Differential Algebra Computational Engine (DACE)

Workshop on Nonlinear Propagation of Uncertainties using Differential Algebra

September 22nd, 2015 *Escape Dance Room*, *ESTEC*

DINAMICA srl
Registered Office:
Piazza della Repubblica, 10 - 20121 - Milano (Italy)
Operational Headquarters:
Via Morghen, 13 - 20156 - Milano (Italy)
Phone +39 02 8342 2930
Fax +39 02 3206 6679
e-mail: dinamica@dinamicatech.com
website: www.dinamicatech.com

Outline

- Purpose of the DACE
- Architecture Design
- Implementation
  - Core
  - Interfaces (C++ & Matlab)
- Performances
- Conclusions

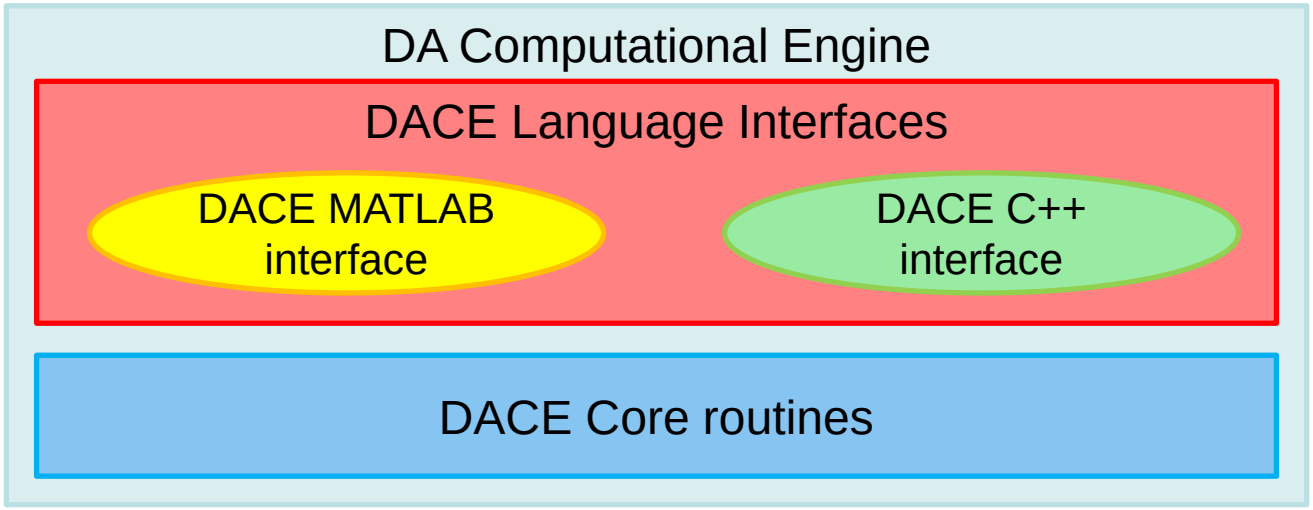
Overview of the DACE Implementation

Purpose of the DACE

- The DA Computational Engine (DACE) is an implementation of basic DA routines
  - Each routine approximates the result of an operation by its Taylor expansion around 0
  - After each operation we obtain an approximation, yielding eventually to the Taylor expansion of arbitrarily complex expressions
- The DACE also provides a user interface to use these routines conveniently
  - Allows writing mathematical expressions in typical computer programming way, and evaluating them using DA and double precision numbers.

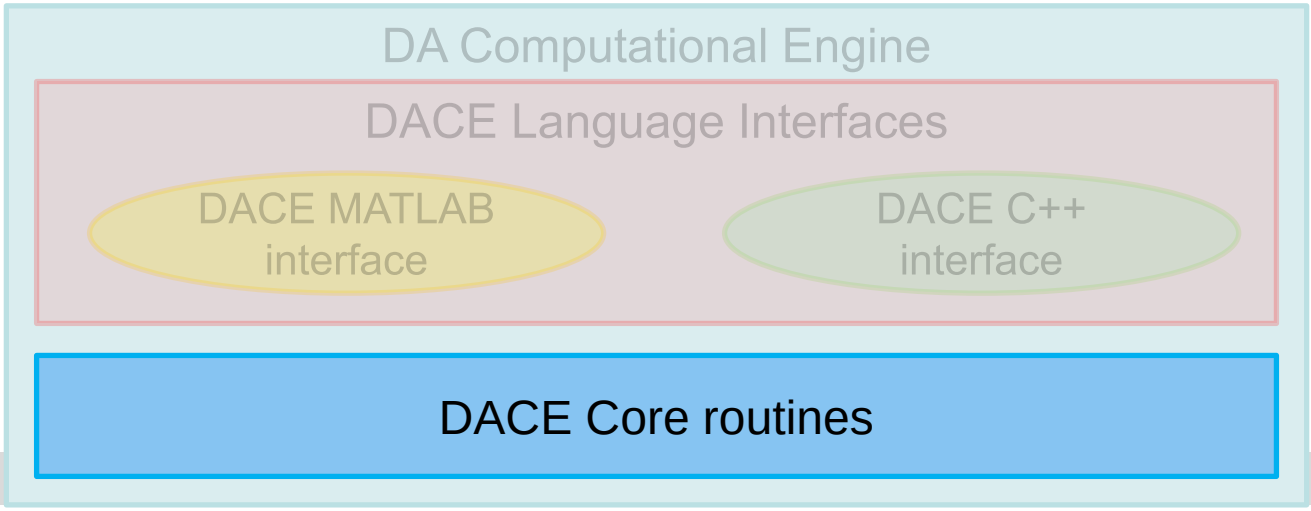
Architecture Overview

- DACE Core routines: Fortran 95
  - Initialization, Memory management, error handling, operations, intrinsic functions, and much more
- Interfaces: C++, MATLAB



DACE core implementation

- DACE core routines
  - Procedural basic DA routines in dynamically loaded library
    - Similar functionality as the standard C floating point math library but for DA
    - Not intended to be used directly by the user
  - Language: Fortran 95 chosen over C99 after tradeoff
    - Fast and efficient implementation
    - Modern memory managment (wrt F77)
    - Available on all considered platform (Win, OS X, Linux)
    - Easy interfaceability to most other languages



DACE core implementation

Public routines currently exported by the DACE core library

Initialization and State
- Query DACE version
- Initialization
- Set cutoff
- Get cutoff
- Set truncation order
- Get truncation order
- Get machine epsilon
- Get max order
- Get max variables

Memory Management
- Allocate DA object
- Deallocate DA object

Error Handling
- Query error state
- Reset error state

DA creation
- Create constant
- Create variable
- Create monomial
- Create random DA
- Create a copy of DA

DA access
- Extract constant part
- Extract linear part
- Extract coefficient
- Set coefficient
- List all coefficients

Input/Output
- Read from strings
- Write to strings
- Export as BLOB*
- Import from BLOB*

Arithmetic Operations
- Addition (DA, FP)
- Subtraction (DA, FP)
- Multiplication (DA, FP)
- Division (DA, FP)
- Integration
- Derivation

Norms & Estimation
- Absolute Value
- Norm
- Order sorted norm
- Order estimation
- Bounding

Evaluation
- Partial plug
- Evaluation tree

Intrinsic Functions
- Square
- (n-th) Power
- Square Root
- Inverse square root
- (n-th) Root
- Truncate
- Round
- Modulo
- Exponential
- Natural logarithm
- Arbitrary logarithm
- Sine
- Cosine
- Tangent
- Arcsine
- Arccosine
- Arctangent
- Arctangent2
- Hyperbolic Sine
- Hyperbolic Cosine
- Hyperbolic Tangent
- Hyperbolic Arcsine
- Hyperbolic Arccosine
- Hyperbolic Arctangent

