Numerical Experimentation in Differential Geometry

Simon David Burton

A thesis submitted for the degree of
Graduate Diploma in Science
Department of Physics
The Australian National University

Supervisor: Dr. Susan Scott

June, 2007

Declaration

This thesis is an account of research undertaken between February 2005 and June 2007 at The Department of Physics, Faculty of Science, The Australian National University, Canberra, Australia.

Except where acknowledged in the customary manner, the material presented in this thesis is, to the best of my knowledge, original and has not been submitted in whole or part for a degree in any university.

Simon Burton
June, 2007

Abstract

A software package, `pydx,` is developed for numerical experimentation in differential geometry. The architecture builds upon several important techniques. Interval arithmetic provides exact quantification of errors, effectively allowing mathematical proof. Automatic differentiation provides a way to naturally generate taylor expansions for solutions of the initial value problem, and find derivatives of the metric as needed for the geodesic equation. Use of the `pydx` system in physics is demonstrated by an exploration of the geodesic structure of the spatial sections of the Curzon spacetime.

Acknowledgments

I firstly thank my supervisor Susan Scott, for showing a greatly appreciated excitable attitude, and for providing clear guidance. I thank Antony Searle for initially defining the bold technical scope of this research as a whole. I also acknowledge my dear friend Patrick Lesslie, whose moral suppport and optimistic outlook had great positive impact on this work. Also thanks go to Erika Mordek, who continually pestered me into completing this thesis. Lastly, I would like to thank the python community, in particular for their warm response to this work at the 2007 python conference (PyCon).

Contents

Introduction

Automatic differentiation provides a way to naturally generate taylor expansions for solutions of the initial value problem. It is also a way of finding derivatives of the metric needed in the geodesic equation. Combination of both of these is possible via multivariate automatic differentiation.

Once we are ready to perform calculations with actual numbers the question remains of what number system to use. We provide access to three different number systems, the CPU's native (double precision) floating point type, an *arbitrary* precision floating point type and an interval type. The latter keeps track of a lower and upper bound on each operation, and in this way can stand as computer generated mathematical proof.

All of these ideas are orchestrated at the highest level by concepts from differential geometry. We will show how these concepts are used and implemented in `pydx`,[1] and how the language of automatic differentiation naturally fits in with the geometric constructions.

As application in theoretical physics, we present computation of geodesics in the axisymmetric Curzon spacetime.

1.1 Overview of `pydx`

The bulk of the `pydx` software is implemented in the Python scripting language [22]. The use of Python enables rapid prototyping, as well as clear expression of mathematical ideas and algorithms. `pydx` is the third complete rewrite of the envisioned "system for computing with differential geometry". Python is a programming language famous for its use amongst programming "beginners" and the experts alike.

`pydx` is not just a tool for obtaining numerical results, but also a way of investigating, understanding, and communicating the theory of differential geometry.

A high-level overview of `pydx` is shown below.

[1] Pronounced: pie-dee-ecks.

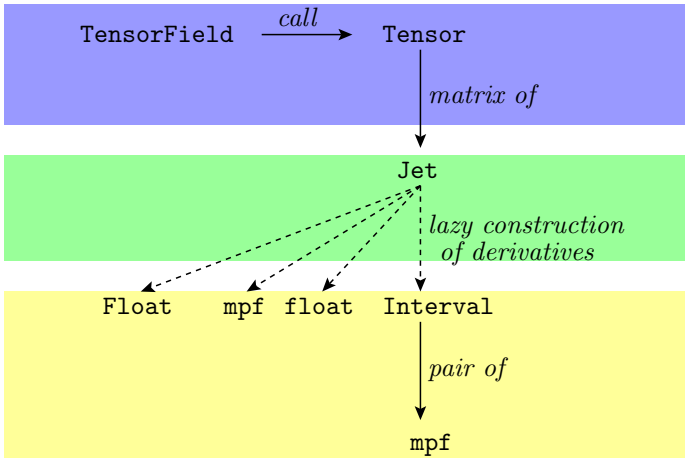


Fig 1.1.1 `pydx` Components

The first thing to note is that every node in this graph (corresponding to a `class` in Python) describes a type that we will be performing algebraic operations on.

The classes in the bottom layer are used to represent scalar values. The `float` type is the native `C`-compiler's `double` type. The `mpf` type is an arbitrary precision floating point type. The `Interval` type stores a pair of `mpf`'s representing the lower and upper bound on a computation. The `Float` type is for performing calculations symbolically.

The next layer implements differentiation; derivatives are stored in the components of a `Jet`. Arithmetic operations induce appropriate operations on these components. The scalar type used for these components is switchable between the types shown in the bottom layer, as indicated by the dashed lines.

The top layer of `pydx` involves tensor calculus. A tensor field is conceived of as a function (a *callable*) that returns a tensor. Operations on these fields are *pointwise*; this is implemented as lazy evaluation.

`pydx` makes extensive use of lazy evaluation[2] techniques. The idea is as follows. Suppose we have two objects `lazy_a` and `lazy_b` of type `Lazy`. Now we perform some operation, such as addition: `lazy_c = lazy_a + lazy_b`. No actual arithmetic is performed at this stage. The type of `lazy_c` is `LazyAdd`: these objects now respond to *requests* by performing the corresponding request on `lazy_a` and `lazy_b` and returning the appropriate addition of the results. The three types `TensorField`, `Jet` and `Float` all follow this pattern.

[2] Also known as *evaluation on-demand*, or *partial evaluation*.

1.2 Chapter Outline

We begin in Chapter 2 with a brief introduction to the basics of the Python language. Throughout the text we will interleave Python code excerpts and the formal mathematical development.

In Chapter 3 we discuss using computers as calculating machines. The main point is rounding error, and the technique used to overcome this, *interval arithmetic*, is also a means of (the computer) generating mathematical proofs.

Chapter 4 is concerned with *automatic differentiation*. This turns out to be a way of generating Taylor polynomials for solutions of ordinary differential equations, which is discussed in Chapter 5, and also fits nicely within the framework of differential geometry which is discussed in Chapter 6.

Automatic differentiation has the effect of promoting the differential of a value to the same status as the value itself. This is borne out in the implementation: we are able to calculate, eg. solve ODE's, with functions that are defined using derivatives of other functions.

Chapter 7 looks at one particular pseudo-Riemannian manifold, the Curzon space-time, and the solutions to the geodesic equation.

As the text proceeds, we work through each of these concepts following a (more or less) standard template:

- Introductory remarks.
- Mathematical formalism.
- Discussion of what parts of the formalism were chosen for the implementation, what parts were left out, and why.
- Examples, where we show how to use the software to perform calculations.
- Implementation section, where we discuss how the formalism is expressed using the constructs available in the Python language.
- Unit tests, which not only serve as excellent examples of how to use the software, but also demonstrate an important aspect of the software development process.
- Bibliographic notes, where we place the research in a wider context.
- References to the code listing, in Appendix B.

1.3 Notation

Differential geometry is known for its abuse of notation. Our case is no different. There is no way we can hope to find a mathematical notation that can withstand the confluence of ideas found herein. Indeed, this is one good reason to include the (highly verbose

and entirely rigorous) program implementation as this is a way we can simultaneously account for all concepts involved.

We use `typewriter type` to denote computer code. In Chapter 6 this is abused slightly as some computer code makes its way into mathematical notation. Note the mathematicians like single letter variables, as two letters side by side denote multiplication, so we may understand an `identifier` to be a single letter when it appears within the mathematical discourse.

Boldface is used in Chapter 3 to denote interval quantities, in Chapter 4 it denotes multi-indices, and in Chapter 6 it denotes tensors.

Python Survival Guide

The built-in types include `bool`, `int`, `float` and `str`. The sequence types are `list` and `tuple`. `str` is also a sequence type.

Any of these types can be used as constructors, eg. `a=int(b)` or `a=str(b)`. These types can also be instantiated via a *literal* form, eg.

```
some_bool = True
some_int = 1
some_float = 1.0
some_str = "1"
some_list = [1,2,3]
some_tuple = (1,2,3)
```

Comments begin with a # sign, and continue to the end of the line.

Multiplication of sequences produces a repeating sequence, eg. `(0,)*rank` is a `rank` length tuple of zero `int`'s.

Dictionaries (associative arrays) are another container type that store their entries indexed by arbitrary "keys". The type is `dict` and literal forms look like this `{"name":"barry", "age":22}`. There is also a `set` type. It has no literal form.

Accessing and setting of elements in sequences and dictionaries is done with *array index* notation. eg. `a[5]` is the `5` element of `a`. Note that sequences are *zero* based, so this would be the *sixth* element of a sequence.

The difference between a `tuple` and a `list` is that once a `tuple` is created, its elements cannot be changed. It is *immutable*. `str`, `int` and `float` are also immutable types.

Two other useful functions are the `len` function which returns the number of elements in a container (sequence or dictionary), and the `type` function which returns the type of its argument.

We show this example using the interactive Python interpreter. The `>>>` prompt indicates an input line, the `...` prompt indicates a continuation of a compound statement (suite) entry, and the lines without a prompt are the `print`'ed values of the previous line.

Basic control flow includes `if`, `elif`, `else` condition statements, `while` loops and `for` loops. `for` loops are peculiar in that they operate by iterating over a given sequence. So to implement a simple counting loop we iterate over a sequential list created with the `range` function:

```
for i in range(10):
    print i
```

Notice the "body" of the loop is indented. This indented section is known as a *suite*. The line previous to a suite will always end with a colon.

Simple `for`-loops for making lists can be inlined in a special form called a *list comprehension*:

```
squares = [i**2 for i in range(10)]
```

`zip` and `enumerate` operate on lists (or iterators):

```
>>> zip([2,3,4], [5,6,7])
[(2, 5), (3, 6), (4, 7)]
>>> enumerate("hi there!")
<enumerate object at 0xb7d7026c>
>>> [item for item in enumerate("hi there!")]
[(0, 'h'),
 (1, 'i'),
 (2, ' '),
 (3, 't'),
 (4, 'h'),
 (5, 'e'),
 (6, 'r'),
 (7, 'e'),
 (8, '!')]
```

Functions (and methods) are created with a `def` suite:

```
def sum(a, b):
    return a+b
```

Assignments to arguments occurring in the function definition line refer to default values for those arguments.

```
>>> def add(a, b=1):
...     return a+b
...
```

```
>>> add(3)
4
>>> add(3, 2)
5
```

In an interactive session it is helpful to be able to see the attributes and methods available on an object. This is done with the `dir` function:

```
>>> message = "Hi there"
>>> dir(message)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
 '__eq__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
 '__getslice__', '__gt__', '__hash__', '__init__', '__le__',
 '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
 '__rmul__', '__setattr__', '__str__', 'capitalize', 'center',
 'count', 'decode', 'encode', 'endswith', 'expandtabs',
 'find', 'index', 'isalnum', 'isalpha', 'isdigit',
 'islower', 'isspace', 'istitle', 'isupper', 'join',
 'ljust', 'lower', 'lstrip', 'replace', 'rfind',
 'rindex', 'rjust', 'rsplit', 'rstrip', 'split',
 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
 'translate', 'upper', 'zfill']
```

The methods following the double underscore convention are special "hook" methods. These are called implicitly from other Python syntax. For example, help can be obtained via the `__doc__` attribute or the `help` function:

```
>>> help(message.upper)
upper(...)
    S.upper() -> string

    Return a copy of the string S converted to uppercase.
```

We write functions that take arbitrarily many arguments with the `*args` specifier.

```
>>> def add(*items):
...     result = 0
...     for item in items:
...         result += item
...     return result
...
>>> add(1,2,3)
6
```

A `lambda` is an anonymous function that can only return an expression (eg. no control flow is allowed).

```
sum = lambda a, b: a+b
```

`assert` statements raise an exception when the argument is logically false:

```
>>> assert 1+1==3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AssertionError
```

Classes are defined with a `class` suite. Class methods always have explicit mention of the object as the first argument, usually called `self`.

```
class Student(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Objects of class `Student` are now created by calling the class (just like the basic types):

```
student = Student("barry", 22)
if student.age < 5:
    print "too young"
```

Operator overloading (such as multiplication for sequences) is implemented with further special "hook" methods. eg. multiplication is implemented with the `__mul__` method.

operation	*syntax*	*special method*
initialization	`some_type(...)`	`__init__`
string	`str(some_object)`	`__str__`
arithmetic	`+ - * / **`	`__add__ __sub__ __mul__ __div__ __pow__`
right arithmetic	`+ - * / **`	`__radd__ __rsub__ __rmul__ __rdiv__ __rpow__`
in-place arithmetic	`+= -= *= /= **=`	`__iadd__ __isub__ __imul__ __idiv__ __ipow__`
comparison	`< <= == >= >`	`__lt__ __le__ __eq__ __ge__ __gt__`
array indexing	`some_array[i]`	`__getitem__ __setitem__`
length	`len(some_container)`	`__len__`
function call	`callable(...)`	`__call__`
attribute access	`some_object.attr`	`__getattr__ __setattr__`

The terms *class* and *type* will be used synonymously.
See also the Python tutorial [23].

Interval Arithmetic

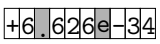
A major stumbling block of numerical algorithms is their susceptibility to computer rounding errors. Obtaining a theoretical appreciation of this effect can be done for certain algorithms. However, often the only recourse of the practitioner is to increase the precision until the result "stabilizes".

This chapter provides another technique *interval arithmetic*. This arithmetic has the effect of taking a strict (albeit pessimistic) account of rounding error. Once we can do this, we can rely on the results without further theoretical introspection. In fact, the computer has generated a mathematical proof.

3.1 Floating Point Arithmetic

The representation of real valued numbers in the computer is done with finite precision *floating-point* arithmetic. The real number is represented similarly to scientific notation, with one digit to the left of the decimal point and an exponentiated factor.

For example, 6.626×10^{-34} would be represented as



Here we have four digits "`6626`", and a sign bit "`+`", that make up what is known as the *mantissa*, along with two digits "`34`", and a sign bit "`-`", for the *exponent*. In practice, these digits are stored as *binary* numbers (base 2) and the exponent has a base of 2. Python's `float` type uses the native `C` compiler's `double` type. This is typically a 64 bit number, with 52 bits for the mantissa, 11 bits for the (signed) exponent, and one sign bit.

A consequence of this fixed size representation is that we can represent only finitely many numbers. For all the other numbers we have to decide on which floating-point number we approximate with. This process is called rounding and is the primary source of errors in a numerical computation.[1] If the result of an arithmetic operation cannot

[1] Due to the fixed range in the exponent, we may also raise an error if the magnitude of the number is too large, or too small but non-zero.

be exactly represented by the floating-point system, then we lose information.

Consider a division operation.

```
4.56789 / 2.00000 = 2.283945
```

We round up, losing the last digit:

```
4.56789 / 2.00000 = 2.28395
```

Sometimes a number has so little information that it becomes very misleading:

```
4.56789 - 4.56788 = 1.00000e-5
```

The question arises: how do we prevent this from happening ?

3.2 Implementation

The Python `float` type uses the native `C`-compiler's `double` type.

To get extended precision, we use the Python `gmpy` [13] module, which is a wrapper around the `libgmp` [3] `C` library. The `gmpy` library exports the `mpf` (*multi precision float*) type to Python. The size of the exponent is fixed (16 bits on a 32-bit computer), and the size of the mantissa (which includes the sign) is user selectable from a minimum of 64 bits upwards in units of "limbs" (32 bits on a 32-bit computer). The exponent counts limbs, ie. with a 32 bit limb the exponent has a base of 2^{32}. In this case we can represent numbers in the range $2^{-68719476768}$ - $2^{68719476736}$. On a 64 bit machine the range will be larger.

One serious limitation of `libgmp` is that it does not implement any of the transcendental functions. In later chapters we will get around this limitation by writing these functions as solutions to first or second order ODE's.

3.3 Example

To take an example, we turn to the computation of a particularly difficult power series. This is the (truncated) Taylor series of $f(x) = e^{-x}$:

$$f_n(x) = \sum_0^n (-1)^j \frac{1}{j!} x^j$$

In factored it form it looks like this:

$$f_n(x) = 1 - \ \ldots \frac{1}{n-2} x \ \Big(\ 1 - \frac{1}{n-1} x \ \big(1 - \frac{1}{n} x\big) \ \Big).$$

In this form it is much easier to compute because there are no big $n!$ numbers involved.

We define a Python function,

```
def f(x, n):
    r = 1.0
    i = n
    while i > 0:
        r = 1.0 - x*r/i
        i -= 1
    return r
```

and set about comparing it to the built-in ** operator.

```
>>> from math import e
>>> e**-1
0.36787944117144233
>>> f(1.0, n=10)
0.36787946428571427
```

This is reasonable accuracy for a series with 10 terms. Things get a bit more difficult if we move to the point $x = 20$:

```
>>> e**-20
2.0611536224385599e-09
>>> f(20.0, n=10)
1859623.680776014
```

Now we need to use many more terms to get any accuracy:

```
>>> f(20.0, n=70)
1.5517533924480631e-09
```

But even with $n = 1000$ there is no impact on the error:

```
>>> f(20.0, n=1000)
1.5517533924480631e-09
```

At $x = 50$ we become hopelessly lost:

```
>>> f(50.0, n=100)
280505281350.00238
>>> f(50.0, n=500)
-81347.387567684738
>>> f(50.0, n=1000)
-81347.387567684738
```

The only solution is to use a floating point type with more precision.

We repeat the calculation at $x = 20$ with 64-bit (mantissa) floating point numbers:

```
>>> from gmpy import mpf
>>> f(mpf(20.0, 64), n=100)
mpf('2.0611536224390286940e-9',64)
```

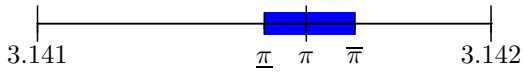
At $x = 50$ we need 128-bit numbers:

```
>>> e**-50
1.928749847963923e-22
>>> f(mpf(50.0, 64), n=1000)
mpf('7.0089572128619110240e-9',64)
>>> f(mpf(50.0, 128), n=1000)
mpf('1.928753399978943643001205448767949337928e-22',128)
```

We have found the answer, but we knew what we were looking for.

3.4 Introduction to Interval Arithmetic

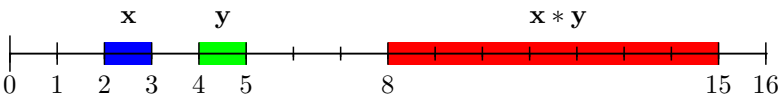
Interval Arithmetic is based on two ideas. First, instead of using higher and higher precision to represent real numbers, we store lower and upper bounds (an *interval*) that are represented exactly using machine floating point numbers.



We don't know what the number is exactly but we know it is inside the interval somewhere (maybe at one of the endpoints).

The second idea is that we carry out arithmetic operations on these intervals such that the resulting interval encloses all possible values. Denoting a generic operation with a $\bullet$, we require

$$\mathbf{x} \bullet \mathbf{y} \supseteq \{x \bullet y | x \in \mathbf{x}, y \in \mathbf{y}\}.$$



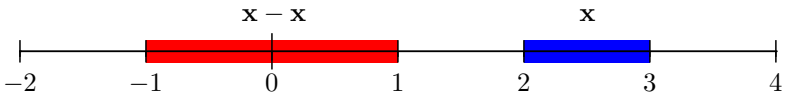
We enlarge the result if necessary so that the endpoints are represented exactly in the machine.

Calculations involving intervals amount to a (computer generated) mathematical proof: it *guarantees* that the result of the calculation is inside an interval,

$$x \bullet y \in \mathbf{x} \bullet \mathbf{y}.$$

Unfortunately, this guarantee can sometimes become extremely weak, as the interval explodes. There are two problems encountered. The first is the *dependency* problem: as the calculation progresses we lose information about the correlations between the values in the calculation. The second problem involves higher dimensional calculations: multi-dimensional intervals represent axes aligned rectangular regions. These regions do not behave well under rotations.

A simple example of the dependency problem is given by subtracting an interval from itself. The "real" answer is zero, a very small interval, but we end up with something considerably larger:



Methods exist to counter this effect by augmenting the interval object with information describing how its value depends on another value; for example, derivative information. For example, the Affine Arithmetic of Figueiredo and Stolfi [12] tracks first derivative information for each variable in a calculation.

The wrapping effect becomes evident when we are computing in higher dimensions. Consider the following rotation:

$$\mathbf{x}_1 = \frac{1}{\sqrt{2}}(\mathbf{x}_0 - \mathbf{y}_0)$$
$$\mathbf{y}_1 = \frac{1}{\sqrt{2}}(\mathbf{x}_0 + \mathbf{y}_0)$$

Every application of the rotation will cause the width of the resulting intervals to be multiplied by a factor greater than one.

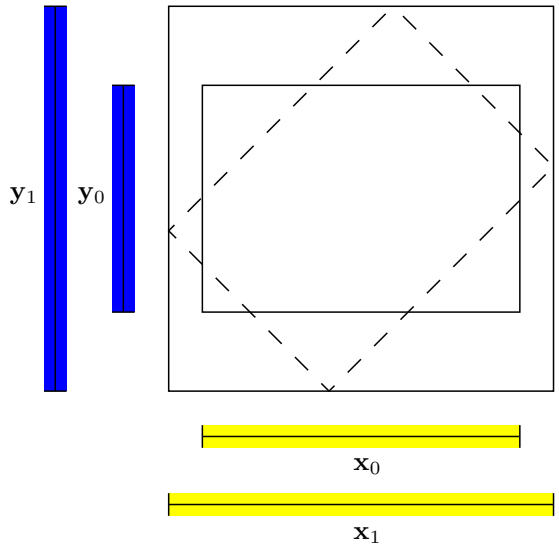


Fig. 3.4.1 The wrapping effect: every rotation causes an interval box to expand.

3.5 Real Interval Arithmetic

We now develop the formal apparatus of "ideal" interval arithmetic. It sets a first limit on what we can expect from a practical implementation.

Definition 3.5.1. *We define an $\mathbb{R}$-interval to be the closed (non-empty) set:*

$$[a,b] = \{x | a \leq x \leq b\}, \quad \textit{where } a,b \in \mathbb{R},\ a \leq b$$

and define I to be the set of all such intervals:

$$I = \{[a,b] | a \leq b\}$$

We use boldface font to denote interval variables: $\mathbf{x}$.

Definition 3.5.2. *Given an interval $\mathbf{x} = [a,b]$, the lower and upper points are written as $\underline{\mathbf{x}} := a$ and $\overline{\mathbf{x}} := b$, respectively.*

Definition 3.5.3. *Given a binary operation $\bullet : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, the interval extension is defined:*

$$\mathbf{x} \bullet \mathbf{y} := \bigcap \{\mathbf{z} \in I | \mathbf{z} \supset \{x \bullet y | x \in \mathbf{x}, y \in \mathbf{y}\}\}$$

that is, $\mathbf{x} \bullet \mathbf{y}$ is the smallest interval containing the set $\{x \bullet y | x \in \mathbf{x}, y \in \mathbf{y}\}$.

For arithmetic operations, we can calculate results as follows.

Proposition 3.5.4.

(1) $[a,b]+[c,d]=[a+c,b+d]$

(2) $[a,b]-[c,d]=[a-d,b-c]$

(3) $[a,b][c,d]=[\min\{ac,ad,bc,bd\},\max\{ac,ad,bc,bd\}]$

(4) $[a,b]/[c,d]=[\min\{a/c,a/d,b/c,b/d\},\max\{a/c,a/d,b/c,b/d\}]$, *where* $0<c$ *or* $d<0$.

The algebra of intervals inherits certain properties from the underlying operations.

Proposition 3.5.5.

(1) Sub-inverse for addition and multiplication:

$$0\in\mathbf{x}-\mathbf{x}$$

If $0\notin\mathbf{x}$*:*

$$1\in\frac{\mathbf{x}}{\mathbf{x}}.$$

(2) Associativity and commutativity of addition and multiplication:

$$(\mathbf{x}+\mathbf{y})+\mathbf{z}=\mathbf{x}+(\mathbf{y}+\mathbf{z})$$

$$\mathbf{x}+\mathbf{y}=\mathbf{y}+\mathbf{x}$$

$$(\mathbf{xy})\mathbf{z}=\mathbf{x}(\mathbf{yz}).$$

$$\mathbf{xy}=\mathbf{yx}$$

(3) Sub-distributive law:

$$\mathbf{x}(\mathbf{y}+\mathbf{z})\subseteq\mathbf{xy}+\mathbf{xz}.$$

The sub-distributive law is the first indication of where things can go wrong. The two occurrences of $\mathbf{x}$ in the RHS become *uncorrelated*; the interval arithmetic machinery has no way of knowing that these two intervals come from the same variable.

Other operations defined for intervals:

Definition 3.5.6.

(1) The width *of an interval is defined:*

$$\text{width}([a,b]):=b-a$$

(2) The hull *of two intervals is the smallest interval containing both:*

$$\text{hull}([a,b],[c,d]):=[\min\{a,c\},\max\{b,d\}]$$

3.6 Numerical Interval Arithmetic

The implementation of intervals in computer hardware is somewhat softer than the above "sharp" arithmetic. The essential difference is that we are approximating ideal intervals (in $\mathbb{R}$) with slightly larger intervals, whose endpoints can be exactly represented in the floating point arithmetic of the machine.

To this end, we let $\mathbb{F}$ (the set of machine "floats") be some subset of $\mathbb{R}$.

Definition 3.6.1. *We define an $\mathbb{F}$-interval to be the closed set:*

$$[a,b] = \{x \in \mathbb{R} | a \leq x \leq b\}, \quad \textit{where } a,b \in \mathbb{F},\ a \leq b$$

and define $I_{\mathbb{F}}$ to be the set of all such intervals:

$$I_{\mathbb{F}} = \{[a,b] | a \leq b,\ a,b \in \mathbb{F}\}$$

An $\mathbb{F}$-interval is simply an $\mathbb{R}$-interval with endpoints in $\mathbb{F}$.

We now take a disciplined approach to arithmetic in $\mathbb{F}$. It is not enough to define the addition of two floats as being the nearest float to the true sum (in $\mathbb{R}$).

Definition 3.6.2. *We define the lower and upper $\mathbb{F}$ sum, difference, multiplication and division using the corresponding operations on $\mathbb{R}$:*

$$\begin{aligned}
a \dot{+} b &:= \min\{c \in \mathbb{F} | c \geq a + b\} \\
a \dot{+} b &:= \max\{c \in \mathbb{F} | c \leq a + b\} \\
a \dot{-} b &:= \min\{c \in \mathbb{F} | c \geq a - b\} \\
a \dot{-} b &:= \max\{c \in \mathbb{F} | c \leq a - b\} \\
a \dot{*} b &:= \min\{c \in \mathbb{F} | c \geq a * b\} \\
a \dot{*} b &:= \max\{c \in \mathbb{F} | c \leq a * b\} \\
a \dot{/} b &:= \min\{c \in \mathbb{F} | c \geq a / b\} \\
a \dot{/} b &:= \max\{c \in \mathbb{F} | c \leq a / b\}
\end{aligned}$$

These operations provide a way to round "up" or "down".

Definition 3.6.3. *Given a binary operation $\bullet : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, the $\mathbb{F}$-interval extension is defined:*

$$\mathbf{x} \bullet \mathbf{y} := \bigcap \{\mathbf{z} \in I_{\mathbb{F}} | \mathbf{z} \supset \{x \bullet y | x \in \mathbf{x}, y \in \mathbf{y}\}\}$$

that is, $\mathbf{x} \bullet \mathbf{y}$ is the smallest $\mathbb{F}$-interval containing the set $\{x \bullet y | x \in \mathbf{x}, y \in \mathbf{y}\}$.

We use the rounding operations above to calculate $\mathbb{F}$-interval operations.

Proposition 3.6.4.

(1) $[a,b]+[c,d]=[a\dot{+}c, b\dot{+}d]$

(2) $[a,b]-[c,d]=[a\dot{-}d, b\dot{-}c]$

(3) $[a,b]*[c,d]=[\min\{a\dot{*}c, a\dot{*}d, b\dot{*}c, b\dot{*}d\}, \max\{a\dot{*}c, a\dot{*}d, b\dot{*}c, b\dot{*}d\}]$

(4) $[a,b]/[c,d]=[\min\{a\dot{/}c, a\dot{/}d, b\dot{/}c, b\dot{/}d\}, \max\{a\dot{/}c, a\dot{/}d, b\dot{/}c, b\dot{/}d\}]$, *where* $0<c$ *or* $d<0$.

Because $\mathbb{F}$ may be arbitrarily sparse, the arithmetic laws have become somewhat weaker.

Proposition 3.6.5.

(1) Sub-inverse for addition and multiplication:

$$0\in \mathbf{x}-\mathbf{x}$$

If $0\notin \mathbf{x}$*:*

$$1\in \frac{\mathbf{x}}{\mathbf{x}}.$$

(2) Associativity and commutativity of addition and multiplication:

$$(\mathbf{x}+\mathbf{y})+\mathbf{z}\cap \mathbf{x}+(\mathbf{y}+\mathbf{z})\neq\phi$$
$$\mathbf{x}+\mathbf{y}=\mathbf{y}+\mathbf{x}$$
$$(\mathbf{xy})\mathbf{z}\cap \mathbf{x}(\mathbf{yz})\neq\phi$$
$$\mathbf{xy}=\mathbf{yx}$$

(3) Sub-distributive law:

$$\mathbf{x}(\mathbf{y}+\mathbf{z})\cap \mathbf{xy}+\mathbf{xz}\neq\phi.$$

As we are mostly interested in ensuring that the widths of intervals do not grow too large, we define the width of an $\mathbb{F}$-interval using the upper subtraction of the endpoints[2].

Definition 3.6.6. *The* width *of an* $\mathbb{F}$*-interval is defined:*

$$\text{width}([a,b]) := b\dot{-}a$$

[2] We could also define the width of an $\mathbb{F}$-interval to be another $\mathbb{F}$-interval by using both lower and upper subtraction.

The definition of the hull of two $\mathbb{R}$-intervals does not need modification for $\mathbb{F}$-intervals.

Finally, we need to recognize that $\mathbb{F}$ is typically a finite set, and so the rounded arithmetic operations will not always be defined. To handle this case we extend $\mathbb{F}$ to include a special "undefined" element `NaN`.

3.7 Implementation

Exact rounding semantics is provided by the `libmpfr` library [16]. This library uses a similar arbitrary-precision floating point data type to `libgmp`. However, in the operations we are required to specify weather the result is rounded up or down (or nearest). Furthermore, this result is guaranteed to be the closest value to the true value. `libmpfr` also implements the commonly used transcendental functions (unlike `libgmp`).

The `libmpfi` library [18] uses the exact rounding semantics of `libmpfr` to define operations on interval types, as in the above formalism.

Both of these libraries are implemented in `c`.

The `pydx.scalar.mpfi` module provides a type `Interval`, which is a Python wrapper around the `libmpfi` interval type.

3.8 Example

We continue our example of a pathological calculation.

```
    >>> from pydx.scalar.mpfi import Interval, set_default_prec
    >>> set_default_prec(128)
    >>> y=f(Interval(1), 10)
    >>> y
    Interval(
mpf('3.67879464285714285714285714285714285137e-1'),
mpf('3.67879464285714285714285714285714285167e-1'))
    >>> y.width()
    mpf('2.9387358770557187699 2e-39',128)
```

This time we are confidant that the value of the function did not lose substantially to rounding error.

Now if we sum the series at $x = 50$:

```
    >>> y=f(Interval(50), 1000)
    >>> y
    Interval(
mpf('-1.16290171183914073834003405567966100931 9e-17'),
mpf('1.16971685442333635837233121437692089771 9e-17'))
    >>> y.width()
```

```
    mpf('2.33261856626247709671e-17',128)
```

we find that our interval has a width that swamps its value.[3] With another limb of precision, we arrive at our destination, and we *know* it:

```
    >>> set_default_prec(192)
    >>> y=f(Interval(50), 1000)
    >>> y
    Interval(
mpf('1.92874984796391137104428234080312105688023472064381908608 79e-22'),
mpf('1.92874984796392528618612828749576847594840002143386082086 48e-22'))
    >>> y.width()
    mpf('1.39151418459466926474e-36',128)
```

3.9 Unit Tests

Many ideas present themselves for use as a unit test. The one we choose is the simplest, based on the fundamental property of intervals, $x \bullet y \in \mathbf{x} \bullet \mathbf{y}$. A skeletal outline of such a test is shown below.

