Differential Algebra Lab

AstroNet II – Fourth Training School

Alexander Wittig*

Milan, February 2 – 6, 2015

This document contains a hands-on introduction to Differential Algebra (DA) techniques and some of their applications to engineering to be used as reference for the DA lab session of the school.

The key goal of DA is to allow for the automatic and efficient expansion of arbitrary functions as a polynomial by replacing floating point operations with corresponding DA operations on a computer. DA concepts, both theoretical and most of all practical, are introduced using fully coded examples written in C++ using the *Dinamica Differential Algebra Core Engine* (Dinamica DACE). The different sections of this document cover topics ranging from the very first DA steps up to the simulation of a PID controller. Plenty of exercises are provided for each section, and fully coded solutions to all exercises are made available for reference.

In the appendix, some notes on DA are given to briefly give an idea of the technique to those not familiar with it. Various standard numerical algorithms as well as engineering and physics concepts used in the exercises are also provided for convenience. Lastly, the appendix includes some notes on C++ to briefly introduce the relevant concepts required for these exercises. While the Dinamica DACE makes heavy use of advanced C++ techniques such as object oriented programming and templates internally, its interface is designed in such a way that it can be used without knowledge of either subject in a purely procedural manner (C style).

*Postdoctoral Researcher at Politecnico di Milano, Milan (Italy). e-mail: alexander.wittig@polimi.it.

Introduction

The exercises are designed to complement the lab on systems control. You will work in the same teams as for the control lab. Each team can chose to work on any number of the following exercises based on your interests. You are encouraged to play with the DA tools, the goal is not to finish all the exercises. In fact, there are more exercises provided than can be done during the school. Some of the exercises are recommended, they are marked by a (*) before the exercise. These will build on each other and should be done in order, eventually they lead to simulating a PID controller.

If you are not familiar with the basic ideas of DA, you should first read the Notes on Differential Algebra in Appendix A. If you are not familiar with C++, you are encouraged to first read the Notes on C++ in Appendix B.

Additional materials, including presentations, documentation, solutions and pre-made code snippets useful to reduce implementational effort, are available at the school website at

https://dib.aero.polimi.it/data/public/6507e13484.php.

Software Setup

The DA implementation we will use is called Dinamica Differential Algebra Core Engine (Dinamica DACE). It provides a convenient and powerful C++ interface to basic as well as advanced DA routines. Its computational core is based on the same algorithm as COSY INFINITY, one of the oldest implementations of DA. The Dinamica DACE is currently being developed under ESA contract ITT 7570: "Nonlinear Propagation of Uncertainties in Space Dynamics based on Taylor Differential Algebra" for the European Space Agency.

For simplicity, we will provide a fully set up Linux system ready to develop and run DA enabled programs along with kdevelop, a convenient integrated development environment (IDE). This system is distributed as a bootable USB stick that can be run on any Intel/AMD x64 based PC or laptop.

Dinamica SRL is planning to release the Dinamica DACE software free of charge once all legal and technical hurdles have been cleared with ESA. If you wish to be notified when the Dinamica DACE officially is launched publicly, please let us know and we'll put you on an email list so you don't miss it.

Notation

In this document we use the following notation:

- Given a sufficiently smooth function $f(x)$, its Taylor expansion around $x = 0$ is written as $\mathcal{T}_f(x)$. If the order n of the expansion is important it is denoted as $\mathcal{T}_f^n(x)$.
- Unless specifically noted, the expansion point of any Taylor expansion is assumed to be 0.

- We denote by $f^{(i)}(x)$ the $i-th$ derivative of a function $f(x)$ with the convention that $f^{(0)}(x) = f(x)$. We will also use the notation f', f'', ... in the text to denote derivatives.

- In mathematical expressions, in particular when describing algorithms, it sometimes is necessary to explicitly denote DA objects (elements of the truncated polynomial algebra). We do so by square brackets, i.e. $[x], [p]$. To distinguish from other variables, the independent variables of the expressions they represent are prefixed by δ. Thus, the expression $[x] = x_0 + \delta x$ indicates that $[x]$ represents a polynomial of the form $P(\delta x) = x_0 + \delta x$ either as an element of the truncated polynomial algebra (mathematically) or as a DA object (in the computer implementation).

- Additionally, when referred outside the context of a DA object as above, the independent DA variables are simply referred to as $x_1, x_2, x_3, \ldots$ or, in the special cases of operations in $\mathbb{R}$, $\mathbb{R}^2$, and $\mathbb{R}^3$, also as x, y, z.

1. DA Basics

Initializing the Dinamica DACE and using the DA class which forms the core of the system to perform simple DA expansions

```
1  #include <DA/dace.h>
2  #include <cmath>
3  using namespace std;
4  using namespace DACE;
5
6  int main( void )
7  {
8      DA::init( 10, 1 );
9      DA x = DA(1);
10     cout << "x" << endl << x << endl;
11     cout << "sin(x)" << endl << sin(x);
12 }
```

This code illustrates the basic structure of a DA enabled program. It computes the Taylor expansion of $\sin(x)$ up to order 10 and outputs the result to the screen. The details of the code are described line-by-line in the following.

DA setup

Lines 1-4 include the `DA/dace.h` header which contains the core `DA` class as well as the system `cmath` header for math routines, and import the standard `std` and `DACE` namespace. While importing the namespace is not strictly required, it is strongly recommended to prevent obscure problems with old-style C math function overloading. You can just consider those three lines required setup for the rest of your code. They will be present in all your programs.

Line 8 contains a very important command that should always be the very first line of code executed. It calls `DA::init( order, variables )` which initializes the DACE to perform computations up to the given order with the given number of independent variables $x_1, x_2, \ldots$. You must not use any DACE facilities (including declaration of DA variables) before calling this function.

Remark 1. During the execution, you can temporarily reduce the order to a smaller value, but it can never be increased beyond the order specified here. Similarly, you are free to use fewer variables than you initialize the DACE with, but you cannot later increase this limit. There is some overhead associated with extra orders and variables, so for peak performance you should specify the numbers actually used in your program here.

In order to reduce the computation order temporarily, it is most convenient to use the command `DA::pushTO(new_order)`, which will set the truncation order to the `new_order` value given. To reset it to whatever it was before you called this command, call `DA::popTO()`. That is, if you need to reduce the computation order, wrap

your code into a set of `DA::pushTO()` and `DA::popTO()` calls so as to leave the computation order unaffected for code that may be executed later.

DA operations

Line 9 declares `x` to be a variable of type `DA` and sets it to `DA(1)`, corresponding to the identity in the first independent variable, that is the polynomial $P(\vec{x}) = x_1$ or simply x_1. Using our bracket notation, we also sometimes write this as $[x] = \delta x_1$. You can obtain the identities in the other independent variables using `DA(2)`, `DA(3)` and so on.

Remark 2. It is often convenient to declare C++ DA variables with names such as `x, y, z` or `x1, x2, x3` and initialize them to the values `DA(1), DA(2), DA(3)` respectively in the beginning of your program. Using `x` instead of `DA(1)` in the subsequent code increases the readability.

Line 11 uses the `sin` function with a DA argument to compute the sine of x_1 in DA arithmetic. Apart from the `sin` function, all other intrinsic math functions in the C++ math library (such as `log` or `atan`) are also available as DA functions. C++ automatically detects if the argument is a DA object and uses the correct function. It is thus possible to write almost any expression as for floating point numbers and have it evaluated directly in DA arithmetic.

Remark 3. As the function names of common mathematical operations for DA and regular double precision floating point operations are the same, it is possible to (re)use the same code list of a function for evaluation with `double` or `DA` data types without writing the function again. To do so, use *C++ function templates* as introduced in Section B.

Output

Line 10 and 11 output the variable `x` and the result of the DA function evaluation `sin(x)` to the screen. Outputting DA variables to a file or the screen is done the way as any other data type: using the C++ stream facilities. In particular, the stream `stdout` is already defined in every C++ program and represents the screen. You can output DA variables like any C++ type using the `<<` output operator.

The output of a DA variable on screen looks like this:

```
I  COEFFICIENT                ORDER EXPONENTS
1     1.000000000000000         1   1
2    -0.1666666666666667        3   3
3     0.8333333333333333E-02    5   5
4    -0.1984126984126984E-03    7   7
5     0.2755731922398589E-05    9   9
-----------------------------------------------
```

It consists of a numbered list of coefficients of the resulting polynomial, sorted by order, giving one coefficient per line. Along with the value of each coefficient, its order and the exponents are shown. As we have initialized the DACE to use only one variable in the example above, only one column is shown under EXPONENTS. If the DACE is initialized with more than one variable there will be a column with an exponent shown for each independent variable. Note that only the non-zero terms are shown, zero coefficients are automatically suppressed in the output.

The DA output reported above thus corresponds to the polynomial (coefficients truncated for readability):

$$P(x_1) = 1.0 \cdot x_1^1 - 0.167 \cdot x_1^3 + 0.00834 \cdot x_1^5 - 0.000198 \cdot x_1^7 + 0.00000276 \cdot x_1^9.$$

Remark 4. At first it may seem a bit cumbersome to have a tabularized output of this form instead of a mathematical polynomial expression. However, this output format is much easier to read and handle in a text output once the polynomial contains more than just a few dozen coefficients.

Choice of Expansion Point

As we have seen in these examples, DA always expands a function around the point $x = 0$. This is not a restriction at all as it is very simple to expand a function around another point x_0. We simply take that the Taylor expansion of the function $f(x_0 + x)$, instead of just $f(x)$, around $x = 0$. Since

$$\frac{d^i}{dx^i} f(x_0 + x) = f^{(i)}(x_0 + x)$$

the resulting Taylor expansion around $x = 0$ is of the form

$$P(x) = \sum_{i=0}^{N} \frac{f^{(i)}(x_0)}{i!} x^i.$$

This is exactly the Taylor expansion of $f(x)$ around the point x_0.

In the DA arithmetic, this is very easily achieved, too. Function calls with DA can be "chained" just like they can be with floating point numbers. That is, instead of calling the function `f` with the DA identity `x`, we can instead also pass any other DA expression including the expression `x0+x`.

1.1. Exercises

1. Verify that the expansion of the function $sin(x)$ is correct by manually computing the coefficients of the Taylor series around 0.

2. Instead of $\sin(x)$ compute the expression $3(x + 3) - x + 1 - (x + 8)$. What should the result be?

3. (*) Change the program to compute the Taylor expansion of the *sin* function around 1 instead of 0, that is compute $\mathcal{T}_{sin(1+x)}(x)$.
   Check the result manually or using another tool for automatic differentiation.

4. Compute $sin(x)^{11}$ and output the result. What is happening?

5. Write a function that takes two arguments `x` and `y` of any type and returns the expression $\sqrt{1+x^2+y^2}$.

1.2. Advanced Exercises

1. Compute the Taylor expansions of $sin(x)^2$ and $cos(x)^2$ and print both to the screen. Then print their sum and explain the result. Try other expressions for which you know the correct answer.

2. (*) Write a function to evaluate the sombrero function

$$f(x,y) = \sin\left(\sqrt{x^2+y^2}\right)/\sqrt{x^2+y^2}$$

   in both DA and double precision. Evaluate the function at $(0,0)$ in double precision arithmetic and then try to expand around $(0,0)$ in DA arithmetic. Compare the results to the analytically obtained limit of f at the origin and its Taylor expansion $\mathcal{T}_f(x,y)$ there.
   Hint: If your C++ runtime library does not automatically show error messages associated with exceptions, use the C++ exception construct `try { f(x,y); } catch( exception &ex ) { cout << ex.what(); }` to see what is going wrong (see Section B).

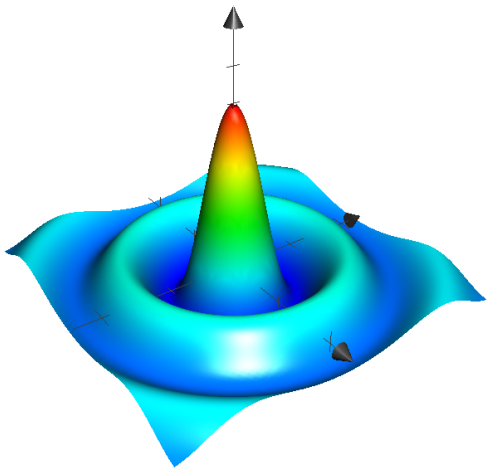

Figure 1.1: The sombrero function.

2. Automatic Derivation and Integration

Using Differential Algebra to automatically compute accurate arbitrary partial derivatives and integrals of functions

```
1  #include <DA/dace.h>
2  #include <cmath>
3  using namespace std; using namespace DACE;
4
5  DA f( DA x ) { return x/(1.0+x); }
6
7  int main( void ) {
8      DA::init( 10, 2 );
9      DA x = DA(1);    DA y = DA(2);
10     DA fx, dfx, ddfx, Sfx;
11     double x0 = 1.0;
12
13     fx = f( x );
14     dfx = fx.deriv( 1 );
15     ddfx = dfx.deriv( 1 );
16     cout << "f(x)\n" << fx << "\n(df/dx)(x)\n" << dfx
17     << "\n(d^f/dx^2)(x)\n" << ddfx << endl;
18
19
20     fx = f( x + x0 );
21     dfx = fx.deriv( 1 );
22     ddfx = dfx.deriv( 1 );
23     cout << "\nf(x+x0)\n" << fx
24     << "\n(df/dx)(x+x0)\n" << dfx
25     << "\n(d^f/dx^2)(x+x0)\n" << ddfx << endl;
26     cout << "f(x0)\n" << cons(fx)
27     << "\n(df/dx)(x0)\n" << cons(dfx)
28     << "\n(d^f/dx^2)(x0)\n" << cons(ddfx) << endl;
29
30     fx = f( x );
31     Sfx =  fx.integ(1);
32     cout << "\nintegral[f(x)]\n" << Sfx << endl;
33 }
```

This code illustrates how to use the derivative and integral operators that in DA are just ordinary operations just as addition or division. Lines 1-3 are the usual header, while line 5 defines a simple function `f` to evaluate the expression $\frac{x}{1+x}$ in DA. Lines 6-10 initialize the DACE and declare some variables.