* BLOB = Binary large object (lossless binary representation of the data in a DA)

DACE core implementation

- Memory management in the DACE core library
  - DA objects are identified by integers assigned by the DACE
  - Memory for DA is transparently (re)allocated within the DACE library
  - No limits on the number of DA objects (except system memory)

- Procedural programing example

```
DACEinit(10,3);
A = DACEalloc(); B = DACEalloc();
DACEvar(A,1);
DACEadd(A,1.0,B); DACEsin(B,A);
DACEprint(A);
DACEfree(A); DACEfree(B);
```

$$A = \sin(1 + x)$$

not supposed to be used directly by end users

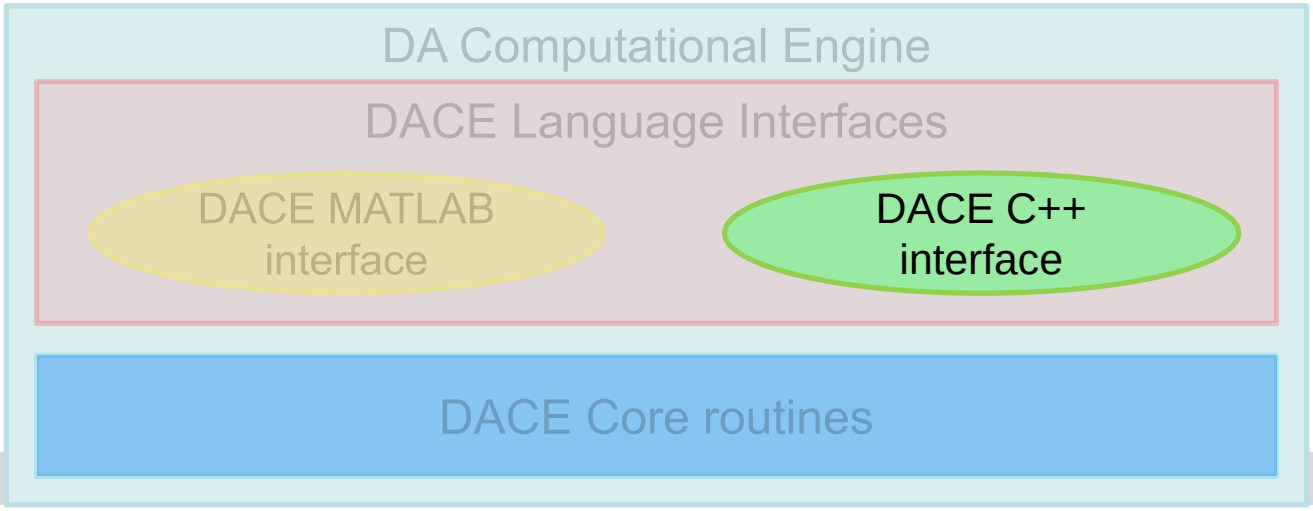
DACE core implementation

- Error handling in the DACE core library
  - Error conditions are signalled by setting an error state and severity
  - Accessible by the interface for handling, details left to the interface
  - All errors are recoverable except out of system memory

- Error severity classes
  - 0: Information, no action required
  - 1: Warning, possibly wrong use of the DACE
  - 6: Error, invalid operation or arithmetic error (returns zero DA)
  - 9: Severe, DACE initialization failure
  - 10: Fatal, out of memory (terminates the program)

C++ interface implementation

- C++ Interface
  - C++ is a mature, wide spread object oriented language
    - Full support for custom arithmetic data types
    - Templates allow reuse of code with different data types
  - Easy to interface directly with DACE core routine library
  - Very little overhead
  - Optimal for full applications with high performance requirements



C++ interface implementation

- Provides access to all DACE core library routines
- Handles DACE internals such as DA allocation and error states
- Blends in with advanced C++ language concepts
  - Strict typing / type safety to avoid coding errors
  - Operator overloading for intuitive coding
  - Namespaces to avoid naming collisions
  - Templates / C++ standard template library (STL)
  - C++ I/O streams for seamless input and output

C++ interface implementation

Routines available in the DACE C++ Interface

Initialization and State
- Query DACE version
- Initialization
- Set cutoff
- Get cutoff
- Set truncation order
- Get truncation order
- Get machine epsilon
- Get max order
- Get max variables

Memory Management
- Allocate DA object
- Deallocate DA object

Error Handling
- Query error state
- Reset error state

DA creation
- Create constant
- Create variable
- Create monomial
- Create random DA
- Create a copy of DA

DA access
- Extract constant part
- Extract linear part
- Extract coefficient
- Set coefficient
- List all coefficients

Input/Output
- Read from strings
- Write to strings
- Export as BLOB*
- Import from BLOB*

Arithmetic Operations
- Addition (DA, FP)
- Subtraction (DA, FP)
- Multiplication (DA, FP)
- Division (DA, FP)
- Integration
- Derivation

Norms & Estimation
- Absolute Value
- Norm
- Order sorted norm
- Order estimation
- Bounding

Evaluation
- Partial plug
- Evaluation tree

Intrinsic Functions
- Square
- (n-th) Power
- Square Root
- Inverse square root
- (n-th) Root
- Truncate
- Round
- Modulo
- Exponential
- Natural logarithm
- Arbitrary logarithm
- Sine
- Cosine
- Tangent
- Arcsine
- Arccosine
- Arctangent
- Arctangent2
- Hyperbolic Sine
- Hyperbolic Cosine
- Hyperbolic Tangent
- Hyperbolic Arcsine
- Hyperbolic Arccosine
- Hyperbolic Arctangent

* BLOB = Binary large object (lossless binary representation of the data in a DA)

C++ interface implementation

- DACE::DA class encapsulates one single DA object
- DACE initialization required to set maximum order and maximum number of independent variables
  - Should happen only once in the very beginning
  - Flushes all currently existing DA objects
  - Computation order can be dynamically adjusted up to max order in code

```
using namespace DACE;
DA::init(10,2);           // order 10, 2 independent variables
cout << DA::getOrder();   // 10
cout << DA::setTruncationOrder(5);
cout << DA::getOrder();   // 5

DA x = DA(1), y = DA(2);
cout << x+y;
```

C++ interface implementation

Mathematical expressions

- All operators are overloaded for DA and double operations
- Independent variables (generators of the algebra) created by simple DA constructor

  `DA(1), DA(2), …`
- Overloaded intrinsic routines available for functional programming style

  `sin(x)` instead of `x.sin()`

Very clear and easy to understand code

```
DA x = -1 + DA(1) + DA(2);    // Creation of DA variables
DA y = sin(x) + 1.9;          // Functional notation
DA z = x.sin() + 1.9;         // Object oriented notation
```

C++ interface implementation

Mathematical expressions

- Notation is virtually identical to built in numerical C++ types (e.g. `double`, `int`) and C/C++ math library
- New code can be written easily
- Existing code can be adapted to DA with minimal changes
  - Does not mean old code will “just run”!
    Non-trivial code always requires some changes due to conceptual differences, not the interface!

```
double f(double x)
{
    double res = (sin(x)+2*x)/(1+cos(x));
    for(int i = 0; i<3; i++) res = sqrt(res);
    return res;
}
```