```
 1 #!/usr/bin/env python
 2
 3 from random import random
 4 from gmpy import mpf
 5 from pydx.scalar.mpfi import Interval
 6
 7 x = [random(), random()]
 8 x = [mpf(x[0], 1024), mpf(x[1], 1024)]
 9
10 xx = [Interval(x[0], x[0]), Interval(x[1], x[1])]
11
12 op = lambda x: x[0]+x[1]
13
14 assert op(xx).contains(op(x))
```

On line 7 we choose two random values. These numbers are promoted to rather high precision floating point numbers on line 8. This is done in order to approximate the real

[3] this calculation succeeded with the 128-bit `mpf` types but has failed in this case because of the pessimistic nature of the `Interval`'s internal rounding.

arithmetic operation, $x \bullet y$. On line 10 we create singleton intervals (the end-points are equal) from these two numbers. On line 12 we choose the operation, and line 14 assert that the interval result encloses the floating point result.

3.10 Bibliographic Notes

Interval arithmetic was developed in the 1960's by Ramon E. Moore [14].

In Knuth's book [11], section 4.2.2, we find a comprehensive account of errors arising from floating point arithmetic. He also praises interval arithmetic, and recommends that "efforts should be made to increase its availability and to make it as user-friendly as possible".

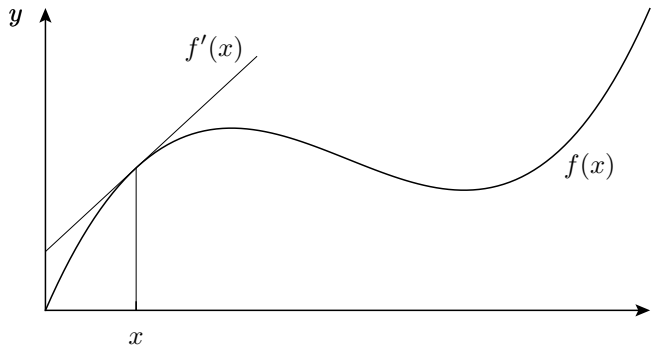
Wikipedia has an excellent article on the IEEE 754 floating point standard [1]. This is the implementation of floating point calculation common on todays hardware.

See also [5], and [12], for a good overview of interval arithmetic and applications.

Automatic Differentiation

4.1 Introduction

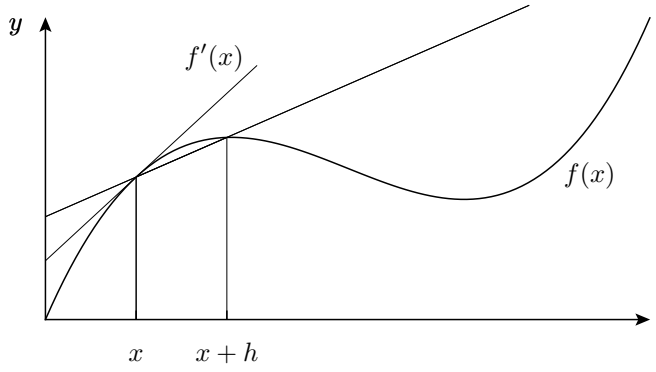
Given a function $f(x)$, how do we find the derivative, $f'(x)$?



Method 1. *Use finite differences.* An example formula:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

This is a brute force method.



It is approximate: we don't know how close the answer is to $f'(x)$, all we know is that the error in the method is reduced as $h \to 0$. The method can be improved by adding in more terms (evaluating f at more points).

This method has the advantage that we can treat f as a *black box*: its algebraic form (or otherwise) does not need to be known. Of course, we *do* need to know that $f(x)$ is differentiable.

Method 2. *Calculate symbolically (analytic method).* This is what we learned in high school. Given an algebraic expression for $f(x)$:

$$f(x) = x^2,$$

we manipulate this expression using some well-defined recursive rules, to obtain the derivative:

$$f'(x) = 2x.$$

Unlike **method 1**, this is an *exact* answer. Although we may be hampered by our inability to actually *find* numerical value for the RHS, at *this* point we have not degraded our knowledge of f' in any way. For reasonably simple functions, the form of the derivative may provide significant insight into the problem at hand. For larger functions, of several variables, the expression for the derivative (or higher derivatives) may span several pages making it useless for any comprehension.

However this method *does* require that we have an algebraic expression for f, and the corresponding rules for differentiating the component algebraic operations.

Furthermore, as we calculate higher derivatives (by iterating this procedure) the formula usually grows in size, *exponentially*:

$$f(x) = \exp(x^2)$$
$$f'(x) = 2x \exp(x^2)$$
$$f''(x) = 2 \exp(x^2) + 4x^2 \exp(x^2)$$
$$f^{(3)}(x) = 4x \exp(x^2) + 8x \exp(x^2) + 8x^3 \exp(x^2)$$
$$\ldots$$

and $f^{(10)}$ has 1631 operations.

This is a problem not only for pen and paper calculation, but also for (orders of magnitude faster) computer program calculation.

Method 3. *Calculating to first order.* This is the physicist's favorite trick. Introduce a new symbol ε, which is understood to be a *tiny* value, so small that ε^2 is considered to be *zero*.

Given $f(x) = x^2$,

$$f(x+\varepsilon) = (x+\varepsilon)^2 = x^2 + 2x\varepsilon + \varepsilon^2 = x^2 + 2x\varepsilon$$

and the derivative of f is now read off as the coefficient of ε:

$$f'(x) = 2x.$$

The point of this is *not* to produce a (possibly huge) formula for f'. What we have now is an *algorithm*. At each stage in the calculation we are only storing two numbers: a value and a derivative (the coefficient of ε). That is, we directly compute using the formula for f, but the objects operated on store a "value" and a "derivative".

So, this method is similar to **method 2**. We have an exact algorithm, and we still need an algebraic form for f, and an extension of the operations to these "first order" numbers [1].

It turns out that we can add more of these ε symbols, and corresponding algebraic identities, to encode higher order derivatives and even arbitrary *partial* derivatives of multivariate functions.

To calculate $f^{(10)}(x)$ of $f(x) = exp(x^2)$ now takes only 59 operations[2]. To see how this is at least *possible*, consider the equation for $f^{(3)}(x)$ above. Note the redundancy; for example $\exp(x^2)$ appears 3 times. In effect, this method is able to eliminate such redundant computation that typically comes from differentiating.

These techniques are studied by the mathematicians in *Synthetic Differential Geometry*. We will use a slightly different formulation of the same idea, known by the computer scientists as *Automatic Differentiation*.

[1] Note also, the way complex numbers extend the reals by adding a new symbol i whose square is -1.
[2] See `pydx.test.test_nops`.

4.2 Recursive Formulae for the Calculation of Normalized Derivatives

We will make extensive use of *multi-index* notation, denoted with a boldface letter:

$$\mathbf{n} = (n_1, ..., n_r)$$

We also define various operations on multi-indices. The norm of a multi index, $|\mathbf{n}|$, is the sum of its components:

$$|\mathbf{n}| := \sum_i n_i.$$

We add, subtract and compare multi-indices *component-wise*:

$$\mathbf{n} \pm \mathbf{j} := (n_1 \pm j_1, ..., n_r \pm j_r),$$

$$\mathbf{j} \leq \mathbf{n} \quad \text{iff} \quad n_1 \leq j_1, \text{ and } ..., n_r \leq j_r,$$

and similarly for other multi-index comparisons. The zero multi-index $\mathbf{0}$ is zero in all its components. The unit multi-index $\mathbf{e}_i$ is zero in every component except for the *i'th* component which is one. We define factorial as:

$$\mathbf{n}! := \prod_{i=1}^{i=r} n_i!$$

Given a function $f : \mathbb{R}^r \to \mathbb{R}$ we use multi-indices to refer to a specific partial derivative:

$$f^{(\mathbf{n})}(x) := \Big(\frac{\partial}{\partial x^1}\Big)^{n_1} ... \Big(\frac{\partial}{\partial x^r}\Big)^{n_r} f(x).$$

We will also have use for the raising of a vector to a multi-index power:

$$h^{\mathbf{n}} := \prod_i h_i^{n_i}, \quad h \in \mathbb{R}^r$$

The *normalized* $\mathbf{n}$-*derivative* is defined in terms of the derivative:

$$f^{[\mathbf{n}]}(x) := \frac{1}{\mathbf{n}!} f^{(\mathbf{n})}(x).$$

This is now in the form of a Taylor series coefficient. Working with normalized derivatives has the advantage of simplifying many of the equations below. For example:

Theorem 4.2.1. (Generalized Leibniz's Rule) *If f and g are real-valued C^∞ functions on (an open domain in) $\mathbb{R}^r$, then*

$$(fg)^{[\mathbf{n}]}(x) = \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{n}} f^{[\mathbf{j}]}(x) g^{[\mathbf{n}-\mathbf{j}]}(x)$$

where the sum is taken over all multi-indices $\mathbf{j}$ *such that* $\mathbf{0} \leq \mathbf{j} \leq \mathbf{n}$.

Proof. We use induction on the Leibniz rule $(fg)'(x) = f'(x)g(x) + f(x)g'(x)$ to obtain:

$$(fg)^{(\mathbf{n})}(x) = \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{n}} \binom{n_1}{j_1} \ldots \binom{n_r}{j_r} f^{(\mathbf{j})}(x) g^{(\mathbf{n}-\mathbf{j})}(x)$$

Now expand the choice symbols, and rewrite in terms of normalized derivatives:

$$\mathbf{n}!\ (fg)^{[\mathbf{n}]}(x) = \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{n}} \frac{\mathbf{n}!}{\mathbf{j}!(\mathbf{n}-\mathbf{j})!}\ \mathbf{j}!\ f^{[\mathbf{j}]}(x)\ (\mathbf{n}-\mathbf{j})!\ g^{[\mathbf{n}-\mathbf{j}]}(x)$$

All the products of indices cancel and we have the result.

□

The following lemma shows how to compose normalized derivatives.

Lemma 4.2.2. *Let* f *be a real-valued* C^∞ *function on (an open domain in)* $\mathbb{R}^r$. *Then,*

$$f^{[\mathbf{n}][\mathbf{k}]}(x) = \frac{(\mathbf{n}+\mathbf{k})!}{\mathbf{n}!\mathbf{k}!} f^{[\mathbf{n}+\mathbf{k}]}(x).$$

Proof. We use the definition of the normalized derivative, together with the fact that derivatives behave like exponents; $f^{(\mathbf{n})(\mathbf{k})}(x) = f^{(\mathbf{n}+\mathbf{k})}(x)$.

$$\begin{aligned} f^{[\mathbf{n}][\mathbf{k}]}(x) &= \frac{1}{\mathbf{n}!\mathbf{k}!} f^{(\mathbf{n})(\mathbf{k})}(x) \\ &= \frac{1}{\mathbf{n}!\mathbf{k}!} f^{(\mathbf{n}+\mathbf{k})}(x) \\ &= \frac{(\mathbf{n}+\mathbf{k})!}{\mathbf{n}!\mathbf{k}!} f^{[\mathbf{n}+\mathbf{k}]}(x) \end{aligned}$$

□

The next corollary is the key ingredient in finding normalized derivatives of the transcendental functions in **Proposition 4.2.4**. It tells us how to "bootstrap" an equation involving first order differentials to arbitrary order.

Corollary 4.2.3. *Let* f, g *and* h *be real-valued* C^∞ *functions on (an open domain in)* $\mathbb{R}^r$ *such that* $f^{[\mathbf{e}_i]}(x) = g(x)h^{[\mathbf{e}_i]}(x)$. *If* $\mathbf{e}_i \leq \mathbf{k}$, *then*

$$f^{[\mathbf{k}]}(x) = \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{k}-\mathbf{e}_i} \frac{(k_i - j_i)}{k_i} g^{[\mathbf{j}]}(x) h^{[\mathbf{k}-\mathbf{j}]}(x)$$

Proof. Let $f^{[\mathbf{e}_i]}(x) = g(x)h^{[\mathbf{e}_i]}(x)$. We take the $\mathbf{k}-\mathbf{e}_i$'th normalized derivative of both sides:

$$\begin{aligned} f^{[\mathbf{e}_i][\mathbf{k}-\mathbf{e}_i]}(x) &= \frac{\mathbf{k}!}{\mathbf{e}_i!(\mathbf{k}-\mathbf{e}_i)!} f^{[\mathbf{k}]}(x) && (\textbf{Lemma}) \\ &= k_i f^{[\mathbf{k}]}(x) \end{aligned}$$

$$\begin{aligned} \big(g(x)h^{[\mathbf{e}_i]}(x)\big)^{[\mathbf{k}-\mathbf{e}_i]} &= \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{k}-\mathbf{e}} g^{[\mathbf{j}]}(x) h^{[\mathbf{e}_i][\mathbf{k}-\mathbf{e}_i-\mathbf{j}]}(x) && (\textbf{Leibniz}) \\ &= \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{k}-\mathbf{e}} g^{[\mathbf{j}]}(x) \frac{(\mathbf{k}-\mathbf{j})!}{\mathbf{e}_i!(\mathbf{k}-\mathbf{e}_i-\mathbf{j})!} h^{[\mathbf{k}-\mathbf{j}]}(x) && (\textbf{Lemma}) \\ &= \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{k}-\mathbf{e}} g^{[\mathbf{j}]}(x) \frac{(k_i-j_i)!}{(k_i-1-j_i)!} h^{[\mathbf{k}-\mathbf{j}]}(x) \\ &= \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{k}-\mathbf{e}} (k_i-j_i) g^{[\mathbf{j}]}(x) h^{[\mathbf{k}-\mathbf{j}]}(x) \end{aligned}$$

And the result follows.

□

The following proposition forms the heart of the recursive implementation of automatic differentiation described in section 4.6.

Proposition 4.2.4. *Given g and h real-valued C^∞ functions on (an open domain in) $\mathbb{R}^r$ and multi-index $\mathbf{n}$ such that $\mathbf{e}_i \le \mathbf{n}$, we have the following:*

(1) If $f(x) = g(x) \pm h(x)$, then

$$f^{[\mathbf{n}]}(x) = g^{[\mathbf{n}]}(x) \pm h^{[\mathbf{n}]}(x).$$

(2) If $f(x) = g(x)h(x)$, then

$$f^{[\mathbf{n}]}(x) = \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{n}} g^{[\mathbf{n}-\mathbf{j}]}(x) h^{[\mathbf{n}]}(x).$$

(3) If $f(x) = g(x)/h(x)$, then

$$f^{[\mathbf{n}]}(x) = \frac{1}{h^{[\mathbf{0}]}(x)} \Big[g^{[\mathbf{n}]}(x) - \sum_{\mathbf{j}>\mathbf{0}}^{\mathbf{n}} h^{[\mathbf{j}]}(x) f^{[\mathbf{n}-\mathbf{j}]}(x) \Big].$$

(4) If $f(x) = e^{g(x)}$, *then*

$$f^{[\mathbf{n}]}(x) = \frac{1}{n_i} \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{n}-\mathbf{e}_i} (n_i - j_i) f^{[\mathbf{j}]}(x) g^{[\mathbf{n}-\mathbf{j}]}(x).$$

(5) If $f(x) = \ln g(x)$, *then*

$$f^{[\mathbf{n}]}(x) = \frac{1}{g^{[\mathbf{0}]}(x)} \Big[g^{[\mathbf{n}]}(x) - \frac{1}{n_i} \sum_{\mathbf{j}>\mathbf{0}}^{\mathbf{n}-\mathbf{e}_i} (n_i - j_i) g^{[\mathbf{j}]}(x) f^{[\mathbf{n}-\mathbf{j}]}(x) \Big].$$

(6) If $f(x) = \sqrt{g(x)}$, *then*

$$f^{[\mathbf{n}]}(x) = \frac{1}{f^{[\mathbf{0}]}(x)} \Big[\frac{1}{2} g^{[\mathbf{n}]}(x) - \frac{1}{n_i} \sum_{\mathbf{j}>\mathbf{0}}^{\mathbf{n}-\mathbf{e}_i} (n_i - j_i) f^{[\mathbf{j}]}(x) f^{[\mathbf{n}-\mathbf{j}]}(x) \Big].$$

(7) If $f(x) = \tan^{-1} g(x)$, *define* $h(x) = \frac{1}{1+g(x)}$, *then*

$$f^{[\mathbf{n}]}(x) = \frac{1}{n_i} \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{n}-\mathbf{e}_i} (n_i - j_i) h^{[\mathbf{j}]}(x) g^{[\mathbf{n}-\mathbf{j}]}(x).$$

(8) If $f(x) = \sin^{-1} g(x)$, *define* $h(x) = \frac{1}{\sqrt{1-g^2(x)}}$, *then*

$$f^{[\mathbf{n}]}(x) = \frac{1}{n_i} \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{n}-\mathbf{e}_i} (n_i - j_i) h^{[\mathbf{j}]}(x) g^{[\mathbf{n}-\mathbf{j}]}(x).$$

(9) If $f(x) = \sin h(x)$ *and* $g(x) = \cos h(x)$, *then*

$$f^{[\mathbf{n}]}(x) = \frac{1}{n_i} \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{n}-\mathbf{e}_i} (n_i - j_i) g^{[\mathbf{j}]}(x) h^{[\mathbf{n}-\mathbf{j}]}(x).$$

$$g^{[\mathbf{n}]}(x) = -\frac{1}{n_i} \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{n}-\mathbf{e}_i} (n_i - j_i) f^{[\mathbf{j}]}(x) h^{[\mathbf{n}-\mathbf{j}]}(x).$$

Proof. (1) is obvious. (2) is Leibniz's formula. To prove (3), apply (2) to $g(x) = f(x)h(x)$.

For the next items, we write an equation matching the form of the hypothesis of corollary 4.2.3, so that we can bootstrap to arbitrary orders.

(4) $f^{[\mathbf{e}_i]}(x) = f(x) g^{[\mathbf{e}_i]}(x)$,
(5) $g^{[\mathbf{e}_i]}(x) = g(x) f^{[\mathbf{e}_i]}(x)$,
(6) $g^{[\mathbf{e}_i]}(x) = 2 f(x) f^{[\mathbf{e}_i]}(x)$,
(7) and *(8)* $f^{[\mathbf{e}_i]}(x) = h(x) g^{[\mathbf{e}_i]}(x)$,
(9) $f^{[\mathbf{e}_i]}(x) = g(x) h^{[\mathbf{e}_i]}(x)$, and $g^{[\mathbf{e}_i]}(x) = -f(x) h^{[\mathbf{e}_i]}(x)$.

□

We are now able to justify the efficiency claims made in the introduction.

Corollary 4.2.5. *The number of arithmetic operations required to compute the normalized derivatives of a function on $\mathbb{R}^r$ up to order $\mathbf{n}$ is $O(\prod_i n_i^2)$.*

Other operations can be derived in terms of the proceeding formulae. For example,
(10) If $f(x) = g(x)^t$, then $f(x) = e^{r \ln g(x)}$, and
(11) If $f(x) = \sinh(g(x))$, then $f(x) = \frac{1}{2}(e^{g(x)} - e^{-g(x)})$.

4.3 Taylor's Theorem

The following theorem is a key ingredient of the ODE solver described in Chapter 5. We also use it in the unit tests (section 4.8).

Theorem 4.3.1. Taylor's Theorem *Given real-valued C^∞ function f defined on $\mathbb{R}^r$, and some $h \in \mathbb{R}^r$, there exists $\xi \in \mathbb{R}^r$ with $|\xi_i| \leq |h_i|$, such that*

$$f(x+h) = \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{n}} h^{\mathbf{j}} f^{[\mathbf{j}]}(x) \; + \sum_{|\mathbf{j}|=|\mathbf{n}|+1} h^{\mathbf{j}} f^{[\mathbf{j}]}(\xi).$$

An exciting thing to note is that this can be reformulated in terms of *intervals*[3]

Theorem 4.3.2. Taylor's Theorem (interval form) *Let f be a real-valued C^∞ function defined on $\mathbb{R}^r$, and $h \in \mathbb{R}^r$. Define the vector ξ with interval components $\xi_i = \text{hull}(0, h_i)$. Then we have*

$$f(x+h) \in \sum_{\mathbf{j}=\mathbf{0}}^{\mathbf{n}} h^{\mathbf{j}} f^{[\mathbf{j}]}(x) \; + \sum_{|\mathbf{j}|=|\mathbf{n}|+1} h^{\mathbf{j}} f^{[\mathbf{j}]}(\xi).$$

4.4 Jets of Functions

Loosely speaking, the *jet* of a function $f : \mathbb{R} \to \mathbb{R}$ at a point $x = a \in \mathbb{R}$ is the sequence of its (normalized) derivatives. For the univariate case we use the bracket notation:

$$\langle f^{[0]}(a), f^{[1]}(a), ... \rangle.$$

So, a constant function $f(x) = b$ will have jet $\langle b, 0 \rangle$, the linear function $f(x) = x$ will have jet $\langle x, 1 \rangle$.

[3] In this chapter we refrain from using boldface font for interval values as this would conflict with the notation for multi-indices.

The idea now is that we "forget" about the function and compute just with the jets themselves. An example calculation would proceed

$$\langle x, 1\rangle^2 = \langle x^2, 2x\rangle.$$

What is going on here? We have a jet $\langle x, 1\rangle$, which we represent with the function $g(x) = x$. We take another function, $f(x) = x^2$. Then, we are calculating the zeroth and first derivative of $f \circ g$ using the chain rule. The chain rule is merely a recursive formula for finding these derivatives. The bracket notation embodies this recursion by placing the derivative information in the foreground.

More formally, we define a jet as the *equivalence class* of all (smooth) functions that "go through" the jet. That is, the functions in each equivalence class share all derivatives up to a specified order.

Definition 4.4.1. *Given some multi-index* $\mathbf{n}$*, the equivalence relation* $\sim_{\mathbf{n}}$ *at a fixed base point* $x = a \in \mathbb{R}^r$*, is defined between smooth functions* $\mathbb{R}^r \to \mathbb{R}$ *as*

$$f \sim_{\mathbf{n}} g \quad \textit{iff} \quad f^{[\mathbf{j}]}(a) = g^{[\mathbf{j}]}(a) \quad \textit{for} \quad \mathbf{0} \leq \mathbf{j} \leq \mathbf{n}.$$

Each of the resulting equivalence classes is known as a rank-r $\mathbf{n}$*-jet, and the set of all these equivalence classes is denoted*

$$J^r_{\mathbf{n}}(a) := \{f : \mathbb{R}^r \to \mathbb{R} | f \textit{ is smooth}\}/\sim_{\mathbf{n}}.$$

Note that the rank r is exactly the length of the multi-index n. So, for $r = 0$, there is only one possible index $\mathbf{n}$, the empty tuple. Also, the only function in a rank-0 jet is a nullary function, ie. a constant. In this way we can identify $J^0_{()}(a)$ with $\mathbb{R}$.

A specific equivalence class in $J^r_{\mathbf{n}}(a)$ can be referred to by specifying the derivatives of one of the functions in the equivalence class.

Definition 4.4.2. Univariate bracket notation $\langle y_0, y_1, ..., y_n\rangle \in J^r_n(a)$ *is defined to be the equivalence class containing the function*

$$f(x) = \sum_{j=0}^{n} y_j (x - a)^j.$$

Calculating with jets makes sense because of the following lemma.

Lemma 4.4.3. *Given smooth functions* f*,* g*,* h*, with* $h \sim_{\mathbf{n}} g$ *at* $x = a$*,*

$$(f \circ g)^{[\mathbf{n}]}(a) = (f \circ h)^{[\mathbf{n}]}(a)$$

That is, $(f \circ g)^{[\mathbf{n}]}(a)$ *depends only on the derivatives of* $g(x)$*, at* $x = a$*, up to order* $\mathbf{n}$*.*

Proof. Induction using the chain rule. □

Indeed, we use the notation in expressions such as $f(\langle y_0, ..., y_n \rangle)^{[j]}$ to imply that we choose some function $g \in \langle y_0, ..., y_n \rangle$ and compute $(f \circ g)^{[j]}(a)$. The lemma tells us that as long as we don't differentiate $f \circ g$ too much (ie. $j \leq n$), the choice of g does not matter. And what is the variable x ? It is the dependent variable, but we don't care; the data contained in the components of the jet allow us to calculate the function value $f(g(a))$ and derivatives (with respect to x) via the chain rule. We do not carry the base point a through the calculation, we only carry the data contained in the jet(s).

Note the careful placement of the $[j]$ superscript in the previous paragraph. The expression $f(\langle y_0, ..., y_n \rangle)^{[j]}$ is extracting the j-th component from a jet, which is quite different from $f^{[j]}(\langle y_0, ..., y_n \rangle)$: computing *with* the j-th derivative of f.

We emphasize this notation for two reasons. In the next section we will show that these jet objects are precisely the objects that are implemented in `pydx.` And, in Chapter 6, we will show how an initial value problem *defines* a jet.

4.5 Example: Computing with `Jet`'s

In this section we look at how automatic differentiation works in `pydx.` We will show examples of constructing jet objects, computing with them, then extracting components from the resulting jets.

Central to the implementation is the `Jet` type, defined in the `pydx.mjet` module. We use this type to construct jets manually. Behind the scenes there are other types of jet objects: these are used for each of the (algebraic) operations on jets.

For a first example, we take the function $f(x) = x^2$. We define this in Python with a `lambda`

```
>>> f = lambda x: x**2
```

Suppose we seek the first derivative of f at the point $x = 3$. This corresponds to the expression

$$f(\langle 3, 1 \rangle)^{[1]}.$$

Translating directly into code, we use the `Jet` class from the `pydx.mjet` module:

```
>>> from pydx.mjet import Jet
>>> f(Jet([3.0, 1.0]))[1]
6.0
```

The object `Jet([3.0, 1.0])` is a univariate jet. We instantiate these using lists (in this case `[3.0, 1.0]`), and extract components using a single index.

The components of *multivariate* `Jet`'s are indexed using a tuple of indices. Given two (independent) parameters, `x` and `y`, with values `2.0` and `3.0` respectively, their `Jet`'s are defined as:

```
>>> x = Jet({(0,0):2.0, (1,0):1.0})
>>> y = Jet({(0,0):3.0, (0,1):1.0})
```

These are now *rank-2* jets, and we can use them to find partial derivatives. For example, to determine the first derivative of $\sin(x^2 + y)$ with respect to x:

```
>>> from pydx.scalar.fmath import sin
>>> sin(x**2+y)[1,0]
3.01560901737
```

Note how we extract the component `[1,0]`. This represents the first differential w.r.t x and the zeroth differential w.r.t. y.

To find

$$\frac{\partial}{\partial x^2 \partial y^3} \sin(x^2 + y)$$

we would extract the component `[2,3]`

```
>>> sin(x**2+y)[2,3]
1.1147007722442044
```

The question that arises is, what happens when we compute the expression `sin(x**2+y)`? Exactly how many derivatives are stored? The answer is that no derivatives are computed or stored, not even the value (the zeroth derivative) is computed. The object merely remembers how it was constructed, and then responds to our indexing requests appropriately. That is, the derivatives are computed *on demand* (also known as *lazy* evaluation). This is key to the ease of use of this system, as we do not need to specify up front (globally or per-variable) how many derivatives will be needed.

In the first example above we produced the lazy square of a `Jet`:

```
>>> Jet([3.0,1.0])**2
SqrJet(<0:3.0 1:1.0>)
```

And the multivariate example is a compound of several types:

```
>>> sin(x**2+y)
SinJet(AddJet(SqrJet(<00:2.0 10:1.0>),<00:3.0 01:1.0>))
```

We will discuss this zoo of types further in section 4.6.

Note the brevity of the string representation of the `Jet`'s:

```
>>> x
<00:2.0 10:1.0>
>>> y
<00:3.0 01:1.0>
```

It is possible to modify a `Jet` after it has been constructed:

```
>>> x[0,0]=99.
>>> x
<00:99.0 10:1.0>
```

If we try and set or access components of a `Jet` with the wrong number of indices, an exception is raised:

```
>>> x[0]=99.
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "pydx/pydx/mjet.py", line 904, in __setitem__
    assert len(idxs)==self.rank
AssertionError
```

Sometimes a calculation involves rank-0 `Jet`'s. As noted in the remark following definition 4.4.1, these are essentially scalar values (constants). The value is accessed with the empty tuple `()`:

```
>>> x=Jet(rank=0)
>>> x
<:0.0>
>>> x[()]
0.0
>>> x[()]=5.0
>>> x[()]
5.0
```

4.5.1 The Scalar Stack

The `Jet` objects are sensitive to the type of scalar used for the components. If we try to set a component using the wrong type we will get an error. For example if we use an integer type:

```
>>> x=Jet(rank=3)
>>> x[0,0,0]=456
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "pydx/pydx/mjet.py", line 906, in __setitem__
    assert isinstance(c, self.scalar.type)
AssertionError: attempt to set item (0, 0, 0) to non-scalar 456
```

The way we use different scalar types is with the `pydx.scalar` module. Here we request the 64 bit precision `mpf` scalar type:

```
>>> from pydx.scalar import set_mpf_scalar
>>> set_mpf_scalar(64)
```

The `Jet` components are promoted to whatever the current scalar type is set to

```
>>> x=Jet({(0,0):2.0,(1,0):1.0})
>>> x
<00:mpf('2.e0',64) 10:mpf('1.e0',64)>
```

These scalar types are stored on a stack, so when we are done we can restore to the previous the scalar type:

```
>>> from pydx.scalar import restore_scalar
>>> restore_scalar()
```

Note that several of the components of `pydx` effectively subscribe to the current scalar type. For example, the `Tensor` and `TensorField` types (which we will meet in Chapter 6) are also scalar sensitive.

4.5.2 The Symbolic Scalars

The techniques in this section will allow us to greatly speed repeated evaluation of jet expressions. We will also use these techniques to manipulate computations symbolically; in chapter 7 we will be able to express the exponential function as the solution of an initial value problem.

The symbolic scalars are a lazy scalar type. They defer calculation in a similar way as does the `Jet` machinery.