Differentiation

Lines 13-17 demonstrate the basic use of the `deriv()` operator of the `DA` data type. This operator is attached with a period to the end of a DA expression and takes one

argument, the independent DA variable with respect to which the derivative is to be taken. Line 13 computes a DA representation (i.e. the Taylor expansion $\mathcal{T}_f(x)$ in the first independent DA variable x_1) of the function $f(x) = \frac{1}{1+x}$ into the DA variable `fx`. As we have seen in the previous section, the DA expansion in `fx` resulting from the evaluation of $f(x)$ in DA is a Taylor polynomial of the form $P(x) = \sum_{i=0}^{N} \frac{f^{(i)}(0)}{i!} x^i$. Thus, in addition to obtaining a *Taylor approximation* of the function and its derivatives in some neighborhood around $x = 0$, it is also possible to extract the *exact*[1] value of the derivatives of $f(x)$ at the expansion point $x = 0$ up to the computation order N. This is because the Taylor expansion is by definition exact when evaluated at $x = 0$.

Line 14 now takes the derivative of `fx` with respect to the first independent variable x and stores the resulting DA expansion in `dfx`. Thus now `dfx` contains the Taylor expansion of $\frac{df}{dx}(x) = f'(x)$. Similarly, the second derivative is computed into `ddfx`. The resulting expansions of the derivative functions are then output to the screen.

In order to compute the derivatives at any point x_0, recall from the previous section that we can change the expansion point of our DA expansion. Expanding in DA the function $f(x_0 + x)$ instead of just $f(x)$, we obtain the Taylor expansion of $f(x_0 + x)$ around $x = 0$, which contains the exact derivatives up to order N of $f(x)$ at the point x_0.

To extract those coefficients elegantly, we make use of the `cons()` function. It returns the constant part of a DA expression as a `double`. As such it acts like evaluating a DA expansion at 0. Hence applying this to the DA derivatives `dfx` and `ddfx` of the function $f(x)$, we obtain directly the desired values of the derivatives $f'(x_0)$ and $f''(x_0)$.

Remark 5. Note that since we are using a finite computation order N, the function $f(x)$ is expanded up to that order N. However, with each derivative we loose one order, so that e.g. $f'(x)$ is only expanded up to order $N - 1$. This must be taken into account in the choice of computation order.

Integration

Lines 30-32 demonstrate the inverse process. Starting again from a DA expansion `fx` of the function $f(x)$, we this time apply the `integ()` operator. Also this operator takes one argument, the independent DA variable with respect to which the integral is performed. It returns the Taylor expansion of $g(x) = \int_0^x f(\widetilde{x})d\widetilde{x}$. Note that in this operation the integration constant (lower bound of the integral) is in principle arbitrary, but in DA is fixed to be 0. Thus the resulting expansion of $g(x)$ has zero constant part.

Also for this operation, the expansion point at which the integral is to be computed can be changed easily by observing that

$$\int_0^x f(\widetilde{x} + x_0)d\widetilde{x} = \int_{x_0}^{x_0+x} f(\widetilde{x})d\widetilde{x}.$$

Thus once again shifting the expansion point by a constant changes the point around which the integral function is expanded.

[1] In practice the value is exact up to floating point errors accumulated during the computation, which are typically on the order of 10^{-15}.

2.1. Exercises

1. (*) Compute the partial derivatives ∂_x, ∂_y, ∂_x^2, ∂_y^2, $\partial_x\partial_y$, and ∂_x^3 of the sombrero function
$$f(x,y) = \sin\left(\sqrt{x^2+y^2}\right)/\sqrt{x^2+y^2}$$
at the point $(2,3)$. Then do the same computation using divided differences. Compare the relative errors $|\partial_{i,DA} - \partial_{i,DD}|/\partial_{i,DA}$ for various values of the finite difference. Can you get all relative errors of the divided differences to be less than 10^{-8}?
Now try the function $f(x,y) = \sin(y)\cdot\exp(x^2)$.

2. Compute the indefinite integral $F(x) = \int_0^x f(\widetilde{x})d\widetilde{x}$ of the function $f(x) = \frac{1}{1+x^2}$ and verify that it is correct by comparing it to the Taylor expansion of the analytical solution of the indefinite integral.

3. Compute the 30^{th} order Taylor expansion $\mathcal{T}_{erf}(x)$ of the error function $erf(x) = \frac{2}{\sqrt{\pi}}\int_0^x \exp(-t^2)dt$.

2.2. Advanced Exercises

1. (*) Use DA to build a Newton solver for any function $f(x)$ given as a C++ function. Instead of requiring an analytical derivative for f, use a first order DA computation to calculate the values of $f(x_n)$ and $f'(x_n)$ that is required in each step.
Test it by computing a root of the function $f(x) = x\sin(x) + \cos(x)$ as well as Kepler's equation $f(x) = x - e\sin(x) - M$ (for some suitable values of e and M)

2. Compute the Taylor expansion of the integral function
$$F(x) = \frac{2}{\sqrt{\pi}}\int_{x_0}^{x_0+x} \exp(-\widetilde{x}^2)d\widetilde{x}$$
around any given expansion point x_0.
Hint: use a change of variables to transform the bounds of the integral.

3. Polynomial Evaluation

Evaluating DA polynomials at discrete points and with other DA polynomials (function concatenation)

```
1  #include <DA/dace.h>
2  #include <cmath>
3  using namespace std; using namespace DACE;
4
5  int main( void )
6  {
7      DA::init( 10, 3 );
8      DA x = DA(1); DA y = DA(2); DA z = DA(3);
9      DA f = x/(1.0+x);
10     DA g = (x-z*exp(x+y))/cos(z);
11
12     cout << f.evalScalar(0.1) << " "
13          << 0.1/(1.0+0.1) << endl;
14     cout << f.evalScalar(sin(x)) << " "
15          << sin(x)/(1.0+sin(x)) << endl;
16     cout << f.evalScalar(y) << endl;
17
18     vector<double> arg(3);
19     arg[0] = 0.1; arg[1] = 0.2; arg[2] = 0.3;
20     cout << g << g.eval( arg ) << " "
21          << (0.1-0.3*exp(0.1+0.2))/cos(0.3) << endl;
22 }
```

An important operation acting on any function is of course that of evaluating the function. DA objects, as computer representations of functions, provide the same capability.

Pointwise Evaluation

The most simple way to evaluate a DA polynomial of one single variable is using the `evalScalar()` routine of the polynomial as demonstrated in lines 12 and 13. The variable `f` contains the DA expansion of the function $x/(1+x)$, and it is then evaluated at $x = 2.0$. The `evalScalar()` routine is a convenience routine that takes a single argument which is then used as the value for the first independent DA variable (i.e. `DA(1)`) in the DA expression being evaluated. All remaining independent DA variables (e.g. `DA(2)` or `DA(3)`) are automatically assumed to be 0. In particular for simple DA expressions this allows for very easy and intuitive evaluation of a DA variable.

Remark 6. When evaluating any DA object it is important to keep in mind that it is a Taylor expansion and hence a local representation of the function in some neighborhood of the origin. The actual radius of convergence within which an expansion yields sufficiently accurate results depends on the function that is expanded, the expansion point at which it is expanded, the computation order (together determining the constant C),

as well as of course the accuracy requirements. As in all numerical computations, sometimes even how it was computed plays a role.

Function Concatenation

The evaluation of DA expressions is in no way limited to single points. Instead of arguments of type double, any arithmetic type that supports addition and multiplication by a double can be used to evaluate polynomials. In particular, the DA representation of a function can again be evaluated with DA expressions for the independent variables. This is the equivalent of function concatenation, that is computing and approximation of $(f \circ g)(x) = f(g(x))$ provided an DA representations of f and g.

Lines 14 and 15 illustrate this by evaluating the DA representation of the function $f(x) = x/(1+x)$ with the DA representation of the function $\sin(x)$ to obtain a DA representation of the concatenated function $\sin(x)/(1+\sin(x))$. Line 16 shows a particular example of the use of function concatenation which replaces all occurrences of the first independent variable by the second, that is it changes $f(x)$ into $f(y)$.

However, also in the general case of function concatenation the question of convergence remains relevant:

Remark 7. Evaluating the Taylor expansion $\mathcal{T}_f$ of any function f with another Taylor expansion $\mathcal{T}_g$ of a function g such that $g(0) = 0$, that is $\mathcal{T}_g$ has no constant part, yields the exact Taylor expansion $\mathcal{T}_{f\circ g}$ of the concatenation $f \circ g$ (up to floating point error).

This is, however, not true in the general case where $\mathcal{T}_g$ has non-zero constant part. If, in particular, already the constant part of $g(x)$, that is $g(0)$, is too far from the expansion point of f for its Taylor expansion to converge well there, then the expansion of $f \circ g$ will not converge well anywhere.

Evaluation with Several Argument

In order to evaluate DA vectors with more than one independent DA variable, it is necessary to pass a C++ `vector` (essentially a C++ array, not to be confused with the mathematical notion of a vector) of the values for each independent DA variable. The function for this kind of evaluation is called `eval()`. Its use is illustrated in lines 18 - 21, showing the declaration of a vector of doubles of length 3 in line 18, followed by the assignment of a value for each element of the vector in line 19. Note that C++ vectors start counting at 0, so `arg[0]` is the value for the first independent DA variable (i.e. `DA(1)`), `arg[1]` the value for the second (i.e. `DA(2)`) and so on.

As before for the evaluation with a single scalar, the evaluation can be performed with any arithmetic data type as long as it is the same for every argument. The C++ vector must simply be declared to be of the correct type and the correct length. If the vector contains fewer elements than there are independent DA variables in the DA expression evaluated, the missing elements are assumed to be 0. In any case, evaluation returns a single scalar of the same type as the elements of the vector as a single DA object represents a function $\mathbb{R}^v \to \mathbb{R}$.

Remark 8. For repeated evaluation of the same DA polynomial it is not efficient to use the `evalScalar()` or `eval()` routines of the `DA` class. In order to evaluate polynomials they are compiled into an internal format more suitable for efficient evaluation. By caching the result of this compilation significant savings of several orders of magnitude in computational time can be obtained. However, for the purpose of this introduction we will ignore this consideration.

3.1. Exercises

1. (*) Compute the Taylor expansion of $f(x) = e^{-x^2}$ and plot the resulting polynomial by evaluating it over a grid of points around the expansion point. Compare the plot to the correct pointwise function values. Try with different orders and expansion points to see the effect.

2. Compute the Taylor expansion of the sombrero function

$$f(x,y) = \sin\left(\sqrt{x^2+y^2}\right)/\sqrt{x^2+y^2}$$

around the expansion point $x_0 = (1,1)$ and evaluate it over a grid of $N \times N$ points in the interval $[-1,1] \times [-1,1]$. Plot the resulting surface along with the result of pointwise evaluation of $f(x,y)$ to estimate where the expansion converges. Repeat for different expansion points and computation orders.

3. Set the computation order to 10. Then like in the example discussed above, compute the expression $\sin(x+2)$ once directly in DA arithmetic, and once by concatenating (evaluating) the Taylor expansion of $\sin(x)$ with the DA representation of the function $x+2$. Compare the two results. What is different between the example and this exercise?

4. (*) Using the error function erf from the previous section's exercises, compute the value of the integral

$$\frac{2}{\sqrt{\pi}} \int_{-1}^{1} e^{-x^2} dx = erf(1) - erf(-1)$$

using different expansion orders and plot the logarithmic error as a function of the computation order (correct result ~ 1.685401585899429).

3.2. Advanced Exercises

1. Instead of increasing the computation order to compute the value of the integral $\frac{2}{\sqrt{\pi}} \int_{-1}^{1} e^{-x^2} dx$, fix the computation order to 9 but split the integration domain $[-1,1]$ into N equally spaced subdomains $D_i = [a_i, a_{i+1}]$ with $a_i = -1 + (i-1)\frac{2}{N}$. Then compute the integral $\frac{2}{\sqrt{\pi}} \int_{-1}^{1} e^{-x^2} dx$ as $\sum_i \frac{2}{\sqrt{\pi}} \int_{a_i}^{a_{i+1}} e^{-x^2} dx$ by Taylor

expanding $erf(x)$ around the center point c_i of each D_i to compute the integrals. Plot the logarithmic error of the result as a function of N (while keeping the computation order constant) and compare the results to the integral using a single expansion from the previous exercise.

2. Repeat the previous exercise with the increased integration domain of $[-2, 2]$. What is the maximum accuracy you can reach using a single high-order expansion?

3. Prove Remark 7.
   Hint: recall that some elements, like x, are nilpotent in DA arithmetic.

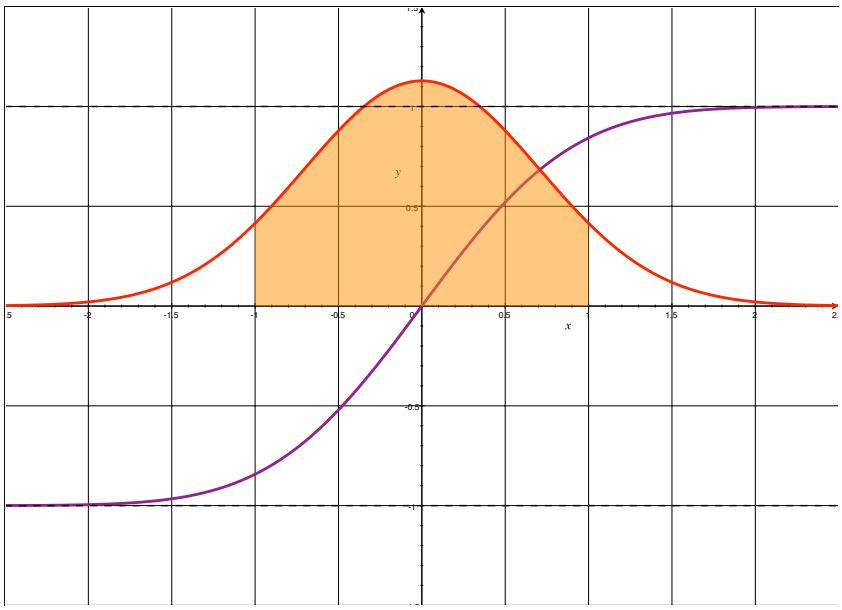

Figure 3.1: The error function (purple) and the Gaussian (red) with the area over the interval $[-1, 1]$ highlighted.