Existing double code

C++ interface implementation

Mathematical expressions

- Notation is virtually identical to built in numerical C++ types (e.g. `double`, `int`) and C/C++ math library
- New code can be written easily
- Existing code can be adapted to DA with minimal changes
  - Does not mean old code will “just run”!
    Non-trivial code always requires some changes due to conceptual differences, not the interface!

```
DA f(DA x)
{
    DA res = (sin(x)+2*x)/(1+cos(x));
    for(int i = 0; i<3; i++) res = sqrt(res);
    return res;
}
```

New DA code

C++ interface implementation

Mathematical expressions

- Using C++ templates allows maximum code flexibility
- DA class is template safe
- C++ automatically “writes” code for any data type used in function call
- Most versatile, and recommended, way to write functions for use with DA and double

```
template<class T> T f(T x)
{
    T res = (sin(x)+2*x)/(1+cos(x));
    for(int i = 0; i<3; i++) res = sqrt(res);
    return res;
}
cout << f(5.0);      // call f with double
cout << f(DA(1));    // call f with DA
```

Hybrid template code

C++ interface implementation

Input/Output

- Uses human readable formatted ASCII output
- DA class integrates smoothly with C++ iostreams
- Intuitive handling for C++ programmers
- Reads both COSY INFINITY and DACE format for interoperability

```
I  COEFFICIENT                ORDER EXPONENTS
1     1.000000000000000         1   1  0
2    -0.1666666666666667        3   3  0
3     0.8333333333333333E-02    5   5  0
4    -0.1984126984126984E-03    7   7  0
5     0.2755731922398589E-05    9   9  0
-----------------------------------------
```

x^1y^0 x^3y^0 x^5y^0 x^7y^0 x^9y^0

C++ interface implementation

Input/Output

- Uses human readable formatted ASCII output
- DA class integrates smoothly with C++ iostreams
- Intuitive handling for C++ programmers
- Reads both COSY INFINITY and DACE format for interoperability

```
DA x, y;
// ... Computations ...
cout << x << y;                 // print to screen
fstream file(“test.dat”);       // write to file
file << x << y;
file.close();
file.open(“test.dat”);          // read from file
file >> x >> y;
file.close();
```

C++ interface implementation

Error Handling

- Errors are handled via C++ exceptions
- DACEException class provides detailed, human readable error messages
- Errors are recoverable, the DACE can continue computation afterwards

```
DA x = -1 + DA(1) + DA(2);
try {
    cout << sqrt(x);
} catch( DACEException ex ) {
    cout << ex;
}
```

```
Error in ./DACE/cinterface/DA.cpp at line 750
630 DACEROOT: negative or zero constant part for even root
```

C++ interface implementation

Evaluation

- DA transparently compiled into efficient tree structure for evaluation
- Templated implementation in the interface allows evaluation with any C++ arithmetic type
- For high efficiency, compiled tree structure can be cached
- Extremely fast repeated evaluation of same polynomial, very useful for DA Monte-Carlo type simulations

```
DA x = -1 + DA(1) + DA(2);
vector<double> args(2);
args[1] = 0.1; args[2] = -2.3;
double res = x.eval(args);
```

C++ interface implementation

Evaluation

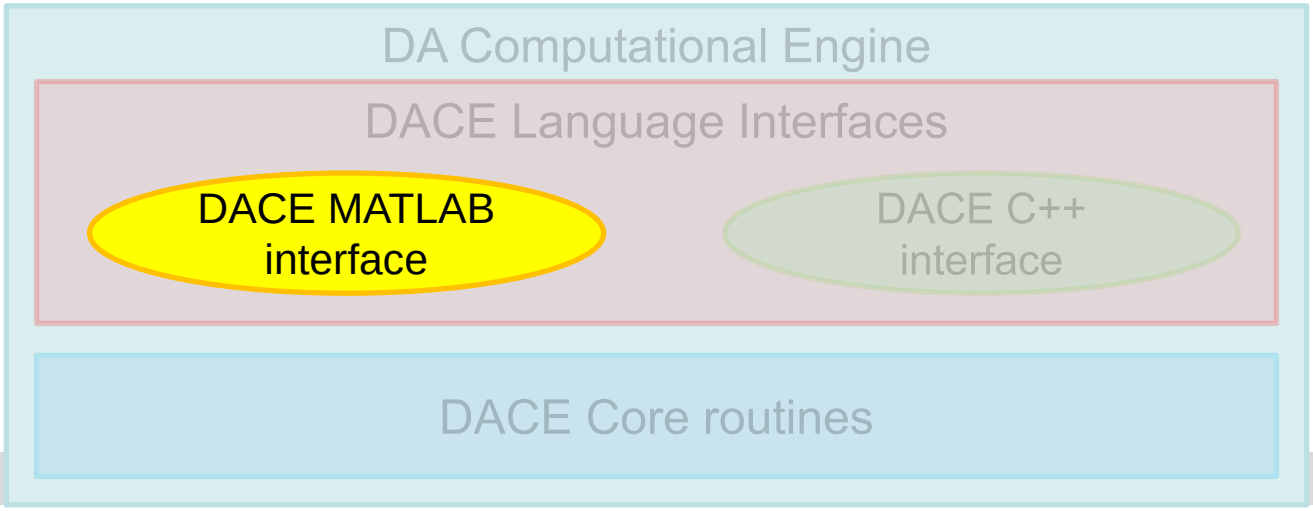
- DA transparently compiled into efficient tree structure for evaluation
- Templated implementation in the interface allows evaluation with any C++ arithmetic type
- For high efficiency, compiled tree structure can be cached
- Extremely fast repeated evaluation of same polynomial, very useful for DA Monte-Carlo type simulations

```
DA x = -1 + DA(1) + DA(2);
vector<double> args(2);
args[1] = 0.1; args[2] = -2.3;
compiledDA cda(x);
double res = cda.eval(args);
```

Huge time savings!

Matlab Interface implementation

- MATLAB Interface
  - MATLAB is popular for scientific computations
    - Easy to use interactive interface
    - Hard to be enhanced by custom arithmetic data types
  - Must interface directly with DACE core routine library
  - Very slow
  - Good for interactive testing of simple DA algorithms



Matlab Interface implementation

MATLAB Features	MATLAB Limitations
Interactive	
Procedural Language	Object Oriented Framework incomplete
Automatic Memory Management - Automatic creation and destruction of variables - Automatic Polymorphism of variables	Only internal data type are supported, limited possibility of user-defined data types
Huge availability of built-in functions	Only internal data types supported
Extensible with external library	Limited support for C and Fortran, No support for C++ or Fortran 2003
Closed source	Only published interfaces available

Matlab Interface implementation

- Object oriented framework to implement a user friendly experience in Matlab
  - New Data Type representing a DA object → Two class types are available: Value and Handle
  - Constructors and Destructor Overloading → Only Handle has a destructor; No copy constructor available
  - Operator Overloading → Assignment operator can not be overloaded
  - Static Member Function

  Copy of class properties is performed → Only Value Class can be used

- Link to external core library
  - Centralized memory management in external library → Not possible with Value class as the destructor is needed
  - Data passing to external library by reference → Possible with “libpointer” but inefficient as the internal memory is not referenced directly but copied to temporary objects

Matlab Interface implementation

Implementation solutions investigated:

- ~~New data type in a C++ mex file~~ → C++ object are supported only inside mex functions, cannot be exported to MATLAB Workspace
- ~~Handle Class with calls to C++ mex file~~
- ~~Handle Class with calls to C mex file~~