```
>>> from pydx.scalar.symbolic import Var
>>> from pydx.scalar import fmath
>>> x = Var('x')
>>> y = Var('y')
>>> z = fmath.sin(x**2+y)
>>> print z.deepstr()
SinFloat(AddFloat(IntPowFloat(Var('x'), 2), Var('y')))
```

The "seed" type here is the `Var` type. It represents a variable we will be calculating with. Instances of the compound types `SinFloat`, `AddFloat`, `IntPowFloat`, were then created in response to the operations in `fmath.sin(x**2+y)`.

Once we have constructed a symbolic object there are several things we can ask for. One possibility is to get a Python expression:

```
>>> print z.expr()
fmath.sin(((x ** 2) + y))
```

We have also implemented functionality for producing expressions suitable for use in the Taylor software package of Jorba and Zou [9]:

```
>>> print z.jz_expr()
sin((x ^ 2) + y)
```

Because the symbolic scalars are part of the scalar subsystem of `pydx`, we can use them as a scalar type when performing computations with jets. In the following example, we show how to produce successive derivatives of $\exp(x^2)$, symbolically:

```
>>> from pydx.scalar import fmath
>>> from pydx.mjet import Jet
>>> from pydx.scalar import set_symbolic_scalar, restore_scalar
>>> from pydx.scalar.symbolic import Var, OneFloat
>>> set_symbolic_scalar()
>>> x = Jet([Var('x'), OneFloat()])
>>> y = fmath.exp(x**2)
>>> print y[0]
fmath.exp((x ** 2))
>>> print y[1]
(fmath.exp((x ** 2)) * (2.0 * x))
>>> print y[2]
((((2.0 * fmath.exp((x ** 2))) * (2.0 / 2.0)) +
((fmath.exp((x ** 2)) * (2.0 * x)) * (2.0 * x))) / 2.0)
>>> restore_symbolic_scalar()
```

Expressions produced this way can easily contain thousands of operations, as noted in section 4.1. Indeed, if we use the `deeplen` method of the symbolic type we find how many operations (and constants) the system has produced for the tenth derivative:

```
>>> y[10].deeplen()
1720
```

We focus now on the symbolic object `y[2]` which is our symbolic representation of $f''(x) = \frac{d}{d^2x} exp(x^2)$.

The trick is, the jet machinery is *re-using* calculations. It is only when we display the "flattened" expression for `y[2]` that we see two occurrences of `fmath.exp((x ** 2))`. This becomes more clear if we examine the composition of the symbolic types.

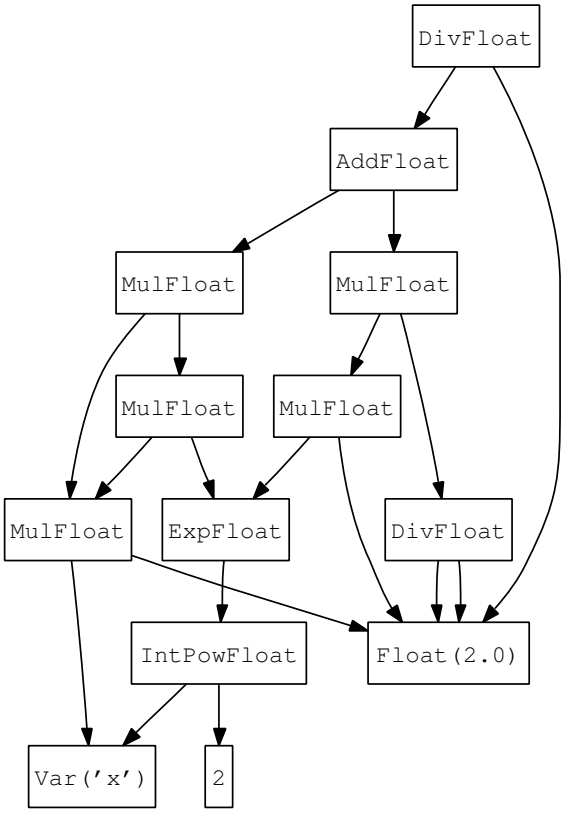


Fig 4.5.2.1 Structure of the symbolic representation of `y[2]`.

This diagram shows how the calculation of `fmath.exp((x ** 2))` is re-used twice (see how the `ExpFloat` node is referenced twice) in the calculation of `y[2]`.

So we cannot hope to work with the flattened expressions, we must use temporary variables so that we can re-use calculations.

The following example shows how to do this. The `ComputationContext` records symbolic calculation as it proceeds.

```
>>> from pydx.scalar.symbolic import ComputationContext
>>> set_symbolic_scalar()
>>> temp = open('temp.py', 'w')
>>> ctx = ComputationContext('func', ['x'], temp)
>>> ctx.assign('result', y[2])
>>> func = ctx.finalize('result')
>>> restore_symbolic_scalar()
```

We end up with a temporary file `temp.py` containing a function for computing `y[2]`.

```
def func(x):
    temp_1 = 2.0
    temp_2 = x
    temp_3 = (temp_2 ** 2)
    temp_4 = fmath.exp(temp_3)
    temp_5 = (temp_1 * temp_4)
    temp_6 = (temp_1 / temp_1)
    temp_7 = (temp_5 * temp_6)
    temp_8 = (temp_1 * temp_2)
    temp_9 = (temp_4 * temp_8)
    temp_10 = (temp_9 * temp_8)
    temp_11 = (temp_7 + temp_10)
    temp_12 = (temp_11 / temp_1)
    result = temp_12
    return result
```

And again, we see how the computation of `fmath.exp((x ** 2))` is reused (twice) by referencing the variable `temp_4`.

Now we have a Python function that we can use to calculate $f''(x) = \frac{d}{d^2x} exp(x^2)$. Moreover, we can use any of the different scalar types with this function.

```
>>> from pydx.scalar import set_mpf_scalar
>>> from gmpy import mpf
>>> x = mpf(1.2)
>>> set_mpf_scalar(128)
>>> func(x)
16.376299769946621924384014975142307801 11
>>> restore_scalar()
```

It was found that this technique of symbolic evaluation and code generation sped up computation many thousands of times when applied to the problems in differential geometry described in chapter 6 and 7.

4.6 Implementation: `pydx.mjet`

All the `Jet` object types derive from a "Meta-Jet" abstract base class. These objects store their rank, which is the dimensionality of the Jet (how many free variables we are differentiating with respect to). The current scalar is set as a class variable.

```
class MJet(object):

    scalar = None
```

```
    def __init__(self, rank):
        self.rank = rank
    ...
```

An `MJet` does not have a way of accessing its components (this is left for subclasses to implement), but every `MJet` *does* know how to participate in algebraic operations. Crucial to this is being able to "promote" various other objects to the same status as `self`. For example, scalars need to be promoted to a "constant" `Jet`.

```
class MJet(object):
    ...
    def promote(self, item):

        if not isinstance(item,MJet):
            item = self.scalar.promote(item)
            x = Jet({(0,)*self.rank:item})
            return x
        elif item.rank == self.rank:
            return item
        ...
```

This piece of code uses a "concrete" subclass of `MJet`, one whose components are directly stored, called simply a `Jet`.

```
class Jet(MJet):
    """ This is a concrete Jet.
        All components are stored in a dictionary.
    """

    def __init__(self, cs={}, rank=None):
        ...
        MJet.__init__(self, rank)
        self.cs = {} # coefficients, components, ...
        for idxs, value in cs.items():
            self[idxs] = self.scalar.promote(value)
```

Upon creation of a `Jet`, the components are promoted to the current scalar type.

The bulk of the `MJet` implementation is made up of the special methods that implement various algebraic operations. The key here is that no computation of components takes place. Instead, a *lazy* object is created that knows how to compute its components from those of its children. For example, this method implements the addition of two `MJet`'s.

```
class MJet(object):
    ...
    def __add__(self, other):
        return AddJet(self, other)
```

All the lazy `MJet`'s inherit from a `JetOp` class that manages a cache of components. This is so that when components are repeatedly requested we can eliminate recomputing them. A speed-up of around 30 times was observed by using this cache strategy.

```
class JetOp(MJet):
    """ Lazy Jet with item cache.
    """
    def __init__(self, rank):
        MJet.__init__(self, rank)
        self._cache = {} # speeds up things by *30
```

The `__getitem__` for a `JetOp` merely examines the cache and then calls `self.getitem` (implemented in the subclass) if the value is not found.

The meat of the actual computation of components is found in these `getitem` methods:

```
class AddJet(JetOp):

    def __init__(self, a, b):
        JetOp.__init__(self, a.rank)
        b = self.promote(b)
        self.a = a
        self.b = b

    def getitem(self, idxs):
        return self.a[idxs]+self.b[idxs]
```

Here we see how `AddJet` requests components from it's "children" and then adds them[4].

A more tricky example comes from the lazy multiplication `JetOp`[5]:

```
class MulJet(JetOp):
    ...
    def getitem(self, idxs):
        assert type(idxs)==tuple
        idxss = [ _ for _ in multi_range(idxs) ]
```

[4]See proposition 4.2.4, item (2).
[5]See proposition 4.2.4, item (3).

```
            _idxss = [ multi_sub(idxs,_idxs) for _idxs in idxss ]
            return sum((self.a[a_idxs] * self.b[b_idxs]
                for a_idxs, b_idxs in zip(_idxss,idxss)), self.scalar.zero)
```

This time we need to do some manipulation of the multi-index objects. The `multi_range` function yields multi-indices starting from the zero multi-index going up to the argument (*inclusive*):

```
    def multi_range(idxs):
        if len(idxs):
            for idx in range(0,idxs[0]+1):
                for rest in multi_range(idxs[1:]):
                    yield (idx,)+rest
        else:
            yield ()
```

`multi_sub` implements subtraction of multi-indices:

```
    def multi_sub(a_idxs, b_idxs):
        c_idxs = tuple(a_idxs[i]-b_idxs[i] for i in range(len(a_idxs)))
        return c_idxs
```

4.7 Implementation: Taylor Expansion

Evaluation of the Taylor polynomial (see **Theorem 4.3.1**) is computed with the `expand` method.

```
    class MJet(object):
        ...
        def expand(self, x, order):
            assert len(x)==self.rank
            r = self.scalar.zero
            for j in multi_range_order(self.rank, 0, order):
                s = self.scalar.one
                for i in range(self.rank):
                    if j[i]>0:
                        s *= x[i]**j[i]
                r = r + self[j] * s
            return r
```

The interval version (see **Theorem 4.3.2**) is calculated with the `expand_err` method.

```
    class MJet(object):
        ...
```

```
    def expand_err(self, x, order, err):
        assert len(x)==self.rank
        r = self.scalar.zero
        for j in multi_range_order(self.rank, 0, order):
            s = self.scalar.one
            for i in range(self.rank):
                if j[i]>0:
                    s *= x[i]**j[i]
            r = r + self[j] * s
        for i,j in enumerate(
                multi_range_order(self.rank, order+1, order+1)):
            s = self.scalar.one
            for k in range(self.rank):
                if j[k]>0:
                    s *= x[k]**j[k]
            r = r + err[i] * s
        assert i==len(err)-1
        return r
```

4.8 Unit Tests: Taylor Expansion

All the unit tests involve using the `MJet` machinery to compute coefficients of Taylor polynomials of functions. The polynomials are then evaluated at different points, and compared to the original function they approximate.

We use two ways of verifying this comparison. The first uses regular Python floating point scalars and an assertion of a bound on the (relative) error of the Taylor polynomial. The second uses the exact form of Taylor's theorem, along with the `Interval` scalars, and asserts that the value of the polynomial *overlaps* with the value of the function.

The `Interval` method verifies that we "do no wrong", while the floating-point method verifies that we are "approximately right".

Since an `MJet` can be interpreted as the coefficients of a Taylor polynomial, evaluation of such a polynomial is implemented as an `expand` method of the `MJet` class. Here we employ this method in asserting a relative error bound:

```
def test_jet_univariate_simple():
    f = lambda x: 1.0/(x+1.0)    # define some function
    order = 5
    x0 = random() - 0.5
    x = Jet([x0, 1.0])           # Concrete MJet
    h = 0.01*random()            # small float
    fx = f(x)                    # MJet
    fxh = f(x0+h)                # float
```

```
    fxe = fx.expand((h,), order) # float
    error = abs(fxe - fxh)
    relative_error = error / max(1e-6, error)
    assert relative_error < 1e-10
```

The bound `1e-10` is chosen in an *ad-hoc* fashion, as is the range of the `h` variable.

The `Interval` test is more involved (even more so for the multivariate tests). This time we produce an error `Interval` and pass it to the `MJet` method `expand_err`:

```
def test_jet_expand_2():
    f = lambda x: 1.0/(x+1.0)
    order = 5
    set_interval_scalar()
    x0 = Interval(random()-0.5)
    x = Jet([x0,1.0])
    h = Interval(random()-0.5)
    xx0 = x0.hull(x0+h)
    xx = Jet([xx0,1.0])
    err = f(xx)[order+1]
    fx = f(x)
    fxh = f(x0+h)
    fxe = fx.expand_err((h,), order, (err,))
    assert fxh.overlapping(fxe)
    restore_scalar()
```

This is a much more satisfying test; there is no ad-hoc choice of bounds, the `Interval` class takes care of all of that.

4.9 Bibliographic Notes

The first reference to automatic differentiation found is in Wengert [24] from 1964.

Griewank [4] discusses in detail the advantages automatic differentiation has over the symbolic methods.

We use multivariate automatic differentiation which is thoroughly worked out in Neidinger [17] in terms of recursive algorithms presented as pseudo-code. Jorba and Zou [10] have a much cleaner approach to automatic differentiation although they only develop the univariate case, which is sufficient for their purposes.

Our exposition is essentially a re-write of Neidinger's algorithms and data structures in terms of normalized-derivatives, following the style found in Jorba and Zou. We also use lazy-evaluation techniques which greatly simplifies the overall implementation and use of `pydx`. This means it is not necessary to know *apriori* to what order we will evaluate derivatives, whereas Neidinger's algorithms compute all derivatives statically.

Ordinary Differential Equations

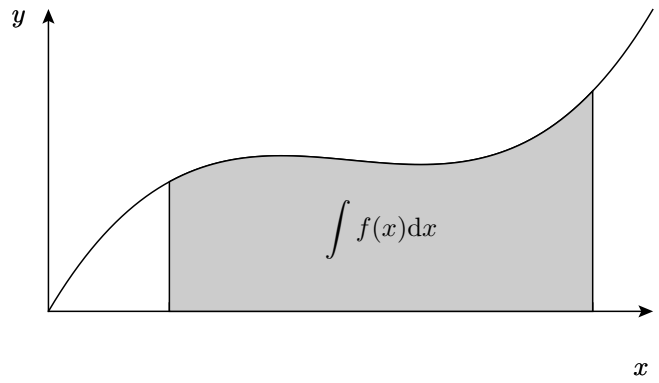
In this chapter we will use the tools we have developed to attack the problem of numerical solution of ordinary differential equations. Our goal will be the development of an arbitrary order solver based on the Taylor expansion, and one that will provide rigorous bounds on the solution. Along the way we will discover that much of traditional calculus is already framed in a way that makes it clear how to apply interval arithmetic.

In this chapter the theoretical exposition will be carried out in the one dimensional case, in order to ease the burden on the notation.

The theory developed here, as well as in the previous chapter, also forms the basis of the `Taylor`[1] ODE solver [9]. We will be using `Taylor` in chapter 7.

5.1 The Riemann Integral

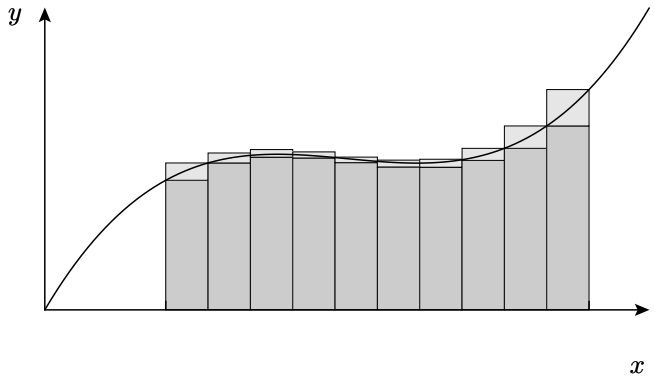
The integral of a function $f : \mathbb{R} \to \mathbb{R}$ is thought of geometrically as the "area under the curve":



The Riemannian integral is defined using the following squeeze formula

[1] We use `typewriter type` to denote this software package.

$$\sum_i \inf_{[x_i, x_i+h]} hf(x) \leq \int f(x)\mathrm{d}x \leq \sum_i \sup_{[x_i, x_i+h]} hf(x)$$



This picture suggests the following interval arithmetic approach. We evaluate the function using disjoint intervals that cover the domain of integration, and use the resulting intervals as bounds on the inf and sup in the above equation

$$\sum_i h\underline{f([x_i, x_i + h])} \leq \int f(x)\mathrm{d}x \leq \sum_i h\overline{f([x_i, x_i + h])}.$$

We have shown a flavour of performing integration with interval arithmetic, which we now proceed to generalize.

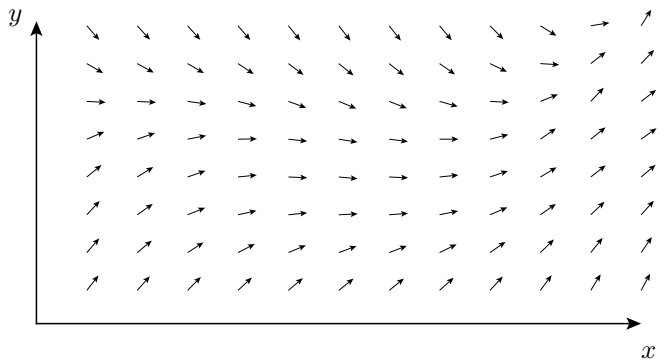
5.2 The Initial Value Problem

We are given a function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. The initial value problem (IVP) asks for a solution y to the following equation:

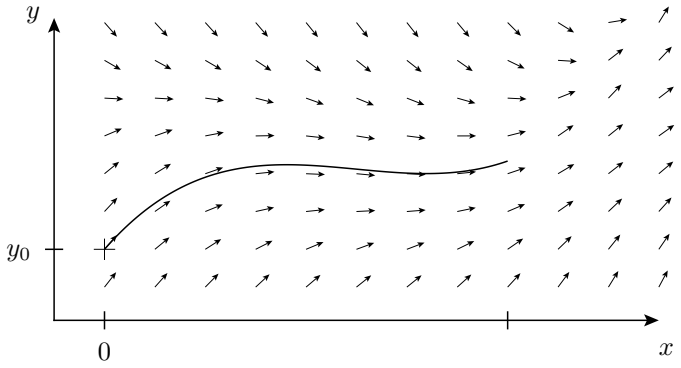
$$y'(x) = f(x, y(x)), \text{ and } y(0) = y_0$$

at least for x in some small range $[0, \varepsilon]$.

$f(x, y)$ specifies a gradient for each point in the plane



The IVP asks us to start at an initial point $(0, y_0)$ and follow the gradient function f:



The central proposition in the theory of ordinary differential equations tells us that these solutions do exist and are unique.

Proposition 5.2.1. *Given sufficiently nice*[2] $f(x, y)$*, the solution to the initial value problem exists and is unique for some domain* $x \in [0, \varepsilon]$*.*

Proof. Given a function $y(x)$ we define another function $(Ty)(x)$ as follows:

$$(Ty)(x) = \int_{x_0}^{x} f(x, y(x))dx, \quad x \in [x_0, x_1].$$

This map is a contraction:

$$||Ty_1 - Ty_2||_\infty \leq \frac{1}{2}||y_1 - y_2||_\infty$$

which implies there is a unique solution $y(x)$ to the initial value problem:

$$y'(x) = f(x, y(x)), \quad y(0) = y_0.$$

[2] *eg. smooth.*

□

This procedure is known as Picard iteration. In proposition (5.4.1) we will use an interval form of this to generate bounds on the range of y.

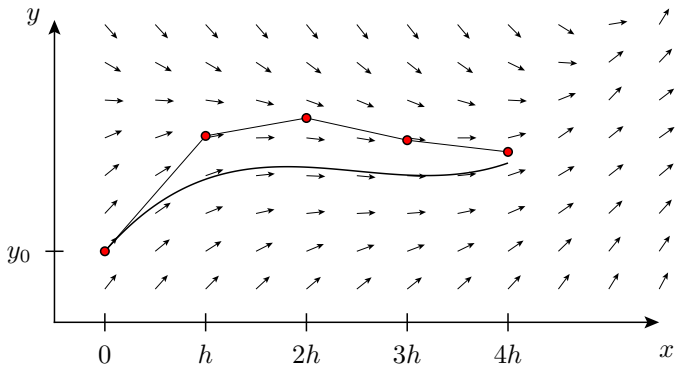
5.3 Euler Method

For a first attempt at a numeric solution, we discretize the domain into steps of size h and approximate f with a function that is constant over each step. This gives us a sequence of points (x_i, y_i) as follows

EULER STEP:

$$y_{i+1} = y_i + hf(x_i, y_i), \quad x_i = ih$$

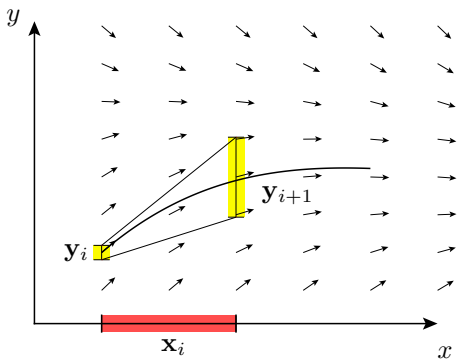
Geometrically, we are piecing together linear segments by following the gradient arrows at each step:



5.4 Interval Euler Method

We set about discovering an interval version of the preceding method.

From a bound $y(x_i) \in \mathbf{y}_i$ we can find a bound $y(x_{i+1}) \in \mathbf{y}_{i+1}$ using the derivative $y'(x)$. Observe that on the interval $\mathbf{x}_i := [x_i, x_{i+1}]$, $y(x)$ can increase no faster than the maximum of its derivative, and no slower than the minimum of its derivative:



$$\min_{x\in\mathbf{x}_i} y'(x) \le \frac{y(x_{i+1}) - y(x_i)}{h} \le \max_{x\in\mathbf{x}_i} y'(x)$$

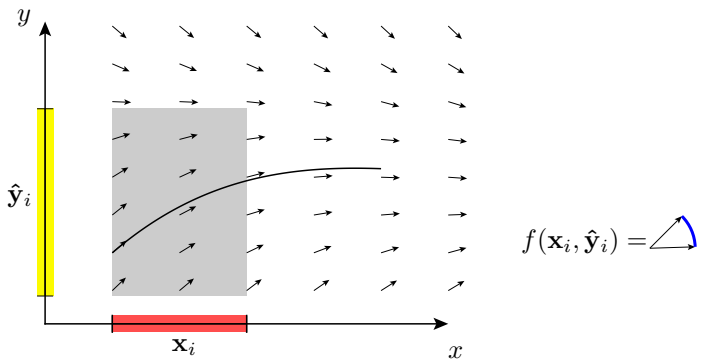
This is rephrased naturally as a statement involving intervals

$$\mathbf{y}_{i+1} := \mathbf{y}_i + h\mathbf{y}'([x_i, x_{i+1}]).$$

We can also see this as a kind of mean value theorem

$$y(x_{i+1}) = y(x_i) + hy'(\xi), \quad \xi \in [x_i, x_{i+1}].$$

The only missing piece is a bound for $y'(x)$ over $\mathbf{x}_i$. To this end, suppose that we manage to bound the *solution* $y(x)$ on this interval $\mathbf{x}_i$ with some interval $\hat{\mathbf{y}}_i$:



$$\mathbf{y}(\mathbf{x}_i) \subseteq \hat{\mathbf{y}}_i$$

Computing f with this 2-dimensional interval generates a bound on $y'(x)$

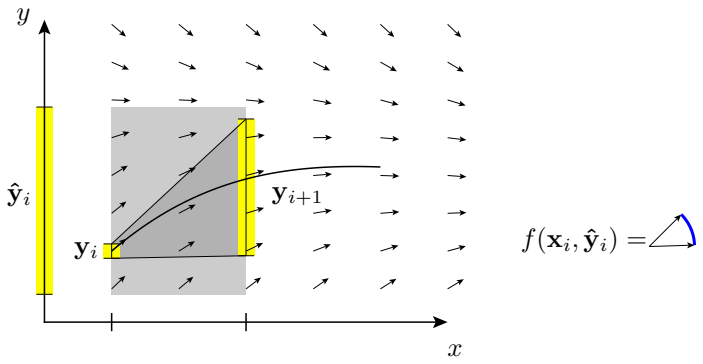
$$\mathbf{y}'(\mathbf{x}_i) \subseteq \mathbf{f}(\mathbf{x}_i, \hat{\mathbf{y}}_i),$$

ie. we are looking at the range of slopes of the little arrows in the shaded rectangle.

The interval Euler step is now in the form

$$\mathbf{y}_{i+1} := \mathbf{y}_i + h\mathbf{f}(\mathbf{x}_i, \hat{\mathbf{y}}_i).$$

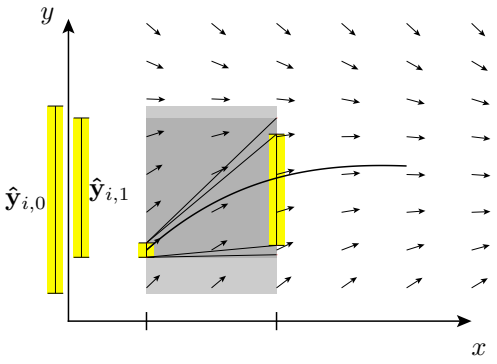
Not having access to the function $y'(x)$, we see that $\mathbf{y}_{i+1}$ is much wider than before:



Taking pause, we examine the darkly shaded quadrilateral above, and see that it must enclose the solution! We have found a (hopefully) *tighter* bound for $y(x)$,

$$\mathbf{y}(\mathbf{x}_i) \subseteq \mathbf{y}_i + [0, h]\mathbf{f}(\mathbf{x}_i, \hat{\mathbf{y}}_i).$$

A recipe for iteration – a kind of *interval Picard iteration* – presents itself, graphically and algebraically:



$$\hat{\mathbf{y}}_{i,k+1} = \mathbf{y}_i + [0, h]\mathbf{f}(\mathbf{x}_i, \hat{\mathbf{y}}_{i,k}).$$

It turns out that this iteration will also furnish us with a correct $\hat{\mathbf{y}}_i$ when we start with *any* interval $\hat{\mathbf{y}}_i$ and then iterate until we get a contraction:

Proposition 5.4.1. *Suppose:*

$$\hat{\mathbf{y}} \subseteq \mathbf{y}(\mathbf{x}) + [0, h] f(\mathbf{x}, \hat{\mathbf{y}})$$

then $\mathbf{y}(\mathbf{x}) \subseteq \hat{\mathbf{y}}$*, ie.* $y(x) \in \hat{\mathbf{y}}$ *for* $x \in \mathbf{x}$*.*

Proof. (Omitted.) □

In summary, we iterate $\hat{\mathbf{y}}_{i,k}$ to obtain a bound on $\mathbf{y}(\mathbf{x}_i)$, then take a step.

CONTRACT:

$$\hat{\mathbf{y}}_i \subseteq \mathbf{y}_i + [0, h]\ \mathbf{f}(\mathbf{x}_i, \hat{\mathbf{y}}_i)$$

STEP:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\ \mathbf{f}(\mathbf{x}_i, \hat{\mathbf{y}}_i)$$

5.4.1 Example: Interval Picard Contraction

Here we examine the equation $y'(x) = y(x)$, which is generated by the IVP defined by the function $f(x, y) := y$. We choose an initial bound of $[0, 0]$ and apply the contraction iteration some arbitrary number of times. At the end we ask if the contraction hypothesis is satisfied, and if so assert that the interval contains the (known) solution.

```
from pydx.scalar.mpfi import Interval

def f(x,y):
    return y

y0 = Interval(1.0)

h = 0.1 # Stepsize
hh = Interval(0.0, h)

xx = Interval(0.0, 0.1)

yy0 = Interval(0.0) # Initial bound
yy1 = yy0

for i in range(30):
```

```
        yy0 = yy1
        yy1 = y0 + hh * f(xx,yy0) # CONTRACT

    if yy0.contains(yy1):
        assert yy1.contains(xx.exp())
        print "bound:", yy1, "of", xx.exp()
    else:
        print "no bound"
```

This generates the output:

```
bound: [1.000000000,1.111111111] of [1.000000000,1.105170918]
```

5.5 Second Order Method

Now that we have a way of finding bounds on solutions to the IVP we seek to extend this technique to include higher order derivative information.

A second order Euler step would take the following form

$$y_{i+1} = y_i + hy'(x_i) + \frac{1}{2}h^2 y''(x_i).$$

We know that

$$y'(x_i) = f(x_i, y_i).$$

In order to find $y''(x_i)$ we differentiate $y'(x) = f(x, y(x))$ with respect to x

$$y''(x_i) = f_x(x_i, y(x_i)) + f_y(x_i, y(x_i))y'(x_i).$$

To use intervals we recast the Taylor theorem in the exact form

$$y(x_{i+1}) = y(x_i) + hy'(x_i) + \frac{1}{2}h^2 y''(\xi), \quad \xi \in [x_i, x_{i+1}],$$

and note that ξ is really just the interval $\mathbf{x}_i$

$$y(x_{i+1}) \in y(x_i) + hy'(x_i) + \frac{1}{2}h^2 \mathbf{y}''(\mathbf{x}_i).$$

We would like to write a step equation

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\mathbf{y}'_i + \frac{1}{2}h^2 \mathbf{y}''(\mathbf{x}_i).$$

Let us pause to examine the various terms involved. The $\mathbf{y}_i$ and $\mathbf{y}'_i$ are the initial conditions for the step, we hope their widths stay small. The $\frac{1}{2}h^2\mathbf{y}''(\mathbf{x}_i)$ is an error term:

$\mathbf{y}''(\mathbf{x}_i)$ is a bound for $y''(x)$ over *all* of $\mathbf{x}_i$ (not just at one of the endpoints). The only way we can control the size of this error is by making h small[3].

It is actually not necessary to simultaneously maintain all these values at each step. We can get by with just $\mathbf{y}_i$, our bound on the solution $y(x)$ at $x = x_i$.

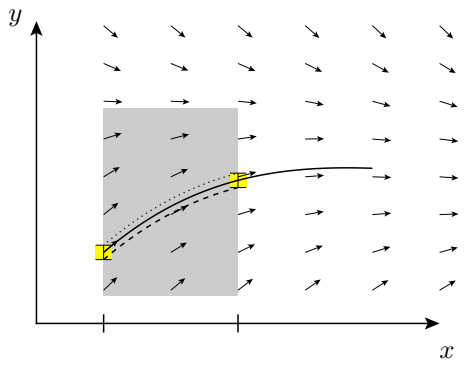
We bootstrap[4] using $f(x, y)$:

$$\mathbf{y}'_i \;:=\; f(x_i, \mathbf{y}_i)$$

For the error term, once we have a contraction (on an interval) $\hat{\mathbf{y}}_i$, we can bootstrap to the next higher order:

$$\begin{aligned} \hat{\mathbf{y}}'_i \;&:=\; f(\mathbf{x}_i, \hat{\mathbf{y}}_i) \\ \mathbf{y}''(\mathbf{x}_i) \;\subseteq\; \hat{\mathbf{y}}''_i \;&:=\; f_x(\mathbf{x}_i, \hat{\mathbf{y}}_i) + f_y(\mathbf{x}_i, \hat{\mathbf{y}}_i)\hat{\mathbf{y}}'_i \end{aligned}$$

A somewhat vague picture would look like this:



[3] In the next section we will further diminish this error by raising the exponent of h.
[4] A term meaning to produce results without outside help.

CONTRACT:

$$\hat{\mathbf{y}}_i \subseteq \mathbf{y}_i + [0,h]\ \mathbf{f}(\mathbf{x}_i, \hat{\mathbf{y}}_i)$$

GENERATE:

$$\begin{aligned} \mathbf{y}_i' &:= f(x_i, \mathbf{y}_i) \\ \hat{\mathbf{y}}_i' &:= f(\mathbf{x}_i, \hat{\mathbf{y}}_i) \\ \hat{\mathbf{y}}_i'' &:= f_x(\mathbf{x}_i, \hat{\mathbf{y}}_i) + f_y(\mathbf{x}_i, \hat{\mathbf{y}}_i)\hat{\mathbf{y}}_i' \end{aligned}$$

STEP:

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\mathbf{y}_i' + \frac{1}{2}h^2\hat{\mathbf{y}}_i''.$$

5.6 A General Interval Taylor Method

We examine an arbitrary order Taylor step (note the normalized derivatives):

$$y(x_{i+1}) = \sum_{j=0}^{n-1} h^j y^{[j]}(x_i) + h^n y^{[n]}(\xi),\ \xi \in [x_i, x_{i+1}].$$

This suggests an interval stepping equation

$$\mathbf{y}_{i+1} = \sum_{j=0}^{n-1} h^j \mathbf{y}_i^{[j]} + h^n \mathbf{y}^{[n]}(\mathbf{x}_i).$$

Differentiating $y'(x) = f(x, y(x))$ repeatedly, as in the last section, gives exponential growth in the size of the equations. Hence, as explained in Chapter 4, we seek a way to use automatic differentiation.

The next lemma describes how the components of the jet of the solution y are constrained by the initial value problem.

Lemma 5.6.1.

$$y^{[n]}(x) = \frac{1}{n} f(x, y(x))^{[n-1]}$$

Proof. We use lemma 4.2.2 which tells us how to compose normalized derivatives:

$$y^{[n]}(x) = \frac{1}{n} y^{[1][n-1]}(x) = \frac{1}{n} f(x, y(x))^{[n-1]}$$

□

The combination of this lemma with lemma (4.4.3) gives a recursive formula for $y^{[n]}(x)$

$$y^{[n]}(x) = \frac{1}{n} f(\langle x, 1\rangle, \langle y^{[0]}, ..., y^{[n-1]}\rangle)^{[n-1]}.$$

In other words, the initial value problem defines a jet! This is perhaps not so surprising, since the jet is merely that defined by the unique solution to the initial value problem. What is interesting is that we can use the algebraic form for f to drive a recursive formula for generating the components of the y jet.

Now we can extend the interval stepping algorithm to order n. At each step we maintain a $\mathbf{y}_i$, and find a contraction on some $\hat{\mathbf{y}}_i$. Then we use the recursive formula to generate $\mathbf{y}_i^{[j]}$ for $0 < j \leq n-1$ and $\hat{\mathbf{y}}_i^{[j]}$ for $0 < i \leq n$:

CONTRACT:

$$\hat{\mathbf{y}}_i \subseteq \mathbf{y}_i + [0, h]\ \mathbf{f}(\mathbf{x}_i, \hat{\mathbf{y}}_i)$$

GENERATE:

$$\mathbf{y}_i^{[n-1]} = \frac{1}{n} f(\langle x_i, 1\rangle, \langle \mathbf{y}_i^{[0]}, ..., \mathbf{y}_i^{[n-2]}\rangle)^{[n-2]}$$

$$\hat{\mathbf{y}}_i^{[n]} = \frac{1}{n} f(\langle x_i, 1\rangle, \langle \hat{\mathbf{y}}_i^{[0]}, ..., \hat{\mathbf{y}}_i^{[n-1]}\rangle)^{[n-1]}$$

STEP:

$$\mathbf{y}_{i+1} = \sum_{j=0}^{n-1} h^j \mathbf{y}_i^{[j]} + h^n \hat{\mathbf{y}}_i^{[n]}$$

This theory readily extends to the multidimensional case. There is still only one x variable, but now we have one y variable for each dimension, and one f function for each dimension:

$$y_0'(x) = f_0(x, y_0, ..., y_n)$$

$$...$$

$$y_n'(x) = f_n(x, y_0, ..., y_n).$$

Then we use the multivariate machinery developed in Chapter 4.

5.6.1 Implementation

The jet defined by the initial value problem is encapsulated in the `SprayItem` class. This is another `JetOp` that implements the recursive formula for the components of the jet. Therefore, once it has been instantiated, we can use it like any other jet. The `Spray` class handles the multi dimensional IVP. It behaves like a list of `MJet`'s (`SprayItem`'s).

The following three lines (from the `pydx.ode.ODE` class) construct the order n interval Taylor step:

```
y0 = Spray(f, x0, y0)
y = Spray(f, x, y)
y1 = [y0i.expand_err((h,), n-1, (yi[n],)) for y0i,yi in zip(y0,y)]
```

`x0` and `y0` are the initial conditions for the step: x_i and $\mathbf{y}_i$. `x`, `y` are the intervals $\mathbf{x}_i$ and $\hat{\mathbf{y}}_i$. `y1` is then a list containing $\mathbf{y}_{i+1}$ values, one for each dimension.

The `Spray` class behaves like a *list* of jets:

```
class Spray(object):
    def __init__(self, f, x, y):
        self.f = f
        self.x = x
        self.y = [SprayItem(self,idx,yi)
            for idx,yi in enumerate(y)]
        self.dy = f(x, self.y) # a list of MJet's
    def __getitem__(self, idx):
        return self.y[idx]
```

Each `SprayItem` implements the recursive rule for an IVP defining a jet[5]:

```
class SprayItem(JetOp):
    """ Created by Spray.
        These are the (MJet) components of a Spray object.
    """
    def __init__(self, spray, idx, yi):
        JetOp.__init__(self, rank=1)
        self.spray = spray
        self.idx = idx
        self.yi = MJet(1).promote(yi)
    def getitem(self, order):
        order, = order
        if order==0:
```

[5]See also the discussion from section 4.6.