4. Application: An Implicit Equation Solver

Using DA to expand the solution of an implicit equation with respect to a parameter and obtain the solution manifold

```
#include <DA/dace.h>
#include <cmath>
using namespace std; using namespace DACE;

template<typename T> T Nf( T x, T p )
{ return x - (x*x-p)/(2*x); }

int main( void )
{
     int order = 10;
     DA::init( order, 1 );

     double p0 = 1; double x0 = 1;
     DA p = p0 + DA(1);
     DA x = x0;

     int i = 1;
     while( i <= order )
     {
          x = Nf( x, p );
          i *= 2;
     }
     cout << x << endl << sqrt(p)-x;
}
```

In this section, we will study our first application of the DA methods introduced so far. In Section 2, we already implemented a general Newton method working on any function f given as a computer code list. There, we used DA only to automatically compute the derivative of f at each point without the need to manually compute an analytic derivative and code it.

We now take this idea one step further and consider a function $f(x;p)$ where p is a (fixed) parameter. Instead of computing a single value x^* for a given fixed p such that $f(x^*;p) = 0$, we want to compute an expansion $x^*(p)$ that gives the correct root as a function of the parameter p.

That is, we want to use DA to build a parametric implicit equation solver that returns the solution of the equation as a function of parameter(s). This problem is of great practical importance and is a staple of DA algorithms for engineering problems.

Parametric Implicit Equation Solver

We first present the algorithm for a DA based parametric implicit equation solver and then illustrate it using a simple example. However, as will also become clear in the

exercises, this algorithm works for any (sufficiently smooth) parametric implicit equation.

In general, out goal is to solve the parametric implicit equation

$$f(x;p) = 0$$

for a solution $x(p)$ satisfying the above equation. As we are dealing with a DA expansion, the solution $x(p)$ will be expanded around a reference value p_0, that is the solution is given as a polynomial $X(\delta p)$ approximating the exact solution x for the value $p = p_0+\delta p$: $X(\delta p) \approx x(p_0 + \delta p)$.

The basic algorithm to compute this in DA arithmetic is surprisingly simple and consists of only a few steps. Denote by

$$N_f(x;p) = x - \frac{f(x;p)}{f'(x;p)}$$

the Newton operator of the function f.

Naively one may be tempted to simply perform the entire Newton iteration in DA arithmetic and iterate until all coefficients of the polynomial have converged. As the Newton method is contracting, this will eventually happen and a solution could be obtained this way.

However, by making a slight change to this algorithm it is possible to benefit from substantial computational savings and observe one of the marvels of DA. Instead of starting the Newton iteration in DA from any arbitrary initial condition, we start the iteration using a constant part that is already the solution of the implicit parametric equation at the expansion point. Then within a finite number of iterations all orders will converge to the exact solution.

In particular, we perform the following algorithm:

1. Fixing $p = p_0$, compute a pointwise reference solution x_0 such that $f(x_0; p_0) = 0$. This is typically done performing a classical Newton iteration using N_f on the real numbers.

2. Let $[p] = p_0 + \delta p$ and $[x_0] = x_0$.

3. Perform the iteration of the Newton operator N_f in DA arithmetic to compute the sequence $[x_{n+1}] = N_f([x_n]; [p])$.

4. Stop the iteration after computing $[x_n]$ such that $2^n > O$ where O is the computation order. Then the final $[x_n]$ contains the desired solution expanded to order O.

One amazing property of the Newton iteration in DA is that it can be proven to roughly double the number of correct orders in each iteration (as always, this is up to floating point errors). More precisely, if the DA expansion x_n in the nth step is correct up to order k, then after the next iteration the DA expansion x_{n+1} is correct up to order $2k+1$. Thus, as we start with x_0 which has the zeroth order (constant part) already correct,

we have that after the first iteration x_1 is correct up to order $2 \cdot 0 + 1 = 1$, then x_2 is correct up to order $2 \cdot 1 + 1 = 3$, and x_3 already correct up to order $2 \cdot 3 + 1 = 7$.

Note that this is very different from the case of the Newton method on real numbers, where each iteration ideally gets closer to the correct result, but the convergence rate depends strongly on the initial guess and the particular problem considered and the exact solution in general is never attained. In DA, however, for any parametric implicit equations the above method converges to the exact solution in a finite, predetermined number of steps.

Remark 9. We forgo a detailed theoretical study of these methods at this point, but we point out that methods such as these, called strongly convergent iterations, form the core of almost any DA algorithm and in practice are a powerful and efficient tool. The reader is encouraged to observe this amazing behavior in one of the following exercises.

Remark 10. The structure of the algorithm seen here is very common for a large class of DA algorithms. First, the correct reference solution is computed using traditional methods, which forms the zeroth order of the resulting DA expansion. Then it is followed by a DA iteration of a suitably crafted iteration function that computes the additional correct orders.

Example: Square Root

A well known example of the Newton method is the computation of the root of any positive real number p by considering the function $f(x) = x^2 - p$. Considering p as a parameter, the function becomes $f(x; p)$, and the algorithm above allows us to compute the square root of any positive value p.

The goal then is to compute an expansion $x^*(p)$ that gives the correct root as a function of p such that

$$f(x^*(p); p) = x^*(p)^2 - p = 0.$$

In this particular example, we of course already know that the solution $x^*(p) = \sqrt{p}$ and we can thus expand this easily around some reference value $p_0 > 0$ using the built in square root function `sqrt()` of the DACE with what we have learned in the previous sections. But in general, of course, implicit equations do not have a closed form solution that can be expressed as a function of simple intrinsic functions.

In lines 5 and 6 we set up the Newton operator

$$N_f(x; p) = x - \frac{f(x; p)}{f'(x; p)} = x - \frac{x^2 - p}{2x}$$

as a templated function so it can be called with both DA arguments as well as regular double precision numbers.

This is followed by the values of p_0 and x_0 in line 13. In this particular example, we already know that the solution of the equation $f(x, 1) = 0$ is $x = \sqrt{1} = 1$ so for simplicity we hard code these values here. In general, of course, the reference solution x_0 corresponding to p_0 has to be found first (see exercises).

This is followed by the setup of $[p]$ and the initial $[x]$ in lines 14 and 15, which lead to the actual iteration of the Newton operator in lines 18-22 until the correct order has been reached.

In line 23 we output the resulting expansion as well as the difference to the correct reference result $\sqrt{[p]}$ to see if the iteration worked.

Remark 11. Note that in this application we manually computed the Newton operator N_f and hence the derivative of f, unlike we did in the previous exercise. Combining both, the automatic derivation and the expansion in terms of parameters, is also possible. But it is a bit more technical so for demonstration purposes here we assume that the Newton operator N_f is known explicitly.

4.1. Exercises

1. (*) Verify that using the correct zeroth order solution as the initial condition is necessary by using a "wrong" initial condition.

2. (*) Verify the strong convergence of the iteration by looking at the residual in each iteration step.

3. Implement a *naive* DA Newton method starting from any initial condition, and use `abs()` of the difference between two iterations as the criterion for ending the iteration.
   Compare the number of iterations required compared to the strongly convergent method for different p_0 (e.g. $p_0 \in \{0.1, 1, 10\}$) and initial guesses (correct and not correct).

4. Expand $x(p) = \sqrt{\cos(p)}$ *without* using the built-in square root routine (you may use `cos`).
   Hint: the parameter $[p]$ used in the iteration can of course itself be any DA, not just $[p] = p_0 + \delta p$.

4.2. Advanced Exercises

1. (*) Extend the example code in this section to a full DA Newton solver according to the algorithm presented. That is, add a traditional double precision Newton iteration to compute the correct constant part.

2. (*) Expand the eccentric anomaly E as a function of the mean anomaly M for fixed eccentricity by solving Kepler's equation.
   Plot the solution as a function of the mean anomaly and compare with a pointwise solver (e.g. using the Newton solver from Section 2.2) at different computation orders. Estimate the convergence radius of the solution based on the plots.

3. Extend the Kepler's equation solver to expand the true anomaly as a function of both mean anomaly M and eccentricity e.

5. Some DA vector calculus

Introducing DA in vector spaces and some simple uses.

```
#include <DA/dace.h>
#include <cmath>
using namespace std; using namespace DACE;

int main( void )
{
     DA::init( 5, 2 );

     AlgebraicVector<DA> x(3), y(3);
     AlgebraicVector<double> z(3);
     x[0] = DA(1); x[1] = 1.0; x[2] = cos(DA(2));
     y[0] = 1.0; y[1] = DA(2); y[2] = sin(DA(1));
     z[0] = sqrt(0.5); z[1] = 0.0; z[2] = sqrt(0.5);

     cout << "Sum:\n" << x+y << "Product:\n" << x*y
          << "Expression:\n" << (2*x+sin(z))/(y+1.0);

     cout << "Dot product:\n" << x.dot(z)
          << "Cross product:\n" << x.cross(y)
          << "Vector norm:\n" << y.vnorm( )
          << "Unit vector:\n" << z.cross(x).normalize( );

     cout << "Derivative:\n" << x.deriv(2)
          << "Integral:\n" << x.integ(1);

     DA f = sin(DA(1))*cos(DA(2));
     cout << "Gradient:\n" << f.gradient( );

     cout << AlgebraicVector<DA>::identity(2);
}
```

DA objects represent functions that map the v-dimensional space $\mathbb{R}^v$ into the one-dimensional space $\mathbb{R}$. That is, they represent polynomials of v variables. In many applications, however, we deal with vector valued functions, that is functions that map $\mathbb{R}^v$ into the p-dimensional space $\mathbb{R}^p$. These functions can easily be represented using p polynomials, one for each component of the function. This is the concept of a DA vector.

DA vectors

A p-dimensional DA vector $[\vec{x}]$ is given simply by its p components $[x_i]$, which themselves are DA objects. It can be seen as a representation of a vector valued function $x : \mathbb{R}^v \to \mathbb{R}^p$.

In the code, this concept of a DA vector is represented by the `AlgebraicVector` class. The word "*Algebraic*" in the name indicates that this vector class, unlike the C++

standard `vector` class, is actually able to perform algebraic operations such as addition and multiplication. All those operations are performed component-wise, that is, not just the vector addition is defined in the canonical way, where each component is added to the other, but also all other binary algebraic operations are implemented the same way. That is

$$\vec{x} \circledast \vec{y} = \begin{pmatrix} x_1 \circledast y_1 \\ x_2 \circledast y_2 \\ \vdots \\ x_p \circledast y_p \end{pmatrix}$$

where $\circledast$ is any one of the operations $+, -, \cdot, /$. Furthermore, also all intrinsic functions such as sin, cos, exp, etc. are defined the same way, that is

$$f(\vec{x}) = \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_p) \end{pmatrix}.$$

You may be familiar with this kind of operations from other programming languages such as MATLAB. These features allow for very convenient notation in programs and also in some cases for very efficient code as it can make use of vectorization features of the hardware.

In lines 9 and 10, we declare the variables `x`, `y`, and `z` to be AlgebraicVectors of length 3. The AlgebraicVector class is not just a class for vectors with DA components. In fact, the components of an AlgebraicVector can have any (arithmetic) type. When declaring an AlgebraicVector, the type of its component must be given in angled brackets like `AlgebraicVector<double>` (a vector containing double precision elements) or `AlgebraicVector<DA>` (a vector containing DA elements). In our declaration, `x` and `y` are vectors with DA elements, while `z` is a vector of doubles.

Vector Algebra

Lines 11-16 show some basic algebraic operations on `AlgebraicVector` objects. First we initialize the elements of the new `AlgebraicVector<DA>` to represent the functions

$$[\vec{x}] = \begin{pmatrix} \delta x \\ 1 \\ \cos(\delta y) \end{pmatrix}, \; [\vec{y}] = \begin{pmatrix} 1 \\ \delta y \\ \sin(\delta x) \end{pmatrix}$$

and the `AlgebraicVector<double>` to the vector

$$\vec{z} = \begin{pmatrix} \sqrt{\frac{1}{2}} \\ 0 \\ \sqrt{\frac{1}{2}} \end{pmatrix}.$$

Note that in C++ the counting of the elements starts at 0, so we initialize components $0, 1, 2$ of the vector. This is done like for regular C style arrays, that is by appending the index of the element to access in square brackets to the variable name, such as `x[0]`.

In lines 15 and 16 some simple arithmetic operations are being performed on the vectors, such as computing their sums, products, and some complicated algebraic expression. Each of the results is printed to the screen. This is done exactly the same way as for single DA objects: using the stream insertion operator `<<`. All of these computations are performed component-wise, and it is possible to mix `AlgebraicVectors` defined for different algebraic types as long as the types are compatible (i.e. if you can compile `x[0]*y[0]` also `x*y` will compile). This is illustrated in line 16 where the algebraic expression involves both vectors of `DA` and `double` elements. In the same expression, also the mixing of scalar `double` values and `AlgebraicVector` objects is illustrated. The operations are still performed component-wise, e.g. adding the same scalar to each component.