  → Mex file support a single function for each C/C++ file, object are lost between successive calls to mex file, so no external storage is possible with mex file
- ~~Handle Class with calls to external DLL~~ → Copy cannot be implemented with handle class, as only pointer to the data is copied, not the actual data
- ~~Value class with calls to C mex file~~ → Object are lost between successive calls to mex file, so no external storage is possible with mex file
- Value class with calls to external DLL → Objects are stored in Value Class and passed to external DLL for operations

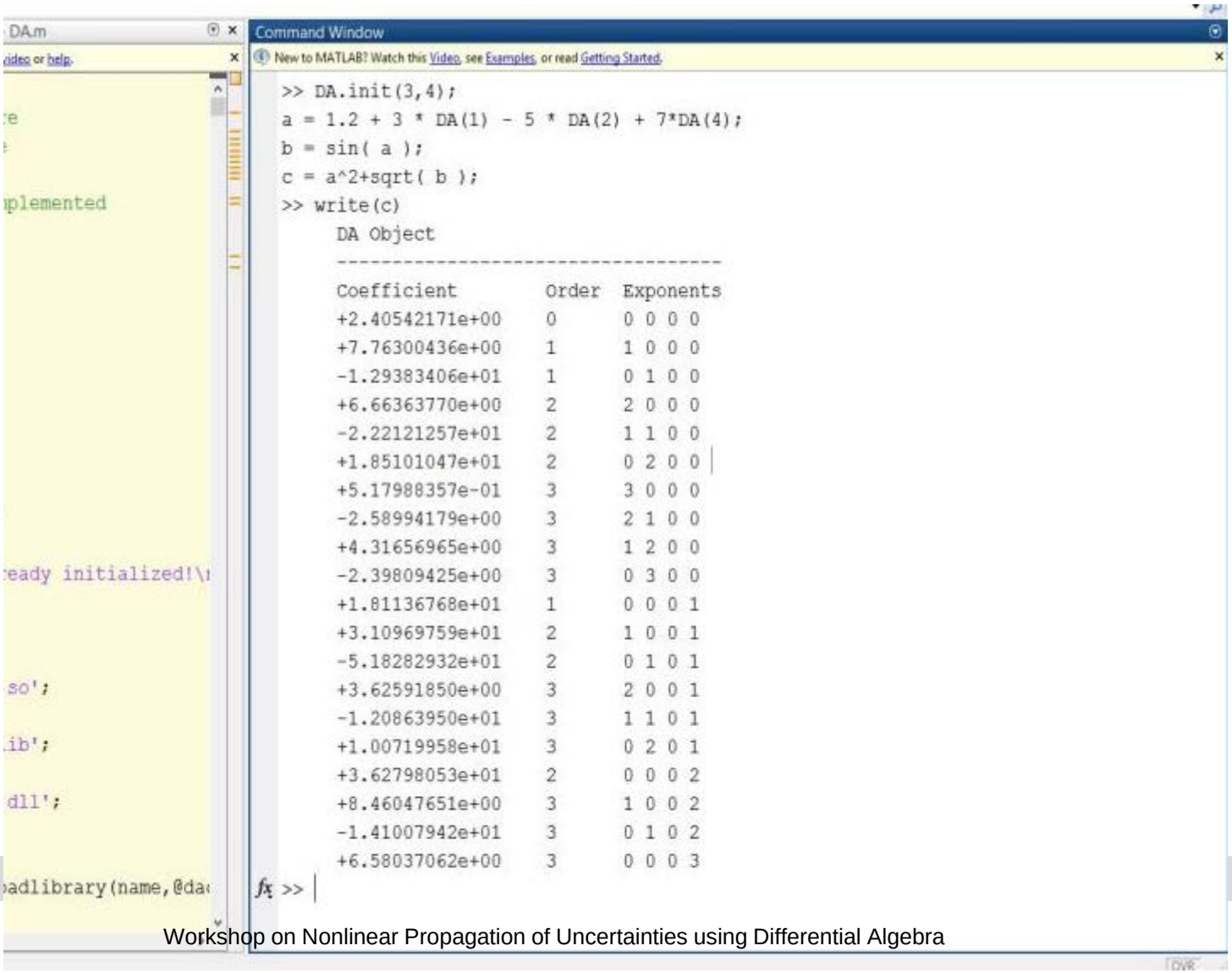
Matlab Interface implementation

Value class with calls to external DLL

- Require use of “loadlibrary” → Supported only if an external compiler is present
- Requires use of “callib” function → introduces a big overhead
- The DA object coefficient are stored inside the Matlab class → Coefficient need to be continuously copied to and from the external DLL
- Requires the use of “libpointer” for retrieving data from external functions → Inefficient implementation which create a pointer to a new memory region with a copy of the original data. Big overhead introduced for copy operations
- Allows the desired syntax:

```
>> DA.init(3,4);                                  % Static method to initialize DA environment
>> a = 1.2 + 3 * DA(1) - 5 * DA(2) + 7*DA(4);     % Assignment using constructor and overloaded operators
>> b = sin(a);                                    % Assignment using overloaded function
>> c = a^2+sqrt(b);
```

Matlab Interface implementation



DACE Testing and Validation

- The DACE has been subjected to an extensive testing regime to assure accurate and correct operation.
- Includes the core routines as well as the C++ interface
- Tests performed:
  - Test 1: DACE Initialization
  - Test 2: Single variable functions
  - Test 3: Multivariate functions
  - Test 4: Advanced tests
  - Test 5: Derivation and anti-derivation
  - Test 6: Convergence radius and polynomial bounder
  - Test 7: Computational efficiency

DACE Testing and Validation

- The most mature DA implementations is COSY INFINITY by Berz and Makino at Michigan State University
  - extensively tested in the field by its wide set of users
  - arithmetic operations validated in the work by Revol et al.
  - over 25 years of active development
- The DACE expansions at various orders and numbers of variables have been validated coefficient by coefficient with the same expansion computed using COSY INFINTIY
  - Both expansions match up to small errors due to floating point errors

DACE Testing and Validation

- DA vs. analytic expansion of selected fundamental functions

TEST ID	Function
2.1	$f(x) = (1+x)^n$, where $n = 1, \dots, 10$
2.2	$f(x) = \frac{1}{1-x}$
2.3	$f(x) = \sqrt[n]{1+x}$, where $n = 2, \dots, 5$
2.4	$f(x) = e^x$
2.5	$f(x) = \log(1+x)$
2.6	$f(x) = \sin(\mathrm{x})$
2.7	$f(x) = \cos(\mathrm{x})$
2.8	$f(x) = \tan(\mathrm{x})$

TEST ID	Function
2.9	$f(x) = \mathrm{asin(x)}$
2.10	$f(x) = \mathrm{acos(x)}$
2.11	$f(x) = \mathrm{atan(x)}$
2.12	$f(x) = \sinh(\mathrm{x})$
2.13	$f(x) = \cosh(\mathrm{x})$
2.14	$f(x) = \tanh(\mathrm{x})$
2.15	$f(x) = \mathrm{asinh(x)}$
2.16	$f(x) = \mathrm{atanh(x)}$