```
            return self.yi[(0,)]
        dyi = self.spray.dy[self.idx]
        dyi = MJet(1).promote(dyi)
        r = (self.scalar.one/order) * dyi[order-1]
        return r
```

5.6.2 Unit Tests

We examine a second order differential equation

$$y''(x) = -y(x).$$

This is reduced to two first order equations:

$$y_0'(x) = y_1(x)$$
$$y_1'(x) = -y_0(x)$$

These are implemented in the `__call__` method of an `ODE` subclass:

```
class Pendulum(ODE):
    def __init__(self, x0, y0):
        ODE.__init__(self, x0, y0)
        assert len(y0)==2
    def __call__(self, x, ys):
        y, dy = y
        result = Tensor((Tensor.up,), self.dim)
        result[0] = dy
        result[1] = -y
        return result
```

The `Tensor` object is used to store the 2-dimensional result. It is another `list`-like class.

The test suite (`pydx.test.test_ode`) uses this class to test the various stepping algorithms described above. For example, this is a test of the first order interval stepper. Note the assertion which uses the interval valued cosine method.

```
# test first order interval method
# y''(x) = -y(x), y(0) = 1, y'(0) = 0
# solution: y(x) = cos(x)
x = ODE.scalar_zero
y, dy = ODE.scalar_one, ODE.scalar_zero
ode = Pendulum(x, [y,dy])
for i in range(steps):
```

```
x = Interval(ode.x).lower
Y0 = ode.contract1(Interval(x,(x+h).upper), ode.y)
ode.istep1(h, Y0)
y, dy = ode.y
assert ode.y[0].overlapping(ode.x.cos())
```

Differential Geometry

Differential geometry is the language used in general relativity to describe the geometry of space-time in a way that is independent of "point of view."

In this chapter we discuss the mathematics of differential geometry, and show how to perform the corresponding calculations in `pydx`. Section 6.2 and 6.3 introduce the `Tensor` class. Section 6.5 discusses the `TensorField` class and associated operations. In section 6.6 we show how to calculate some of the tensors found in Riemannian geometry, and in section 6.7 we discuss the geodesic equation.

6.1 Motivation

In this section we will motivate some of the formalism of differential geometry with two examples. The first example highlights a notation deficiency when doing calculations naively. The mathematical formalism essentially is designed to account for the domain and range of participating functions. The second motivating example shows why it is not only sometimes convenient, but can also be necessary, to use multiple coordinate systems.

We also jump in with a code example that shows some of how the formalism appears in `pydx`.

6.1.1 Coordinate Transforms

Suppose we have a 2-dimensional scalar field $f : \mathbb{R}^2 \to \mathbb{R}$,

$$f(x, y) = x^2 + y^2.$$

Here we may decide to work in polar coordinates. In this case, we write x and y as functions of the new variables

$$x(r, \theta) = r \cos\theta$$
$$y(r, \theta) = r \sin\theta$$

and substitute these into the equation for f to get

$$f(r, \theta) = r^2.$$

But these calculations get rather confusing, as we quickly lose track of what is a function and what is a (free) variable, and, what is operating on what, to produce what? Somehow we need to include the choice of coordinates in our formalism. Proceeding abstractly, we consider the scalar field f as a real valued function defined over some space $\mathcal{M}$

$$f : \mathcal{M} \to \mathbb{R}.$$

Now, when we specify a cartesian coordinate (x, y) or a polar coordinate (r, θ), we are using two different ways to refer to points in $\mathcal{M}$. We make this explicit by defining the maps

$$\phi_c : \mathcal{M} \to \mathbb{R}^2$$

and

$$\phi_p : \mathcal{M} \to \mathbb{R}^2.$$

These maps label points $p \in \mathcal{M}$: $\phi_c(p)$ is the cartesian coordinate of p and $\phi_p(p)$ is the polar coordinate of p.

Now we write the formula for our scalar field, with reference to the cartesian coordinate system

$$(f \circ \phi_c^{-1})(x, y) = x^2 + y^2.$$

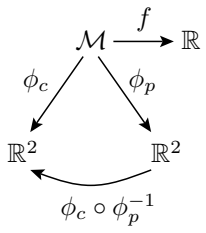
The change of coordinates

$$(\phi_c \circ \phi_p^{-1})(r, \theta) = (r \cos\theta, r \sin\theta),$$

yields the scalar field in the polar coordinate system

$$(f \circ \phi_c^{-1} \circ \phi_c \circ \phi_p^{-1})(r, \theta) = (f \circ \phi_p^{-1})(r, \theta) = r^2.$$

This is all summarized neatly by the arrow diagram



.

Next we would like to differentiate f. Concretely, given a real valued multivariate

function $f : \mathbb{R}^r \to \mathbb{R}$ the directional derivative of f at $x \in \mathbb{R}^r$ is a linear map

$$D_x f : \mathbb{R}^n \to \mathbb{R}.$$

Going back to the abstract function $f : \mathcal{M} \to \mathbb{R}$, it turns out that we can "lift" the directional derivative to a linear map

$$D_p f : T_p\mathcal{M} \to \mathbb{R}$$

which is coordinate free. Here, $T_p\mathcal{M}$ is an abstract r-dimensional *vector space*, the "tangent vectors" to the manifold at a point $p \in \mathcal{M}$. Once again, given a coordinate chart ϕ we can specify elements of $T_p\mathcal{M}$ using elements of $\mathbb{R}^r$.

For more details on how this is done, see the excellent notes by Ben Andrews [2] or the more physics oriented book by John Stewart [8].

6.1.2 The 2-Sphere

Let $\mathcal{M}$ be the following 2-dimensional subspace of $\mathbb{R}^3$

$$\mathcal{M} := \{(x, y, z) \in \mathbb{R}^3 | x^2 + y^2 + z^2 = 1\}.$$

Define a coordinate chart

$$\phi_N(x, y, z) := \big(\frac{1}{1-z}x, \frac{1}{1-z}y\big).$$

The chart is undefined at the point $z = 1$, so it can only label an (open) subset of $\mathcal{M}$

$$\phi_N : \big(\mathcal{M} - (0, 0, 1)\big) \to \mathbb{R}^2.$$

Another coordinate chart is necessary, to cover the entire space

$$\phi_S : \big(\mathcal{M} - (0, 0, -1)\big) \to \mathbb{R}^2,$$

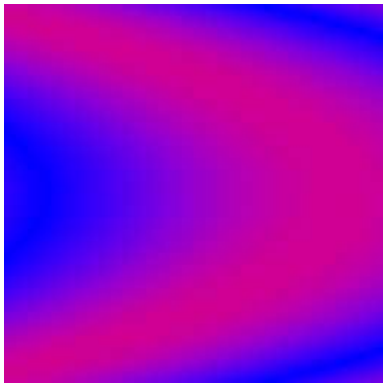
$$\phi_S(x, y, z) := \big(\frac{1}{z-1}x, \frac{1}{z-1}y\big).$$

6.1.3 Example

We define a scalar "heat" field:

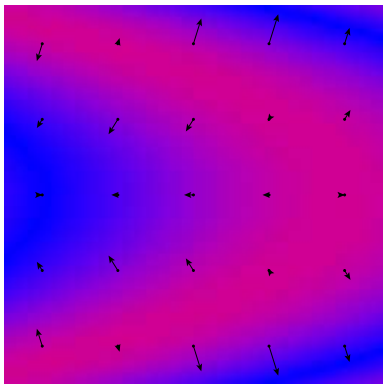
```
>>> from pydx.field import ConcreteTensorField
>>> from pydx.scalar.fmath import sin
>>> f = lambda x, y:sin(x**2 + y)
>>> r = ConcreteTensorField(dim=2)
```

```
>>> r[()] = f
```



And differentiate this to find the heat flow:

```
>>> v = -r.comma()
```



We can take this vector field, which is an instance of `TensorField` and slowly peel off the layers, as we descend the `pydx` stack[1]:

$$\texttt{TensorField} \to \texttt{Tensor} \to \texttt{MJet} \to \texttt{float}.$$

The first step is evaluating the field at a point:

```
>>> vp = v(1.0,1.0)
>>> print vp
NegJet(<:-0.832293673094>) NegJet(<:-0.416146836547>)
```

This is now a `Tensor` object with 2 `MJet` components. We extract these `MJet`'s as `vp[0]` and `vp[1]`. These are rank-0 `MJet`'s, containing a `float`:

[1] See Fig. 1.1.1.

```
>>> a = vp[0]
>>> print a[()]
0.832293673094
```

6.2 Tensors

Given a vector space V we define the space of (m, n)-tensors as the tensor product

$$T^m_n := \bigotimes_1^m V \otimes \bigotimes_1^n V^*.$$

For our purposes the ordering of the factors is important. For example we will distinguish between $V \otimes V^* \otimes V$ and $V \otimes V \otimes V^*$. For an (m, n)-tensor we will specify a *valence*, which is an $m + n$ length sequence of elements taken from the binary set {`up`, `dn`}, such than there is exactly m `up` elements and exactly n `dn` elements[2]. For example, $V \otimes V^* \otimes V$ has the valence (`up`, `dn`, `up`), and $V \otimes V \otimes V^*$ has valence (`up`, `up`, `dn`). The reason we need to do this is so that we can refer to a factor in the tensor product by its place (an integer index) in the product. We will refer to the space of tensors with valence p as p-tensors.

Given a basis $\{\mathbf{v}_a\}$ for V and dual basis $\{\mathbf{v}^a\}$ for V^*, we can define elements $\mathbf{t} \in T^m_n$ by their scalar components. For example, an element of $V \otimes V \otimes V^*$ is written[3]:

$$\mathbf{t} = t^{ab}{}_c \mathbf{v}_a \otimes \mathbf{v}_b \otimes \mathbf{v}^c.$$

The implementation `pydx.tensor.Tensor`, stores these basis-dependent components.

To construct a `Tensor` object, one specifies a valence, and the dimension of the underlying space V:

```
>>> from pydx.tensor import Tensor, up, dn
>>> t = Tensor((up, up, dn), 4)
```

The implementation also exposes the `rank` of the tensor, which is defined as the length of the valence, and the `shape` which is a tuple containing the dimension of each of the vector space factors:

```
class Tensor(object):
    def __init__(self, valence = (), dim = 4):
        self.valence = tuple(valence) # sequence of up/dn
```

[2]Note we are mixing mathematical notation with `computer code`.

[3] Note that we employ the Einstein summation convention whereby any repeated index is summed over.

```
        self.rank = len(valence)
        self.shape = (dim,)*self.rank
        self.dim = dim
        self.elems = {} # map index tuple to value
        ...
```

Note how the elements are stored as a mapping from some index tuple (a multi index) to the value of that component. Accessing elements works similarly to the `Jet` objects. To set $t^{ab}{}_{c} = 1$, we do the following:

```
>>> t[a, b, c] = 1.0
```

6.3 Operations on Tensors

Tensors with common dimension and valence live in a vector space. This means we have a zero element, we can add and subtract them, and scalar multiply:

```
class Tensor(object):
    ...
    def zero(self):
        tensor = self.__class__(self.valence, dim=self.dim)
        return tensor

    def __add__(self, other):
        tensor = self.zero()
        for idx in self.genidx():
            tensor[idx] = self[idx]+other[idx]
        return tensor

    def __rmul__(self, other):
        tensor = self.zero()
        for idx in self.genidx():
            tensor[idx] = other * self[idx]
        return tensor
```

The `genidx` method iterates over the indices of each component of the tensor.

6.3.1 Inner and Outer Products

Given a valence p and valence q, the sum $p+q$ is defined as the concatenation of the two sequences. Given p-tensor $\mathbf{s}$ and q-tensor $\mathbf{t}$ the *outer product* (or *tensor product*) $\mathbf{s} \otimes \mathbf{t}$ is a $(p+q)$-tensor defined by taking products of components.

This is computed with the `*` operator.

For example, the outer product of a vector and a co-vector:

$$t^a{}_b = u^a v_b,$$

is computed:

```
>>> t = u*v
```

To calculate inner products we use the `mul` method. This requires a list of "pairs" which associate an index in the left operand with an index in the right operand. For example, to find:

$$t^a{}_c = u^{ab} v_{bc},$$

we specify a pairing of `(1,0)` which pairs the second index of u with the first index of v:

```
>>> from pydx.tensor import Tensor, up, dn
>>> u = Tensor((up, up), 4)
>>> v = Tensor((dn, dn), 4)
>>> t = u.mul(v, (1,0)) # <--- pair (1,0)
```

The code becomes much more complicated when the same tensor is referred to more than once, with a different ordering of indices. Then the `transpose` method is used. It takes as arguments a permutation of $(0, ..., r-1)$ where r is the rank of the tensor. It is not valid to permute a contravariant index with a covariant index.

For an example, we add a $(0, 2)$-tensor to its transpose:

```
>>> u = Tensor((dn,dn), 2)
>>> u[0,1]=1. # <--- set off-diagonal element
>>> print u
0.0 1.0
0.0 0.0
>>> print u + u.transpose(1,0)
0.0 1.0
1.0 0.0
```

Further examples of using `transpose` appear in section 6.4 below.

6.4 Definition of a Manifold

Definition 6.4.1. *Given a space*[4] $\mathcal{M}$*, a chart on* $\mathcal{M}$ *is a one-to-one map* $\phi : U \to \mathbb{R}^n$*, where* $U \subset \mathcal{M}$ *is open.*

[4] *A hausdorff, paracompact topological space.*

Definition 6.4.2. *Two charts, $\phi_1 : U_1 \to \mathbb{R}^n$, and $\phi_2 : U_2 \to \mathbb{R}^n$ are C^∞-related if both the map*

$$\phi_2 \circ \phi_1^{-1} : \phi_1(U_1 \cap U_2) \to \phi_2(U_1 \cap U_2),$$

and its inverse are C^∞. A collection of C^∞-related charts whose domains cover $\mathcal{M}$ forms an atlas. The set of all such C^∞-related charts forms a maximal atlas. If $\mathcal{M}$ is a space and A its maximal atlas, the pair $(\mathcal{M}, A)$ is a C^∞-differentiable manifold. The dimension of the manifold is n.

6.5 Tensor Fields

A *tensor field* X of type p is defined as a function that evaluates to a p-tensor at each point of the manifold:

$$X : M \to T^p.$$

When we have a coordinate chart $\phi : U \to \mathbb{R}^n$ we can represent X as a function on $\phi(U)$ that produces p-tensors. A `TensorField` object is a function (callable) that returns a `Tensor` object.

The constructor is similar to the `Tensor` constructor.

```
class TensorField(object):
    def __init__(self, valence = (), dim = 4):
        self.valence = tuple(valence) # sequence of up/dn
        self.rank = len(valence)
        self.shape = (dim,)*self.rank
        self.dim = dim
        ...
```

However, the `TensorField` class is *abstract*: the implementation of `__call__` is left to various subclasses.

```
class TensorField(object):
    def __call__(self):
        raise NotImplementedError, "abstract base class"
    ...
```

`ConcreteTensorField` is one such subclass. It uses component functions to generate `Tensor`'s at a point. For example, we define a concrete 2-dimensional scalar field

$$r(x, y) = x^2 + y^2$$

by specifying its (single) component function:

```
>>> from pydx.field import ConcreteTensorField
>>> r = ConcreteTensorField((), 2)
>>> r[()] = lambda x,y:x**2+y**2
```

In this case the component is indexed with the empty tuple.
We evaluate the field at a point and get a `Tensor` object:

```
>>> r(3,4)
(25)
>>> type(r(3,4))
<class 'pydx.tensor.Tensor'>
```

As another example, we define the metric of a two dimensional flat-space:

$$g(x, y) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

using the `identity` constructor:

```
>>> from pydx.field import TensorField, up, dn
>>> metric = TensorField.identity((dn, dn), 2)
```

Calling the metric will compute its value at a point:

```
>>> metric(3, 4)
1.0 0.0
0.0 1.0
```

6.5.1 Operations on Tensor Fields

Operations on `TensorField` objects are lazy. This directly represents the mathematical definition of pointwise function operations. For example, the addition of two tensor fields is a tensor field that evaluates to the sum of the values of the two fields:

$$(X + Y)(x) := X(x) + Y(x)$$

We can now take this as a recipe for calculating the value of the sum of two tensor fields and implement it in the `AddTensorField` subclass. These objects are then created in the special `__add__` method of `TensorField`:

```
class TensorField(object):
    def __add__(self, other):
        return AddTensorField(self, other)
    ...
```

The implementation follows the recipe. Notice how we `assert` that the valence and dimension of the two fields match.

```
class AddTensorField(TensorField):
    def __init__(self, a, b):
        assert a.valence == b.valence
        assert a.dim == b.dim
        TensorField.__init__(self, a.valence, a.dim)
        self.a = a
        self.b = b

    def __call__(self, *xs):
        return self.a(*xs) + self.b(*xs)
```

All of the algebraic tensor operations are implemented similarly: there is a lazy class that constructs the result from tensor operand(s).

Continuing with our example, multiplying the metric and the scalar field produces a new `MulTensorField` object:

```
>>> rmetric = r*metric
>>> rmetric(3, 4)
25.0 0.0
0.0 25.0
```

6.5.2 Differentiation of Tensor Fields

The next important ingredient is being able to differentiate tensor fields.

Partial derivatives can be obtained by evaluating tensor fields with `MJet`'s and then extracting the appropriate components.

```
>>> from pydx.mjet import Jet
>>> x=Jet([3, 1])
>>> y=4
>>> r(x,y)
(AddJet(SqrJet(<0:3.0 1:1.0>),<0:16.0>))
>>> r(x,y)[()]
AddJet(SqrJet(<0:3.0 1:1.0>),<0:16.0>)
>>> r(x,y)[()][1]
6.0
```

This is a little tricky, and the string representation produced by the interactive interpreter are perhaps not verbose enough. We evaluate `r` with `Jet` argument `x` and constant argument `y`. This produces `r(x,y)` which is now a `Tensor` object with a (single)

`MJet` component. The first indexing operation `[()]` extracts this component. The next indexing operation `[1]` extracts the first derivative from this `Jet`.

Using some suggestive notation, we are extracting the coefficient of ε, from $r(3{+}\varepsilon,\ 4)$.

We can construct partial derivatives w.r.t. *both* variables with a single evaluation: $r(x+\varepsilon_x,\ y+\varepsilon_y)$. In code this looks like:

```
dr = r(Jet({(0,0):x, (1,0):1}), Jet({(0,0):y, (0,1):1}))
```

We would like to then use the two first order partial derivatives, `dr[()][1,0]` and `dr[()][0,1]`, as components in a new covariant (one `dn` valence) tensor.

```
def call(x, y):
    dr = r(Jet({(0,0):x, (1,0):1}), Jet({(0,0):y, (0,1):1}))
    v = Tensor((dn,), 2)
    v[0] = dr[()][1, 0]
    v[1] = dr[()][0, 1]
    return v
```

Ideally, we would like (something similar to) this function to become the `__call__` method of a new subclass `CommaTensorField`. This is complicated by the fact that `call` as it stands does not properly accept `MJet` arguments. We expect any `TensorField` to accept `MJet` arguments. In this way way can treat `TensorField`'s built from differentiation in the same way as any other `TensorField`, including being able to further differentiate.

The actual implementation of `CommaTensorField.__call__` proceeds in several steps.

```
 1 class CommaTensorField(TensorField):
 2     def __init__(self, tfield):
 3         valence = tfield.valence + (dn,)
 4         TensorField.__init__(self, valence, tfield.dim)
 5         self.tfield = tfield
 6 
 7     def __call__(self, *xs):
 8         dim = len(xs)
 9         assert dim==self.dim
10         if not isinstance(xs[0],MJet):
11             # must be scalar .. ie. a rank-0 Jet
12             xs = [MJet(0).promote(_x) for _x in xs]
13         # now we need dim extra ranks
14         rank0 = xs[0].rank
15         rank = rank0 + dim
16         xx = []
17         for i,_x in enumerate(xs):
```

```
18              _x = MJet(rank).promote(_x)
19              idxs = [0]*rank
20              idxs[rank0 + i] = 1
21              _x[tuple(idxs)] = self.scalar_one
22              xx.append(_x)
23
24          tensor = self.tfield(*xx)
25
26          comma = Tensor(self.valence, self.dim)
27
28          for _idxs in tensor.genidx():
29              g = tensor[_idxs]
30              g = MJet(rank).promote(g)
31              # now we drop back to rank0
32              gs = []
33              for i in range(dim):
34                  idxs = [0]*dim
35                  idxs[i] = 1
36                  idxs = tuple([slice(None)]*rank0+idxs)
37                  g0 = g[idxs]
38                  g0 = MJet(rank0).promote(g0)
39                  gs.append(g0)
40              comma[_idxs] = gs
41          return comma
```

On lines 10-12 we promote the arguments to `MJet`'s if they are not already so.

Lines 16-22 build a list of new arguments, `xx`, which will be used in the call to the underlying `TensorField` on line 24.

Line 26 constructs the `Tensor` object, and lines 28-40 set its elements from the appropriate derivative components of the `MJet`'s.

6.5.3 Coordinate Transforms

The defining property of a tensor is its transformation property under coordinate transforms. `pydx` at present does not enforce a transform policy on `Tensor` objects, ie. we use the `Tensor` class to represent non-tensorial data (such as the Christoffel symbols). In particular we store a coordinate transform in a rank-1 contravariant `TensorField` object.

```
class Transform(ConcreteTensorField):
    """Coordinate Transform: a 'vector' of ScalarFields """
    def __init__(self, fns, inverse=None, g=None):
```

```
            dim = len(fns)
            self.dim = dim
            self.inverse = inverse
            if inverse is not None:
                assert dim == inverse.dim
                self.inverse.inverse = self
            elems = [ ((i,),fn) for i,fn in enumerate(fns) ]
            ConcreteTensorField.__init__(self, (Tensor.up,), self.dim, g, elems)
            self.partial = self.comma()
```

The important thing here is to be able to access the matrix of partial derivatives: `self.partial`.

```
    class TensorField(object):
        def transform(self, coord_trans):
            return TransformTensorField(self, coord_trans)
```

The general tensor transform rule is implemented in `TransformTensorField.__call__`:

```
class TransformTensorField(TensorField):
    def __init__(self, tfield, coord_transform):
        TensorField.__init__(self, tfield.valence, tfield.dim)
        self.tfield = tfield
        self.coord_transform = coord_transform

    def __call__(self, *xs):
        assert len(xs)==self.dim
        _xs = self.coord_transform(*xs)
        tensor = self.tfield(*_xs)
        if Tensor.up in self.valence:
            inverse_partial = self.coord_transform.inverse.partial(*_xs)
        if Tensor.dn in self.valence:
            partial = self.coord_transform.partial(*xs)
        for i, updn in enumerate(self.valence):
            if updn == Tensor.up:
                tensor = tensor.mul(inverse_partial, (i,1))
                idxs = range(self.rank)
                idxs = idxs[:i]+[idxs[-1]]+idxs[i:-1]
                tensor = tensor.transpose(*idxs)
                assert self.valence==tensor.valence
            elif updn == Tensor.dn:
                tensor = tensor.mul(partial, (i,0))
```

```
            idxs = range(self.rank)
            idxs = idxs[:i]+[idxs[-1]]+idxs[i:-1]
            tensor = tensor.transpose(*idxs)
            assert self.valence==tensor.valence
    return tensor
```

6.6 Computation in (Semi-)Riemannian Geometry

`TensorField`'s actually carry around a reference to a metric. This way we can easily raise and lower indices.

The following computations are implemented in the `pydx.manifold` module in the class `RManifold`.

First we calculate the Christoffel symbols:

$$\Gamma^a{}_{bc} = \frac{1}{2} g^{ad}(g_{db,c} + g_{dc,b} - g_{bc,d})$$

This is spelled in code as:

```
g.p = g.comma()
gamma = (self.scalar_one/2) \
    * g.uu.mul(g.p + g.p.transpose(2,1,0) \
        - g.p.transpose(2,0,1), (1,1)).transpose(0,2,1)
christoffel = gamma
```

Next we calculate the Riemann tensor:

$$R^d{}_{cab} = \Gamma^d{}_{bc,a} - \Gamma^d{}_{ac,b} + \Gamma^d{}_{cb}\Gamma^d{}_{ad} - \Gamma^d{}_{ca}\Gamma^d{}_{bd}$$

In code:

```
gamma.p = gamma.comma()
gamma_gamma = gamma.mul(gamma, (0,2))
riemann = gamma.p.transpose(0,2,3,1) \
    - gamma.p.transpose(0,2,1,3) \
    + gamma_gamma.transpose(2,0,3,1) \
    - gamma_gamma.transpose(2,0,1,3)

riemann.dddd = riemann.mul(g, (0,0)).transpose(3,0,1,2)
riemann.uuuu = riemann.mul(g.uu, (1,0)).transpose(0,3,1,2)
riemann.uuuu = riemann.uuuu.mul(g.uu, (2,0)).transpose(0,1,3,2)
riemann.uuuu = riemann.uuuu.mul(g.uu, (3,0))
```

And the Ricci, and scalar curvature:

$$R_{bd} = R^a{}_{bad}$$

$$R = R^a{}_a$$

```
ricci = riemann.contract((0,2))
ricci.ud = ricci.mul(g.uu, (0,0))
curvature = ricci.ud.contract((0,1))
```

The Kretschman scalar:

$$K = R^{abcd} R_{abcd}$$

```
kretschman = riemann.uuuu.mul(riemann.dddd, (0,0), (1,1), (2,2), (3,3))
```

6.7 Geodesics

This is a system of 4 second order differential equations:

$$\ddot{x}^a + \Gamma^a{}_{bc} \dot{x}^b \dot{x}^c = 0 \tag{6.5.1}$$

We reduce this to a system of *first* order equations by introducing new variables y^a:

$$\dot{x}^a = y^a$$
$$\dot{y}^a = -\Gamma^a{}_{bc} y^b y^c$$

This is implemented in the `__call__` method of the `pydx.geodesic.Geodesic` class:

```
class Geodesic(ODE):
    def __init__(self, manifold, x0, y0):
        ODE.__init__(self, x0, y0)
        self.manifold = manifold
        self.gamma = manifold.gamma
    def __call__(self, t, xs):
        assert len(xs)==self.dim
        dx = xs[self.dim/2:] # these are the derivatives of x
        x = xs[:self.dim/2]
        result = Tensor((Tensor.up,), self.dim)
        gamma = self.gamma(*x)
        for i in range(self.dim/2):
```

```
            result[i] = dx[i]
        for i in range(self.dim/2):
            r = sum([ -gamma[i,j,k]*dx[j]*dx[k]
                        for j in range(self.dim/2)
                        for k in range(self.dim/2) ], self.scalar_zero)
            result[self.dim/2+i] = r
        return result
```

6.8 Bibliographic Notes

For a similar fusion of computer science and differential geometry, see the work "Functional Differential Geometry" by Gerald Jay Sussman and Jack Wisdom [20]. They define the concepts of differential geometry interleaved with code fragments in the Scheme language.

Other software tools for performing calculations in differential geometry include `GRworkbench` [15], and the maple package `GRTensor II`.

A comprehensive treatment of differential geometry for general relativity is given in Stewart's book "Advanced General Relativity" [8].

The Curzon Spacetime

In this chapter we examine numerical solutions to the geodesic equation in the Curzon spacetime.

The experiments in section 7.3, 7.4 and 7.5 use `pydx` in conjunction with the `Taylor` ODE solver [9]. The experiment in section 7.6 uses the full `pydx` stack to produce verified geodesics.

7.1 Introduction

The Weyl metric is given by

$$ds^2 = -e^{2\lambda}dt^2 + e^{2(\nu-\lambda)}[dr^2 + dz^2] + r^2e^{-2\lambda}d\phi^2,$$

where the functions λ and ν satisfy

$$\lambda_{rr} + \lambda_{zz} + r^{-1}\lambda_r = 0,$$

and

$$\nu_r = r(\lambda_r^2 - \lambda_z^2), \quad \nu_z = 2r\lambda_r\lambda_z.$$

This metric is *static*: it does not depend on the t coordinate, and *axisymmetric*: it only depends on the r and z coordinates.

The Curzon metric [21] [19] comes from using the monopole potential

$$\lambda = -m/R \text{ and } \nu = -m^2r^2/2R^4 \text{ where } R = (r^2 + z^2)^{1/2}.$$

This is a diagonal metric, ie. g_{ab} is nonzero only when $a = b$. So,

$$\Gamma^a{}_{bc} = \frac{1}{2}g^{aa}(g_{ab,c} + g_{ac,b} - g_{bc,a}) \qquad \text{no sum (n.s.)}$$

And the nonzero components of $\Gamma^a{}_{bc}$ are:

$$\Gamma^a{}_{ca} = \Gamma^a{}_{ac} = \frac{1}{2} g^{aa} g_{aa,c} \qquad \text{n.s.}$$
$$\Gamma^a{}_{bb} = \frac{1}{2} g^{aa} [2 g_{ab,b} - g_{bb,a}] \qquad \text{n.s.}$$
$$= -\frac{1}{2} g^{aa} g_{bb,a}, \ a \neq b \qquad \text{n.s.}$$
$$\Gamma^a{}_{aa} = \frac{1}{2} g^{aa} g_{aa,a} \qquad \text{n.s.}$$

The metric only depends on the r and z so from (6.5.1) we deduce that any geodesic with initial condition $\dot{t} = 0$ has constant t coordinate, and any geodesics with initial condition $\dot{\phi} = 0$ has constant ϕ coordinate. Throughout this chapter we will only concern ourselves with geodesics with constant t coordinate.

7.2 Implementation

Here we show the implementation of the Curzon metric in `pydx`. We define the `SpatialCurzonMetric` class as a subclass of `TensorField`.