Vector Operations

Objects of the AlgebraicVector type also support some typical vector operations as shown in lines 18-21. The `x.dot(y)` routine computes the dot product of AlgebraicVector $[\vec{x}]$ with AlgebraicVector $[\vec{y}]$, while `x.cross(y)` computes the cross product $[\vec{x}] \times [\vec{y}]$ in the special case of two vectors of length 3. The `x.normalize()` routine returns a vector parallel to $[\vec{x}]$ but of unit length, while `x.vnorm()` returns the usual vector norm. Operations can easily be performed one after the other by concatenating the operations with a `.` (dot) as shown in line 21. In any of these routines, if the dimension of the vectors does not agree an exception is thrown.

Remark 12. Be careful not to confuse `vnorm()` and `norm()`, they are very different routines. The first is the usual vector norm, the second is a special DA norm corresponding to the function of the same name in the `DA` class. It is only useful in advanced DA algorithms, and we will not describe it here.

While the above mentioned methods are available for AlgebraicVectors with components of any algebraic type, some special routines that only have meaning within the context of DA components are only defined for `AlgebraicVector<DA>`. In particular, this applies to the following routines.

To compute derivatives of the vector valued function represented by $[\vec{x}]$ the routine `x.deriv()` can be used. Its syntax is the same as for single `DA` objects, and it returns a new `AlgebraicVector<DA>` containing the derivative of each component with respect to the given independent DA variable number as illustrated in line 23. Analogously, there is the `x.integ()` function, which allows integration of each component of $[\vec{x}]$, returning again an Algebraic vector representing the indefinite integral as illustrated in line 24.

Another useful routine related to AlgebraicVectors is the `gradient()` routine of a single DA object representing a function f, which returns an AlgebraicVector containing

the gradient of the function f, that is

$$\nabla f = \begin{pmatrix} \partial_1 f \\ \partial_2 f \\ \vdots \\ \partial_p f \end{pmatrix}$$

where ∂_i stands for the derivative with respect to the ith independent variable. This is illustrated in lines 26 and 27.

Lastly, the AlgebraicVector class provides a convenience routine to return the k dimensional identity function, that is a vector of DA elements of the form

$$[\vec{id}] = \begin{pmatrix} [\delta x_1] \\ [\delta x_2] \\ \vdots \\ [\delta x_k] \end{pmatrix}.$$

This is achieved by simply calling `AlgebraicVector<DA>::identity(k)` as in line 29. Note that, of course, you cannot request an identity of dimension k larger than the number of independent DA variables set in the call to `DA::init()`.

Remark 13. Calling any of the DA specific routines on any other type of AlgebraicVector will cause a compiler error.

5.1. Exercises

1. Compute the two tangent vectors as well as the normal vector of a surface given by $\begin{pmatrix} x \\ y \\ f(x,y) \end{pmatrix}$ as a function of x, y.
   Use the function
   $$f(x,y) = -0.3 \cdot \left(x^2 + \frac{y^2}{2} \right) + 0.8 \cdot \exp\left(\frac{x}{2} + \frac{y}{4} \right).$$

2. (*) Implement the equations of motion of the (uncontrolled) inverted pendulum as given in Appendix D.2 using AlgebraicVector objects for input and output.
   Add an extra argument for the value of the control u and include it in the equations of motion.

3. (*) Obtain the linearized equations of motion of the inverted pendulum around the equilibrium point using DA and compare with the analytical result (see notes of the control lab).

5.2. Advanced Exercises

1. The function $f(x, y)$ given in the previous exercise on the domain $[-1, 1]^2$ is used to model the surface of a roof of one of the EXPO 2015 pavilions.
   Calculate how hot the different parts get at noon when the sun is shining right down along the z axis. Make an intensity plot of the solar flux.
   Hint: the solar flux is proportional to the dot product of the direction of light and the surface normal.

2. Create a two-dimensional vector of DA that maps the domain $[-1, 1]^2$ onto the set of points satisfying both of the following inequality constraints:

$$\begin{aligned} y &\leqslant 1 - x^2 \\ y &\geqslant x^3 - x. \end{aligned}$$

   Hint: interpolate linearly between (x, y_1) and (x, y_2) using the expression

$$y(t) = \frac{y_1 \cdot (t + 1) + y_2 \cdot (1 - t)}{2}$$

   for $t \in [-1, 1]$.

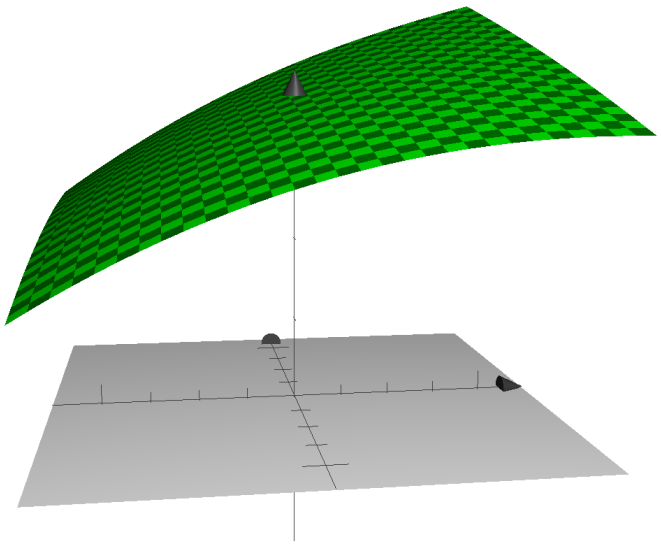

Figure 5.1: Our test surface, doubling as an EXPO 2015 pavilion roof.

6. ODE Integration (Runge-Kutta)

Using DA to expand the solution of an ODE with respect to initial conditions and system parameters

```
#include <DA/dace.h>
#include <cmath>
using namespace std; using namespace DACE;

template<typename T> T euler( T x0, double t0, double t1,
                              T (*f)(T,double) )
{
    const double hmax = 0.001;
    int steps = ceil((t1-t0)/hmax);
    double h = (t1-t0)/steps;
    double t = t0;

    for( int i = 0; i < steps; i++ )
    {
        x0 = x0+h*f(x0,t);
        t += h;
    }

    return x0;
}

template<typename T> T rhs( T x, double t )
{ return x*cos(t+x); }

int main( void )
{
    DA::init( 20, 1 );

    double y = 1.0;
    cout << "Initial:\n" << y << endl
         << "Final:\n" << euler( y, 0, 10, rhs ) << endl;

    DA x = y + 0.5*DA(1);
    cout << "Initial:\n" << x
         << "Final:\n" << euler( x, 0, 10, rhs );
}
```

In this section we will introduce DA algorithms for the solution of initial value problems of an ordinary differential equation (ODE). Mathematically, the problem is phrased in the form

$$\begin{cases} \frac{d\vec{x}}{dt} &= f(\vec{x},t) \\ \vec{x}(0) &= \vec{x}_0. \end{cases}$$

That is, given the right hand side (possibly non-autonomous) vector field $f(\vec{x},t)$ and an

initial value $\vec{x}_0$, find $\vec{x}(t)$ such that it satisfies the ODE.

There are many traditional numerical algorithms to solve this problem for a single initial value expressed as a double precision number. We will introduce instead a DA variant of a group of these algorithms, allowing the automatic expansion of the solution of the ODE at any given time in terms of e.g. the initial conditions or system parameters. These algorithms form one of the most important applications of DA to a vast range of problems in engineering and science. A single DA integration of the ODE, for example, is often sufficient to propagate an entire set of points instead of just a single initial condition at a time.

Euler Integration

We begin with the description of a very simple Euler integrator shown in the code above to illustrate the concept. The forward Euler method is the simplest possible numerical ODE integrator, and it can be viewed as a member of the larger class of explicit Runge-Kutta (RK) style integrators. The basic idea of the forward Euler is to approximate the solution $\vec{x}(t+h)$ of the ODE after a small time step of size h linearly using the equation

$$\vec{x}(t+h) \approx \vec{x}(t) + h \cdot f(\vec{x}(t), t). \tag{6.1}$$

It requires one evaluation of the right hand side and then just assumes that the derivative over the entire time step h remains constant. By performing N of these simple Euler steps one after the other starting with $\vec{x}_0 = \vec{x}(0)$, one finally arrives at the approximate solution $\vec{x}_f \approx \vec{x}(N \cdot h)$ at time $t_f = N \cdot h$. It can be shown that this method is consistent[2], which means that as $h \to 0$ the single step error also goes to zero. Furthermore, the method is 0-stable, which means that also the cumulative error between the approximation $\vec{x}_f$ and the correct result $\vec{x}(t_f)$ of the method goes to 0 as $h \to 0$ and $N \to \infty$.

Remark 14. In real applications, you should never user this integration method. It is a first order method, which means that for an accurate final state $\vec{x}_f$ very small step sizes h are required. This causes huge computational overhead compared to other RK methods (see Appendix C.2), that provide much faster convergence and sometimes also automatic step size control, and are thus far superior in both performance and accuracy. However, the Euler method is useful for illustrative purposes to understand how the DA Runge-Kutta ODE integration process works.

First, we observe that each integration step consists of two main operations:

1. The evaluation of the right hand side f of the ODE at $\vec{x}(t)$ and t,
2. Simple arithmetic operations between the value of f and $\vec{x}(t)$.

Assume now that the initial condition $\vec{x}_0$ is given as a DA object instead of as a double precision number. From the previous exercises we already know that evaluating a function $f(\vec{x}, t)$ given by its code list using DA arithmetic is straight forward (provided it is

[2]Fun fact to know and tell: This is the *only possible* consistent one stage RK method.

sufficiently often differentiable at the origin). Even more trivial are the multiplication by a scalar, h, and the addition to the current state $\vec{x}(t)$.

We conclude that it is conceptually no problem to evaluate the single step of the Euler integration scheme given in Eq. 6.1 in DA arithmetic instead of double precision arithmetic. Then, of course, there is also no problem in iterating the Euler step until reaching the final time t_f, yielding a DA expansion $\vec{x}_f$. This is the basic idea behind the fixed step-size DA Euler method.

Remark 15. These observations of course also holds for the more complex case of higher order Runge-Kutta schemes, as well as various other explicit integration schemes (e.g. explicit linear multistep methods such as Adams-Bashforth). Even methods with automatic step size control can be evaluated in DA arithmetic with at most minor modifications to the algorithms.

DA Implementation

Before we have a closer look at the meaning and uses of the DA expansion of the final state, we will study the C++ implementation shown in the code above. As can be seen, implementing a DA version of the actual Euler integrator is very simple. The function `euler` defined in lines 5-6 contains the entire Euler integrator for any type `T` of initial conditions (as usual using a C++ template). It takes the initial condition `x0` and initial time `t0` as well as the final time to propagate to as arguments. The last (rather cryptic looking) argument `T (*f)(T, double)` on line 6 is a C++ function pointer. It allows passing a function for the right hand side to the `euler` routine. Inside, the `euler` routine, the function that is passed can simply be called as `f(x,t)` where `x` is of type `T` and `t` is a `double`, and it returns a value of type `T`.

This is exactly what is done in line 15, which contains the single Euler step as in Eq. 6.1. It evaluates the right hand side and then updates the state in `x0` to contain the new state after one time step. The time `t` is updated in line 16 which completes the time step. This code is wrapped in a for loop that performs `steps` single Euler steps. Line 19 finally returns the resulting state.

Various constants are declared in lines 8-11. This includes the fixed (maximum) step size `hmax` to be used (line 8), as well as the number of steps `steps` that need to be performed (line 9). Obviously with a fixed step size only times spans that are multiples of the step size can be reached. To allow arbitrary time intervals $[t_0, t_1]$ to be integrated, the number of steps in `steps` and effective step size `h` (line 10) are chosen such that $h < h_{max}$ and $h \cdot steps = (t_1 - t_0)$.

In order to complete the program, in lines 22 and 23 we define a simple scalar right hand side for the ODE

$$\dot{x} = f(x,t) = x * cos(t + x).$$

Note that the function for the right hand side is again templated, and has a definition of the form `T rhs(T x, double t)` which matches exactly the definition of the function pointer in the definition of `euler`. This is why we can subsequently pass this function to the `euler` routine as the right hand side in the main program (lines 31 and 35).

Because we declared everything using templates, we can now use the exact same code to propagate both double precision numbers (i.e. single points) as well as DA initial conditions. In lines 29-31 the initial value $x_0 = 1$ is propagated from time $t_0 = 0$ to time $t_1 = 1$ and the result is printed. In line 33, instead, we propagate the DA expression $[x] = 1 + \frac{\delta x_1}{2}$ over the same time interval and the result is again output. For $\delta x_1 = 0$, i.e. at the expansion point, both the DA expansion and the pointwise propagation must agree. This can easily be checked in the output by comparing the constant part of the second DA integration to the result of the pointwise propagation before.

Remark 16. The ability to use the same integrator for many data types is an extremely useful feature made possible by the combination of C++ templates and careful design of the DACE classes. In fact, since `T` can represent any type, the exact same integrator code can also be used for ODEs of any dimension by simply using vectors of the correct type, such as `AlgebraicVector<double>` or `AlgebraicVector<DA>`, as introduced in Section 5. This makes it trivial to integrate also higher dimensional ODEs by only providing an appropriate function for the right hand side without unnecessary code duplication in the integrator (see exercises).

Remark 17. Obviously, a lot more work is required to develop an industrial strength DA integrator. Typical integration schemes used in real applications are 7/8 order RK pairs with automatic step size control like the Dormand-Prince method. Apart from technical details addressing the differences in measuring the norms in the step size controller, a lot of effort is spent designing the interface to make it simple to use yet powerful and flexible enough to support a wide range of use cases all the while not sacrificing too much performance.

Computing the State Transition Matrix and Higher Derivatives

So what exactly is this expansion of $\vec{x}_f$ that is obtained as the result of the DA ODE integration and how can it be useful? It turns out depending on how one looks at the problem there are several ways this can be useful.