DACE Testing and Validation

- Fundamental identities: Complicated functions that are identically 0

$$f_1(\boldsymbol{x}) = \boldsymbol{x} - \boldsymbol{x} = 0$$

$$f_2(\boldsymbol{x}) = \boldsymbol{x} \cdot \frac{1}{\boldsymbol{x}} - 1 = 0$$

$$f_3(\boldsymbol{x}) = \sqrt[p]{(1+\boldsymbol{x})^p} - 1 - \boldsymbol{x} = 0, \text{ for } p = 1, \dots, 5$$

$$f_4(\boldsymbol{x}) = \log(e^{\boldsymbol{x}}) - \boldsymbol{x} = 0$$

$$f_5(\boldsymbol{x}) = (\sin \boldsymbol{x})^2 + (\cos \boldsymbol{x})^2 - 1 = 0$$

$$f_6(\boldsymbol{x}) = \operatorname{asin}(\sin \boldsymbol{x}) - \boldsymbol{x} = 0$$

$$f_7(\boldsymbol{x}) = \operatorname{acos}(\cos \boldsymbol{x}) - \boldsymbol{x} = 0$$

$$f_8(\boldsymbol{x}) = \tan \boldsymbol{x} - \frac{\sin \boldsymbol{x}}{\cos \boldsymbol{x}} = 0$$

$$f_9(\boldsymbol{x}) = (\cosh \boldsymbol{x})^2 - (\sinh \boldsymbol{x})^2 - 1 = 0$$

$$f_{10}(\boldsymbol{x}) = \cosh \boldsymbol{x} + \sinh \boldsymbol{x} - e^{\boldsymbol{x}} = 0$$

$$f_{11}(\boldsymbol{x}) = \operatorname{asinh}(\sinh \boldsymbol{x}) - \boldsymbol{x} = 0$$

$$f_{12}(\boldsymbol{x}) = \operatorname{acosh}(\cosh \boldsymbol{x}) - \boldsymbol{x} = 0$$

$$f_{13}(\boldsymbol{x}) = \tanh \boldsymbol{x} - \frac{\sinh \boldsymbol{x}}{\cosh \boldsymbol{x}} = 0$$

DACE Testing and Validation

Derivation and Anti-Derivation: Compare known integrals/derivatives

$f(x)$	$g(x)$
e^x	e^x
$\ln(1+x)$	$\frac{1}{1+x}$
$\sin x$	$\cos x$
$\cos x$	$-\sin x$
$\tan x$	$\frac{1}{(\cos x)^2}$
$\tan x$	$1+(\tan x)^2$
$\arctan x$	$\frac{1}{1+x^2}$

DACE Testing and Validation

- All tests can be considered to be successful:
  - Few tests showed an error slightly greater than 10^{-15}
  - Considering typical truncation errors related to floating point arithmetic, the tolerance value of 10^{-15} should be relaxed

- The DACE has been validated:
  - The computational engine has been tested
  - The C++ and MATLAB interfaces have been used to perform the tests

Performances

- DACE will be tested against the software COSY INFINITY
  - Both the C++ and MATLAB interface will be tested
  - COSY INFINITY and the DACE will be compiled using the same compiler flags, compiler optimizations, and on the same platform
- Consider the general dynamical system: $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$
- The test is based on multiple evaluations of Picard–Lindelöf operator

$$\boldsymbol{x}(t) = \boldsymbol{x}_0 + \int_{t_0}^{t} \boldsymbol{f}(\boldsymbol{x}(\tau))\, \mathrm{d}\tau$$

  - Makes use of both algebraic operations and anti-derivation
  - Its recursive use converges to the expansion of the solution of $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$

Performances

- Test program: for any six-dimensional dynamics
  - Initialize *i* of the 6 phase-space variables as DA numbers
  - Evaluate the Picard–Lindelöf operator 10^4 times with COSY INFINITY and DACE in its form

$$\boldsymbol{x}_{j+1} = \boldsymbol{x}_j + \int \boldsymbol{f}(\boldsymbol{x}_j)\, \mathrm{d}\tau, \text{ for } j = 0, 1, \ldots, 10^4$$

  - Store the computational times in $t_{k,i}^{DACE}$ and $t_{k,i}^{COSY}$
  - Repeat the test program 100 times, average $t_{k,i}^{DACE}$ and $t_{k,i}^{COSY}$ and compute

$$\tau_{k,i} = \frac{\tilde{t}_{k,i}^{DACE}}{\tilde{t}_{k,i}^{COSY}}$$

Performances

- Dynamics to be used:
  - 2-body dynamics

$$\ddot{\boldsymbol{r}} = -\frac{\mu}{r^3}\boldsymbol{r}$$

		Number of variables			
		1	2	…	6
Order	1	$\tau_{1,1}$	$\tau_{1,2}$	…	$\tau_{1,6}$
	2	$\tau_{2,1}$	$\tau_{2,2}$	…	$\tau_{2,6}$
	…	…	…	…	…
	10	$\tau_{10,1}$	$\tau_{10,2}$	$\tau_{10,3}$	$\tau_{10,6}$

  - Circular restricted 3-body dynamics

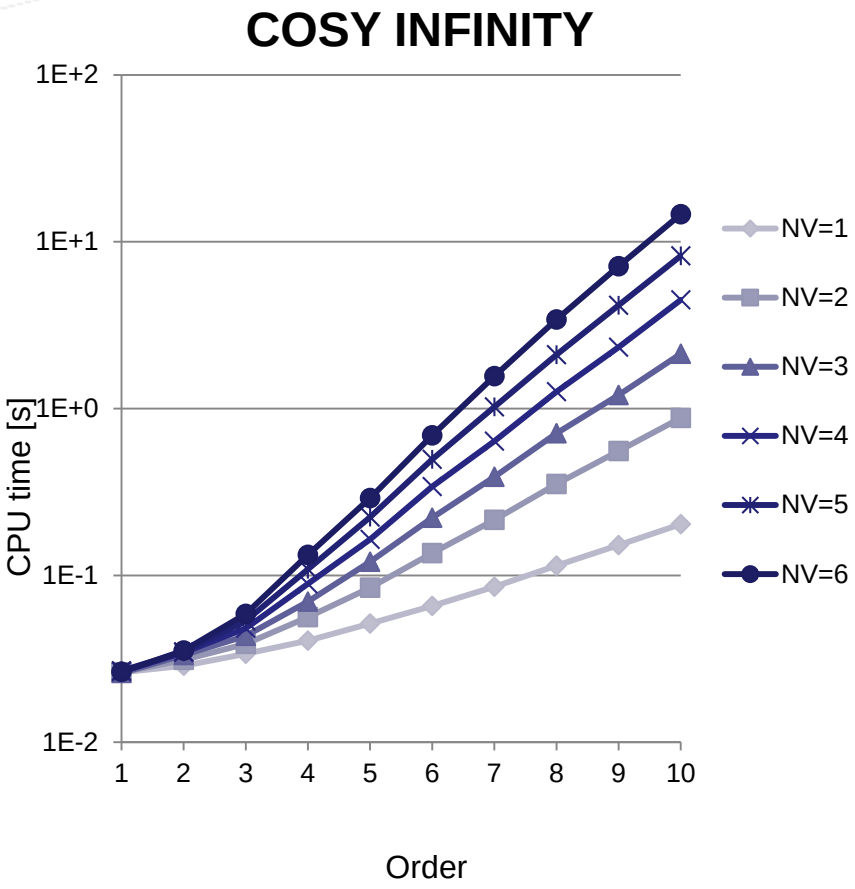
$$\begin{cases} \ddot{x} &= 2\dot{y} + \Omega_x \\ \ddot{y} &= -2\dot{x} + \Omega_y \\ \ddot{z} &= \Omega_z \end{cases}$$