```
class SpatialCurzonMetric(TensorField):
    def __init__(self, m):
        self.m = m
        TensorField.__init__(self, (Tensor.dn, Tensor.dn), 3)

    def __call__(self, r, z, phi):
        zero = self.scalar_zero
        m = self.m
        R2 = sqr(r)+sqr(z)
        nu = -sqr(m)*sqr(r) / (2.0 * sqr(R2))
        lmda = -m/sqrt(R2)
        c = exp(2*(nu-lmda))
        dr_dr = c
        dz_dz = c
        dphi_dphi = sqr(r)*exp(-2.0*lmda)
        g = [
            [ dr_dr,  zero,  zero,      ],
            [ zero,   dz_dz, zero,      ],
            [ zero,   zero,  dphi_dphi, ],
        ]
        g = Tensor(self.valence, self.dim, elems=g)
        return g
```

There is another class, `RZCurzonMetric`, for geodesics restricted to constant ϕ coordinate. This class is defined similarly to `SpatialCurzonMetric`, but excludes the ϕ coordinate.

7.3 Geodesics with Constant ϕ

The experiment in this section uses `pydx` with symbolic scalars to generate a system of differential equations suitable as input to the Taylor ODE solver of Jorba and Zou [9]. This solver is then compiled and run with the GNU multi-precision floating point numbers. As mentioned in Chapter 3, `libgmp` does not implement the exponential function. We get around this limitation by defining this function as a solution to a differential equation, and enlarging the system of ODE's accordingly.

In greater detail, suppose we have a system

$$y_i'(x) = f_i(x, y_1, ..., y_n).$$

We replace each occurrence of the exponential function

$$y_i'(x) = ... \exp\big(g(x, y_1, ..., y_n)\big)...$$

with a new variable

$$z := \exp\big(g(x, y_1, ..., y_n)\big),$$

and expand the initial value problem with the new equation

$$z' = z \sum_i \frac{\partial}{\partial y_i} g(x, y_1, ..., y_n) y_i' + z \frac{\partial g}{\partial x}, \quad z(0) = \exp\big(g(0, y_1(0), ..., y_n(0))\big).$$

Note this still requires an evaluation of the exponential function, which we do using an `Interval`. These mechanisms are implemented in the module `pydx.jz.jz`.

This is a plot of several geodesics starting from the point $(r, z) = (-0.01, 0.1)$.

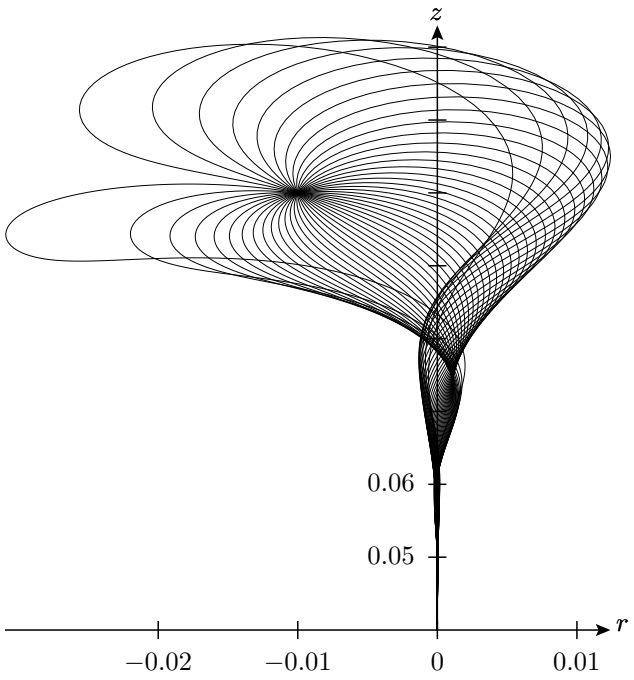


Fig. 7.3.1 Geodesics with constant ϕ emanating from the point $(r, z) = (-0.01, 0.1)$.

Note the distinction amongst geodesics based on the sign of the initial value $z'(0)$. Geodesics with $z'(0) < 0$ refocus at one location, while geodesics with $z'(0) > 0$ appear to refocus at another location further down the z-axis.

7.4 Geodesics in the Compactified Coordinate System

In order to explore the singularity at $(r, z) = (0, 0)$, Scott and Szekeres [19] define a new compactified coordinate system for the quarter-plane $r \geq 0$, $z \geq 0$,

$$x = \tan^{-1}[(r/m)e^{m/z}] + \tan^{-1}[(r/m)e^{-(2^{1/2}m/r)^{2/3}}]$$
$$y = \tan^{-1}\left\{3\frac{z}{m} - \frac{(z/m)^2 e^{\psi}}{(R^8 + 1 + \frac{1}{3}[r/m]^2 R^{-4})^{1/4}}\right\}$$

where

$$\psi = \nu - \lambda = (m/R) - (m^2 r^2/2R^4).$$

The previous geodesics are too close to $(r, z) = (0, 0)$ to be very revealing. Here we look at geodesics starting from the point $r = 0.79$, $z = 1.06$. The techniques are the same as in the previous section, but we plot the results after they have been transformed to the compactified coordinates.

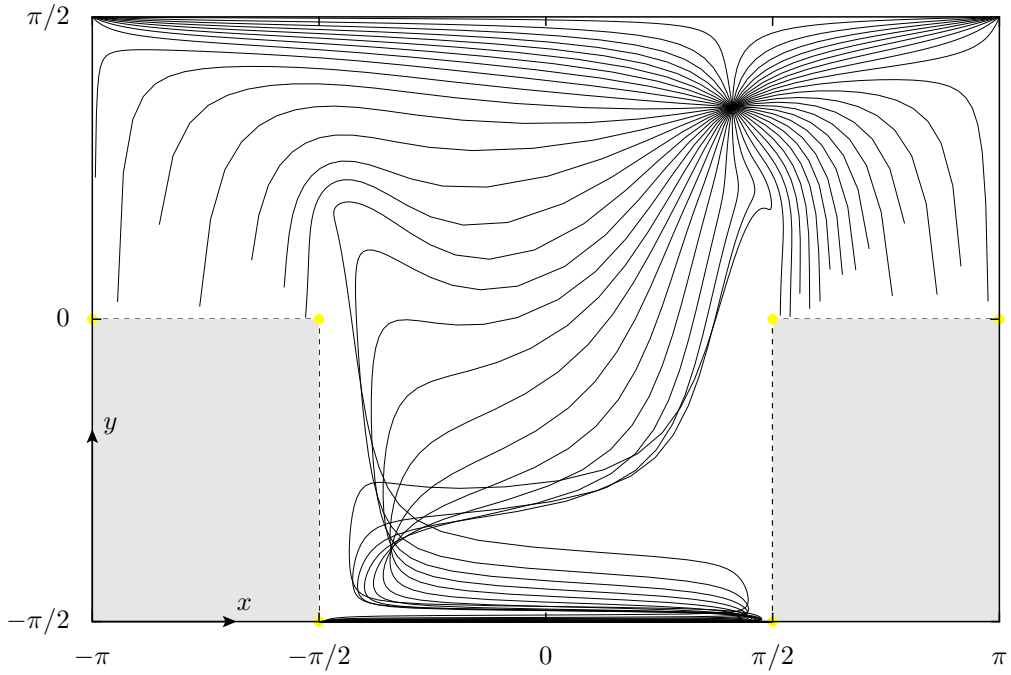


Fig. 7.4.1 Geodesics with constant ϕ in the compactified coordinates.

We see the geodesics separate into three classes:

- geodesics limiting to either of the infinite points $(x, y) = (\pm\pi, \pi/2)$,
- geodesics crossing into the gray region from $y > 0$, and
- geodesics oscillating in the bottom well, limiting to $x \in [-\pi/2, \pi/2], y = -\pi/2$.

Also note how the geodesics diverge from the points $x = 0$, $y = \pm\pi/2$.

7.5 Spatial Geodesics

Here we extend the methods of the previous section to include the spatial geodesics with ϕ coordinate.

The yellow rings represent the points $(x, y) = (-\pi, \pi/2)$, $(-\pi, 0)$, $(-\pi/2, 0)$, and $(-\pi/2, -\pi/2)$, rotated in ϕ, also marked as yellow spots in the previous figure.

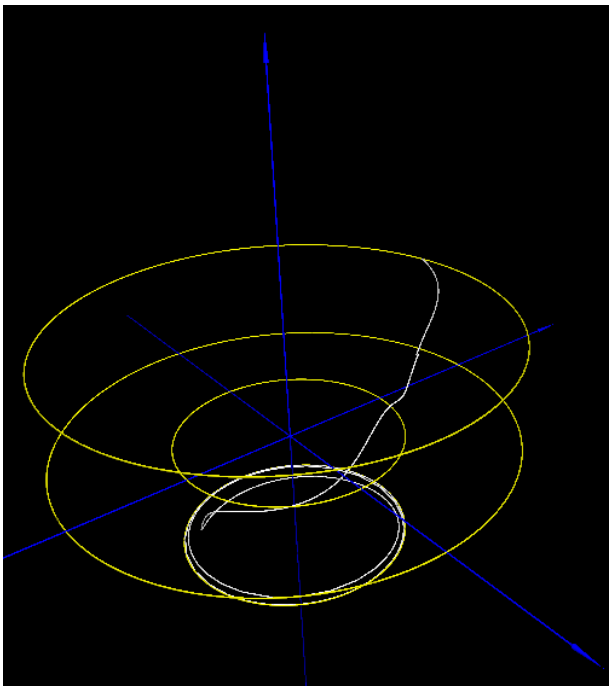


Fig. 7.5.1 A spatial geodesic in the compactified coordinates.

This figure shows a typical geodesic from the oscillating class above. They all limit to the ring $(x, y) = (-\pi/2, -\pi/2)$. Such geodesics with initial condition $\phi'(0)$ close to zero show an elliptic path that also expands into the circular limit at $(x, y) = (-\pi/2, -\pi/2)$.

7.6 Verified Geodesics

We examine solutions to the geodesic equation in the r, z plane. This is a plot of several geodesics starting from the point $(r, z) = (-0.01, 0.1)$.

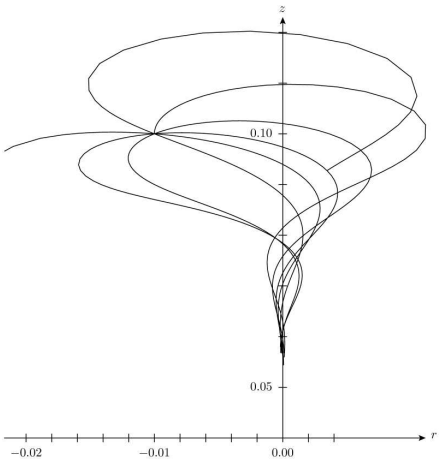


Fig. 7.6.1 Verified geodesics with constant ϕ.

Now we fix attention on the geodesic with initial condition $r'(0) = 5$, and look at the effect of the order of the method. The stepsize is fixed at $h = 10^{-4}$. For the higher orders we boost the precision until there is negligible improvement in the results.

The following graph plots the step number versus the width of the solution (the sum of the widths[1] over each dimension):

[1]See definition 3.6.6

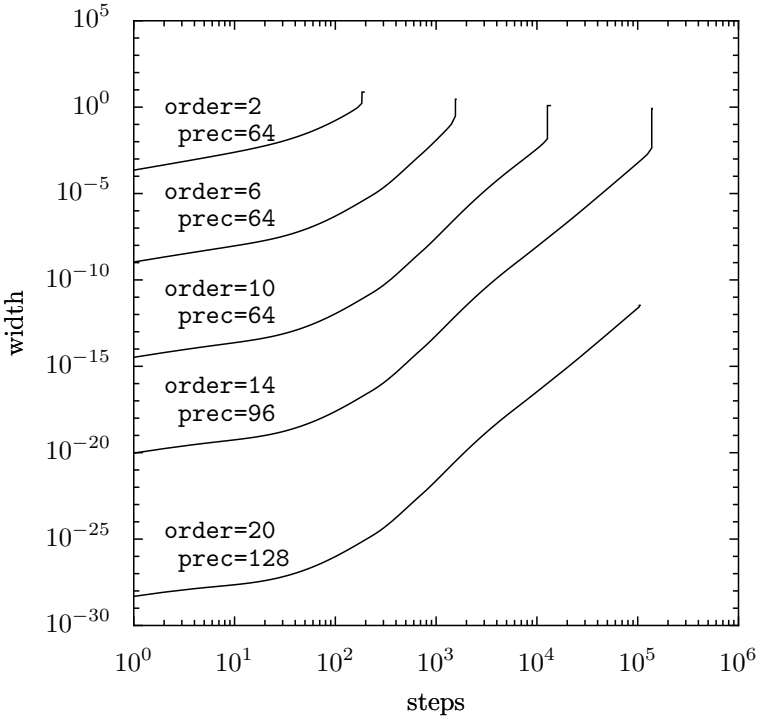


Fig. 7.6.2 Behavior of solution width at different orders.

Execution time is vastly longer than in the previous sections. The longest plot here, of about 10^5 steps represents a calculation taking several days, as compared to only seconds for the non-verified geodesics. Partly this is because `Taylor` only uses a few hundred steps, high order and large step size is the key to the success of the `Taylor` solver. The verified stepper in `pydx` also does not have step-size control while the `Taylor` solver does. This would be fairly easy to implement (search for an interval over which a contraction is found), but it is not clear how helpful this would be.

Conclusion and Further Work

We have presented the software `pydx`, which is a complete implementation of computational differential geometry using automatic differentiation. This is the seen to be the "right" way to perform these types of calculations, both from an algorithmic point of view, and a theoretical point of view.

We note some words on the implementation language, Python. It is known that the raw algorithmic performance of Python code is approximately 100 times slower than in the compiled languages. We circumvent this problem in `pydx` by using aggressive code generation techniques. The resulting code has minimal overhead, and could potentially be compiled using tools like `pyrex` or `rpython`. However, the main reason Python was chosen is its ease of implementation. It is also felt that the resulting code is also remarkably clear, compared to for example, the `C++` code in [15].

8.1 Further Work

The interface for calculating with tensors is made entirely difficult and awkward by the use of the `transpose` method. The author found it immensely difficult to write the correct expressions of Riemannian geometry. It would be desirable to implement a way that would encode the way the formulae are written with indices directly. This would allow us to write something like the following for the calculation of the Christoffel symbols,

$$\Gamma^a{}_{bc} = \frac{1}{2} g^{ad}(g_{db,c} + g_{dc,b} - g_{bc,d})$$

as

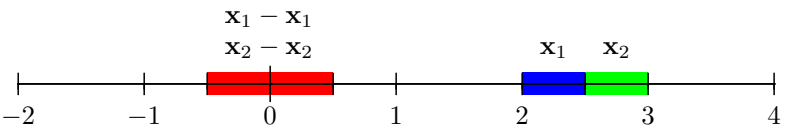
```
gp = g.comma()
gamma = (self.scalar_one/2) * g.uu.ad * (g.p.dbc + g.p.dcb - g.p.bcd)
```

where we have used the same trick of over-riding the dot (`getattr`) operation as we did for the raising and lowering indices, eg. `g.uu`.

The Pypy project [6] has implemented the Python interpreter in Python itself. Along the way they have also developed a type inference engine that allows a restricted subset

of Python code to be compiled to `C`. This results in performance gains of up to 300 times. It would be exciting to leverage this technology in `pydx.`

The `pydx` framework is designed so that adding new scalar types is reasonably straight forward. It would be interesting to use the automatic differentiation in the implementation of a smart interval scalar type that tracks derivative information, as in [12]. The author also experimented with a way of dealing with the dependency problem. We treat a calculation *lazily*, ie., as an interval valued function of intervals. Then, to find a tight interval enclosing the result we use a minimization/maximization algorithm which has been extended to handle interval values. For example, as we apply a bisection algorithm to the calculation $f(x) = x - x$ we see that the resulting interval approaches zero.



In `GRworkbench` [15], they implement mechanisms for working with charts. These are only implicit in `pydx,` it would be relatively easy to at least implement some book-keeping to make this explicit. Furthermore, it is possible that some kind of `AbstractManifold` class would help to clarify the connection between the modern (coordinate free) treatment of differential geometry and the concrete implementation in `pydx.`

Further study of synthetic differential geometry would be useful as it is the mathematical theory of differential geometry developed with "infinitesimals", which is exactly what automatic differentiation is. This study would help to illuminate the connection between `Jet`'s and `TensorField`'s.

It would be nice to be able to use solutions of IVP's as first class functions within `pydx.` This would "close the loop", allowing us to treat such solutions as *functions* (curves), and compute derivatives, or coordinate transforms (eg. for co-moving coordinates, or exponential normal coordinates). These things are possible now, but require explicit manipulation.

We note that it is possible to compute with expressions involving *arbitrary* partial derivatives of geometric objects. It is felt that the tools and capabilities of `pydx` have only barely begun to be used. The author hopes that future exploration of differential geometry with `pydx` will show this.

Bibliography

[1] IEEE floating-point standard. *http://en.wikipedia.org/wiki/IEEE_floating-point_standard*.

[2] Ben Andrews. *Differential Geometry*. Unpublished course notes, Mathematical Sciences Institute, The Australian National University, 1997.

[3] Torbjorn Granlu. The GNU Multiple Precision Arithmetic Library. *http://gmplib.org*.

[4] Andreas Griewank. On Automatic Differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, 1989.

[5] R. Hammer, M. Hocks, Ulrich Kulisch, and Dietmar Ratz. *Numerical Toolbox for Verified Computing I, Basic Numerical Algorithms, Theory, Algorithms, and Pascal-XSC Programs*. Number 21 in Springer Series in Computational Mathematics. Springer-Verlag, New York, 1991.

[6] Holger Krekel, Martijn Faassen, Armin Rigo, Carl Friedrich Bolz, Guido Wesdorp, Christian Tismer, Michael Hudson and Maciej Fijalkowski. The PyPy Project. *http://codespeak.net/pypy/*.

[7] Jens-Uwe Mager, Holger Krekel, Martijn Faassen, Stephan Schwarzer, Brian Dorsey, Grigh Gheorghiu, Armin Rigo, Carl Friedrich Bolz, Guido Wesdorp and Maciej Fijalkowski. The py lib. *http://codespeak.net/py/dist/*.

[8] John M. Stewart. *Advanced General Relativity*. Cambridge University Press, 1990.

[9] Angel Jorba and Maorong Zou. A software package for the numerical integration of ODE by means of high-order Taylor methods. *http://citeseer.ist.psu.edu/jorba01software.html*.

[10] Angel Jorba and Maorong Zou. A software package for the numerical integration of ODE by means of high-order Taylor methods. *Experimental Mathematics*, 14:99–117, 2005.

[11] Donald E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd Edition)*. Addison-Wesley Professional, November 1997.

[12] Luiz H. de Figueiredo and Jorge Stolfi. *Self-Validated Numerical Methods and Applications*. Brazilian Mathematics Colloquium.

[13] Alex Martelli. GMPY: general multiprecision python. *http://gmpy.sourceforge.net/*.

[14] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[15] Andrew Moylan. Numerical Experimentation within GRworkbench. *B.Sc. (Honours) thesis, Department of Physics, The Australian National University*, 2003.

[16] INRIA Lorraine MPFR team (SPACES project and LORIA). The MPFR Multiple Precision Floating-Point Reliable Library. *http://www.mpfr.org/*.

[17] Richard D. Neidinger. An Efficient Method for the Numerical Evaluation of Partial Derivatives of Arbitrary Order. *ACM Trans. Math. Software*, 18(2):159–173, June 1992.

[18] Fabrice Rouillier and Nathalie Revol. The MPFI library. *http://perso.ens-lyon.fr/nathalie.revol/software.html*.

[19] Susan M. Scott and P. Szekeres. The Curzon Singularity. I: Spatial Sections. II: Global Picture. *General Relativity and Gravitation*, 18:557–583, 1986.

[20] Gerald Jay Sussman and Jack Wisdom. Functional Differential Geometry. *http://www-swiss.ai.mit.edu/users/wisdom/AIM-2005-003.pdf*.

[21] P. Szekeres and F. H. Morgan. Extensions of the Curzon Metric. *Commun. math. Phys.*, 32:313–318, 1973.

[22] Guido van Rossum. The Python Interpreter. *http://python.org/*.

[23] Guido van Rossum. The Python Tutorial. *http://docs.python.org/tut/tut.html*.

[24] R. E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463 – 464, 1964.

Appendix A: pydx Installation

A.1 pydx Dependencies

The main dependency is the Python interpreter itself.

- The Python interpreter [22], version 2.4 or higher.

The following libraries are required in order to compute with other number systems apart from the Python `float` type.

- GNU Multiple Precision Arithmetic Library[3]. libgmp 4.1.0 or higher. Although version 3 seems to be sufficient. You may already have it installed.
- MPFR [16], mpfr-2.2.0. **You will need to apply the cumulative patch.**
- MPFI [18], version 1.3.4.
- GMPY [13]. This provides Python bindings for libGMP. Get the CVS version.

A.2 Obtaining and Building pydx

Checkout the code from the subversion repository:

```
~/downloads$ svn co http://gr.anu.edu.au/svn/people/sdburton/pydx
```

This will create a directory called pydx.
Option 1 (recommended) is to build using make:

```
~/downloads$ cd pydx
~/downloads/pydx$ make
```

This will build the modules in-place. (Note that make produces lots of warnings: this is OK).

In order for Python to always find `pydx` you will need to add the pydx subdirectory to your `PYTHONPATH`:

```
~/downloads/pydx$ export PYTHONPATH=$PYTHONPATH:‘pwd‘
```

Option 2 for installing is with the `setup.py` script:

```
~/downloads/pydx$ python setup.py build
~/downloads/pydx$ sudo python setup.py install
```

The test suite is designed to be driven from py.test [7]:

```
~/downloads/pydx$ py.test pydx
...
```

This is example output of a run of the `pydx` test suite.

```
$ py.test
inserting into sys.path: /home/simon/local/pypy
====================== test process starts ======================
testing-mode: inprocess
executable:   /usr/bin/python  (2.4.3-final-0)
using py lib: /home/simon/local/pypy/py <rev 35227>

test/test_field.py[2] ..
test/test_geodesic.py[2] ..
test/test_interval.py[2] ..
test/test_manifold.py[1] .
test/test_metric.py[3] ...
test/test_mjet.py[7] .......
test/test_mpfi.py[1] .
test/test_ode.py[1] .

========== tests finished: 19 passed in 581.53 seconds ==========
```

Appendix B: Code Listing - pydx

We include the source of pydx here in full as it is the final and unambiguous specification of the system itself.

B.1 scalar/__init__.py

```
class Scalar(object):
    stack = []
    clients = []
    current = None
    def __init__( self, type, promote, zero, one, callback=None, cb_args=(), restore_cb=None ):
        self.type = type
        self.promote = promote
        self.zero = zero
        self.one = one
        self.callback = callback
        self.cb_args = cb_args
        self.restore_cb = restore_cb

    def notify_client(self, client):
        client.scalar = self
        # deprecated:
        client.scalar_type = self.type
        client.scalar_promote = self.promote
        client.scalar_zero = self.zero
        client.scalar_one = self.one

    def set( cls, scalar_type, scalar_promote, scalar_zero, scalar_one, callback=None, cb_args=(), restore_cb=None ):
        cls.stack.append( cls.current )
        scalar = cls(scalar_type, scalar_promote, scalar_zero, scalar_one, callback, cb_args, restore_cb)
        if scalar.callback is not None:
            scalar.callback( *scalar.cb_args )
        cls.current = scalar
        for client in cls.clients:
            scalar.notify_client(client)
    set = classmethod(set)

    def restore(cls):
        scalar = cls.current
        if scalar.restore_cb is not None:
            scalar.restore_cb( *scalar.cb_args )
        scalar = cls.stack.pop()
        if scalar.callback is not None:
            scalar.callback( *scalar.cb_args )
        for client in cls.clients:
            scalar.notify_client(client)
        cls.current = scalar
    restore = classmethod(restore)

    def add_client(cls, client):
        if client not in cls.clients:
            cls.clients.append(client)
            if cls.current is not None:
                cls.current.notify_client(client)
    add_client = classmethod(add_client)

Scalar.current = Scalar(float, float, 0., 1.)

def set_scalar( *args, **kw ):
    Scalar.set(*args, **kw)
def restore_scalar():
    Scalar.restore()
```

```
def set_mpf_scalar(prec=53):
    import gmpy
    scalar_zero = gmpy.mpf(0.0)
    scalar_one = gmpy.mpf(1.0)
    scalar_type = type(scalar_zero)
    scalar_promote = gmpy.mpf
    set_scalar( scalar_type, scalar_promote, scalar_zero, scalar_one,
        gmpy.set_minprec, (prec,) )

def set_interval_scalar(prec=53):
    from mpfi import Interval, set_default_prec
    scalar_zero = Interval(0.0)
    scalar_one = Interval(1.0)
    scalar_type = type(scalar_zero)
    scalar_promote = Interval
    set_scalar( scalar_type, scalar_promote, scalar_zero, scalar_one,
        set_default_prec, (prec,) )

def set_symbolic_scalar():
    from symbolic import Float, ZeroFloat, OneFloat
    # push current scalar onto Float's "scalar" stack
    # See eg. compile method
    Float.push_scalar(
        Scalar.current.zero,
        Scalar.current.one,
        Scalar.current.promote,
        Scalar.current.type,
    )
    scalar_type = Float
    scalar_promote = Float.promote
    scalar_zero = ZeroFloat()
    scalar_one = OneFloat()
    set_scalar( scalar_type, scalar_promote, scalar_zero, scalar_one,
        restore_cb = Float.restore_scalar )
```

B.2 scalar/fmath.py

```
"""
We want to do something like this:

class ExpMathFunc(object):
    def __call__(self, x):
        return math.exp(x)

exp = ExpMathFunc()

so that we can dynamically change the behaviour of exp() depending on
what scalars we are using.
"""

import sys
import math
math.sqr = lambda self, x:x**2 # ?


def sqr(x):
    return x**2
def exp(x):
    try:
        return x.exp()
    except:
        return math.exp(x)
def log(x):
    try:
        return x.log()
    except:
        return math.log(x)
def sqrt(x):
    try:
        return x.sqrt()
    except:
        return math.sqrt(x)
def sin(x):
    try:
        return x.sin()
    except:
```

```
        return math.sin(x)
def cos(x):
    try:
        return x.cos()
    except:
        return math.cos(x)
def tan(x):
    try:
        return x.tan()
    except:
        return math.tan(x)
def asin(x):
    try:
        return x.asin()
    except:
        return math.asin(x)
def acos(x):
    try:
        return x.acos()
    except:
        return math.acos(x)
def atan(x):
    try:
        return x.atan()
    except:
        return math.atan(x)
def sinh(x):
    try:
        return x.sinh()
    except:
        return math.sinh(x)
def cosh(x):
    try:
        return x.cosh()
    except:
        return math.cosh(x)
def tanh(x):
    try:
        return x.tanh()
    except:
        return math.tanh(x)

# These are not particularly faster than the above:
if 0:
    def exp(x): return x.exp()
    def log(x): return x.log()
    def sqr(x): return x**2
    def sqrt(x): return x.sqrt()
    def sin(x): return x.sin()
    def cos(x): return x.cos()
    def tan(x): return x.tan()
    def asin(x): return x.asin()
    def acos(x): return x.acos()
    def atan(x): return x.atan()
    def sinh(x): return x.sinh()
    def cosh(x): return x.cosh()
    def tanh(x): return x.tanh()
```

B.3 scalar/mpfi.pyx

```
cdef extern from *:
    ctypedef unsigned int __c_size_t "size_t"

cdef extern from "errno.h":
    int errno
    void perror(char *s)
    char *strerror(int errnum)

cdef extern from "stdio.h":
    ctypedef void FILE
    FILE *stdout
    FILE *stderr

cdef extern from "string.h":
    void *__c_memcpy "memcpy" ( void*, void*, __c_size_t )
    int __c_memcmp "memcmp" ( void*, void*, __c_size_t )
```

```
cdef extern from "stdlib.h":
    void *malloc(size_t)
    void *calloc(size_t, size_t)
    void free(void*)


cdef extern from "Python.h":
    void*  PyMem_Malloc(__c_size_t)
    void*  PyMem_Realloc(void *p, __c_size_t n)
    void   PyMem_Free(void *p)

cdef extern from "gmp.h":
    ctypedef int mp_prec_t
    ctypedef int mp_exp_t
    ctypedef unsigned int mp_limb_t
    struct __mpf_struct:
        # Max precision, in number of `mp_limb_t's.
        # Set by mpf_init and modified by mpf_set_prec.
        # The area pointed to by the _mp_d field contains `prec' + 1 limbs.
        int _mp_prec
        # abs(_mp_size) is the number of limbs the last field points to.
        # If _mp_size is negative this is a negative number.
        int _mp_size
        mp_exp_t _mp_exp  # Exponent, in the base of `mp_limb_t'.
        mp_limb_t *_mp_d  # Pointer to the limbs.