The first and simplest use that comes to mind is to automatically compute the derivatives of the final state $\vec{x}_f$ with respect to the initial condition $\vec{x}_0$. To do so, the initial condition is set up to be the (double precision) reference point $\vec{x}_{ref}$ at which the derivatives of the flow are to be computed plus the DA identity, that is

$$[\vec{x}_0] = \vec{x}_{ref} + \begin{pmatrix} [\delta x_1] \\ \vdots \\ [\delta x_v] \end{pmatrix}.$$

Then after propagation each component of the final state $[\vec{x}_f]$ is a polynomials of the independent DA variables $\delta x_1, \ldots, \delta x_v$, that is

$$[\vec{x}_f] = \vec{P}(\delta x_1, \delta x_2, \ldots, \delta x_v)$$

where the vector over $\vec{P}$ indicates that this is a vector with a polynomial in each component. This represents the Taylor expansion of the result of the Euler method, which,

after all, is just a very long and complicated function of the initial condition $\vec{x}_0$. As the Euler method itself is again an approximation of the real final state $\vec{x}(t_f)$ of the ODE, we have obtained a polynomial approximation of the final state $\vec{P}(\delta x_1, \ldots, \delta x_v) \approx \vec{x}(t_f)$ for the initial condition $\vec{x}_0 = \vec{x}_{ref} + (\delta x_1, \ldots, \delta x_v)^T$.

That means that in order to know where an initial condition $\vec{x}_0' = \vec{x}_{ref} + \Delta\vec{x}$ will end up at time t_f, it is sufficient to evaluate the polynomial $\vec{P}(\Delta x)$ and as long as Δx is within the convergence radius of $\vec{P}$ this will yield the final state $\vec{x}_f'$. Furthermore, as $\vec{P}$ is a Taylor expansion, it of course as usual contains all the derivatives of $\vec{x}_f$ with respect to the δx_i up to the computation order as we have seen in Section 2. In particular, this means that $\vec{P}$ contains the first order derivatives

$$\partial_j \vec{x}_{f,i} = \frac{\partial \vec{x}_{f,i}}{\partial \delta x_j}$$

which the reader will recognize as the entries of the State Transition Matrix

$$STM = \begin{pmatrix} \partial_1 \vec{x}_{f,1} & \partial_2 \vec{x}_{f,1} & \cdots & \partial_v \vec{x}_{f,1} \\ \partial_1 \vec{x}_{f,2} & \partial_2 \vec{x}_{f,2} & & \\ \vdots & & \ddots & \\ \partial_1 \vec{x}_{f,v} & & & \partial_v \vec{x}_{f,v} \end{pmatrix}.$$

So we see that using DA the computation of the STM is nearly trivial. In particular, there is no need to derive and implement variational equations or integrate huge systems of ODEs! Just set the initial condition as above and then take derivatives of the final result using the DA derivative operator. Note that if only the STM is required, a DA computation of order 1 is sufficient, which can be performed very efficiently even for systems with a large number of variables.

Remark 18. It is important not to confuse the order of a Runge-Kutta method with the expansion order of the DA arithmetic. Both have nothing to do with each other. The order of an RK method refers to the order of the time expansion implicitly performed inside each step of the RK algorithm, and it only determines how the one step error scales with the step-size h. The DA expansion order, on the other hand, is only related to the expansion order in the DA variables, the accuracy of which is only affected by the correct choice of the step-size h. In particular, this means that it is possible to compute arbitrary order expansions of the final state even with a first order RK method such as the Euler method.

Advanced Topic: Propagation of Sets of Initial Conditions

Another way to use the flow expansion is to efficiently propagate entire sets of initial conditions through any given dynamics using a single DA integration instead of thousands of individual pointwise integrations. Taking this idea further, we will now devise a method to propagate entire sets of points using DA integration.

Classical Set Representation

To begin with we take a step back and have a look at mathematical sets in general. The main problem when talking about a set of points is how to represent it. Mathematically, it is rather easy to define even complicated sets, however there is an important problem with such set definitions in applied mathematics: it is often very difficult to actually compute an element of a mathematically well defined and useful set. Many sets have complicated definitions and can be shown to be non-empty, but it is still not easily possible to even obtain a single element, much less all elements. Consider for example the set

$$X = \{ x \in \mathbb{R} | \, f(x) = 0 \}$$

for some continuous function $f : \mathbb{R} \to \mathbb{R}$ such that $f(+\infty) > 0$ and $f(-\infty) < 0$. This set is well defined, and non-empty as f has at least one root in $\mathbb{R}$ by the Intermediate Value Theorem. We don't even need to know f in order to be able to know these things about the set. And yet, even for a given f the definition of the set is abstract in the sense that it defines properties of the elements without actually naming them.

What is needed is a way to actually represent sets on the computer. The naïve way found often in simple numerical applications is discretization, that is using some algorithm to compute a finite number of elements of the set, and then use the finite set X' containing all of them instead of X. This is exactly what you do when you put a grid on a domain in order to then perform operations on each grid point in order to deduce some insight about the behavior also between the grid points.

This method works for simple grids in 2 or sometimes 3 dimensions, but it suffers from many problems. The entire structure of the set is lost, instead it is just approximated by a cloud of points. The spacing between the points required to obtain a good approximation is unknown a priori, and finally the curse of dimensionality makes the number of points grow rather quickly. Just as an example, to sample a 6 dimensional phase space typical for mechanical problems with a measly 10 points in each direction already yields a set of 10^6 or one million points taking up over 45.7 megabytes of memory in double precision.

DA Set Representation

With DA, we want to go another route. Instead of storing single points close together, we will use the DA expansions to represent sets of points. To do so, we do not need to do much as it turns out. The most important change is a change in our view of what a DA expansion represents. This will be a conceptual change only, there is no change whatsoever in the code, algorithms or definitions we used so far. The only difference is in our head as we change perspective a bit to look at the same DA objects from a different angle.

Instead of merely considering the DA objects as functions, we make one important "mental" change by attaching a domain to each independent DA variable δx_i. The choice of this domain is arbitrary, but for numerical and practical reasons $[-1, 1]$ is a good choice and commonly used in the field. Now the value of each δx_i is "limited" to $[-1, 1]$. Note that this is not a strict limit enforced anywhere in the code, of course the

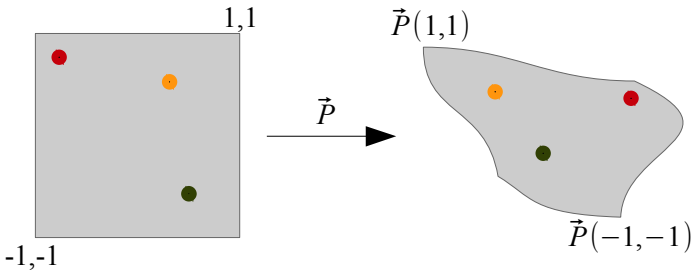

Figure 6.1: Interpreting a DA expansion as a representation of a set.

DA expansion can still be evaluated with any values for each δx_i. The domain is merely a conceptual limit.

What do we gain from this? Consider any vector of polynomials $\vec{P}$ with n components and of v independent variables. This function $\vec{P}$ then maps the set $D = [-1, 1]^v$, called its *domain*, into a corresponding set of points $R(\vec{P}) \subset \mathbb{R}^n$, namely its range (see Figure 6.1). For every point $\vec{y} \in R(\vec{P})$ there now is some point $\vec{x} \in D = [-1, 1]^v$ such that $\vec{P}(\vec{x}) = \vec{y}$. This mapping is not necessarily unique, there could be several $\vec{x}$ mapping into the same point $\vec{y}$. We now have associated the set of points $R(\vec{P})$ with the polynomial $\vec{P}$, because the set $R(\vec{P})$ is parametrized by $\vec{P}$. We can now use the function $\vec{P}$ to fully describe the set of points $R(\vec{P})$. Note that for mathematicians, this idea is conceptually very close to the concept of a *manifold* (albeit with just a single chart).

This representation of sets has various advantages:

1. It is relatively compact in the sense that it allows the representation of an infinite set using only the coefficients of the Taylor expansion
2. It maintains a certain analytical structure not present in mere point sets (e.g. tangents to the set)
3. As a DA function representation, it can be used in all of the DA arithmetic we introduced so far.

The third property is the most important one in practice. It allows the propagation of entire sets through a function by means of simple DA arithmetic. All that is needed is the DA set representation $[X]$ of the starting set X. Then to find out how X transforms under some function f, all that is needed is a simple DA evaluation $[Y] = f([X])$. The resulting DA set representation $[Y]$ now automatically represents the set of points given by

$$Y = \{ f(x) | \, x \in X \} .$$

Of course this is, as always, subject to the convergence of the expansion of f over the domain $[-1, 1]^v$. So if the function f is too strongly non-linear over the initial DA set representation $[X]$, then the final DA set representation $[Y]$ is likely to be a bad representation of the real set Y. However, as before, in some neighborhood of the expansion point of the set, the description is accurate.

So how is this used in practice? Say you have a certain set of initial conditions of an ODE that you can describe by a DA set representation. Typically, you will have Gaussian uncertainties attached to the initial conditions, so for simplicity you can take as the initial condition the set that encloses all the points within $-\sigma$ to σ of the mean of the associated Gaussian distribution. This set is easy to represent as it is just a box of the form $[x] = x_{mean} + \sigma \cdot \delta x_1$. Then you perform the DA integration on this initial condition set, and you obtain a final DA expansion $[x_f]$.

This final expansion over the domain $[-1, 1]$ represents the result of propagating all points within 1σ of the mean through the ODE. You can evaluate $[x_f]$ on a grid in order to visualize it, or you can evaluate it at randomly sampled points based on the initial distribution in order to perform a Monte-Carlo simulation without having to integrate each sample separately. But depending on what you want to do, you could also use it for post processing, for example by performing some optimization process in order to find the minimum distance between the set and a given point in space.

Or you can simply bound the resulting expansion to obtain estimates of the worst case scenarios for e.g. a controller. This is what we will do in Section 7. But in order to prepare for that, we first give a few exercises on DA integration.

6.1. Exercises

1. (*) Replace the Euler integrator by the explicit mid-point rule integrator (see Appendix C.2).

2. (*) Implement a model of the (uncontrolled, i.e. $u = 0$) inverted pendulum using the fixed step size mid-point rule integration algorithm.
   Propagate an initial condition, e.g. corresponding to the pendulum being pushed slightly out of equilibrium, using `double` and `DA` for a couple turns and plot the results to check if it works.

3. Propagate the two dimensional set $[1, 3] \times [-1, 1]$ in the dynamics given by the equation
   $$f(\vec{x}, t) = (1 + \alpha \cdot |\vec{x}|) \cdot \begin{pmatrix} -x_2 \\ x_1 \end{pmatrix} \tag{6.2}$$
   for $\alpha = 0$ from $t_0 = 0$ to $t_1 = 2\pi$. Plot the propagated set every $2\pi/6$ time units and inspect the final state. What do you notice about the set and the final state expansion?
   Now try the same for different values of α such as $\alpha = 0.01$, $\alpha = 0.1$, or $\alpha = 1$ and compare.

4. Compute the State Transition Matrix (STM) from $t_0 = 0$ to $t_1 = 2\pi$ of the ODE in the previous exercise at $(1, 1)$ using DA for different values of α.
   Does the accuracy of the STM depend on the value of α?

5. Still in the same ODE as the previous exercise, instead of expanding the solution with respect to the initial condition now fix the initial condition to be $(1, 1)$. Let

the parameter α be a DA within the interval $[0, 0.1]$ and compute the final state after time 2π as a function of α.

6.2. Advanced Exercises

1. Implement a fixed step size 4^{th} order Runge-Kutta integrator (RK4, see C.2) for DA initial conditions.

2. Using either the simple Euler or the RK4 to propagate the set of points satisfying the constraints

$$\begin{aligned} y &\leqslant 1 - x^2 \\ y &\geqslant x^3 - x \end{aligned}$$

for various times t and $\alpha = 0$ in the rotational dynamics given in Eq. 6.2 and plot the resulting sets.
Hint: See exercise 2 in Section 5.2.

3. Implement the circular restricted three-body problem (CR3BP, see Appendix D.3) and compute the State Transition Matrix required for the exercise on stable manifolds in the second AstroNet school in Glasgow using DA (you should use the RK4 for this to be sufficiently accurate in reasonable time).
Reminder[3]: the mass parameter used at that school was $\mu = 0.30404234 \cdot 10^{-5}$, the period is $T = 3.05923$, and the initial point on the periodic orbit is $x = (0.9888426847,\ 0,\ 0.0011210277,\ 0,\ 0.0090335498,\ 0)^T$.

4. Redo exercise 3 in Section 6.1, but this time using a different parametrization of the initial condition that is more suitable for the right hand side in Eq. 6.2. For the same values of α as above, what is the accuracy you can obtain now?
Note: you don't have to use the exact same initial condition box as in the previous exercise, just chose one of comparable size.
Hint (engineers): it's a rotation with radially changing angular velocity.
Hint (mathematicians): action/angle.

5. In the same setting as the previous exercise, compute the STM at $(1, 1)$ as a function of the parameter α.
Hint: to compute the STM as a (high-order) expansion of α you must perform the STM computation at full order and then evaluate the derivatives at the origin.

6.3. Very Advanced Exercise

This exercise is only recommended if you are already familiar with DA and/or want to skip Section 7 on PID control to work on this instead. Furthermore, for this exercise no solution is provided.