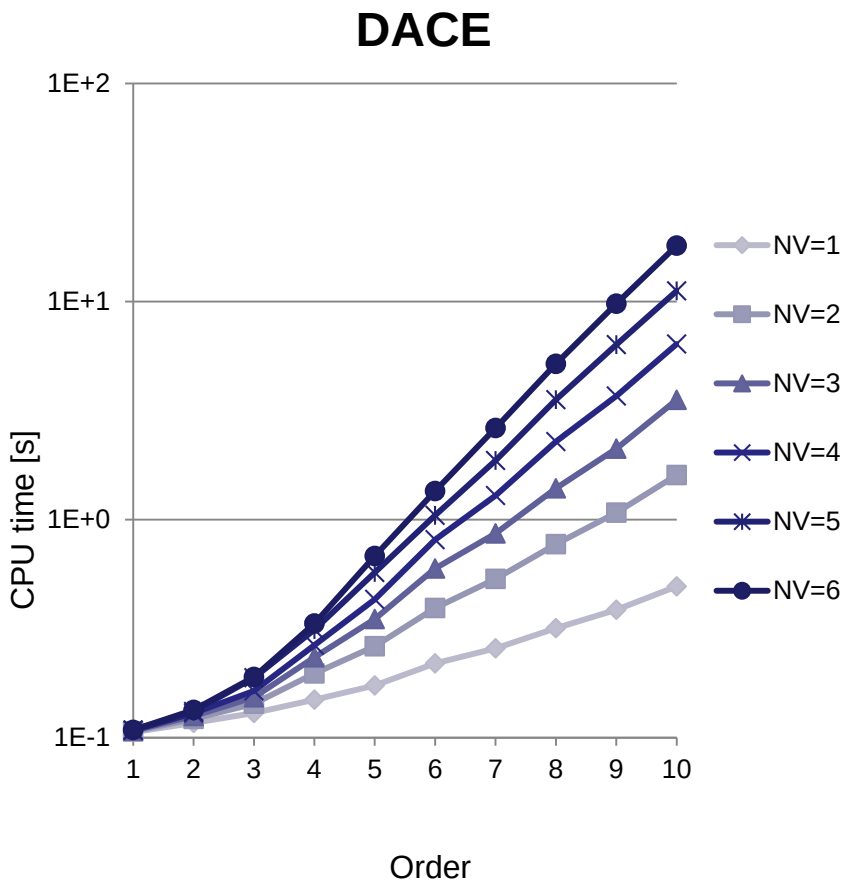
$$\Omega = \frac{1}{2}(x^2 + y^2) + \frac{1-\mu}{r_1} + \frac{\mu}{r_2}$$

Remarks

- All tests for computational efficiency are performed on a machine with the following specifications
  - Processor: Intel® Core™ i7-4930K CPU @ 3.40GHz
  - RAM: 32 Gb
  - OS: Kernel Linux 3.11.10-7-default (64 bit)
  - Compiler: gcc/GNU Fortran (SUSE Linux) 4.8.1 20130909
- NB: Results may vary on different machines and using different compilers
- CPU time used to measure required time
  - Amount of time used by CPU to process instructions of a computer program
  - Functions required for Windows and Linux/MAC OS are different
  - "Clocks per seconds" must be known

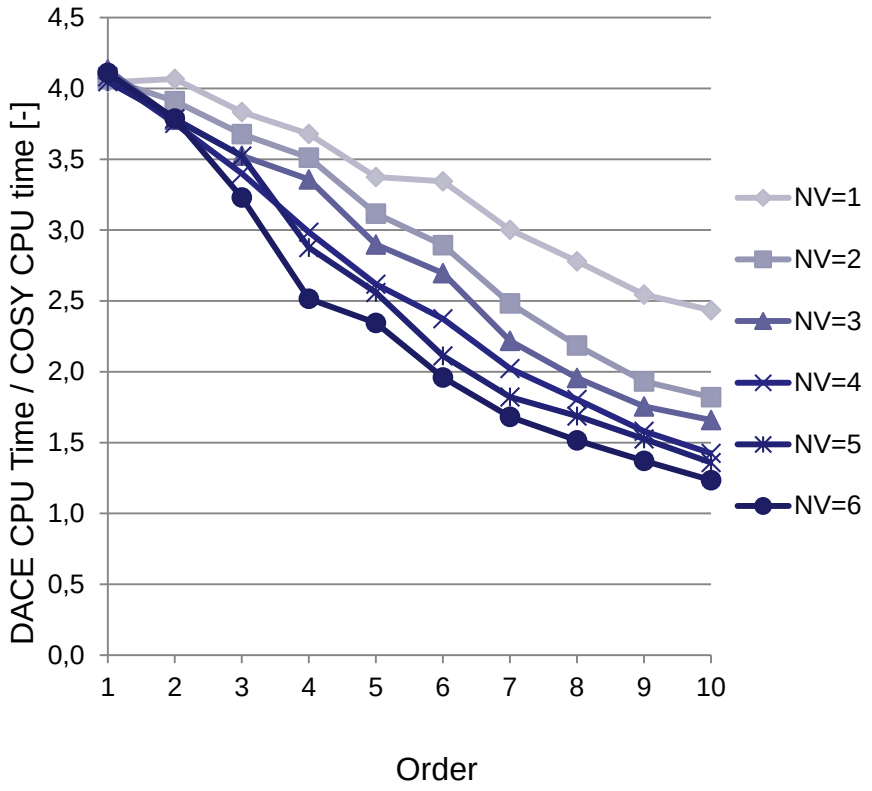
C++ interface CPU time (2BP)





C++ interface performance (2BP)

TEST 7.1		Number of variables					
		1	2	3	4	5	6
Order	1	4.042	4.067	4.134	4.082	4.048	4.110
	2	4.067	3.910	3.779	3.753	3.786	3.787
	3	3.833	3.678	3.525	3.399	3.522	3.230
	4	3.678	3.511	3.358	2.985	2.877	2.516
	5	3.375	3.116	2.898	2.618	2.560	2.345
	6	3.344	2.894	2.696	2.374	2.112	1.960
	7	3.002	2.482	2.219	2.023	1.822	1.682
	8	2.780	2.186	1.956	1.805	1.688	1.516
	9	2.544	1.932	1.756	1.581	1.526	1.371
	10	2.434	1.821	1.660	1.424	1.360	1.235



Matlab interface performance (2BP)

- Matlab is considerably slower than COSY (between 20 and 8500 times)

TEST 7.2		Number of variables					
		1	2	3	4	5	6
Order	1	8494.66	8491.62	8553.76	8399.55	8313.81	8429.54
	2	7729.14	7148.08	6696.81	6466.11	6408.73	6333.05
	3	6571.43	5729.44	5137.66	4625.31	4176.27	3796.80
	4	5481.30	3958.34	3205.06	2562.61	2099.47	1726.42
	5	4321.49	2636.62	1887.31	1390.23	1029.63	796.17
	6	3390.55	1645.79	1031.09	674.94	468.60	341.41
	7	2600.63	1037.56	588.26	363.56	230.43	154.74
	8	1949.39	645.26	324.28	185.96	114.89	74.20
	9	1471.07	411.80	193.28	102.77	60.33	38.06
	10	1100.96	262.54	110.30	54.63	32.18	20.52

Matlab interface CPU time (2BP)