    ctypedef __mpf_struct *mpf_t
    ctypedef __mpf_struct *mpf_ptr
    ctypedef __mpf_struct *mpf_srcptr
    void mpf_init( mpf_ptr )
    void mpf_set( mpf_t, mpf_t )
    void mpf_set_d( mpf_t, double )
    int mpf_get_prec( mpf_t )
    void __c_mpf_set_default_prec "mpf_set_default_prec"( int )
    double mpf_get_d( mpf_srcptr )
    __c_size_t mpf_out_str (FILE *, int, __c_size_t, mpf_srcptr)
    char *mpf_get_str  (char *, mp_exp_t *, int, size_t, mpf_srcptr)


cdef extern from "gmpy.h":
    void import_gmpy()

    ctypedef struct PympfObject:
        mpf_t f
        int rebits
    object Pympf_new(int bits)
    int Pympf_Check(o)


cdef extern from "mpfr.h":
    ctypedef enum mpfr_rnd_t:
        GMP_RNDN # nearest
        GMP_RNDZ # zero
        GMP_RNDU # up
        GMP_RNDD # down
        GMP_RND_MAX
        GMP_RNDNA

    ctypedef int mpfr_prec_t
    ctypedef int mpfr_sign_t

    ctypedef struct __mpfr_struct:
        mpfr_prec_t  _mpfr_prec
        mpfr_sign_t  _mpfr_sign
        mp_exp_t     _mpfr_exp
        mp_limb_t   *_mpfr_d

    ctypedef __mpfr_struct mpfr_t[1]
    ctypedef __mpfr_struct *mpfr_ptr
    ctypedef __mpfr_struct *mpfr_srcptr


    void mpfr_set_default_prec(mp_prec_t)
    mp_prec_t mpfr_get_default_prec()

    void mpfr_init( mpfr_ptr )
    void mpfr_init2( mpfr_ptr, mpfr_prec_t )
    void mpfr_clear( mpfr_ptr )
```

```
    mpfr_prec_t mpfr_get_prec( mpfr_srcptr )

    int mpfr_set_f (mpfr_ptr, mpf_srcptr, mpfr_rnd_t)
    int mpfr_get_f(mpf_ptr, mpfr_srcptr, mpfr_rnd_t)
    int mpfr_set_d (mpfr_ptr, double, mpfr_rnd_t)
    double mpfr_get_d (mpfr_srcptr, mpfr_rnd_t)

    int mpfr_pow (mpfr_ptr, mpfr_srcptr, mpfr_srcptr, mpfr_rnd_t)
    int mpfr_pow_si (mpfr_ptr, mpfr_srcptr, long int, mpfr_rnd_t)
    int mpfr_add (mpfr_ptr, mpfr_srcptr, mpfr_srcptr, mpfr_rnd_t)
    int mpfr_sub (mpfr_ptr, mpfr_srcptr, mpfr_srcptr, mpfr_rnd_t)
    int mpfr_mul (mpfr_ptr, mpfr_srcptr, mpfr_srcptr, mpfr_rnd_t)
    int mpfr_div (mpfr_ptr, mpfr_srcptr, mpfr_srcptr, mpfr_rnd_t)

    char*mpfr_get_str (char*, mp_exp_t*, int, __c_size_t, mpfr_srcptr, mpfr_rnd_t)
    int mpfr_strtofr (mpfr_t rop, char *nptr, char **endptr, int base, mpfr_rnd_t rnd)
    void mpfr_free_str (char *str)

    int mpfr_check_range (mpfr_t X, int T, mpfr_rnd_t RND)

    int mpfr_underflow_p ()
    int mpfr_overflow_p ()
    int mpfr_nanflag_p ()
    int mpfr_inexflag_p ()
    int mpfr_erangeflag_p ()

    void mpfr_clear_flags ()



cdef extern from "mpfi.h":

    ctypedef struct __mpfi_struct:
        __mpfr_struct left
        __mpfr_struct right

    ctypedef __mpfi_struct mpfi_t[1]
    ctypedef __mpfi_struct *mpfi_ptr
    ctypedef __mpfi_struct *mpfi_srcptr


    # Rounding
    int mpfi_round_prec(mpfi_ptr,mp_prec_t prec)


    # Initialization, destruction and assignment

    # initializations
    void   mpfi_init       (mpfi_ptr)
    void   mpfi_init2      (mpfi_ptr, mp_prec_t)

    void   mpfi_clear      (mpfi_ptr)

    # mpfi bounds have the same precision
    mp_prec_t mpfi_get_prec(mpfi_srcptr)
    void mpfi_set_prec(mpfi_ptr,mp_prec_t)


    # assignment functions
    int   mpfi_set        (mpfi_ptr, mpfi_srcptr)
    int   mpfi_set_si     (mpfi_ptr, long)
    int   mpfi_set_ui     (mpfi_ptr, unsigned long)
    int   mpfi_set_d      (mpfi_ptr, double)
    int   mpfi_set_z      (mpfi_ptr, mpz_srcptr)
    int   mpfi_set_q      (mpfi_ptr, mpq_srcptr)
    int   mpfi_set_fr   (mpfi_ptr, mpfr_srcptr)
    int   mpfi_set_str    (mpfi_ptr, char *, int)

    # combined initialization and assignment functions
    int   mpfi_init_set        (mpfi_ptr, mpfi_srcptr)
    int   mpfi_init_set_si     (mpfi_ptr, long)
    int   mpfi_init_set_ui     (mpfi_ptr, unsigned long)
    int   mpfi_init_set_d      (mpfi_ptr, double)
    int   mpfi_init_set_z      (mpfi_ptr, mpz_srcptr)
    int   mpfi_init_set_q      (mpfi_ptr, mpq_srcptr)
    int   mpfi_init_set_fr   (mpfi_ptr, mpfr_srcptr)
    int   mpfi_init_set_str    (mpfi_ptr, char *, int)

    # swapping two intervals
    void mpfi_swap (mpfi_ptr, mpfi_ptr)
```

```
# Various useful interval functions
# with scalar or interval results

# absolute diameter
int mpfi_diam_abs(mpfr_ptr, mpfi_srcptr)
# relative diameter
int mpfi_diam_rel(mpfr_ptr, mpfi_srcptr)
# diameter: relative if the interval does not contain 0
# absolute otherwise
int mpfi_diam(mpfr_ptr, mpfi_srcptr)
# magnitude: the largest absolute value of any element
int mpfi_mag(mpfr_ptr, mpfi_srcptr)
# mignitude: the smallest absolute value of any element
int mpfi_mig(mpfr_ptr, mpfi_srcptr)
# middle of y
int mpfi_mid (mpfr_ptr, mpfi_srcptr)
# picks randomly a point m in y
void mpfi_alea (mpfr_ptr, mpfi_srcptr)

# Conversions
double mpfi_get_d (mpfi_srcptr)
void mpfi_get_fr (mpfr_ptr,mpfi_srcptr)

# Basic arithmetic operations

# arithmetic operations between two interval operands
int   mpfi_add        (mpfi_ptr, mpfi_srcptr, mpfi_srcptr)
int   mpfi_sub        (mpfi_ptr, mpfi_srcptr, mpfi_srcptr)
int   mpfi_mul        (mpfi_ptr, mpfi_srcptr, mpfi_srcptr)
int   mpfi_div        (mpfi_ptr, mpfi_srcptr, mpfi_srcptr)

# arithmetic operations between an interval operand and a double prec. floating-point
int   mpfi_add_d      (mpfi_ptr, mpfi_srcptr, double)
int   mpfi_sub_d      (mpfi_ptr, mpfi_srcptr, double)
int   mpfi_d_sub      (mpfi_ptr, double, mpfi_srcptr)
int   mpfi_mul_d      (mpfi_ptr, mpfi_srcptr, double)
int   mpfi_div_d      (mpfi_ptr, mpfi_srcptr, double)
int   mpfi_d_div      (mpfi_ptr, double, mpfi_srcptr)

# arithmetic operations between an interval operand and an unsigned long integer
int   mpfi_add_ui     (mpfi_ptr, mpfi_srcptr, unsigned long)
int   mpfi_sub_ui     (mpfi_ptr, mpfi_srcptr, unsigned long)
int   mpfi_ui_sub     (mpfi_ptr, unsigned long, mpfi_srcptr)
int   mpfi_mul_ui     (mpfi_ptr, mpfi_srcptr, unsigned long)
int   mpfi_div_ui     (mpfi_ptr, mpfi_srcptr, unsigned long)
int   mpfi_ui_div     (mpfi_ptr, unsigned long, mpfi_srcptr)

# arithmetic operations between an interval operand and a long integer
int   mpfi_add_si     (mpfi_ptr, mpfi_srcptr, long)
int   mpfi_sub_si     (mpfi_ptr, mpfi_srcptr, long)
int   mpfi_si_sub     (mpfi_ptr, long, mpfi_srcptr)
int   mpfi_mul_si     (mpfi_ptr, mpfi_srcptr, long)
int   mpfi_div_si     (mpfi_ptr, mpfi_srcptr, long)
int   mpfi_si_div     (mpfi_ptr, long, mpfi_srcptr)

# arithmetic operations between an interval operand and a multiple prec. integer
int   mpfi_add_z     (mpfi_ptr, mpfi_srcptr, mpz_srcptr)
int   mpfi_sub_z     (mpfi_ptr, mpfi_srcptr, mpz_srcptr)
int   mpfi_z_sub     (mpfi_ptr, mpz_srcptr, mpfi_srcptr)
int   mpfi_mul_z     (mpfi_ptr, mpfi_srcptr, mpz_srcptr)
int   mpfi_div_z     (mpfi_ptr, mpfi_srcptr, mpz_srcptr)
int   mpfi_z_div     (mpfi_ptr, mpz_srcptr, mpfi_srcptr)

# arithmetic operations between an interval operand and a multiple prec. rational
int   mpfi_add_q     (mpfi_ptr, mpfi_srcptr, mpq_srcptr)
int   mpfi_sub_q     (mpfi_ptr, mpfi_srcptr, mpq_srcptr)
int   mpfi_q_sub     (mpfi_ptr, mpq_srcptr, mpfi_srcptr)
int   mpfi_mul_q     (mpfi_ptr, mpfi_srcptr, mpq_srcptr)
int   mpfi_div_q     (mpfi_ptr, mpfi_srcptr, mpq_srcptr)
int   mpfi_q_div     (mpfi_ptr, mpq_srcptr, mpfi_srcptr)

# arithmetic operations between an interval operand and a mult. prec. floating-pt nb
int   mpfi_add_fr  (mpfi_ptr, mpfi_srcptr, mpfr_srcptr)
int   mpfi_sub_fr  (mpfi_ptr, mpfi_srcptr, mpfr_srcptr)
int   mpfi_fr_sub  (mpfi_ptr, mpfr_srcptr, mpfi_srcptr)
int   mpfi_mul_fr  (mpfi_ptr, mpfi_srcptr, mpfr_srcptr)
int   mpfi_div_fr  (mpfi_ptr, mpfi_srcptr, mpfr_srcptr)
int   mpfi_fr_div  (mpfi_ptr, mpfr_srcptr, mpfi_srcptr)
```

```
# arithmetic operations taking a single interval operand
int   mpfi_neg        (mpfi_ptr, mpfi_srcptr)
int   mpfi_sqr        (mpfi_ptr ,mpfi_srcptr)
# the inv function generates the whole real interval if 0 is in the interval defining the divisor
int   mpfi_inv        (mpfi_ptr, mpfi_srcptr)
# the sqrt of a (partially) negative interval is a NaN
int   mpfi_sqrt       (mpfi_ptr, mpfi_srcptr)
# the first interval contains the absolute values of
# every element of the second interval
int   mpfi_abs        (mpfi_ptr, mpfi_srcptr)

# various operations
int   mpfi_mul_2exp   (mpfi_ptr, mpfi_srcptr, unsigned long)
int   mpfi_mul_2ui    (mpfi_ptr, mpfi_srcptr, unsigned long)
int   mpfi_mul_2si    (mpfi_ptr, mpfi_srcptr, long)
int   mpfi_div_2exp   (mpfi_ptr, mpfi_srcptr, unsigned long)
int   mpfi_div_2ui    (mpfi_ptr, mpfi_srcptr, unsigned long)
int   mpfi_div_2si    (mpfi_ptr, mpfi_srcptr, long)

# Special functions
int mpfi_log  (mpfi_ptr, mpfi_srcptr)
int mpfi_exp  (mpfi_ptr, mpfi_srcptr)
int mpfi_exp2 (mpfi_ptr, mpfi_srcptr)

int mpfi_cos  (mpfi_ptr, mpfi_srcptr)
int mpfi_sin  (mpfi_ptr, mpfi_srcptr)
int mpfi_tan  (mpfi_ptr, mpfi_srcptr)
int mpfi_acos (mpfi_ptr, mpfi_srcptr)
int mpfi_asin (mpfi_ptr, mpfi_srcptr)
int mpfi_atan (mpfi_ptr, mpfi_srcptr)

int mpfi_cosh (mpfi_ptr, mpfi_srcptr)
int mpfi_sinh (mpfi_ptr, mpfi_srcptr)
int mpfi_tanh (mpfi_ptr, mpfi_srcptr)
int mpfi_acosh (mpfi_ptr, mpfi_srcptr)
int mpfi_asinh (mpfi_ptr, mpfi_srcptr)
int mpfi_atanh (mpfi_ptr, mpfi_srcptr)

int mpfi_log1p (mpfi_ptr, mpfi_srcptr)
int mpfi_expm1 (mpfi_ptr, mpfi_srcptr)

int mpfi_log2 (mpfi_ptr, mpfi_srcptr)
int mpfi_log10 (mpfi_ptr, mpfi_srcptr)

int mpfi_const_log2(mpfi_ptr)
int mpfi_const_pi(mpfi_ptr)
int mpfi_const_euler(mpfi_ptr)

# Comparison functions
# Warning: the meaning of interval comparison is not clearly defined
# customizable comparison functions

int    (*mpfi_cmp)     (mpfi_srcptr,mpfi_srcptr)
int    (*mpfi_cmp_d)   (mpfi_srcptr,double)
int    (*mpfi_cmp_ui)  (mpfi_srcptr,unsigned long)
int    (*mpfi_cmp_si)  (mpfi_srcptr,long)
int    (*mpfi_cmp_z)   (mpfi_srcptr,mpz_srcptr)
int    (*mpfi_cmp_q)   (mpfi_srcptr,mpq_srcptr)
int    (*mpfi_cmp_fr)(mpfi_srcptr,mpfr_srcptr)

int    (*mpfi_is_pos)     (mpfi_srcptr)
int    (*mpfi_is_nonneg)  (mpfi_srcptr)
int    (*mpfi_is_neg)     (mpfi_srcptr)
int    (*mpfi_is_nonpos)  (mpfi_srcptr)
int    (*mpfi_is_zero)    (mpfi_srcptr)
int    (*mpfi_is_strictly_pos) (mpfi_srcptr)
int    (*mpfi_is_strictly_neg) (mpfi_srcptr)

# default comparison functions
int    mpfi_is_pos_default          (mpfi_srcptr)
int    mpfi_is_nonneg_default       (mpfi_srcptr)
int    mpfi_is_neg_default          (mpfi_srcptr)
int    mpfi_is_nonpos_default       (mpfi_srcptr)
int    mpfi_is_zero_default         (mpfi_srcptr)
int    mpfi_is_strictly_neg_default (mpfi_srcptr a)
int    mpfi_is_strictly_pos_default (mpfi_srcptr a)

int    mpfi_cmp_default      (mpfi_srcptr,mpfi_srcptr)
int    mpfi_cmp_d_default    (mpfi_srcptr,double)
```

```
int    mpfi_cmp_ui_default   (mpfi_srcptr,unsigned long)
int    mpfi_cmp_si_default   (mpfi_srcptr,long)
int    mpfi_cmp_z_default    (mpfi_srcptr,mpz_srcptr)
int    mpfi_cmp_q_default    (mpfi_srcptr,mpq_srcptr)
int    mpfi_cmp_fr_default (mpfi_srcptr,mpfr_srcptr)


int mpfi_has_zero (mpfi_srcptr)

int mpfi_nan_p (mpfi_srcptr)
int mpfi_inf_p (mpfi_srcptr)
int mpfi_bounded_p (mpfi_srcptr)

# Interval manipulation

# operations related to the internal representation by endpoints

# get left or right bound of the interval defined by the second argument and put the result in the first one
int   mpfi_get_left    (mpfr_ptr,mpfi_srcptr)
int   mpfi_get_right   (mpfr_ptr,mpfi_srcptr)

int   mpfi_revert_if_needed  (mpfi_ptr)

# Set operations on intervals
# "Convex hulls"
# extends the interval defined by the first argument so that it contains the second one

int   mpfi_put        (mpfi_ptr,mpfi_srcptr)
int   mpfi_put_d      (mpfi_ptr,double)
int   mpfi_put_si     (mpfi_ptr,long)
int   mpfi_put_ui     (mpfi_ptr,unsigned long)
int   mpfi_put_z      (mpfi_ptr,mpz_srcptr)
int   mpfi_put_q      (mpfi_ptr,mpq_srcptr)
int   mpfi_put_fr     (mpfi_ptr,mpfr_srcptr)

# builds an interval whose left bound is the lower (round -infty)
# than the second argument and the right bound is greater (round +infty) than the third one

int   mpfi_interv_d   (mpfi_ptr,double,double)
int   mpfi_interv_si  (mpfi_ptr,long,long)
int   mpfi_interv_ui  (mpfi_ptr,unsigned long,unsigned long)
int   mpfi_interv_z   (mpfi_ptr,mpz_srcptr,mpz_srcptr)
int   mpfi_interv_q   (mpfi_ptr,mpq_srcptr,mpq_srcptr)
int   mpfi_interv_fr  (mpfi_ptr,mpfr_srcptr,mpfr_srcptr)

# Inclusion tests
# tests if the first argument is inside the interval defined by the second one
int   mpfi_is_strictly_inside   (mpfi_srcptr,mpfi_srcptr)
int   mpfi_is_inside         (mpfi_srcptr, mpfi_srcptr)
int   mpfi_is_inside_d       (double, mpfi_srcptr)
int   mpfi_is_inside_ui      (unsigned long, mpfi_srcptr)
int   mpfi_is_inside_si      (long, mpfi_srcptr)
int   mpfi_is_inside_z       (mpz_srcptr,mpfi_srcptr)
int   mpfi_is_inside_q       (mpq_srcptr,mpfi_srcptr)
int   mpfi_is_inside_fr    (mpfr_srcptr,mpfi_srcptr)

# set operations
int   mpfi_is_empty         (mpfi_srcptr)
int   mpfi_intersect        (mpfi_ptr, mpfi_srcptr, mpfi_srcptr)
int   mpfi_union            (mpfi_ptr, mpfi_srcptr, mpfi_srcptr)

# complement... : to do later


# Miscellaneous

# adds the second argument to the right bound of the first one and
# subtracts the second argument to the left bound of the first one
int  mpfi_increase         (mpfi_ptr,mpfr_srcptr)
# keeps the same center and multiply the radius by 2*(1+fact)
int mpfi_blow(mpfi_ptr, mpfi_srcptr, double)
# splits the interval into 2 halves
int mpfi_bisect(mpfi_ptr, mpfi_ptr, mpfi_srcptr)

char * mpfi_get_version()

# Error handling

void   mpfi_reset_error()
void   mpfi_set_error(int)
```

```
    int    mpfi_is_error()

    #define MPFI_FLAGS_BOTH_ENDPOINTS_EXACT       0
    #define MPFI_FLAGS_LEFT_ENDPOINT_INEXACT      1
    #define MPFI_FLAGS_RIGHT_ENDPOINT_INEXACT     2
    #define MPFI_FLAGS_BOTH_ENDPOINTS_INEXACT     3

    #define MPFI_BOTH_ARE_EXACT(x) ( (int)(x) == 0 )
    #define MPFI_LEFT_IS_INEXACT(x) ( (int)(x)%2 )
    #define MPFI_RIGHT_IS_INEXACT(x) ( (int)(x)/2 )
    #define MPFI_BOTH_ARE_INEXACT(x) ( (int)(x) == 3 )

    #define MPFI_REVERT_INEXACT_FLAGS(x) ( ((x)==1) ? 2 : ((x)==2) ? 1 : x )

    #define MPFI_NAN_P(a) ( MPFR_IS_NAN(&(a->left)) || MPFR_IS_NAN (&(a->right)) )
    #define MPFI_INF_P(a) ( MPFR_IS_INF(&(a->left)) || MPFR_IS_INF (&(a->right)) )
    #define MPFI_IS_ZERO(a)  (MPFI_NAN_P(a) ? 0 : ((MPFR_SIGN(&(a->right))==0) && (MPFR_SIGN(&(a->left))==0)))

    #define MPFI_CLEAR(m) {mpfr_clear(&(m->right)); mpfr_clear(&(m->left));}

import_gmpy()

sizeof_mpfi = sizeof(mpfi_t)

def mpf_set_default_prec(prec):
    __c_mpf_set_default_prec(prec)

cdef set_mpf( mpf_t _x, x ):
    " set _x from x, steals a ref "
    assert Pympf_Check(x)
    mpf_set( _x, (<PympfObject*>x)[0].f )

cdef object mpf2py( mpf_t x ):
    " new ref "
    cdef object _x
    cdef int bits
    bits = mpf_get_prec( x )
    _x = Pympf_new( bits )
    mpf_set( (<PympfObject*>_x)[0].f, x )
    return <object>_x

cdef object mpfr2py( mpfr_t x, mpfr_rnd_t rnd ):
    " new ref "
    cdef object _x
    cdef int bits
    bits = mpfr_get_prec( x )
    _x = Pympf_new( bits )
    mpfr_get_f( (<PympfObject*>_x)[0].f, x, rnd )
    return <object>_x

import gmpy
from gmpy import mpf

cdef object mpf_type
mpf_type = type(mpf(1.0))

cdef class Interval

cdef object c_promote(x):
    if isinstance(x,Interval):
        pass
    elif hasattr(x,"__float__") or Pympf_Check(x):
        x = Interval(x)
    else:
        x = None
    return x

def promote(x):
    if isinstance(x,Interval):
        return x
    if hasattr(x,"__float__") or Pympf_Check(x):
        return Interval(x)
    return None

def set_default_prec(prec):
    mpfr_set_default_prec(prec)
    gmpy.set_minprec(prec)

def get_default_prec():
    return mpfr_get_default_prec()
```

```
class DisorderException(Exception):
    pass

cdef class Interval:
    cdef mpfi_t x

    def __init__( self, lower=None, upper=None ):
        cdef Interval _lower
        cdef mpfr_t fr_lower
        cdef mpfr_t fr_upper
        cdef PympfObject *pympf_lower
        cdef PympfObject *pympf_upper

        if isinstance(lower, Interval):
            _lower = lower
            mpfi_init_set( self.x, _lower.x )
            assert upper is None, "but first arg is already an Interval ?!?!"
        else:
            if hasattr(lower,"__float__"):
                mpfi_init_set_d( self.x, lower )
            elif Pympf_Check(lower):
                pympf_lower = <PympfObject*>lower
                mpfr_init(fr_lower)
                mpfr_set_f(fr_lower,pympf_lower[0].f,GMP_RNDD)
                mpfi_init_set_fr( self.x, fr_lower )
                mpfr_clear(fr_lower)
            elif lower is None:
                mpfi_init( self.x )
                assert upper is None
            else:
                mpfi_init( self.x )
                raise TypeError, "what's this: %s"%repr(lower)
            if hasattr(upper,"__float__"):
                mpfi_put_d( self.x, upper )
            elif Pympf_Check(upper):
                pympf_upper = <PympfObject*>upper
                mpfr_init(fr_upper)
                mpfr_set_f(fr_upper,pympf_upper[0].f,GMP_RNDU)
                mpfi_put_fr( self.x, fr_upper )
                mpfr_clear(fr_upper)
            elif upper is not None:
                mpfi_init( self.x )
                raise TypeError, "what's this: %s"%upper
        assert (lower is None and upper is None) or not mpfi_is_empty( self.x ),\
            (self.lower.digits(),self.upper.digits(),lower,upper)

    def __dealloc__( self ):
        mpfi_clear( self.x )

    def get_addr( self ):
        cdef long addr
        addr = <long>self.x
        return addr

    property lower:
        def __get__( self ):
            cdef mpfr_t fr_lower
            mpfr_init( fr_lower )
            mpfi_get_left( fr_lower, self.x )
            lower = mpfr2py( fr_lower, GMP_RNDD )
            mpfr_clear( fr_lower )
            return lower

    property upper:
        def __get__( self ):
            cdef mpfr_t fr_upper
            mpfr_init( fr_upper )
            mpfi_get_right( fr_upper, self.x )
            upper = mpfr2py( fr_upper, GMP_RNDU )
            mpfr_clear( fr_upper )
            return upper

    property prec:
        def __get__( self ):
            return mpfi_get_prec( self.x )

        def __set__( self, prec ):
            mpfi_round_prec( self.x, prec )
```

```
def __repr__( self ):
    return self.to_string()

def __str__( self ):
    return "[%.9f,%.9f]" % (self.lower, self.upper)

def to_string( self ):
    cdef char *_s_lower, *_s_upper
    cdef mp_exp_t exp
    cdef mpfr_t fr_lower, fr_upper

    mpfr_init( fr_lower )
    mpfi_get_left( fr_lower, self.x )
    mpfr_init( fr_upper )
    mpfi_get_right( fr_upper, self.x )

    _s_lower = mpfr_get_str(NULL, &exp, 10, 0, fr_lower, GMP_RNDD )
    s_lower = _s_lower
    if s_lower is None:
        raise Exception
    if s_lower[0]=='-':
        sl = s_lower[:2]+'.'+s_lower[2:]+'e'+str(exp-1)
    else:
        sl = s_lower[:1]+'.'+s_lower[1:]+'e'+str(exp-1)

    _s_upper = mpfr_get_str(NULL, &exp, 10, 0, fr_upper, GMP_RNDD )
    s_upper = _s_upper
    if s_upper is None:
        raise Exception
    if s_upper[0]=='-':
        su = s_upper[:2]+'.'+s_upper[2:]+'e'+str(exp-1)
    else:
        su = s_upper[:1]+'.'+s_upper[1:]+'e'+str(exp-1)

    mpfr_free_str(_s_lower)
    mpfr_free_str(_s_upper)
    mpfr_clear( fr_lower )
    mpfr_clear( fr_upper )
    return "Interval(mpf('%s'), mpf('%s'))" % (sl, su)

def __hash__( self ):
    return hash(self.to_string())

def __richcmp__( _x, _y, op ):
    cdef Interval x, y, z
    x = c_promote(_x)
    if x is None:
        return NotImplemented
    y = c_promote(_y)
    if y is None:
        return NotImplemented
    if op == 0: # <
        if x.upper < y.lower:
            return True
    elif op == 1: # <=
        if x.upper <= y.lower:
            return True
    elif op == 2: # ==
        if x.lower == y.lower and x.upper == y.upper:
            return True
    elif op == 3: # !=
        if x.lower != y.lower or x.upper != y.upper:
            return True
    elif op == 4: # >
        if x.lower > y.upper:
            return True
    elif op == 5: # >=
        if x.lower >= y.upper:
            return True
    return False

def freeze(self):
    return self

def __neg__( x ):
    cdef Interval y
    y = Interval()
    mpfi_neg( y.x, x.x )
    if mpfi_is_empty( y.x ):
        raise DisorderException
```

```
    return y

def __pos__( x ):
    return x

def __abs__( x ):
    cdef Interval y
    y = Interval()
    mpfi_abs( y.x, x.x )
    if mpfi_is_empty( y.x ):
        raise DisorderException
    return y

def __add__( _x, _y ):
    cdef Interval x, y, z
    x = c_promote(_x)
    if x is None:
        return NotImplemented
    y = c_promote(_y)
    if y is None:
        return NotImplemented
    z = Interval()
    mpfi_add( z.x, x.x, y.x )
    if mpfi_is_empty( z.x ):
        raise DisorderException((x, y, z))
    return z

def __sub__( _x, _y ):
    cdef Interval x, y, z
    x = c_promote(_x)
    if x is None:
        return NotImplemented
    y = c_promote(_y)
    if y is None:
        return NotImplemented
    z = Interval()
    mpfi_sub( z.x, x.x, y.x )
    if mpfi_is_empty( z.x ):
        raise DisorderException((x, y, z))
    return z

def __mul__( _x, _y ):
    cdef Interval x, y, z
    x = c_promote(_x)
    if x is None:
        return NotImplemented
    y = c_promote(_y)
    if y is None:
        return NotImplemented
    z = Interval()
    mpfi_mul( z.x, x.x, y.x )
    if mpfi_is_empty( z.x ):
        raise DisorderException((x, y, z))
    return z

def __div__( _x, _y ):
    cdef Interval x, y, z
    x = c_promote(_x)
    if x is None:
        return NotImplemented
    y = c_promote(_y)
    if y is None:
        return NotImplemented
    z = Interval()
    mpfi_div( z.x, x.x, y.x )
    if mpfi_is_empty( z.x ):
        raise DisorderException((x, y, z))
    return z

def __pow__( _x, y, w ):
    cdef Interval x, z
    cdef mpfr_t x_lower, x_upper
    cdef mpfr_t z_lower, z_upper

    x = c_promote(_x)
    if x is None:
        return NotImplemented
    if type(y)==Interval and y.width()==0.0 and int(y.lower)==y.lower:
        doweusethisatall
        y=int(y.lower)
```

```
    if type(y) in (int,float):
        if int(y)==y:
            y = int(y)
            if y==0:
                return Interval(1.0)
            if y==1:
                return x
            if y==2:
                return x.sqr()
            mpfr_init( x_lower )
            mpfr_init( x_upper )
            mpfr_init( z_lower )
            mpfr_init( z_upper )
            mpfi_get_left( x_lower, x.x )
            mpfi_get_right( x_upper, x.x )
            z = Interval()
            mpfi_get_left( z_lower, z.x )
            mpfi_get_right( z_upper, z.x )
            mpfr_pow_si( z_lower, x_lower, y, GMP_RNDD )
            mpfr_pow_si( z_upper, x_upper, y, GMP_RNDU )
            mpfi_set_fr( z.x, z_lower )
            mpfi_put_fr( z.x, z_upper )
            mpfr_clear( x_lower )
            mpfr_clear( x_upper )
            mpfr_clear( z_lower )
            mpfr_clear( z_upper )
            if mpfi_is_empty( z.x ):
                raise DisorderException((x, y, z))
            return z
        elif y==0.5:
            return _x.sqrt()
        z = (y*x.log()).exp()
        if mpfi_is_empty( z.x ):
            raise DisorderException((x, y, z))
        return z
    y = c_promote(y)
    if y is None:
        return NotImplemented
    assert 0, "implement me! "
    return NotImplemented
    # return Interval(x.lower**y,x.upper**y) # BROKEN
    # this is useless:
    # z = Interval(1.0)
    # while y:
    #     z = z*x
    #     y = y - 1
    # return z

def width(x):
    cdef mpfr_t fr_y
    cdef mpf_t f_y
    cdef int bits
    cdef object py_y
    mpfr_init(fr_y)
    mpfi_diam_abs(fr_y, x.x)
    mpf_init( f_y )
    mpfr_get_f( f_y, fr_y, GMP_RNDU )
    bits = mpf_get_prec( f_y )
    py_y = Pympf_new( bits )
    mpf_set( (<PympfObject*>py_y)[0].f, f_y )
    mpfr_clear(fr_y)
    return py_y

def centre(x):
    cdef mpfr_t y
    mpfr_init(y)
    mpfi_mid(y, x.x)
    _y = mpfr2py(y, GMP_RNDN)
    mpfr_clear(y)
    return _y

def overlapping(x, Interval y):
    return x.upper>=y.lower and y.upper>=x.lower

def intersect(x, Interval y):
    cdef Interval z
    z = Interval()
    mpfi_intersect(z.x, x.x, y.x)
    if mpfi_is_empty( z.x ):
        raise DisorderException((x, y, z))
```

```
        return z

    def hull(x, Interval y):
        cdef Interval z
        z = Interval()
        mpfi_union(z.x, x.x, y.x)
        if mpfi_is_empty( z.x ):
            raise DisorderException((x, y, z))
        return z

    def contains( x, _y ):
        cdef Interval y
        if not isinstance(_y,Interval):
            y = Interval(_y)
        else:
            y = _y
        return mpfi_is_inside( y.x, x.x )

    def is_close(self, other, EPSILON=1e-13):
        return abs(self.lower-other.lower)<EPSILON and abs(self.upper-other.upper)<EPSILON

    def bisect(x):
        cdef Interval y
        cdef Interval z
        y = Interval()
        z = Interval()
        mpfi_bisect( y.x, z.x, x.x )
        # return x,x if x width is zero ?
        if mpfi_is_empty( y.x ):
            raise DisorderException()
        if mpfi_is_empty( z.x ):
            raise DisorderException()
        return y, z

    def split(x, n):
        # subdivide x into at least n Intervals
        ivs = [x]
        while len(ivs)<n:
            _ivs = []
            for _x in ivs:
                l, r =_x.bisect()
                _ivs.append(l)
                _ivs.append(r)
            ivs = _ivs
        return ivs