[3] Courtesy of Marta who had to remind me :)

1. Continuing exercise 3 in Section 6.2, (re)do the complete exercise from the AstroNet summer school in Glasgow. That is, compute the stable and unstable eigenvectors ($\vec{v}_s$ and $\vec{v}_u$ of the STM and then propagate them forward, taking snap shots along the orbit. In DA, you can do this elegantly by setting your initial condition to be e.g. $\alpha \cdot \vec{v}_u \cdot [\delta x_1]$ where $0 < \alpha << 1$ is a very constant (such as $\alpha \sim 10^{-5}$). Then at regular time intervals you can just evaluate the linear part to obtain the propagated tangent vector. After normalizing and scaling by α, it forms an initial conditions for the manifold point generation.
   Hint: in order to compute the eigenvalues/eigenvectors of the STM you can use an external C++ library such as the eigen library (http://eigen.tuxfamily.org/). It is *not* covered in these exercises, but is made available on the your system.

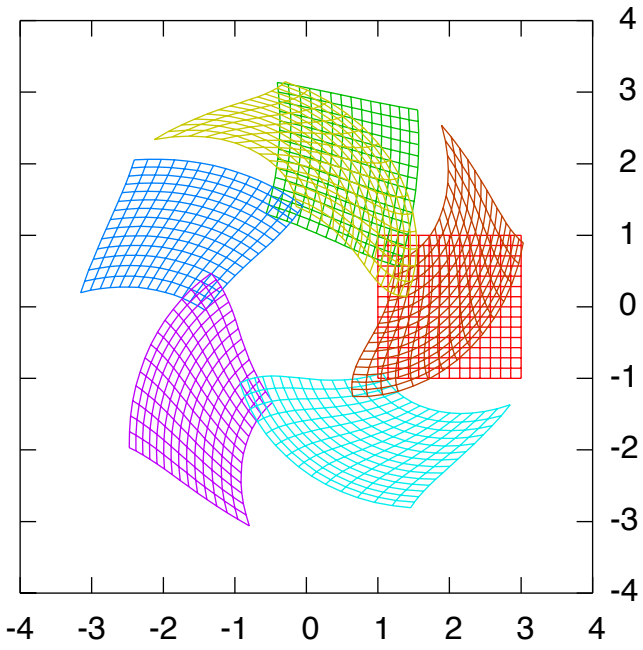

Figure 6.2: Propagation of an initial condition in DA set representation in the dynamics of Eq. 6.2 for $\alpha = 0.1$.

7. Application: A PID Controller

Analyzing the behavior of a PID controller on the inverted pendulum using various of the previously introduced DA techniques

```
#include <DA/dace.h>
#include <cmath>
using namespace std; using namespace DACE;

int main( void )
{
     DA::init( 20, 2 );
     DA x = DA(1); DA y = DA(2);
     DA func = sin(x/2)/(2+cos(y/2+x*x));
     Interval b = func.bound( );

     cout << "func" << endl << func << endl;
     cout << "Bounds:" << endl
          << "[" << b.m_lb << "," << b.m_ub << "]" << endl;
}
```

In this section you will combine the DA concepts introduced in the previous sections to perform a full, non-linear simulation of a PID controller of the inverted pendulum. Before we start with the simulation, we introduce one more feature of DA that is very useful in practice: the bounding of the range of a DA polynomial.

DA Range Estimation

After computing the DA representation of a function, it is often of interest to obtain bounds on the range of the function. That way, all possible outcomes given an input in a certain range can be easily estimated. For example, bounds on the final position of a system can be given based on the range of inputs or the variation of system parameters.

One of the features of DA is the ability to obtain such (outer) estimates of the range easily. The function `bound()` of a DA object returns an upper and lower bound of the polynomial represented by the DA object over some domain for each independent DA variable in the expression. As was the case before, the domain $[-1, 1]$ for each independent DA variable is used by convention.

This bounding process of a polynomial $P : \mathbb{R}^v \to \mathbb{R}$ then yields two numbers, b_l and b_u which satisfy the inequality

$$b_l \leqslant P(\delta x_1, \delta x_2, \ldots, \delta x_v) \leqslant b_u$$

for any $(\delta x_1, \delta x_2, \ldots, \delta x_v) \in [-1, 1]^v$.

Remark 19. It is not guaranteed, and in fact will not be the case for almost all non-linear polynomials P, that the values of b_l and b_u are assumed on the domain $[-1, 1]^v$. That is to say, b_l is not the minimum of P on $[-1, 1]^v$ but a (pessimistic) lower bound for the minimum and the interval $[b_l, b_u]$ overestimates the actual range of the polynomial P.

This makes it particularly suitable for e.g. stability analysis as any point in the range of the polynomial over the domain $[-1,1]^v$ will always be contained within the bounds returned[4]. How well the bounds of the DA polynomial represent the bounds of the original function the DA approximates is as always dependent on how well the Taylor expansion in the DA represents the actual function at the selected computation order.

To compute the bound of a DA variable in the code is simple. As shown in line 10, an object of the `Interval` data type is used to hold the two values b_l and b_u. The interval enclosure of any DA variable can be computed by calling the `bound()` member function as shown in the same line. In order to access the two values within the `Interval` data type, one simply accesses the `m_lb` and `m_ub` fields, corresponding to b_l and b_u respectively, as is shown in line 14.

Remark 20. One of the advantages of the conventional domain $[-1,1]$ for independent DA variables is that it provides a simple way to represent an reference value with an associated error. One can simply use an expression of the form $x_0 + \Delta x \cdot \delta x$ as the initial condition for the computation, where x_0 is the reference value, and Δx is the maximum error with respect to the reference value to be considered. In code, a typical initial condition or value for a system parameter thus looks like `x=x0+dx*DA(1)`. This expression over the domain $[-1,1]$ for `DA(1)` will then cover the interval $[x0-dx, x0+dx]$.

Simple PID Controller Simulation

The basic task of any closed loop controller is to return the value of a control input based on the observed state of the system with the goal of bringing the system into a certain state (the *set point*). A particular controller that is very commonly used is the PID controller (see Appendix D.4). It does not require any explicit knowledge of the system, but has three generic parameters that need to be tuned according to the system.

In our simple model of the PID controller, we will assume the control exerted by the PID controller is piecewise constant. The controller itself can change the value of the applied control only at a fixed frequency, for example 50 Hz. This is to simulate the processing time required by a real world controller to acquire the current system state, process the control, and return the new control to be applied. Thus, in this example the PID controller updates the control u based on the current system state every $\Delta t = 20$ ms.

In the simulation, the state of the system is simulated by propagating it through the dynamics of the inverted pendulum. In a simple loop, the state is propagated for Δt using the fixed value control u, then the PID controller update of u is performed based on the current state, and the loop starts again until the final simulation time is reached or the control of the system is lost (e.g. when the pendulum has fallen over).

This allows the simulation of the performance of the PID controller for a given set of parameters and initial condition. As this is a simulation and not a real time control, at every step of the code additional operations can be performed to monitor and analyze the state, such as plotting the time evolution of state and control.

[4] At this point, the bounds form rigorous outer bounds neglecting floating point error.

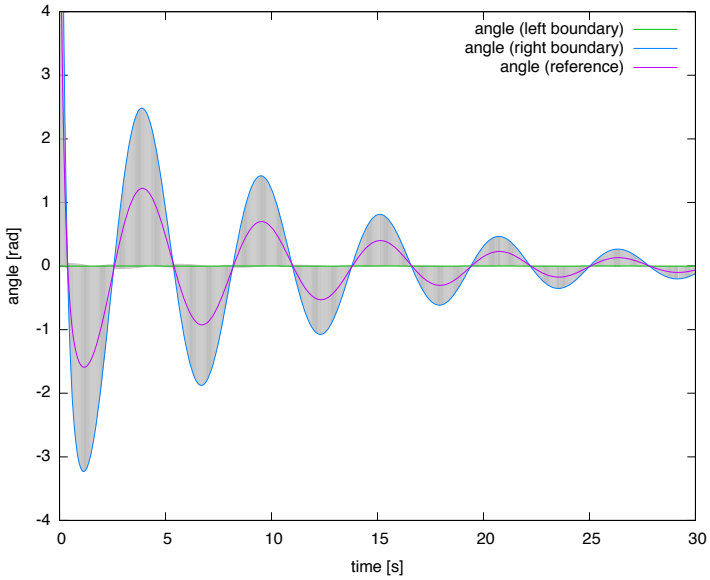

Figure 7.1: Evolution of a range of initial conditions over time with bounds on the angle. There is still some tuning to do on the PID control here.

Remark 21. Note that we are using the full, non-linear (and, in principle, also time dependent) system evolution in our simulation. This sets it apart from commonly used theoretical techniques for determining control stability which is based on transfer functions (obtained via Laplace transformations). There, typically only a linear and time-invariant system is considered.

As we have already seen in the previous sections, the entire system evolution can be implemented and simulated using DA. Also the controller is of a very simple nature and as such can be easily implemented in DA. It is therefore possible to perform the entire simulation using DA.

This way, it is possible to, for example, perform the simulation not just for a single initial condition but an entire range of initial conditions. Using the polynomial range bounding at each controller time step, the evolution of the whole set of initial conditions can be analyzed and even bounds on the worst case state can be obtained (see Figure 7.1). Or the dependence of the controller on various system parameters, such as the masses or the length of the pendulum can readily be expanded and studied.

This is the content of the exercises in this section.

Advanced Topic: Saturation of the Control

When left unchecked, the controller algorithm may decide that a control value should be applied that is not realistically possible due to the constraints of the system. In case of the inverted pendulum, for example, the maximum force that can be applied is limited by the motor pulling the cart. This requires clamping the output of the controller to some

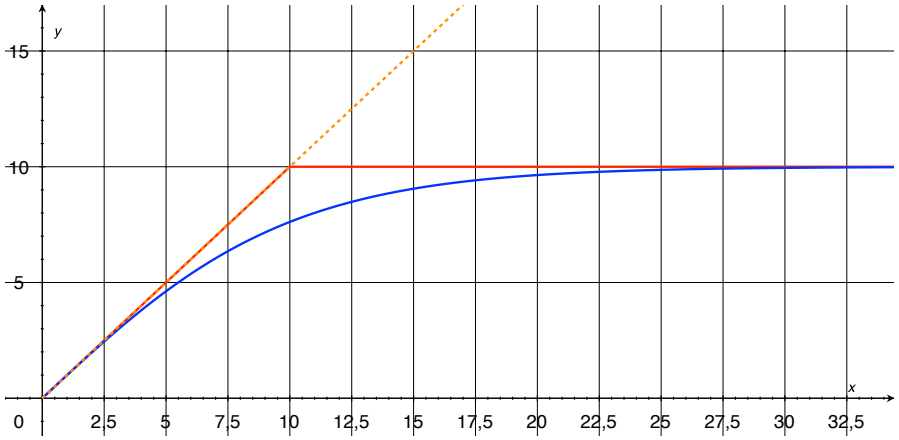

Figure 7.2: Smooth saturation via tanh (blue) versus hard cutoff (red).

reasonable values based on the physical specifications of the system. This is known as saturation of the controller. In reality there are many factors to consider, as saturation can occur in various ways. For example, instead of just hitting a hard upper limit, the increase in the force generated by a motor ΔF when increasing the current by a fixed ΔI may simply become smaller and smaller at higher currents (soft saturation).

However, to simplify the modeling of the system, we will assume that there is some hard maximum u_{max} for the control u that the controller cannot exceed. Mathematically, this corresponds to taking the raw output u^* of the controller, and subjecting it to the saturation function S to obtain the actual u:

$$u = S(u^*) = \min(u^*, u_{max}).$$

Unfortunately, this function S is not differentiable at u_{max} and hence cannot be Taylor expanded at that point. Furthermore, any Taylor expansion of S at a different point will fail to approximate the function on the other side of the switching point.

This means that in DA there cannot be a function such as min which returns the minimum of a DA object and a number or another DA object. Consider the raw control output $u^*(p) = u_0 + u_1 p + u_2 p^2 + \dots$ as a polynomial of some expansion variable p. What we would need is some new polynomial expansion $u(p)$ satisfying $u(p) = \min(u(p), u_{max})$ for any p, or at least for any p in the range of the parameter it represents. But as we just saw, such a polynomial does not exist.

In order to still limit the output of the control, several things can be done. The easiest is to only look at the constant part of the polynomial. If it is above the limit, we set $u(p) = u_{max}$, else we leave $u(p) = u^*(p)$. This ensures that at least for $p = 0$ the control is within the limits, and if $u^*(p) \neq u_{max}$ then also at least for some small neighborhood around $p = 0$ the same is true. Unfortunately, in practice this does not work well as we want to be valid for as large a range of p as possible.

A more clever trick is to modify the saturation function S to make it smooth so we

can expand it more easily. One simple possibility for this is using the function

$$\tilde{S}(u^*) = \tanh(u^*/u_{max}) \cdot u_{max}$$

instead of the original function S. This is a smooth function that asymptotically has the same behavior as S but differs near the saturation point. Its behavior is shown in Figure 7.2 (left). For small and large values of u^* is will behave very similar to the hard cutoff S, but near the cutoff errors tend to grow with a maximum of about 25% error right at the cutoff.

Since the cutoff is typically anyway selected somewhat arbitrarily and with an error margin added to it, this behavior in many applications can be acceptable. For our simulation it will certainly be sufficient.

7.1. Exercises

1. (*) Write a simulator for a PID controller as introduced in this section and connect it it with the model of the inverted pendulum.

2. (*) Using your simulator from the previous exercise, plot the time evolution of θ for some initial values θ_0 and tune the parameters of the controller to obtain stability for 100 seconds.

3. (*) Change your controller to perform all operations using the DA data type and expand the initial angle as a DA of the form $[\theta_0 + \Delta\theta \cdot \delta x_1]$ with $\Delta\theta \approx 5$ deg.
   Verify that your controller still achieves stability for 100 seconds for all initial conditions in the interval $[\theta_0 - \Delta\theta, \theta_0 + \Delta\theta]$ by outputting and inspecting the final state $[\theta_f]$.