- Note that an almost constant offset of about 222 s is present

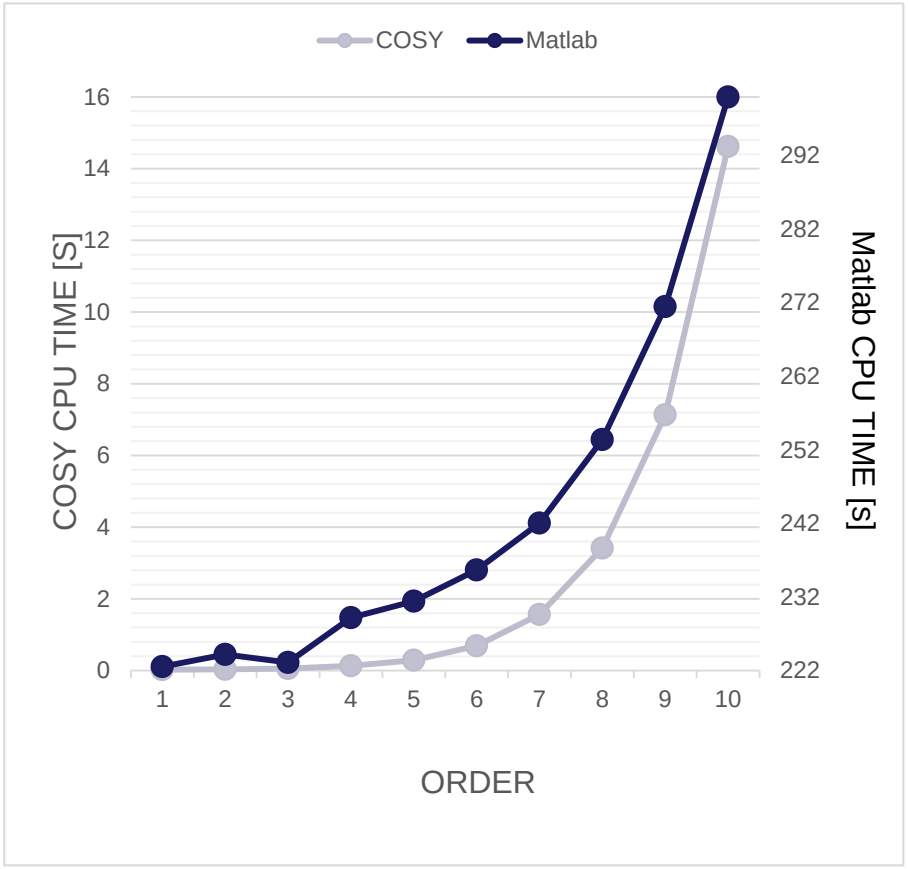
MATLAB	NV=1	NV=2	NV=3	NV=4	NV=5	NV=6
NO=1	222.90	222.82	222.74	222.42	222.81	222.54
NO=2	222.29	223.02	222.87	223.21	223.28	224.19
NO=3	222.64	222.99	223.18	222.94	223.18	223.10
NO=4	222.76	222.30	223.20	227.97	228.59	229.20
NO=5	222.47	222.32	227.76	228.72	230.06	231.43
NO=6	222.42	223.63	228.20	230.10	232.20	235.68
NO=7	222.51	223.49	229.14	231.73	236.09	242.05
NO=8	223.01	227.83	230.50	234.79	241.72	253.41
NO=9	223.25	229.24	232.94	239.68	250.95	271.48
NO=10	223.45	230.84	235.28	244.96	265.03	299.98

COSY	NV=1	NV=2	NV=3	NV=4	NV=5	NV=6
NO=1	0.026	0.026	0.026	0.026	0.027	0.026
NO=2	0.029	0.031	0.033	0.035	0.035	0.035
NO=3	0.034	0.039	0.043	0.048	0.053	0.059
NO=4	0.041	0.056	0.070	0.089	0.109	0.133
NO=5	0.051	0.084	0.121	0.165	0.223	0.291
NO=6	0.066	0.136	0.221	0.341	0.496	0.690
NO=7	0.086	0.215	0.390	0.637	1.025	1.564
NO=8	0.114	0.353	0.711	1.263	2.104	3.415
NO=9	0.152	0.557	1.205	2.332	4.160	7.133
NO=10	0.203	0.879	2.133	4.484	8.237	14.620

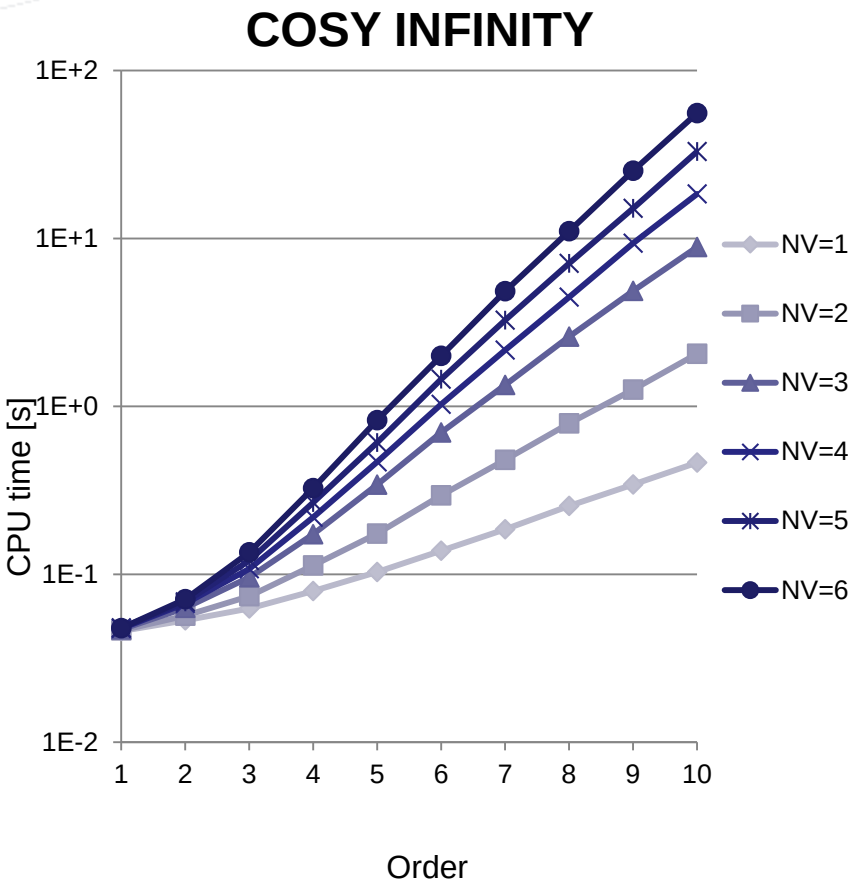
Matlab interface CPU time (2BP)