#    def ceil(x):
#        return Interval(ceil(x._lower), ceil(x._upper))
#    def floor(x):
#        return Interval(floor(x._lower), floor(x._upper))

#    def max(x, y):
#        return Interval(max(x._lower, y._lower), max(x._upper, y._upper))
#    def min(x, y):
#        return Interval(min(x._lower, y._lower), min(x._upper, y._upper))

    def sqr(x):
        cdef Interval y
        y = Interval()
        mpfi_sqr( y.x, x.x )
        if mpfi_is_empty( y.x ):
            raise DisorderException()
        return y

    def sqrt( x ):
        cdef Interval y
        y = Interval()
        mpfi_sqrt( y.x, x.x )
        if mpfi_is_empty( y.x ):
            raise DisorderException()
        return y

    def hemimetric(x, y):
        ms = [
            [ abs(x.lower-y.lower), abs(x.lower-y.upper) ],
            [ abs(x.upper-y.lower), abs(x.upper-y.upper) ],
        ]
        return max(min(ms[0][0],ms[0][1]),min(ms[1][0],ms[1][1]))

    def metric( x, y ):
```

```
    " hausdorff metric "
    return max( x.hemimetric(y), y.hemimetric(x) )

def get_pi( x ):
    cdef Interval y
    y = Interval()
    mpfi_const_pi(y.x)
    return y

def exp( x ):
    cdef Interval y
    y = Interval()
    mpfi_exp( y.x, x.x )
    if mpfi_is_empty( y.x ):
        raise DisorderException()
    return y

def log( x ):
    cdef Interval y
    y = Interval()
    mpfi_log( y.x, x.x )
    return y

def sin( x ):
    cdef Interval y
    y = Interval()
    mpfi_sin( y.x, x.x )
    return y

def cos( x ):
    cdef Interval y
    y = Interval()
    mpfi_cos( y.x, x.x )
    return y

def tan( x ):
    cdef Interval y
    y = Interval()
    mpfi_tan( y.x, x.x )
    return y

def asin( x ):
    cdef Interval y
    y = Interval()
    mpfi_asin( y.x, x.x )
    return y

def acos( x ):
    cdef Interval y
    y = Interval()
    mpfi_acos( y.x, x.x )
    return y

def atan( x ):
    cdef Interval y
    y = Interval()
    mpfi_atan( y.x, x.x )
    return y

def sinh( x ):
    cdef Interval y
    y = Interval()
    mpfi_sinh( y.x, x.x )
    return y

def cosh( x ):
    cdef Interval y
    y = Interval()
    mpfi_cosh( y.x, x.x )
    return y

def tanh( x ):
    cdef Interval y
    y = Interval()
    mpfi_tanh( y.x, x.x )
    return y

def asinh( x ):
    cdef Interval y
    y = Interval()
```

```
        mpfi_asinh( y.x, x.x )
        return y

    def acosh( x ):
        cdef Interval y
        y = Interval()
        mpfi_acosh( y.x, x.x )
        return y

    def atanh( x ):
        cdef Interval y
        y = Interval()
        mpfi_atanh( y.x, x.x )
        return y

def from_string(s):
    # unfortunately, we lose precision going through mpf type.
    assert s.startswith('Interval(')
    s = s[len('Interval('):-1]
    l, u = s.split(',')
    l = mpf(l)
    u = mpf(u)
    return Interval(l,u)
```

B.4 scalar/symbolic.py

```
import sys
import os

from random import random, seed
from time import time

import fmath

try:
    import gmpy
    from gmpy import mpf
except ImportError:
    print " ** no gmpy found ** "

from xinterval import Interval

__module__ = sys.modules[__name__]

class ComputationContext(object):

    def __init__( self, name, args, dumpfile=None ):
        """
            @param name: function name
            @param args: function arg names (can build lvalue's out of these)
            @param dumpfile: dump the function to this file (optional)
        """
        self.lines = []
        self.append( 'def %s(%s):' % (name, ', '.join( [str(v) for v in args] )))
        self.float_cache = {}
        self.var_cache = {}
        self.var_floats = []
        self.name = name
        self.dumpfile = dumpfile

    def append( self, line ):
        self.lines.append(line)

    def assign( self, lvalue, a_float ):
        """
            @param lvalue: a string repr of the left hand side of the assign
            @param a_float: the right hand side (abstract float) of the assign
        """
        #Generate a sequence of unique "assign" statements:
        #  float_cache: map Float -> Var
        #  var_cache: map Var -> Float  (an "assignment")
        #  var_floats: what order to assign the vars in
        float_cache, var_cache, var_floats\
            = a_float.uniq_form(self.float_cache, self.var_cache, self.var_floats)
        assert float_cache is self.float_cache
        assert var_cache is self.var_cache
```

```
        assert var_floats is self.var_floats
        self.var_cache[lvalue] = self.float_cache[a_float] # ASSIGNMENT
        self.var_floats.append( lvalue )

    def finalize(self, rexpr=""):
        """
            Generate the function. (exec the source.)
        """
        for var in self.var_floats:
            self.append( "    %s = %s" % (var, self.var_cache[var]) )
        self.append("    return %s"%rexpr)
        src = '\n'.join(self.lines)
        if self.dumpfile is not None:
            print >>self.dumpfile, src
        self.dumpfile.close()
        exec src
        return locals()[self.name]

#
################################################################################
#

class Float(object):
    scalar_zero = 0.0
    scalar_one = 1.0
    scalar_promote = float
    scalar_type = float
    old_scalar = None

    def push_scalar(cls, zero, one, promote, type): # XX wrong order
        assert cls.old_scalar is None, "stack depth of 1! (already in symbolic mode)"
        cls.old_scalar = cls.scalar_zero, cls.scalar_one, cls.scalar_promote, cls.scalar_type # XX wrong order
        cls.scalar_zero = zero
        cls.scalar_one = one
        cls.scalar_promote = promote
        cls.scalar_type = type
    push_scalar = classmethod(push_scalar)

    def restore_scalar(cls):
        cls.scalar_zero, cls.scalar_one, cls.scalar_promote, cls.scalar_type = cls.old_scalar # XX wrong order
        cls.old_scalar = None
    restore_scalar = classmethod(restore_scalar)

    cache = {} # Fly-weight design pattern
    unop_cache = {} # map name to class, eg. SinFloat

    def __new__(cls, val=None):
        ob = object.__new__(cls)
        ob._hash = None # cache the hash value
        assert isinstance(val,cls.scalar_type), val
        ob.val = val
        if ob.val == cls.scalar_zero:
            assert ZeroFloat._zero_float is None # singleton
        if ob in cls.cache:
            return cls.cache[ob]
        cls.cache[ob] = ob
        return ob

    def __init__(self, *args):
        pass

    def __hash__(self):
        if self._hash is None:
            self._hash = self.hash() # remember this
        return self._hash # cached value

    def hash(self):
        return hash(self.val)

    def __eq__(self, other):
        return type(self)==type(other) and self.val==other.val

    def __cmp__(self, other):
        assert 0, (self,other)

    def promote(self, val):
        if isinstance(val,Float):
            return val
        if isinstance(val,str):
            return Var(val)
```

```
    if type(val) in (int,float,Interval):
        val = self.scalar_promote(val)
    else:
        return None
    if val==self.scalar_zero:
        return ZeroFloat()
    if val==self.scalar_one:
        return OneFloat()
    return Float(val)
promote = classmethod(promote)

def expr(self):
    return repr(self.val)
jz_expr = expr

def deepstr(self):
    return "%s(%s)" % (self.__class__.__name__, self.val)

def deeplen(self):
    return 1

def uniqlen(self, cache=None):
    if cache is None:
        cache = {}
    if self in cache:
        return 0
    else:
        cache[self]=None
        return 1

def deepvisit(self, visitor, cache=None):
    "visit each node"
    if cache is None:
        cache = set()
    if self not in cache:
        visitor(self)
        cache.add(self)

def deepreplace(self, visitor):
    "return a bottom up re-write"
    return visitor(self)

def uniq_form(self, float_cache={}, var_cache={}, var_floats=[]):
    """
        Generate a sequence of unique "assign" statements
            float_cache: map Float -> Var
            var_cache: map Var -> Float  (an "assignment")
            var_floats: what order to assign the vars in
    """
    if self not in float_cache:
        var = Var.temp()
        float_cache[self]=var
        var_cache[var]=self # ASSIGNMENT
        var_floats.append(var)
    return float_cache, var_cache, var_floats

def get_func(self, name, free_vars=[], dumpfile=None):
    float_cache, var_cache, var_floats = self.uniq_form()
    lines = []

    lines.append( 'def %s(%s):' % (name, ', '.join( [str(v) for v in free_vars] )))
    for var in var_floats:
        lines.append( "  %s = %s" % (var, var_cache[var]) )
    lines.append( "  return %s" % (var_floats[-1]) )
    src = '\n'.join(lines)
    if dumpfile is not None:
        print >>dumpfile, src

    namespace = dict(globals())
    exec src in namespace
    func = namespace[name]
    return func

def rewrite(self, var, newvar):
    return self

def __getitem__(self, idx):
    raise IndexError

def __getattr__(self, name):
```

```
    "eg. x.sin() => SinFloat(x) "
    if name.startswith('__'):
        raise AttributeError
    orig_name = name
    name = name[0].upper()+name[1:]+'Float' # eg. SinFloat
    cls = Float.unop_cache.get(name)
    if cls is None:
        cls = type(name, (PostfixOpFloat,), {'op':orig_name})
        Float.unop_cache[name]=cls
    return lambda : cls(self)

def __str__(self):
    return self.expr()

def __pos__( self ):
    return self

def __neg__( self ):
    return NegFloat(self)

def __add__( self, other ):
    other = Float.promote(other)
    if other is None:
        return NotImplemented
    if type(other) == ZeroFloat:
        return self
    return AddFloat(self, other)

def __radd__( self, other ):
    other = Float.promote(other)
    if other is None:
        return NotImplemented
    if type(other) == ZeroFloat:
        return self
    return AddFloat(other, self)

def __sub__( self, other ):
    other = Float.promote(other)
    if other is None:
        return NotImplemented
    if type(other) == ZeroFloat:
        return self
    return SubFloat(self, other)

def __rsub__( self, other ):
    other = Float.promote(other)
    if other is None:
        return NotImplemented
    if type(other) == ZeroFloat:
        return -self
    return SubFloat(other, self)

def __mul__( self, other ):
    other = Float.promote(other)
    if other is None:
        return NotImplemented
    if type(other) == ZeroFloat:
        return ZeroFloat()
    if type(other) == OneFloat:
        return self
    return MulFloat(self, other)

def __rmul__( self, other ):
    other = Float.promote(other)
    if other is None:
        return NotImplemented
    if type(other) == ZeroFloat:
        return ZeroFloat()
    if type(other) == OneFloat:
        return self
    return MulFloat(other, self)

def __div__( self, other ):
    other = Float.promote(other)
    if other is None:
        return NotImplemented
    if type(other) == ZeroFloat:
        return ZeroFloat()
    if type(other) == OneFloat:
        return self
```

```
        return DivFloat(self, other)

    def __rdiv__( self, other ):
        other = Float.promote(other)
        if other is None:
            return NotImplemented
        return DivFloat(other, self)

    def __pow__( self, other ):
        if type(other) != int:
            other = Float.promote(other)
            if other is None:
                return NotImplemented
            return PowFloat(self, other)
        return IntPowFloat(self, other)

    @property
    def jz_op(self):
        return self.op

class ZeroFloat(Float):
    _zero_float = None

    def __new__(cls):
        if cls._zero_float is None:
            cls._zero_float = Float.__new__(ZeroFloat,cls.scalar_zero)
        return cls._zero_float

    def __neg__( self ):
        return self

    def __add__( self, other ):
        other = Float.promote(other)
        if other is None:
            return NotImplemented
        return other

    def __radd__( self, other ):
        other = Float.promote(other)
        if other is None:
            return NotImplemented
        return other

    def __sub__( self, other ):
        other = Float.promote(other)
        if other is None:
            return NotImplemented
        return -other

    def __rsub__( self, other ):
        other = Float.promote(other)
        if other is None:
            return NotImplemented
        return other

    def __mul__( self, other ):
        return self

    def __rmul__( self, other ):
        return self

    def __div__( self, other ):
        return self

    def __rdiv__( self, other ):
        raise ZeroDivisionError


class OneFloat(Float):
    _one_float = None

    def __new__(cls):
        if cls._one_float is None:
            cls._one_float = Float.__new__(OneFloat,cls.scalar_one)
        return cls._one_float

    def __mul__( self, other ):
        other = Float.promote(other)
        if other is None:
            return NotImplemented
```

```
        return other

    def __rmul__( self, other ):
        other = Float.promote(other)
        if other is None:
            return NotImplemented
        return other

    def __rdiv__( self, other ):
        other = Float.promote(other)
        if other is None:
            return NotImplemented
        return other


class Var(Float):
    uid = 0

    def __new__(cls, name):
        ob = object.__new__(cls)
        ob._hash = None
        ob.name = name
        if ob in cls.cache:
            return cls.cache[ob]
        cls.cache[ob] = ob
        return ob

    def temp(cls, prefix='temp_'):
        cls.uid += 1
        return cls('%s%d'%(prefix, cls.uid))
    temp = classmethod(temp)

    def hash(self):
        return hash(self.name)

    def deepstr(self):
        return "%s(%r)" % (self.__class__.__name__, self.name)

    def __eq__(self, other):
        return type(self)==type(other) and self.name==other.name

    def __repr__(self):
        return str(self)

    def expr(self):
        return self.name
    jz_expr = expr

    def rewrite(self, var, newvar):
        if var==self:
            return newvar
        return self

class UnaryOpFloat(Float):

    def __new__(cls, a):
        ob = object.__new__(cls)
        ob._hash = None
        ob.a = a
        if ob in cls.cache:
            return cls.cache[ob]
        cls.cache[ob] = ob
        return ob

    def __getitem__(self, idx):
        return (self.a,)[idx]

    def hash(self):
        return hash((self.op,self.a))

    def __eq__(self, other):
        return type(self)==type(other) and \
            (self.op,self.a)==(other.op,other.a)

    def expr(self):
        return "(%s(%s))"%(self.op, self.a.expr())

    def jz_expr(self):
        return "(%s(%s))"%(self.op, self.a.jz_expr())
```

```
    def deepstr(self):
        return "%s(%s)" % (self.__class__.__name__, self.a.deepstr())

    def deeplen(self):
        return 1+self.a.deeplen()

    def uniqlen(self, cache=None):
        if cache is None:
            cache = {}
        if self in cache:
            return 0
        else:
            cache[self]=self
            return 1+self.a.uniqlen(cache)

    def deepvisit(self, visitor, cache=None):
        if cache is None:
            cache = set()
        if self not in cache:
            visitor(self)
            self.a.deepvisit(visitor, cache)
            cache.add(self)

    def deepreplace(self, visitor):
        a = self.a.deepreplace(visitor)
        self = self.__class__(a)
        return visitor(self)

    def uniq_form(self, float_cache={}, var_cache={}, var_floats=[]):
        if self not in float_cache:
            self.a.uniq_form(float_cache,var_cache,var_floats)
            var = Var.temp()
            a = float_cache[self.a]
            float_cache[self]=var
            var_cache[var]=self.__class__(a) # ASSIGNMENT
            var_floats.append(var)
        return float_cache, var_cache, var_floats

    def rewrite(self, var, newvar):
        return self.__class__( self.a.rewrite( var, newvar ) )

class PostfixOpFloat(UnaryOpFloat):

    def expr(self):
        # this only works for Interval arguments
        # need to do this to produce a function that works with float's
        return "fmath.%s(%s)"%(self.op, self.a.expr())

    def jz_expr(self):
        return "%s%s"%(self.jz_op, self.a.jz_expr())


class NegFloat(UnaryOpFloat):
    op = '-'


# we also build other subclasses of UnaryOpFloat
# dynamically, eg. SinFloat

class BinOpFloat(Float):

    def __new__(cls, a, b):
        ob = object.__new__(cls)
        ob._hash = None
        ob.a = a
        ob.b = b
        if ob in cls.cache:
            return cls.cache[ob]
        cls.cache[ob] = ob
        return ob

    def __getitem__(self, idx):
        return (self.a, self.b)[idx]

    def hash(self):
        return hash((self.op,self.a,self.b))

    def __eq__(self, other):
        return type(self)==type(other) and \
            (self.op,self.a,self.b)==(other.op,other.a,other.b)
```

```
    def expr(self):
        return '(%s %s %s)' % (self.a.expr(), self.op, self.b.expr())

    def jz_expr(self):
        return '(%s %s %s)' % (self.a.jz_expr(), self.jz_op, self.b.jz_expr())

    def deepstr(self):
        return "%s(%s, %s)" % (self.__class__.__name__, self.a.deepstr(), self.b.deepstr())

    def deeplen(self):
        return 1+self.a.deeplen()+self.b.deeplen()

    def uniqlen(self, cache=None):
        if cache is None:
            cache = {}
        if self in cache:
            return 0
        else:
            cache[self]=self
            return 1+self.a.uniqlen(cache)+self.b.uniqlen(cache)

    def deepvisit(self, visitor, cache=None):
        if cache is None:
            cache = set()
        if self not in cache:
            visitor(self)
            self.a.deepvisit(visitor, cache)
            self.b.deepvisit(visitor, cache)
            cache.add(self)
        return self

    def deepreplace(self, visitor):
        a = self.a.deepreplace(visitor)
        b = self.b.deepreplace(visitor)
        self = self.__class__(a, b)
        return visitor(self)

    def uniq_form(self, float_cache={}, var_cache={}, var_floats=[]):
        if self not in float_cache:
            self.a.uniq_form(float_cache,var_cache,var_floats)
            self.b.uniq_form(float_cache,var_cache,var_floats)
            var = Var.temp()
            a = float_cache[self.a]
            b = float_cache[self.b]
            float_cache[self]=var
            var_cache[var]=self.__class__(a,b) # ASSIGNMENT
            var_floats.append(var)
        return float_cache, var_cache, var_floats

    def rewrite(self, var, newvar):
        return self.__class__(
            self.a.rewrite( var, newvar ),
            self.b.rewrite( var, newvar ),
        )

class AddFloat(BinOpFloat):
    op = '+'

class SubFloat(BinOpFloat):
    op = '-'

class MulFloat(BinOpFloat):
    op = '*'

class DivFloat(BinOpFloat):
    op = '/'

class PowFloat(BinOpFloat):
    op = '**'
    jz_op = '^'

class IntPowFloat(BinOpFloat):
    op = '**'
    jz_op = '^'


    def expr(self):
        return '(%s %s %s)' % (self.a.expr(), self.op, self.b)
```

```
    def jz_expr(self):
        return '(%s %s %s)' % (self.a.jz_expr(), self.jz_op, self.b)

    def deeplen(self):
        return 1+self.a.deeplen()+1

    def deepstr(self):
        return "%s(%s, %s)" % (self.__class__.__name__, self.a.deepstr(), self.b)

    def uniqlen(self, cache=None):
        if cache is None:
            cache = {}
        if self in cache:
            return 0
        else:
            cache[self]=self
            return 1+self.a.uniqlen(cache)

    def deepvisit(self, visitor, cache=None):
        if cache is None:
            cache = set()
        if self not in cache:
            visitor(self)
            self.a.deepvisit(visitor, cache)
            cache.add(self)

    def deepreplace(self, visitor):
        a = self.a.deepreplace(visitor)
        self = self.__class__(a, self.b)
        return visitor(self)

    def uniq_form(self, float_cache={}, var_cache={}, var_floats=[]):
        if self not in float_cache:
            self.a.uniq_form(float_cache,var_cache,var_floats)
            var = Var.temp()
            a = float_cache[self.a]
            b = self.b
            float_cache[self]=var
            var_cache[var]=self.__class__(a,b) # ASSIGNMENT
            var_floats.append(var)
        return float_cache, var_cache, var_floats

    def rewrite(self, var, newvar):
        return self.__class__( self.a.rewrite( var, newvar ), self.b )
```

B.5 mjet.py

```
import sys
import os

from random import random, seed
from time import time

from pydx.scalar import fmath
from pydx.scalar import Scalar, set_interval_scalar, set_symbolic_scalar, restore_scalar
from pydx.scalar.symbolic import Float

try:
    import gmpy
    from gmpy import mpf
except ImportError:
    print " ** no gmpy found ** "

try:
    from scalar.xinterval import Interval, get_default_prec, set_default_prec, DisorderException
except ImportError, e:
    print " ** could not import xinterval ** "
    print e

__module__ = sys.modules[__name__]

Scalar.add_client(__module__)

#
################################################################################
#
```

```
#class slice(slice):
#    def __hash__(self):
#        return hash((self.start,self.stop,self.step))
# TypeError: Error when calling the metaclass bases
#         type 'slice' is not an acceptable base type

# do these two functions belong in this module ??
def xcross(idxs):
    if len(idxs):
        idx = idxs[0]
        if type(idx)==slice:
            assert idx.step is None, "Not Implemented"
            for idx in range(idx.start, idx.stop):
                for rest in xcross(idxs[1:]):
                    yield (idx,)+rest
        else:
            for rest in xcross(idxs[1:]):
                yield (idx,)+rest
    else:
        yield ()

def cross(idxs):
    if len(idxs):
        for idx in range(0,idxs[0]):
            for rest in cross(idxs[1:]):
                yield (idx,)+rest
    else:
        yield ()

### This cache doesn't seem to speed things up any.
#_multi_range_order_cache = {}
#def multi_range_order(*args):
#    idxss = _multi_range_order_cache.get(args)
#    if idxss is None:
#        idxss = tuple(idxs for idxs in _multi_range_order(*args))
#        _multi_range_order_cache[args]=idxss
#    return idxss
#def _multi_range_order(rank,start_order,stop_order):
def multi_range_order(rank,start_order,stop_order):
    """
        yield all multi indexes with sum from start_order to stop_order inclusive.
    """
    assert 0<=start_order<=stop_order
    if rank==0:
        assert start_order==stop_order==0
        yield ()
    elif rank==1:
        for i in range(start_order,stop_order+1):
            yield (i,)
    else:
        for i in range(0,stop_order+1):
            for rest in multi_range_order(rank-1,max(0,start_order-i),stop_order-i):
                yield (i,)+rest

def multi_range(idxs):
    if len(idxs):
        for idx in range(0,idxs[0]+1):
            for rest in multi_range(idxs[1:]):
                yield (idx,)+rest
    else:
        yield ()

def multi_zero(rank):
    return (0,)*rank
def multi_unit(rank,i):
    idxs = [0]*rank
    idxs[i] = 1
    return tuple(idxs)
def multi_unit_under(idxs):
    " return unit _idxs s.t. _idxs<=idxs "
    i = 0
    while idxs[i]==0: # IndexError means that idxs all 0
        i += 1
    _idxs=[0]*len(idxs)
    _idxs[i]=1
    return tuple(_idxs)

def multi_lexcmp(a_idxs, b_idxs):
    " lexicographical ordering "
```

```
    assert len(a_idxs)==len(b_idxs)
    i = 0
    while i < len(a_idxs) and a_idxs[i] == b_idxs[i]:
        i += 1
    if i==len(a_idxs):
        return 0
    return cmp(a_idxs[i],b_idxs[i])

def multi_cmp(a_idxs, b_idxs):
    le = True
    ge = True
    eq = True
    for a_idx,b_idx in zip(a_idxs,b_idxs):
        if a_idx>b_idx:
            eq = False
            le = False
        elif a_idx<b_idx:
            eq = False
            ge = False
    if eq:
        return 0
    if le:
        return -1
    if ge:
        return 1

def multi_add(a_idxs, b_idxs):
    assert len(a_idxs)==len(b_idxs)
    return tuple(a_idxs[i]+b_idxs[i] for i in range(len(a_idxs)))

def multi_sub(a_idxs, b_idxs):
    assert len(a_idxs)==len(b_idxs)
    c_idxs = tuple(a_idxs[i]-b_idxs[i] for i in range(len(a_idxs)))
    assert multi_cmp(c_idxs,multi_zero(len(a_idxs))) >= 0
    return c_idxs

def multi_partitions(idxs):
    rank = len(idxs)
    zero = multi_zero(rank)
    if idxs == zero:
        yield []
    else:
        for _idxs in multi_range(idxs):
            if _idxs == zero:
                continue
            for part in multi_partitions(multi_sub(idxs, _idxs)):
                if part and multi_lexcmp(_idxs, part[0]) > 0: # XX use extra parameter
                    continue
                yield [_idxs]+part

def n_choice(n,k):
    " n_choice(n,k): n choose k: n!/(k!(n-k)!) "
    assert k<=n
    if k==0:
        return 1
    if k==1:
        return n
    i = 1
    for j in range(n-k+1,n+1):
        i *= j
    for j in range(1,k+1):
        i /= j
    return i


class MJet(object):

    scalar = None

    def __init__(self, rank):
        self.rank = rank

    def promote(self, item):
        if not isinstance(item,MJet):
            item = self.scalar.promote(item)
            x = Jet({(0,)*self.rank:item})
            return x
        elif item.rank == self.rank:
            return item
        assert self.rank > item.rank
```

```
    if isinstance(item,Jet):
        x = Jet(rank = self.rank)
        extra_zeros = (0,)*(self.rank-item.rank)
        for idxs, value in item.cs.items():
            x[ idxs+extra_zeros ] = value
        return x
    else:
        # lazy promote
        return PromotedJet(item, self.rank)

def str(self, n):
    return "<%s>" % ' '.join(str(self[i]) for i in range(n))

def repr(self, n):
    return "<%s>" % ' '.join(repr(self[i]) for i in range(n))

def get_scalar_component(self):
    return self[(0,)*self.rank]
scalar_component=property(get_scalar_component)

def bdot(cls, a, b, n, i):
    """
        @param a: MJet
        @param b: MJet
        @param n: tuple
        @param i: int
    """
    rank = len(n)
    delta_i = multi_unit(rank, i)
    assert n[i]>0
    r = sum(((n[i]-j[i])*a[j]*b[multi_sub(n,j)] for j in multi_range(multi_sub(n,delta_i))),
                    cls.scalar.zero)
    r = r / n[i]
    return r
bdot = classmethod(bdot)

def expand(self, x, order):
    assert len(x)==self.rank
    r = self.scalar.zero
    for j in multi_range_order(self.rank, 0, order):
        s = self.scalar.one
        for i in range(self.rank):
            if j[i]>0:
                s *= x[i]**j[i]
        r = r + self[j] * s
    return r

#
# On order=5 this has around 0.001% more error: (???)
def expand_factorized(self, x, order):
    # factored taylor series
    assert len(x)==self.rank
    zero_idxs = (0,)*self.rank

    terms = {}
    for j in multi_range_order(self.rank, order, order):
        terms[j] = self[j]
    while order:
        order -= 1
        _terms = {} # new terms for next iteration
        for j in multi_range_order(self.rank, order, order):
            _terms[j] = self[j]
        for idxs in terms:
            i = 0
            while idxs[i]==0:
                i += 1
            value = terms[idxs] * x[i]
            _idxs = multi_sub(idxs,multi_unit(self.rank,i))
            _terms[_idxs] = _terms[_idxs] + value
        terms = _terms
    assert len(terms)==1
    return terms[zero_idxs]

def expand_err(self, x, order, err):
    assert len(x)==self.rank
    assert self.scalar.type in (Interval, Float)
    r = self.scalar.zero
    for j in multi_range_order(self.rank, 0, order):
        s = self.scalar.one
```

```
            for i in range(self.rank):
                if j[i]>0:
                    s *= x[i]**j[i]
            r = r + self[j] * s
        for i,j in enumerate(multi_range_order(self.rank, order+1, order+1)):
            s = self.scalar.one
            for k in range(self.rank):
                if j[k]>0:
                    s *= x[k]**j[k]
            r = r + err[i] * s
        assert i==len(err)-1
        return r

    def expand_err_factorized(self, x, order, err):
        # factored taylor series (less accurate)
        assert len(x)==self.rank
        zero_idxs = (0,)*self.rank

        order += 1
        terms = {}
        for i,j in enumerate(multi_range_order(self.rank, order, order)):
            terms[j] = err[i]
        while order:
            order -= 1
            _terms = {} # new terms for next iteration
            for j in multi_range_order(self.rank, order, order):
                _terms[j] = self[j]
            for idxs in terms:
                i = 0
                while idxs[i]==0:
                    i += 1
                value = terms[idxs] * x[i]
                _idxs = multi_sub(idxs,multi_unit(self.rank,i))
                _terms[_idxs] = _terms[_idxs] + value
            terms = _terms
        assert len(terms)==1
        return terms[zero_idxs]

    def __eq__(self, other):
        return repr(self)==repr(other)
    def __pos__(self):
        return self
    def __neg__(self):
        return NegJet(self)
    def __add__(self, other):
        return AddJet(self, other)
    def __radd__(self, other):
        other = self.promote(other)
        if other is None:
            return NotImplemented
        return AddJet(other, self)
    def __sub__(self, other):
        return SubJet(self, other)
    def __rsub__(self, other):
        other = self.promote(other)
        if other is None:
            return NotImplemented
        return SubJet(other, self)
    def __mul__(self, other):
        return MulJet(self, other)
    def __rmul__(self, other):
        other = self.promote(other)
        if other is None:
            return NotImplemented
        return MulJet(other, self)
    def __div__(self, other):
        return DivJet(self, other)
    def __rdiv__(self, other):
        other = self.promote(other)
        if other is None:
            return NotImplemented
        return DivJet(other, self)
    def __pow__(self, other):
        if other==0:
            r = Jet(rank=self.rank)
            r[(0,)*self.rank] = self.scalar.one
        elif other == 1:
            r = self
        elif other == 2:
            r = SqrJet(self)
```

```
        elif int(other)==other:
            return IntPowJet(self, other)
        elif other==0.5:
            r = SqrtJet(self)
        else:
            other = self.scalar.promote(other)
            r = (other*self.log()).exp()
        return r
    def sqr(self):
        return SqrJet(self)
    def exp(self):
        return ExpJet(self)
    def log(self):
        return LogJet(self)
    def sqrt(self):
        return SqrtJet(self)
    def atan(self):
        return ATanJet(self)
    def asin(self):
        return ASinJet(self)
    def acos(self):
        return ACosJet(self)
    def sin(self):
        return SinJet(self)
    def cos(self):
        return CosJet(self)
    def tan(self):
        return SinJet(self)/CosJet(self)
    def cot(self):
        return CosJet(self)/SinJet(self)
    def reciprocal(self):
        return self.scalar.one/self
    recip = reciprocal
    def sec(self):
        return self.cos().reciprocal()
    def sinh(self):
        return (self.exp()-(-self).exp())/(self.scalar.one*2)
    def cosh(self):
        return (self.exp()+(-self).exp())/(self.scalar.one*2)
    def tanh(self):
        return (self.exp()-(-self).exp()) / (self.exp()+(-self).exp())
Scalar.add_client(MJet)

class JetOp(MJet):
    """ Lazy Jet with item cache.
    """
    def __init__(self, rank):
        MJet.__init__(self, rank)
        self._cache = {} # speeds up things (exponentially!)
    def __getitem__(self, idxs):
        if type(idxs)==int:
            idxs=idxs,
        assert type(idxs)==tuple
        assert len(idxs)==self.rank
        item = self._cache.get(str(idxs)) # XX slice objects are not hashable :(
        if item is None:
            if [idx for idx in idxs if type(idx)==slice]:
                item = SliceJet(self, idxs)
            else:
                item = self.getitem(idxs)
            self._cache[str(idxs)] = item
        return item
    def __str__(self):
        return "%s(%s,%s)"%(self.__class__.__name__,self.a,self.b)
    def __repr__(self):
        return "%s(%s,%s)"%(self.__class__.__name__,repr(self.a),repr(self.b))
    __setitem__ = None

class PromotedJet(JetOp):
    def __init__(self, a, rank):
        JetOp.__init__(self, rank)
        self.a = a
        assert rank > self.a.rank
    def getitem(self, idxs):
        if not idxs[self.a.rank:self.rank]==(0,)*(self.rank-self.a.rank):
            # and it hasn't been set, so:
            return self.scalar.zero
        return self.a[ idxs[:self.a.rank] ]
    def __setitem__(self, idxs, value):
        assert type(idxs)==tuple
```