4. (*) Using the DA propagator, plot the time evolution of the envelope (minimum and maximum) and reference (center point) of θ for the above interval.

5. Compare the DA bounds for the function `func` in the example to the bounds obtained by pointwise sampling of the function for various domains.

7.2. Advanced Exercises

1. (*) Add saturation control using the saturation function $\widetilde{S}$ introduced in this section to both methods.
   Compare the minimum and maximum applied force as well as the convergence of the controller to the previous case.

2. (*) Fixing the initial condition to a single value, propagate an error in the balanced mass of 10% and again plot the time evolution of the envelope of the angle.

3. (*) Expand with respect to various parameters of the system and initial conditions, play with it, and have fun.
   For example, you could try expanding with respect to the gain K and mass m and use the result to find the best K for each mass.

A. Notes on Differential Algebra

In this Appendix we briefly introduce the main idea behind Differential Algebra (DA). Instead of spending a lot of time with theoretical definitions and implementation details, we just focus on the idea and then let the practical examples and exercises above speak for themselves.

But before introducing the concept of DA itself, we start with a reminder of how math with "real" numbers works on a computer.

Floating Point Numbers

The infinite set of real numbers $\mathbb{R}$ on a computer are, by necessity, represented on a computer as a finite approximation called the *floating point numbers* $\mathbb{F}$. Each floating point number represents a certain real number x in a specific form, namely as

$$x = \pm m \cdot 2^e$$

where m is an integer called the mantissa and e is an integer called the exponent, both taken from some finite range of values[5]. Of course it is obvious that not all real numbers have an exact floating point representation, in fact almost all need to be approximated.

These floating point numbers are then equipped with a set of operators and intrinsic functions that mimic the behavior of the real numbers within $\mathbb{F}$. That is, each operation on $\mathbb{F}$ is defined in such a way that its result is another floating point number that approximates the result of the corresponding operations performed in the real numbers. For example, the exponential function in a floating point implementation will compute a result in $\mathbb{F}$ that for every input approximates the correct (real) result on $\mathbb{R}$ by another floating point number.

With this, it is then possible to evaluate even long and complicated mathematical equations and expressions in floating point arithmetic to obtain some floating point number as a result. Since each single operation in the chain leading to the result was such that it approximates the real solution, one expects by induction that the final result will also approximate the result that would have been obtained using real numbers in $\mathbb{R}$. However to determine a priori if that is really the case is in general difficult and requires precise knowledge of the exact representation of floating point numbers and the exact operations performed during the computation. But, as we all know, the general approach is to "just try" and often times the distinction between the floating point approximation and the real result are completely ignored as the results are "good enough".

This process is illustrated in Figure A.1, where the evaluation of the expression $\frac{1}{x+1}$ is illustrated for $x = 2$ once in the real numbers $\mathbb{R}$ (top) and once in a (hypothetical) decimal floating point number system $\mathbb{F}$ with 4 significant digits (bottom). As usual, the evaluation of the full expression is split into a sequence of basic operations to be performed. We start with 2, the value of x, which in both systems is represented correctly.

[5] Also the base 2 used in this expression in principle can be changed, however on binary computers it is practically always 2.

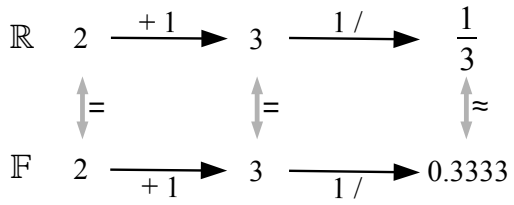

Figure A.1: Step-by-step evaluation of the expression $\frac{1}{x+1}$ for $x = 2$ in the reals (top) and floats (bottom).

Then the operation +1 (add 1) is performed once in real arithmetic (top) and in floating point arithmetic (bottom). In this case, also the real result 3 happens to be representable in the floating point numbers $\mathbb{F}$ without approximation. Then the operation 1/ (1 over) is performed on this result, yielding the real solution $\frac{1}{3}$ and an approximate solution 0.3333 in the floating point numbers $\mathbb{F}$. As can be seen, the sequence of operations performed in the reals in order to evaluate this expression are mirrored exactly in the floating point numbers. The final result of the evaluation in floating point arithmetic is an approximation of the final result of the real computation.

Taken all together, this is called a floating point environment. It combines the theoretical concept of a floating point number, a particular realization of a floating point number representation, as well as a computer implementation of the operators and intrinsic functions to act on these floating point numbers in the way described above.

Remark 22. The previous example illustrates one more important point that is often overlooked. A mathematical expression such as $\frac{1}{x+1}$ is nothing more than a short way to specify a sequence of operations to be performed in order to arrive at a result. When coded on a computer, the programmer must translate this expression into an appropriate sequence of operations to be performed one after the other by the computer. Modern programming languages support the developer in this endeavor by providing an easy way to write mathematical expressions. But internally, the compiler still translates these expressions into a finite code list of operations to be performed in order.

Differential Algebra

Just like real numbers, also functions can be operated on in an algebraic fashion. In the mathematical field of functional analysis, the various operators on function spaces are extensively studied. For example, given two functions f, g it is possible to define the product function $f \cdot g$ as the new function which is the point-wise product $f(x) \cdot g(x)$.

The key feature of Differential Algebra is to efficiently represent (certain sufficiently smooth) functions in a computer environment in a way such that they can be easily created, manipulated and evaluated by means of simple arithmetic expressions. In this, the idea is very similar to that of floating point numbers which represent real numbers. DA "numbers", or DA objects as we more commonly will refer to them, on the other hand represent sufficiently smooth, real functions $f : \mathbb{R}^v \to \mathbb{R}$.

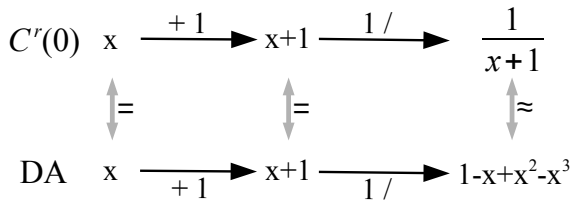

Figure A.2: Step-by-step evaluation of the expression $\frac{1}{x+1}$ in the function space C^r (top) and DA arithmetic (bottom).

Each DA object has a certain internal representation which approximates real functions in a certain way on the computer. Then, the algebraic operators are replaced by specially developed versions acting on DA objects in such a way that the result of an operation is a DA object that once again represents the function after applying the same operation in the real function space. Consider, for example, again the multiplication of two functions $f \cdot g$. If we denote by $[f]$ and $[g]$ the DA representations of these two functions, then the multiplication operator for DA is defined in such a way that the operation $[f] \cdot [g]$ yields another DA representation of the product function $f \cdot g$.

As with floating point numbers, also with DA the representation is an approximation. In fact, internally DA stores the truncated Taylor expansion of f around the origin up to a fixed order (the computation order) of a function. This representation is accurate at the origin, as well as in some neighborhood around it. The size of this neighborhood is, as with floating point numbers, hard to estimate a priori (or even a posteriori), but can easily be validated by comparing the values of the resulting Taylor expansion with those of pointwise evaluation of the original function.

Once more, this process is illustrated in Figure A.2, where the evaluation of the same expression $\frac{1}{x+1}$ as before is illustrated. Now, however, we compare the evaluation in the space $C^r(0)$ of real functions that are r times continuously differentiable at 0 (top) and a (hypothetical) DA arithmetic of one variable and of maximum expansion order 3 (bottom). In both cases we start with the identity function $f(x) = x$ which in both systems is represented correctly. Then the operation +1 (add 1) is performed on both, once in real function space (top) and once in DA arithmetic (bottom). In this case, the resulting function $f(x) = x + 1$ is again represented without approximation in both systems, as it is a polynomial of order less than 3. Lastly, the operation 1/ (1 over) is performed on this result, yielding the solution in the real function space $\frac{1}{x+1}$ and an approximate solution $1 - x + x^2 - x^3$ in DA arithmetic. While the evaluation of a function with the identity in the real function space yields exactly the original function and thus may seem pointless, the same operations performed in DA arithmetic yielded a polynomial that the careful reader will recognize as the Taylor expansion of $\frac{1}{x+1}$, which therefore by Taylor's theorem approximates the original function locally near $x = 0$ with an error of $O(x^4)$, that is to say quite well. Note that, in particular, at $x = 0$ the value of both functions agrees exactly, as is expected for a Taylor expansion.

The term *Differential Algebra* comprises the complete set of the computer representa-

tion of the actual multi-variate polynomial expansion (DA objects) as well as the set of operators and intrinsic functions provided to operate on the polynomials. As such, it is more than just "polynomials" in the same way as a floating point environment is more than just "a mantissa and exponent".

Remark 23. It is important to separate the idea of *what* DA does from the question of *how* it does it. In the previous illustrative example, we do not concern ourselves with the question of how the 1/ operator in DA arithmetic was able to go from the polynomial $x+1$ to the Taylor expansion $\frac{1}{x+1}$. This is a separate question that for both the conceptual understanding of DA as well as the application of DA to real problems is quite irrelevant. The important part to know is that DA provides such an operator 1/ that for any given polynomial $P(x)$ can compute the Taylor expansion of $1/P(x)$ around $x = 0$.

This is actually no different from the floating point arithmetic case. Also there we did not concern ourselves with the question of *how* exactly the 1/ operator on $\mathbb{F}$ can, for any given floating point representation $x = m \cdot 2^e$, find the corresponding m^*, e^* such that $m^* \cdot 2^{e^*}$ is a good approximation of $1/(m \cdot 2^e)$. But because all of us have learned algorithms to manually divide digit by digit (sometimes called "long division") in school we more readily accept the idea that an implementation of a floating point algebra can actually compute such values for m^*, e^*. However, the reader can rest assured that, despite the obvious difficulties, such algorithms can also be devised and implemented in the DA algebra case.

DA Algorithms and Applications

Just as for floating point numbers, many different algorithms to compute accurate and fast expansions of various problems have been developed over time. In the exercises, some of the basic algorithms that form the building blocks of Differential Algebra applications have already been introduced. Some of the many other algorithms available for DA are:

- *Function inversion* to compute a high-order expansion of the inverse function f^{-1} given only the expansion of f.
- *Implicit equation solvers* to solve implicit equations in any dimension with and without deriving a Newton operator.
- *Constraint satisfaction* to expand the constraint manifold around a point in it.
- *Boundary Value Problem solvers* to compute the solution to two point boundary value problems directly in one step.
- *ODE flow expansion* in time, initial conditions, and parameters using Picard-Lindelöf (aka Taylor), Runge-Kutta or Lie-derivative integrators.
- *Range bounding* to compute range bounds of functions.
- *Manifold Expansion* to compute accurate expansions of invariant manifolds.

- *Normal Form Transformations* to automatically compute high-order normal form coordinate transformations.
- *Global optimization* to obtain rigorous enclosures of fixed points and extrema.

Some areas in which DA has been successfully applied include:

- Accelerator physics
  - Particle accelerator analysis and design
  - Transfer map computation
  - Symplectic particle tracking
  - Magnet design and optimization
- Astrodynamics
  - Uncertainty propagation in celestial mechanics
  - Trajectory Design and Optimization
  - MOID and collision probability computation
  - Orbit Determination
  - Robust optimal control
  - High-order Kalman filtering

B. Notes on C++

C++ is a complex and powerful programming language. It can be a bit intimidating to beginners at first, so here we limit ourselves to the practical aspects of various language features that may be useful for the exercises. For a detailed introduction to the language we refer to the pertinent literature.

Structure

All C++ programs have a structure similar to C programs. The first commands in the source are `#include <HEADER>` commands, that load additional definitions of libraries of functions into the program. Typical header files to include are `cmath` for math functions or `iostream` for input/output. This is then followed the declaration of the various functions of the program. A function is declared as

```
TYPE func( TYPE argument, ... ) { ... }
```

where `TYPE` is the data type of each variable. Unlike many scripting languages, C++ requires the specification of a type for every variable used. The first TYPE is the type of the variable returned by the function, or it can be `void` if the function does not return a value. Arguments to the function are declared as a comma separated list of `TYPE name` pairs following the function name in parenthesis. If the function doesn't take arguments, you just have to put empty parenthesis. The code of the function follows in curly brackets.

Each C++ program must have one function which is called at the beginning of the program declared as `int main( void ) { }`. This is where the code starts to execute when the program is run.

Types and Variables

As mentioned, C++ is a strongly typed language, that means each and every variable has a type attached to it. To C++ a character string is different from an integer number which is different yet again from a double precision floating point number. The most important data types in C++ are `int` for (signed) integers, `double` for double precision floating point numbers, and `string` for strings of characters. In order to include a literal string in your program (e.g. to print it), you have to enclose it in double quotes `"like this"`. Integers are simply written as numbers in the code, like `184`, and floating point constants are written with a decimal point, like `184.0`. We will see later that there is one more important custom data type for us that is a DA object and has the type `DA`.

To declare local variables in a function, at any point in the code you can write the type of the variable you want to declare followed by the variable name such as `int i`. Variables exist within the block of code contained in curly braces `{ }` in which they were defined. Once execution goes outside of that block, the variable disappears and cannot be accessed any more. thus variables declared inside a function only exists there, while variables declared e.g. inside a loop (see below) disappear once the loop ends. Variables

can also be declared outside of any function, in which case they are globally available anywhere in your code.

Statements and Flow Control

C++ code inside functions consists of various statements, each ended by a semicolon `;`. Some simple statements are function calls, which are performed by following the function name with the function arguments in parenthesis: `sin(x)` or `print_result()` if there are no arguments. Assignment of a value of an expression to a variable is performed using the = operator. You can combine various types of statement together to declare a variable, call a function and assign the result all in one statement such as `double result = sin(x);`.

