadvantage of the **CVODES integrator** can be observed. This result is of little importance overall as it is the consequence of the constant time transfer of the data block memory required between **Matlab context** and **native C/C++ CVODES integrator**. This can become a problem in case of improper use (e.g. if one is using multiple calls with **small time intervals**, which in turn is triggering a **small number of integration steps**, instead of a **smaller number of calls** with a **larger time intervals**) and can **diminish** the gain based on **parallelization of multiple shooting** above mentioned.

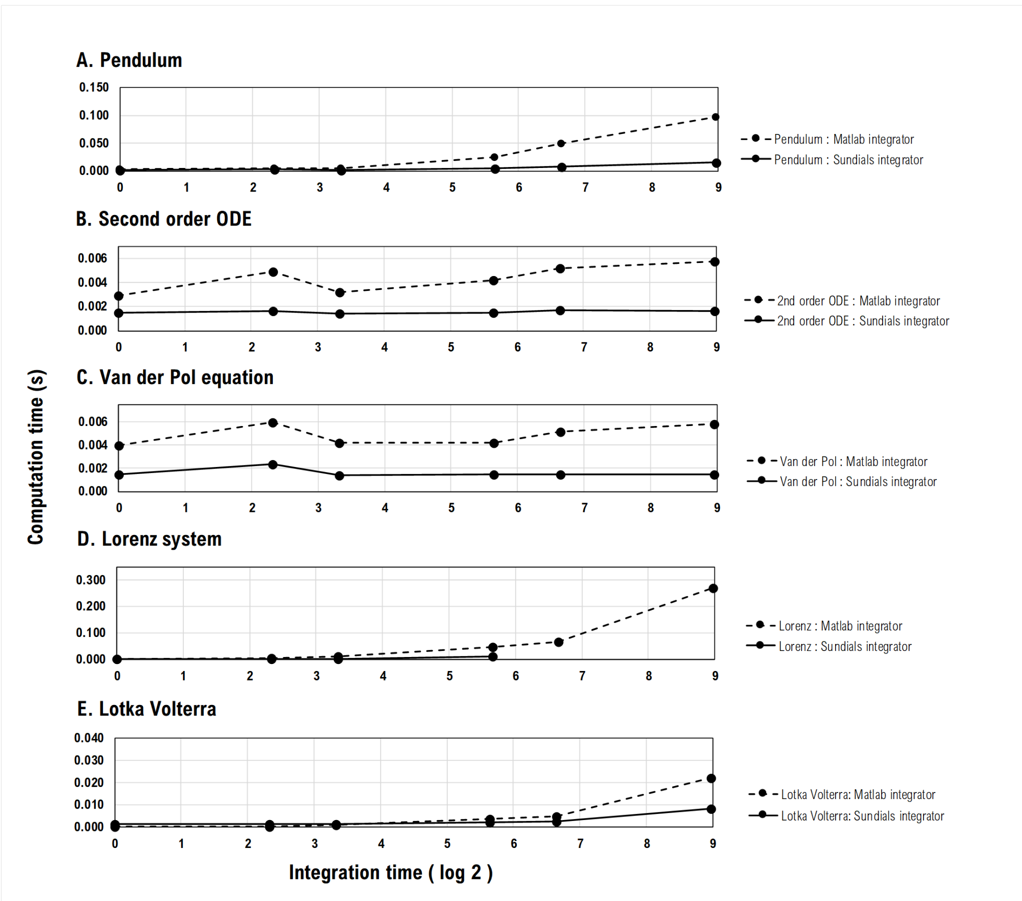

Figure 5: **Integration time** *vs.* **Computation time**

6 Conclusions and Recommendations

The next natural step for this project will be the integration into the **MLI framework**. To this end, the current provided *API* should be the base of a smooth process.

The current implementation contains the complete necessary backend for the initial **ODE** problem introduced in **Section:**3. Besides that, it also contains the necessary implementation for **DAE** and it also partially implements the *API*. The remaining **first** and **second order sensitivity** functionalities should be straightforward to implement based on the counterpart **ODE** implementation.

An experiment can be performed to optimize the number of multiple shooting intervals mentioned in **Section:**5 to optimize for time by means of **parallelization**, modeling the problem as an **OCP** for **producer-consumer** which closely resembles **Lotka-Volterra dynamical system**.

Last but not least, more work should be put into exposing, in an optional way, the rest of the possible configuration parameters that can be part of the **code generation API**.

References

[1] Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. Casadi framework download @ONLINE. https://web.casadi.org/get/, 2021.

[2] Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, In Press, 2018.

[3] Fabian Fröhlich, Daniel Weindl, Yannik Schälte, Dilan Pathirana, Łukasz Paszkowski, Glenn Terje Lines, Paul Stapor, and Jan Hasenauer. Amici: High-performance sensitivity analysis for large ordinary differential equation models. *Bioinformatics*, 04 2021. btab227.

[4] Alan C Hindmarsh, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.