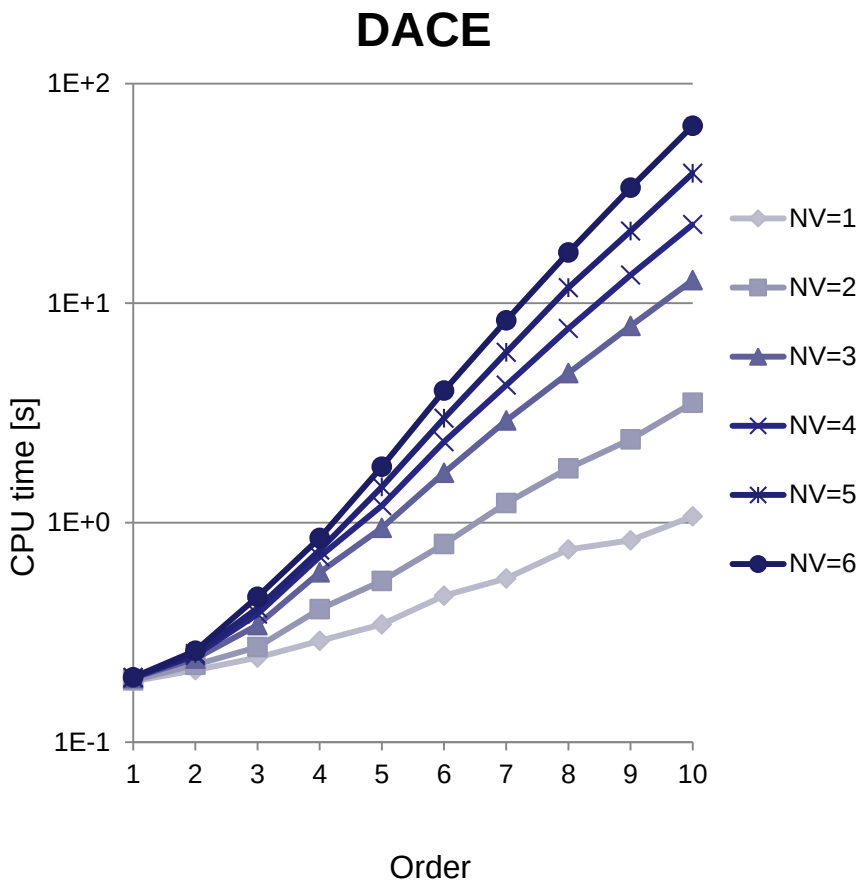
- Similar trend in CPU time once the initial offset is removed

MATLAB	NV=1	NV=2	NV=3	NV=4	NV=5	NV=6
NO=1	222.90	222.82	222.74	222.42	222.81	222.54
NO=2	222.29	223.02	222.87	223.21	223.28	224.19
NO=3	222.64	222.99	223.18	222.94	223.18	223.10
NO=4	222.76	222.30	223.20	227.97	228.59	229.20
NO=5	222.47	222.32	227.76	228.72	230.06	231.43
NO=6	222.42	223.63	228.20	230.10	232.20	235.68
NO=7	222.51	223.49	229.14	231.73	236.09	242.05
NO=8	223.01	227.83	230.50	234.79	241.72	253.41
NO=9	223.25	229.24	232.94	239.68	250.95	271.48
NO=10	223.45	230.84	235.28	244.96	265.03	299.98



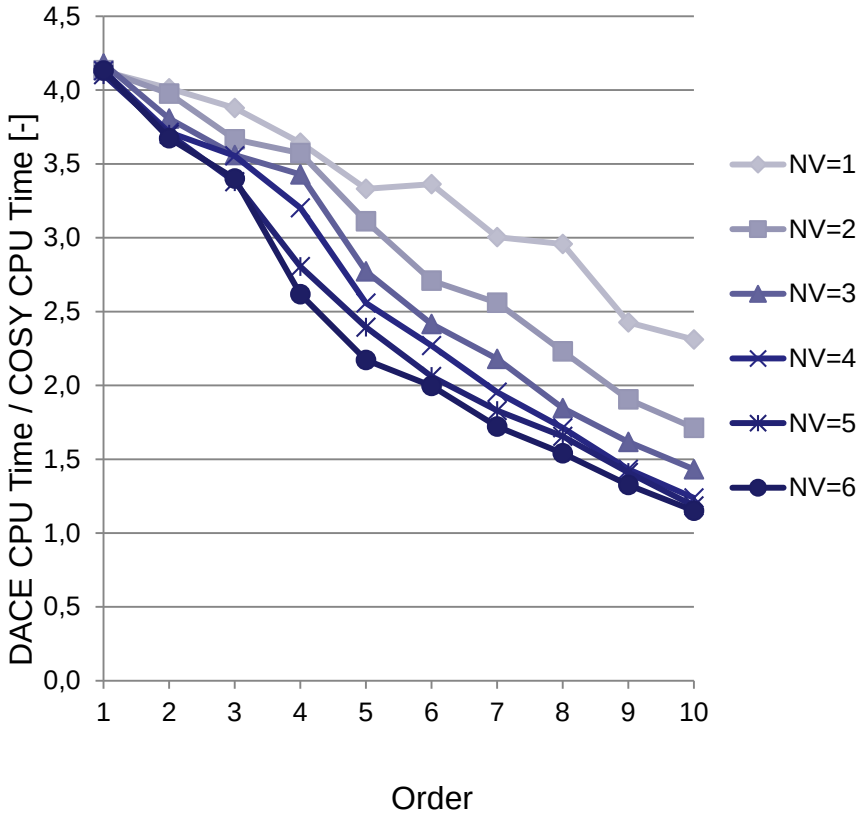
C++ interface CPU time (3BP)





C++ interface performance (3BP)

TEST 7.2		Number of variables					
		1	2	3	4	5	6
Order	1	4.127	4.134	4.180	4.131	4.104	4.132
	2	4.012	3.976	3.806	3.714	3.701	3.674
	3	3.878	3.665	3.560	3.555	3.379	3.401
	4	3.643	3.573	3.429	3.203	2.805	2.617
	5	3.331	3.111	2.772	2.557	2.394	2.172
	6	3.363	2.709	2.416	2.269	2.060	1.996
	7	3.004	2.559	2.180	1.954	1.830	1.721
	8	2.957	2.230	1.846	1.712	1.655	1.540
	9	2.427	1.906	1.617	1.432	1.410	1.325
	10	2.311	1.712	1.432	1.238	1.183	1.151



Matlab interface performance (3BP)

- Matlab has large overhead
- Similar trend of CPU time once inital "offset" is removed

TEST 7.2		Number of variables					
		1	2	3	4	5	6
Order	1	9893.73	9824.44	9689.05	9533.24	9463.18	9504.69
	2	8537.19	8015.45	7187.24	6812.03	6662.10	6417.23
	3	7261.71	6142.52	4743.79	4236.02	3779.78	3381.00
	4	5719.06	4036.29	2631.93	2131.93	1744.25	1440.16
	5	4415.20	2614.28	1362.27	1001.63	772.91	573.02
	6	3288.74	1547.56	668.56	457.46	327.95	241.81
	7	2457.36	949.07	350.70	219.75	148.77	102.76
	8	1785.12	588.14	182.17	108.18	70.47	47.66
	9	1326.69	374.52	98.98	53.13	34.62	22.58
	10	983.86	229.90	55.03	27.86	16.92	11.49

Performances

- DACE C++ interface CPU time for the selected test case is of the same order of magnitude as COSY INFINITY
  - Highest ratio is 4 and is obtained for 1 DA variable
  - As number of variables and order increase the ratio gets closer to 1
  - Similar ratios and behaviour for both test cases
  - Typical settings for uncertainty propagation use 6 variables and order 5 or 6 (DACE about 2 times slower than COSY)
  - Remarkable result if we consider that the DACE has been developed in 4 months
- High overhead for Matlab interface
  - Once overhead is removed similar behaviour of CPU time is observed between COSY INFINITY and Matlab
  - Overhead is due to MATLAB internal objects handling and is unrelated to our interface

Differential Algebra Computational Engine (DACE)

Workshop on Nonlinear Propagation of Uncertainties using Differential Algebra

September 22nd, 2015 *Escape Dance Room*, *ESTEC*

DINAMICA srl
Registered Office:
Piazza della Repubblica, 10 - 20121 - Milano (Italy)
Operational Headquarters:
Via Morghen, 13 - 20156 - Milano (Italy)
Phone +39 02 8342 2930
Fax +39 02 3206 6679
e-mail: dinamica@dinamicatech.com
website: www.dinamicatech.com