In order to control the flow of your program, you can use various constructs. The most common is the `for` loop, which is a very powerful construct but we will use it only to perform a loop over some code for a certain number of times. To do this, you use the form

```
for( i = 0; i < NUMBER; i++ ) {}
```

where `i` is an integer variable you declared before and NUMBER is replaced by the number of times you want the loop to be executed. Within the loop, the value of `i` will run from 0 to $NUMBER - 1$. You put the code of the loop in the curly brackets after the `for` statement. You can also combine the declaration of the loop counter variable `i` with the for loop itself by writing

```
for( int i = 0; i < NUMBER; i++ ) {}
```

and then you don't have to declare `int i` before and `i` only exists inside the loop.

Another useful loop construct is the `while` loop, which performs the loop while a certain expression is true. It is written as

```
while( expression ) {}
```

where `expression` is some logical expression. Logical expressions are typically formed by the comparison operators `<`, `>`, `==` and can be combined using `||` (or) as well as `&&` (and). For example `(abs(x-x0)<err) && (i<1000)` is true if and only if both $|x - x_0| < err$ and $i < 1000$. The parentheses around the two expressions are not required but make it easier to read complex logical expressions.

In order to branch code depending on wether a logical expression is true or not, you can use the `if` construct. It takes the form

```
if( expression ) { } else { }
```

where the code to execute if `expression` is true (i.e. non-zero) is in the first pair of curly braces and the code to execute if `expression` is false (i.e. zero) is in the second pair. You can chain `if` statements to form a decision tree such as

```
if( expression1 ) {} else if( expression2 ) {} else {}
```

Note the space between `else` and `if`, there is no `elseif` in C++!

Classes

C++ is an object oriented language, that means one of its key concept are classes of objects that share certain features and capabilities. As object oriented programming (OOP) is not as commonly taught as the more classical procedural programming (used in e.g. C or MATLAB), we will restrict ourselves to use just very few features of OOP. All the code used int he examples will be written in procedural style, which is also supported by C++.

One of the idea of OOP is that objects not just encapsulate data describing the object (e.g. various parameters of a controller), but that they also have certain "abilities" to act. For example, the controller object may have the ability to compute the value of the control variable. Asking a C++ object to perform one of the actions it knows is done by writing the name of the variable containing the object followed by a dot and the name of the action (it is called like a function, in fact it is a function inside the object). For example the statement

```
u = controller.getControl( x, t );
```

could be used to extract the state of the control variable `u` given the state `x` and time `t` from a controller object.

In mathematical expressions this usage is uncommon. We are used to writing `sin(x)` and not `x.sin()` in order to ask the variable `x` to compute "its" sine value. However, for certain other operations we will learn about later, it is quite natural to use the OOP notation, for example when asking a function f to compute its derivative with respect to the first variable we can write `df=f.deriv(1)`.

Input and Output

In C++ outputting any value to a file or the screen is done using the C++ stream facilities. In particular, the stream object `stdout` is already defined as a global variable in every C++ program and represents the screen. You can output any variable or expression of (almost) any type using the `<<` stream output operator. For example, to print the value of x followed by $cos(x)$ you could write

```
stdout << x << "   " << cos(x) << endl;
```

As you can see, the `<<` operator can be "chained" to output several things to the same stream. To output a string literally, just enclose it in double quotes. The special global variable `endl` represents the end of the line (output will continue in the next line). As an alternative you can also include the character sequence `\n` in a string and it will be replaced by a new line.

To write to a file on your disk, you use the exact same notation. The only difference is that you use a different stream instead of `stdout`. In order to open an output stream for writing, you declare it as a variable of type `ofstream` like this: `ofstream file;`. Then you can open a file for output by writing `file.open( "filename.txt" )` and when you are done outputting you can close it using `file.close()`. To use `ofstream`

you must also include the <iostream> header in the beginning of your program using the command `#include <iostream>`.

Template functions

As mentioned before, C++ is a strongly typed language. However, often times the exact same code can be executed using different data types. This is particularly the case for mathematical expressions. Consider a typical mathematical function $f(x)$ made up of a finite number of algebraic operations and some intrinsic functions (such as sin or exp), for example $f(x) = \frac{1+\sin(x)}{x^2 \exp(x)}$. As all of these operations are defined on both real and complex numbers, you can evaluate the function $f(x)$ with an argument of either type. If used with complex number, all operations have to be performed using the rules of complex arithmetic, while with a real argument x it is understood that real arithmetic is used. In both cases, however the expression of $f(x)$ does not change.

This is a very powerful concept that is at the core of Differential Algebra methods. Operations on real (or, on a computer, floating point) numbers are replaced by corresponding operations on truncated polynomials. Thus, it would be very nice to have the ability also in C++ to call the same function representing a mathematical expression using either `DA` or `double` precision arguments.

In C++, this is very elegantly achieved despite the strict typing by replacing the function written for a specific data type (such as `double`) by a version using *C++ templates* as illustrated in the following code listing for the function

$$f(x) = \frac{1}{\sqrt{x^3}} sin(x).$$

```
1  // function f callable with double only
2  double f( double x )
3  {
4      double a = 1.0/sqrt(x*x*x);
5      return a*sin(x);
6  }
7
8  // same as function f, but callable with both DA and double
9  template<typename T> T g( T x )
10 {
11     T a = 1.0/sqrt(x*x*x);
12     return a*sin(x);
13 }
```

As C++ templates are a very powerful and complex programming tool, we will simply describe this particular case of their use. As in the example, you first just write your function as you would for the `double` data type. Then you add `template<typename T>` before the function. This tells the C++ compiler that from now on the "dummy type" `T` will stand for any valid C++ data type the function may be called with. You then consequently replace every `double` variable in your code by a variable of type `T`. The compiler will then automatically write a version of the function for any data type it is called with later (e.g. `double` or `DA` or `complex`).

Exceptions

In C++ errors are signaled to the programmer using a mechanism called *exception throwing*. This replaces the error-prone and tedious methods used in classical functional programming of checking the return value of each operation for an error code. Instead, when something goes wrong in the code C++ throws an exception which then "bubbles up" through the code until it is caught by the program. If nobody catches the exception, an error is printed to the screen and the program is terminated.

In order to catch exceptions, you use the try-catch block construct:

```
try { } catch( exception &ex ) { }
```

In the first curly brackets you put the code you want to run that may fail and throw an exception. You can put a lot of code there, for example all of the code in your `main()` function. When an exception happens during the execution of this first block of code, the program will jump to the code in the second curly braces. There you can act on the error, inform the user, clean up and do whatever else is needed. If the first block of code executes cleanly without errors, the second block of code will never be executed.

A typical and very useful piece of code to put in the second block of code is the statement `stdout << ex.what();` which will print a human readable error message describing the exception that occurred to the screen. Most implementations of the C++ library will print exactly this to the screen if an uncaught exception ends the program, so to just see what the error is it is typically not necessary to catch the exception.

Plotting data

In some of the exercises of this lab you will be asked to create simple plots of certain data. While C++ comes with many libraries specialized in plotting data, for simple codes an easy method of plotting is using gnuplot. Gnuplot is an external program that reads data from text files and then allows you to plot the data in various ways.

To write data readable by gnuplot (and most other programs), you can simply output numerical values separated by spaces. A typical output command in a loop would be

```
file << order << "  " << x << "  " << result(x);
```

To then plot this file in gnuplot after running your program, you would type `gnuplot` on the shell and then use a command like `plot 'myfile.dat' using 2:3 with lines;` to make a 2D line plot of the second column on the x-axis and the third column on the y-axis. You can automate this process by writing all your gnuplot commands to a file named e.g. *myfile.plot* and then calling gnuplot with this file as the argument. You can even automate this call by adding the command `system("gnuplot myfile.plot");` at the end of your C++ code, which makes your program call gnuplot automatically to visualize the results while your code is running. Note: you will also have to add the line `#include <cstdlib>` at the beginning of your program if you want to use the `system` function to automatically start gnuplot.

Alternatively, you can also generate the text files and then load and plot them in another program of your choice (like Mathematica, Origin, or MATLAB).

Common pitfalls

The following is a list of some common pitfalls encountered during C++ programming.

- C++ is case sensitive! The variable `X` is different from `x`.
- In `for` loops, be very careful with the bounds. Typically, for loops run from 0 to $N - 1$ in order to obtain N executions of the code. This is because arrays and vectors in C++ are indexed starting with index 0.
- Be careful with the type of numerical constants in your code. To C++, `1` is different from `1.0` as the first is an integer and the second is a double precision number. C++ will use certain sets of rules to "magically" transform one into the other, but it is best to write your constants already of the type you want them to be. In particular, be aware that the expression `7/3` in C++ is exactly equal to the integer `2` as both values are integers and integer division of 7 and 3 is 2 (with a remainder of 1). Instead, you most likely want to write `7.0/3.0`.
- All statements must end with a semicolon, else you get an error!
- Pay attention to the difference between assignment (`x=7`) and comparison (`x==7`).
- Note that more "recently" defined local variables "hide" previously defined more global ones of the same name.

C. Standard Numerical Methods

Some additional information on selected standard numerical methods useful for the exercises is provided here for reference.

C.1. Newton Method

The Newton Method is a numerical method to find the root x_0 of a differentiable function $f(x)$ iteratively. The iteration step of the Newton method is given by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{C.1}$$

C.2. Runge-Kutta Integrators

We report a few simple explicit, fixed step-size Runge-Kutta integration methods here, referring the reader to the pertinent literature for a more complete overview.

Let $\dot{x}(t) = f(x,t)$ be the ODE to solve, while x_n and t_n are the state and time in the n-th step of fixed step-size h.

- The simplest RK method is the Euler method, given by the equation

$$x_{n+1} = x_n + h \cdot f(x_n, t_n). \tag{C.2}$$

- The "mid-point rule" (a second order method) offers a slight improvement over the Euler method and is given by the equation

$$x_{n+1} = x_n + h \cdot f\left(x_n + \frac{h}{2} f(x_n, t_n), t_n + \frac{h}{2}\right). \tag{C.3}$$

- The "3/8 rule" is a fourth order Runge-Kutta integrator originally due to Kutta. The equation for its single step are:

$$x_{n+1} = x_n + h \cdot (k_1 + 3k_2 + 3k_3 + k_4)/8 \tag{C.4}$$

where

$$\begin{aligned} k_1 &= f(x_n, t_n) \\ k_2 &= f(x_n + h \cdot k_1/3, t_n + h/3) \\ k_3 &= f(x_n + h \cdot (-k_1/3 + k_2), t_n + 2h/3) \\ k_4 &= f(x_n + h \cdot (k_1 - k_2 + k_3), t_n + h). \end{aligned}$$

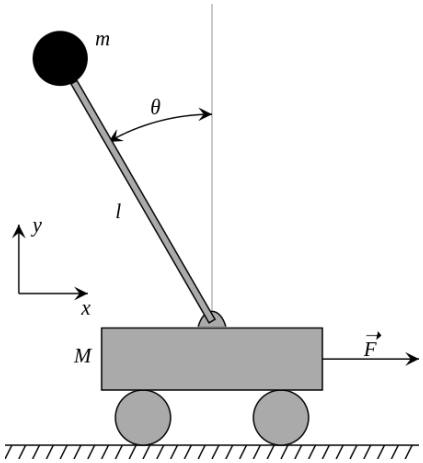

Figure D.1: Inverted Pendulum (Wikipedia)

D. Physics and Engineering

Some additional information on selected physical systems and engineering methods useful for the exercises is provided here for reference.

D.1. Kepler's Equation

Kepler's equation relates the mean anomaly M and the eccentric anomaly E by the equation

$$M = E - e\sin(E) \tag{D.1}$$

where e is the eccentricity of the orbit.

This equation has no closed form inverse, i.e. the function $E(M)$ for a given value of e cannot be written in closed form.

D.2. Dynamics of the Inverted Pendulum

The dynamics of the inverted pendulum without friction are provided below for reference. The definition of the quantities can be found in Figure D.1, the gravitational acceleration on Earth is taken as $g = 9.81\ \mathrm{m/s^2}$.

$$\begin{aligned} (M+m)\,\ddot{x} - m\ell\ddot{\theta}\cos\theta + m\ell\dot{\theta}^2\sin\theta &= F \\ \ell\ddot{\theta} - g\sin\theta &= \ddot{x}\cos\theta \end{aligned} \tag{D.2}$$

See https://en.wikipedia.org/wiki/Inverted_pendulum for more details and derivation.

D.3. The Circular Restricted Three-Body Problem

The equations of motion for the circular restricted three body problem (CR3BP) with the mass parameter μ are given by:

$$\begin{aligned} \ddot{x} &= 2\dot{y} + x - \frac{(1-\mu)(x+\mu)}{r_1^3} - \frac{\mu(-1+x+\mu)}{r_2^3} \\ \ddot{y} &= -2\dot{x} + y - \frac{(1-\mu)y}{r_1^3} - \frac{\mu y}{r_2^3} \\ \ddot{z} &= -\frac{(1-\mu)z}{r_1^3} - \frac{\mu z}{r_2^3} \end{aligned} \tag{D.3}$$

where

$$\begin{aligned} r_1 &= \sqrt{y^2 + z^2 + (x+\mu)^2} \\ r_2 &= \sqrt{y^2 + z^2 + (-1+x+\mu)^2} \end{aligned}$$

D.4. PID Control

A PID controller takes the generic form

$$u(t) = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau)\, d\tau + T_d \frac{d}{dt} e(t) \right) \tag{D.4}$$

where $e(t) = x_{set} - x(t)$ is the error between the targeted set point x_{set} and current value $x(t)$ of the controlled quantity.

The three constants K_p, T_i, and T_d are the proportional gain K_p as well as the integral time T_i and derivative time T_d.

The first and third term of the sum are a linear approximation of the error at time $t + T_d$ with unchanged control u. The second integral term sums all past errors and T_i is roughly the time in which they would be cancelled completely.

The parameters K_p, T_i, and T_d must be chosen based on the controlled system.