```
        self._cache[str(idxs)] = value

class AddJet(JetOp):
    def __init__(self, a, b):
        JetOp.__init__(self, a.rank)
        b = self.promote(b)
        assert b.rank == a.rank
        self.a = a
        self.b = b
    def getitem(self, idxs):
        assert type(idxs)==tuple
        assert isinstance(self.a[idxs],self.scalar.type), (self.a[idxs],self.scalar.type)
        assert isinstance(self.b[idxs],self.scalar.type)
        return self.a[idxs]+self.b[idxs]

class SubJet(JetOp):
    def __init__(self, a, b):
        JetOp.__init__(self, a.rank)
        b = self.promote(b)
        assert b.rank == a.rank
        self.a = a
        self.b = b
    def getitem(self, idxs):
        assert type(idxs)==tuple
        assert isinstance(self.a[idxs],self.scalar.type)
        assert isinstance(self.b[idxs],self.scalar.type)
        return self.a[idxs]-self.b[idxs]

class NegJet(JetOp):
    def __init__(self, a):
        JetOp.__init__(self, a.rank)
        self.a = a
    def getitem(self, idxs):
        assert type(idxs)==tuple
        assert isinstance(self.a[idxs],self.scalar.type)
        return -self.a[idxs]
    def __str__(self):
        return "%s(%s)"%(self.__class__.__name__,self.a)
    def __repr__(self):
        return "%s(%s)"%(self.__class__.__name__,repr(self.a))

class MulJet(JetOp):
    def __init__(self, a, b):
        JetOp.__init__(self, a.rank)
        b = self.promote(b)
        assert b.rank == a.rank
        self.a = a
        self.b = b
    def getitem(self, idxs):
        assert type(idxs)==tuple
        idxss = [ _ for _ in multi_range(idxs) ]
        _idxss = [ multi_sub(idxs,_idxs) for _idxs in idxss ]
        return sum((self.a[a_idxs] * self.b[b_idxs] for a_idxs, b_idxs in zip(_idxss,idxss)), self.scalar.zero)

class DivJet(JetOp):
    def __init__(self, b, c):
        JetOp.__init__(self, b.rank)
        c = self.promote(c)
        assert c.rank == b.rank
        self.b = b
        self.c = c
    def __str__(self):
        return "%s(%s,%s)"%(self.__class__.__name__,self.b,self.c)
    def __repr__(self):
        return "%s(%s,%s)"%(self.__class__.__name__,repr(self.b),repr(self.c))
    def getitem(self, idxs):
        assert type(idxs)==tuple
        idxss = [ _ for _ in multi_range(idxs) if _ != (0,)*self.rank ]
        _idxss = [ multi_sub(idxs,_idxs) for _idxs in idxss ]
        r = self.b[idxs]
        r = r - sum([ self.c[c_idxs] * self[self_idxs]
            for c_idxs, self_idxs in zip(idxss,_idxss) ], self.scalar.zero)
        r = r / self.c[ (0,)*self.rank ]
        return r

class IntPowJet(JetOp):
    def __init__(self, b, n):
        assert type(n)==int
        JetOp.__init__(self, b.rank)
        self.b = b
```

```
        self.n = n
    def __str__(self):
        return "%s(%s,%s)"%(self.__class__.__name__,self.b,self.n)
    def __repr__(self):
        return "%s(%s,%s)"%(self.__class__.__name__,repr(self.b),repr(self.n))
    def getitem(self, idxs):
        parts = multi_partitions(idxs)
        r = self.scalar.zero
        mzero = multi_zero(self.rank)
        for part in parts:
            if len(part)>self.n:
                # only use up to n factors. XX add parameter to multi_partitions XX
                continue
            _r = self.scalar.one
            if len(part)<self.n:
                # make up the extra factors
                _r = self.b[ mzero ]**(self.n-len(part))
            last_idxs = None
            exponents = {}
            for idxs in part:
                exponents[idxs] = exponents.get(idxs,0) + 1
            n = self.n # how many to choose from
            for idxs, exponent in exponents.items():
                _r = n_choice(n, exponent) * _r * (self.b[idxs] ** exponent)
                n -= exponent # we chose this many
            r = r + _r
        return r

#class PowJet(JetOp):
#    def __init__(self, other, alpha):
#        JetOp.__init__(self, other.rank)
#        self.other = other
#        self.alpha = self.scalar.promote(alpha)
#    def __str__(self):
#        return "%s(%s,%s)"%(self.__class__.__name__,self.other,self.alpha)
#    def __repr__(self):
#        return "%s(%s,%s)"%(self.__class__.__name__,repr(self.b),repr(self.n))
#    def getitem(self, n):
#        raise NotImplementedError
#        zero_idxs = multi_zero(self.rank)
#        alpha = self.alpha
#        a, b = self, self.other
#        if n==zero_idxs:
#            return alpha * (b[zero_idxs]**(alpha-self.scalar.one))
#
#        return r

class UnaryJetOp(JetOp):
    def __init__(self, b):
        JetOp.__init__(self, b.rank)
        self.b = b
    def __str__(self):
        return "%s(%s)"%(self.__class__.__name__,self.b)
    def __repr__(self):
        return "%s(%s)"%(self.__class__.__name__,repr(self.b))

class ExpJet(UnaryJetOp):
    """
        a = exp(b)
    """
    def getitem(self, n):
        zero_idxs = multi_zero(self.rank)
        a, b = self, self.b
        if n==zero_idxs:
            return fmath.exp(b[n])
        i = 0
        while n[i]==0:
            i += 1
        r = MJet.bdot(a, b, n, i)
        return r

class LogJet(UnaryJetOp):
    def getitem(self, k):
        # h = log(u)
        zero_idxs = multi_zero(self.rank)
        h, u = self, self.b
        if k==zero_idxs:
            return fmath.log(u[k])
        i = 0
        while k[i]==0:
```

```
            i += 1
        r = self.scalar.zero
        for j in multi_range(multi_sub(k,multi_unit(self.rank,i))):
            if j != zero_idxs and k[i]>j[i]:
                r = r + (k[i]-j[i])*u[j]*h[multi_sub(k,j)]
        r = r/k[i]
        r = (u[k] - r) / u[zero_idxs]
        return r

class SqrtJet(UnaryJetOp):
    def getitem(self, k):
        zero_idxs = multi_zero(self.rank)
        h, u = self, self.b
        if k==zero_idxs:
            return fmath.sqrt(u[k])
        i = 0
        while k[i]==0:
            i += 1
        r = self.scalar.zero
        for j in multi_range(multi_sub(k,multi_unit(self.rank,i))):
            if j != zero_idxs and k[i]>j[i]:
                r = r + (k[i]-j[i])*h[j]*h[multi_sub(k,j)]
        r = r/k[i]
        r = (u[k]/(2*self.scalar.one) - r) / h[zero_idxs]
        return r

class SqrJet(UnaryJetOp):
    def getitem(self, k):
        zero_idxs = multi_zero(self.rank)
        a, b = self, self.b
        if k==zero_idxs:
            return fmath.sqr(b[k])
        i = 0
        while k[i]==0:
            i += 1
        r = self.scalar.zero
        ei = multi_unit(self.rank,i)
        ki = multi_sub(k,ei)
        for j in multi_range(ki):
            r = r + (j[i]+1.0) * b[ multi_sub(ki,j) ] * b[ multi_add(ei,j) ]
        return 2.0 * r / k[i]


class ATanJet(UnaryJetOp):
    def __init__(self, u):
        JetOp.__init__(self, u.rank)
        self.u = u
        self.v = self.scalar.one / (self.scalar.one + u*u)
    def __str__(self):
        return "%s(%s)"%(self.__class__.__name__,self.u)
    def __repr__(self):
        return "%s(%s)"%(self.__class__.__name__,repr(self.u))
    def getitem(self, k):
        zero_idxs = multi_zero(self.rank)
        h, u, v = self, self.u, self.v
        if k==zero_idxs:
            return fmath.atan(u[k])
        i = 0
        while k[i]==0:
            i += 1
        r = self.scalar.zero
        for j in multi_range(multi_sub(k,multi_unit(self.rank,i))):
            r = r + (k[i]-j[i])*v[j]*u[multi_sub(k,j)]
        r = r/k[i]
        return r

class ASinJet(UnaryJetOp):
    def __init__(self, u):
        JetOp.__init__(self, u.rank)
        self.u = u
        self.v = self.scalar.one / (self.scalar.one - u*u).sqrt()
    def __str__(self):
        return "%s(%s)"%(self.__class__.__name__,self.u)
    def __repr__(self):
        return "%s(%s)"%(self.__class__.__name__,repr(self.u))
    def getitem(self, k):
        zero_idxs = multi_zero(self.rank)
        h, u, v = self, self.u, self.v
        if k==zero_idxs:
            return fmath.asin(u[k])
```

```
        i = 0
        while k[i]==0:
            i += 1
        r = self.scalar.zero
        for j in multi_range(multi_sub(k,multi_unit(self.rank,i))):
            r = r + (k[i]-j[i])*v[j]*u[multi_sub(k,j)]
        r = r/k[i]
        return r

class ACosJet(UnaryJetOp):
    def __init__(self, u):
        JetOp.__init__(self, u.rank)
        self.u = u
        raise NotImplementedError
        self.v = self.scalar.one / (self.scalar.one - u*u).sqrt()
    def __str__(self):
        return "%s(%s)"%(self.__class__.__name__,self.u)
    def __repr__(self):
        return "%s(%s)"%(self.__class__.__name__,repr(self.u))
    def getitem(self, k):
        zero_idxs = multi_zero(self.rank)
        h, u, v = self, self.u, self.v
        if k==zero_idxs:
            return fmath.asin(u[k])
        i = 0
        while k[i]==0:
            i += 1
        r = self.scalar.zero
        for j in multi_range(multi_sub(k,multi_unit(self.rank,i))):
            r = r + (k[i]-j[i])*v[j]*u[multi_sub(k,j)]
        r = r/k[i]
        return r

class SinJet(UnaryJetOp):
    def __init__(self, u, cosu=None):
        JetOp.__init__(self, u.rank)
        self.u = u
        if cosu is None:
            cosu = CosJet(u,self)
        self.cosu = cosu
    def __str__(self):
        return "%s(%s)"%(self.__class__.__name__,self.u)
    def __repr__(self):
        return "%s(%s)"%(self.__class__.__name__,repr(self.u))
    def getitem(self, k):
        zero_idxs = multi_zero(self.rank)
        sinu, u, cosu = self, self.u, self.cosu
        if k==zero_idxs:
            return fmath.sin(u[k])
        i = 0
        while k[i]==0:
            i += 1
        r = self.scalar.zero
        for j in multi_range(multi_sub(k,multi_unit(self.rank,i))):
            r = r + (k[i]-j[i])*cosu[j]*u[multi_sub(k,j)]
        r = r/k[i]
        return r

class CosJet(UnaryJetOp):
    def __init__(self, u, sinu=None):
        JetOp.__init__(self, u.rank)
        self.u = u
        if sinu is None:
            sinu = SinJet(u,self)
        self.sinu = sinu
    def __str__(self):
        return "%s(%s)"%(self.__class__.__name__,self.u)
    def __repr__(self):
        return "%s(%s)"%(self.__class__.__name__,repr(self.u))
    def getitem(self, k):
        zero_idxs = multi_zero(self.rank)
        cosu, u, sinu = self, self.u, self.sinu
        if k==zero_idxs:
            return fmath.cos(u[k])
        i = 0
        while k[i]==0:
            i += 1
        r = self.scalar.zero
        for j in multi_range(multi_sub(k,multi_unit(self.rank,i))):
            r = r - (k[i]-j[i])*sinu[j]*u[multi_sub(k,j)]
```

```
        r = r/k[i]
        return r

class SliceJet(JetOp):
    """
        SliceJet(a,idxs): take a (lazy) slice out of 'a'
    """
    def __init__(self, a, idxs):
        assert type(idxs)==tuple
        # every slice gives us a rank
        rank = len([idx for idx in idxs if type(idx)==slice])
        JetOp.__init__(self, rank=rank)
        self.a = a
        self.full_idxs = idxs
        assert len(idxs)==a.rank # full index into a
        idx_map = {}
        i = 0
        for j, idx in enumerate(idxs):
            if type(idx)==slice:
                idx_map[i] = j
                i += 1
                assert idx==slice(None), " shift not implemented "
        self.idx_map = idx_map
    def getitem(self, idxs):
        full_idxs = list(self.full_idxs) # copy
        for i, idx in enumerate(idxs):
            full_idxs[self.idx_map[i]] = idx
        return self.a[tuple(full_idxs)]
    def __str__(self):
        return "%s(%s, rank=%d, idxs=%s)"%(self.__class__.__name__,self.a,self.rank,self.full_idxs)
    def __repr__(self):
        return "%s(%s, rank=%d, idxs=%s)"%(self.__class__.__name__,repr(self.a),self.rank,self.full_idxs)

class SprayItem(JetOp):
    " Created by Spray. These are the (MJet) components of a Spray object. "
    def __init__(self, spray, idx, yi):
        JetOp.__init__(self, rank=1) # yi.rank+1 ?
        self.spray = spray
        self.idx = idx
        self.yi = MJet(1).promote(yi)
    def getitem(self, order):
        order, = order
        if order==0:
            return self.yi[(0,)]
        dyi = self.spray.dy[self.idx]
        dyi = MJet(1).promote(dyi)
        r = (self.scalar.one/order) * dyi[order-1]
        return r

class Spray(object):
    """ Behaves like a list (vector) of MJet's
        Defined by an ODE: dy=f(x,y)
    """
    def __init__(self, f, x, y):
        self.f = f
        self.x = x
        self.y = [SprayItem(self,idx,yi) for idx,yi in enumerate(y)]
        self.dy = f(x, self.y) # a list of MJet's
    def __getitem__(self, idx):
        return self.y[idx]

class Jet(MJet):
    """
        This is a concrete Jet. All components are stored in a dictionary.
    """
    def __init__(self, cs={}, rank=None):
        if type(cs) in (list,tuple):
            if cs and type(cs[0]) in (list,tuple):
                raise NotImplementedError
            cs = dict(((i,),c) for i,c in enumerate(cs))
        if type(cs)!=dict:
            raise ValueError, "expected dict, list or tuple, but got %s"%cs
        if rank is None and not cs:
            raise ValueError, "can't determine rank"
        elif rank is None:
            rank = len(cs.keys()[0])
        for key in cs.keys():
            assert len(key)==rank
        MJet.__init__(self, rank)
        self.bound = [0]*rank
```

```
        self.cs = {} # coefficients, components, ...
        for idxs, value in cs.items():
            self[idxs] = self.scalar.promote(value)

    def __str__(self):
        ss = []
        for idxs in xcross([slice(0,n+1) for n in self.bound]):
            ss.append('%s:%s' % (''.join([str(idx) for idx in idxs]), self[idxs]))
        return '<%s>'%' '.join(ss)

    def __repr__(self):
        ss = []
        for idxs in xcross([slice(0,n+1) for n in self.bound]):
            ss.append('%s:%s' % (''.join([str(idx) for idx in idxs]), repr(self[idxs])))
        return '<%s>'%' '.join(ss)

    def __getitem__(self, idxs):
        if type(idxs)==int:
            idxs=idxs,
        assert type(idxs)==tuple
        assert len(idxs)==self.rank
        # every slice gives us a rank
        rank = len([idx for idx in idxs if type(idx)==slice])
        if rank:
            # Non-scalar return type
            idx_map = {}
            idxs = list(idxs)
            slice_i = []
            for i, idx in enumerate(idxs):
                if type(idx)==slice:
                    slice_i.append(i)
                    assert idx==slice(None), "shift not implemented"
                    idx = slice(0,self.bound[i]+1,None)
                    idxs[i] = idx
            x = Jet(rank = rank)
            for full_idxs in xcross(idxs):
                idx = [full_idxs[i] for i in slice_i]
                x[tuple(idx)] = self[full_idxs]
            return x
        return self.cs.get(idxs, self.scalar.zero)

    def __setitem__(self, idxs, c):
        if type(idxs)==int:
            idxs=idxs,
        assert type(idxs)==tuple
        assert len(idxs)==self.rank
        assert isinstance(c, self.scalar.type), "attempt to set item %s to non-scalar %s" % (idxs, c)
        self.cs[idxs] = c
        for i, idx in enumerate(idxs):
            self.bound[i] = max(idx, self.bound[i])

    def __cmp__(self, other):
        assert type(other)==Jet, "Not implemented"
        for idxs in self.cs.keys()+other.cs.keys():
            r = cmp(self.cs.get(idxs,0.0), other.cs.get(idxs,0.0))
            if r != 0:
                return -1
        return 0

class Derivation(object):
    """
        given f:R^n->R
        Derivation(f):R^n->R^n
            a list of the partial derivative's of f
            extends to arbitrary Jet arguments
    """

    scalar = None

    def __init__(self, f):
        self.f = f

    def __call__(self, *y):
        f = self.f
        dim = len(y)
        assert dim
        if not isinstance(y[0],MJet):
            # must be scalar .. ie. a rank-0 Jet
            y = [ MJet(0).promote(_y) for _y in y ]
        # now we need dim extra ranks
```

```
        rank0 = y[0].rank
        rank = rank0 + dim
        yy = []
        for i,_y in enumerate(y):
            _y = MJet(rank).promote(_y)
            idxs = [0]*rank
            idxs[rank0 + i] = 1
            _y[ tuple(idxs) ] = self.scalar.one
            yy.append(_y)

        ############
        g = f(*yy) #
        ############

        g = MJet(rank).promote(g)
        # now we drop back to rank0
        gs = []
        for i in range(dim):
            idxs = [0]*dim
            idxs[i] = 1
            idxs = tuple([slice(None)]*rank0+idxs)
            g0 = g[idxs]
            g0 = MJet(rank0).promote(g0)
            gs.append(g0)
        return gs

Scalar.add_client(Derivation)

#
################################################################################
#
```

B.6 ode.py

```
from gmpy import mpf

from pydx.mjet import MJet, Jet, Spray
from pydx.scalar import Scalar, set_symbolic_scalar, set_interval_scalar, restore_scalar
from pydx.scalar.symbolic import Var, ComputationContext
from pydx.scalar.mpfi import Interval, DisorderException
from pydx.tensor import Tensor
from pydx.field import TensorField
from pydx.db import Snapshot

#
################################################################################
#


class Info(object):
    def __init__(self,**kw):
        self.__dict__.update(globals())
        self.__dict__.update(kw)
    def __getitem__(self, name):
        return self.__dict__[name]

class ODE(object):

    def __init__(self, x0, y0, paramname='t', varnames=None):
        # XX use get/setattr trickery to hide/expose the param/var attrs XX#
        self.x = x0
        self.y = y0
        ######################################################################
        self.dim = len(self.y)
        if varnames is None:
            varnames = ['x%d'%i for i in range(self.dim)]
        self.varnames = varnames
        self.paramname = paramname
        self.i = 0 # Step number
        self.step_funcs = {} # map order -> func

    def dumpstate(self, file):
        file.write('i = %s\n' % self.i)
        file.write('%s = %r\n' % (self.paramname, self.x))
        for idx, name in enumerate(self.varnames):
            file.write('%s = %r\n' % (name, self.y[idx]))
```

```
    file.flush()

def commit(self, session, **info):
    info['i'] = self.i
    info[self.paramname] = self.x
    for idx, name in enumerate(self.varnames):
        info[name] = self.y[idx]
    session.snapshot(info)

def restore(self, session):
    namespace = sys.modules[__name__]
    info = session.restore(namespace)
    self.i = info['i']
    self.x = info[self.paramname]
    for idx, name in enumerate(self.varnames):
        self.y[idx] = info[name]
    return info

def loadstate(self, file):
    info = Info(i=self.i)
    while self.i==info.i:
        # read lines until the counter increments
        line = file.readline().strip()
        if not line:
            break
        exec line in info.__dict__
    try:
        y = self.y[:]
        x = info[self.paramname]
        for idx, name in enumerate(self.varnames):
            y[idx] = info[name]
        i = info.i
    except Exception, e:
        raise
    self.y[:] = y
    self.x = x
    self.i = i

def __call__(self, x, y):
    """
        ODE(x,y)
            @param x : parameter (scalar)
            @param y : vector

        y'(x) = f(x,y)
    """
    pass

def step1(self, h):
    h = self.scalar_promote(h)
    v = self.y
    t = self.x
    dv = self(t, v)
    for i, dvi in enumerate(dv):
        if isinstance(dvi,MJet):
            dv[i] = dvi[()]
    v = [ v[i] + h*dv[i] for i in range(self.dim) ]
    t = t + h
    self.y = v
    self.x = t
    self.i += 1

def step2(self, h):
    h = self.scalar_promote(h)
    v = self.y
    t = self.x
    dv = self(t, v)
    for i, dvi in enumerate(dv):
        if isinstance(dvi,MJet):
            dv[i] = dvi[()]
    v = [ Jet([v[i],dv[i]]) for i in range(self.dim) ]
    ddv = self(Jet([t, self.scalar_one]), v)
    ddv = [ (self.scalar_one/2) * ddvi[1] for ddvi in ddv ]
    hh = h*h
    v = [ v[i] + h*dv[i] + hh*ddv[i] for i in range(self.dim) ]
    t = t + h
    self.y = v
    self.x = t
    self.i += 1
```

```
def istep1(self, h, yy):
    """
        First order interval step.
        @param yy: bound on solution (got from contract1)
    """
    f = self
    h = self.scalar_promote(h)
    v = self.y
    t = self.scalar_promote(self.x)
    dv = f(t, yy)
    for i, dvi in enumerate(dv):
        if isinstance(dvi,MJet):
            dv[i] = dvi[()]
    v = [ v[i] + h*dv[i] for i in range(self.dim) ]
    t = t + h
    self.y = v
    self.x = t
    self.i += 1

def istep2(self, h, y):
    """
        Second order interval step.
        @param y: bound on solution (got from contract1)
    """
    f = self
    h = self.scalar_promote(h)

    x0 = self.x
    y0 = self.y

    dy0 = f(x0, y0)
    dy0 = [ MJet(0).promote(dy0i)[()] for dy0i in dy0 ]

    x = x0.hull(x0+h)
    dy = f(x, y)
    dy = [ MJet(0).promote(dyi)[()] for dyi in dy ]

    y_jet = [ Jet([y[i],dy[i]]) for i in range(self.dim) ]
    x_jet = Jet([x, self.scalar_one])

    ddy = f(x_jet, y_jet)
    ddy = [ (self.scalar_one/2) * ddyi[1] for ddyi in ddy ]

    hh = h**2
    y = [ y0[i] + h*dy0[i] + hh*ddy[i] for i in range(self.dim) ]
    x1 = x0 + h.lower

    self.y = y
    self.x = x1
    self.i += 1

def _compile_spray(self, order):
    f = self
    dumpfile = open('stepdump.py','w')
    set_symbolic_scalar()
    x = Var('x')
    y = [ Var('y%d'%i) for i in range(self.dim) ]
    vs = [ x ] + y
    name = 'spray'
    args = vs + [ 'rval' ]
    ctx = ComputationContext(name, args, dumpfile)
    rval = [None]*self.dim
    lazy = Spray(f, x, y)
    for i in range(order):
        ctx.assign('rval[%d]'%i, lazy[i])
    func = ctx.finalize()
    restore_scalar()
    return func

def get_spray(self, order):
    if order in self.spray_funcs:
        return self.spray_funcs[order]
    func = self._compile_spray(order)
    self.spray_funcs[order]=func
    return self.spray_funcs[order]

def compile(self,n):
    assert n not in self.step_funcs
    dumpfile = open('stepdump.py','w')
    set_symbolic_scalar()
```

```
    x0 = Var('x0')
    y0 = [ Var('y0_%d'%i) for i in range(self.dim) ]
    h = Var('h')
    x = Var('x')
    y = [ Var('y_%d'%i) for i in range(self.dim) ]
    vs = [x0]+y0+[h,x]+y
    name = 'spray'
    args = vs + [ 'y1' ]
    print "compile n=",n,"args:",args
    f = self

    ################# START COMPUTATION
    print "computing..."
    ctx = ComputationContext(name, args, dumpfile)
    y0 = Spray(f, x0, y0) # a list of MJet's
    y = Spray(f, x, y) # a list of MJet's
    y1 = [ y0i.expand_err((h,), n-1, (yi[n],)) for y0i,yi in zip(y0,y) ]
    for i in range(self.dim):
        ctx.assign('y1[%d]'%i, y1[i])
    print "finalize..."
    func = ctx.finalize()
    ################# FINISH COMPUTATION

    restore_scalar()
    return func

def istepn(self, h, y, n=2):
    """
        Arbitrary order interval step.
        @param h: step size
        @param y: bound on solution (got from contract1)
    """
    func = self.step_funcs.get(n)
    if func is None:
        func = self.compile(n)
        self.step_funcs[n] = func
    x0 = self.x
    y0 = self.y
    h = self.scalar_promote(h)
    x = x0.hull(x0+h)
    y1 = [None]*self.dim
    args = [x0]+y0+[h,x]+[yi for yi in y]+[y1]
    func(*args)
    x1 = x0 + h

    self.y = y1
    self.x = x1
    self.i += 1

def _istepn(self, h, y, n=2):
    """
        Arbitrary order interval step.
        @param h: step size
        @param y: bound on solution (got from contract1)
    """
    f = self
    h = self.scalar_promote(h)

    x0 = self.x
    y0 = self.y
    x = x0.hull(x0+h)

    y0 = Spray(f, x0, y0) # a list of MJet's
    y = Spray(f, x, y) # a list of MJet's

    x1 = x0 + h
    y1 = [ y0i.expand_err((h,), n-1, (yi[n],)) for y0i,yi in zip(y0,y) ]

    self.y = y1
    self.x = x1
    self.i += 1

def contract1(self, x, y0, max_iter=100):
    """
    """
    set_interval_scalar()
    f = self
    hh = Interval(0.0, x.width())
    _y0 = Tensor((Tensor.up,), len(y0))
    for i in range(len(_y0)):
```

```
            _y0[i] = ODE.scalar_promote(y0[i])
        _y = y = y0 = _y0

        cookies = 1 # makes no difference...

        i = 0
        while 1:
            y = _y
            try:
                _y = y0 + hh * f(x,y) # Contract
            except DisorderException:
                raise
            shrink = True
            for i in range(self.dim):
                _y[i] = MJet(0).promote(_y[i])[()]
                if not y[i].contains(_y[i]):
                    shrink = False
            if shrink:
                cookies -= 1
            if cookies <= 0:
                break
            i += 1
            if i > max_iter:
                return None
        restore_scalar()
        return y

Scalar.add_client(ODE)
```

B.7 tensor.py

```
from pydx import scalar
from pydx.mjet import MJet, cross

up = 1; dn = -1;

def genidx(shape):
    if len(shape)==0:
        yield ()
    else:
        for idx in range(shape[0]):
            for _idx in genidx(shape[1:]):
                yield (idx,)+_idx

class Tensor(object):
    """
        NB: not necessarily with the correct transformative properties.
    """
    metric = None # class attr ???
    scalar = None
    up = 1; dn = -1;

    def __init__(self, valence = (), dim = 4, elems=None, diag=None):
        # for now, all Tensors are "square"
        self.valence = tuple(valence) # sequence of up/dn
        for updn in self.valence:
            assert updn in (self.up,self.dn), updn
        self.rank = len(valence)
        self.shape = (dim,)*self.rank
        self.dim = dim
        self.elems = {} # map tuple to value
        # make it dense:
        if elems is None:
            for idxs in self.genidx():
                self.elems[idxs] = self.scalar.zero
        elif type(elems)==dict:
            self.elems = dict(elems)
        else:
            for idxs in self.genidx():
                elem = elems
                for idx in idxs:
                    try:
                        elem=elem[idx]
                    except:
                        raise ValueError, "bad elems"
                self.elems[idxs] = elem
```

```
    if diag is not None:
        assert len(diag) == self.dim
        for i in range(len(diag)):
            self[ (i,)*self.rank ] = diag[i]
    self.v_cache = {}
    self.v_cache[ self.valence ] = self

def identity(cls, valence = (), dim = 4):
    tensor = cls(valence, dim)
    for idxs in tensor.genidx():
        on_diag = True
        if idxs:
            i = idxs[0]
        for j in idxs:
            if j!=i:
                on_diag = False
        if on_diag:
            tensor[idxs] = cls.scalar.one
        else:
            tensor[idxs] = cls.scalar.zero
    return tensor
identity = classmethod(identity)

def zero(self):
    tensor = self.__class__(self.valence, dim=self.dim)
    return tensor

def get_valence(self, v):
    """
        raise or lower indices (get a new valence).
    """
    for updn in v:
        assert updn in (self.up,self.dn), "bad valence: %s"%updn
    assert len(v)==self.rank
    return self.v_cache[v]

def __getitem__(self, idxs):
    if type(idxs)==int:
        idxs = idxs,
    if type(idxs)==slice:
        start, stop, step = idxs.indices(len(self))
        idxs = range(start,stop,step)
        tensor = Tensor(self.valence, len(idxs))
        for idx in idxs:
            tensor[idx-start] = self[idx]
        return tensor
    else:
        assert type(idxs)==tuple, "what's this: %s"%repr(idxs)
        assert len(idxs)<=self.rank
        assert len(idxs)==self.rank, "Not implemented"
        value = self.elems.get(idxs, None)
        if value is None:
            raise IndexError, idxs
        return value

def __len__(self):
    return self.dim

def __setitem__(self, idxs, val):
    assert val is not None
    if type(idxs)==int:
        idxs = idxs,
    assert type(idxs)==tuple
    free_rank = self.rank-len(idxs)
    assert free_rank >= 0, "set valence %s at %s "%(self.valence, idxs)
    if free_rank == 0:
        self.elems[idxs] = val
    elif free_rank == 1:
        assert len(val)==self.dim
        for i in range(self.dim):
            self.elems[idxs+(i,)] = val[i]
    else:
        for _idxs in cross((self.dim)*free_rank):
            self.elems[idxs+_idxs] = val[_idxs]

def scalar(self):
    tensor = Tensor(self.valence, self.dim)
    for idxs in self.genidx():
        r = self[idxs]
        r = MJet(0).promote(r)[()]
```

```
            tensor[idxs] = r
        return tensor

    def is_close(self, other, epsilon=1e-10):
        assert self.valence==other.valence
        assert self.dim==other.dim
        for idxs in self.genidx():
            r = self[idxs]-other[idxs]
            r = MJet(0).promote(r)[()]
            if abs(r)>epsilon:
                return False
        return True

    def idxstr(self, idxs, onebased=0):
        _v = None
        components = []
        for i, v in enumerate(self.valence):
            s = str(idxs[i]+onebased)
            if _v is not v:
                s = {up:"^",dn:"_"}[v]+s
                _v = v
            components.append(s)
        return ''.join(components)

    def str(self, name=""):
        lines = []
        for idxs in self.genidx():
            lines.append("%s%s %s" % (name, self.idxstr(idxs), str(self[idxs])))
        return '\n'.join(lines)

    def __str__(self):
        if self.rank==0:
            return "(%s)"%(self[()])
        ss = []
        for idxs in cross((self.dim,)*(self.rank-1)):
            ss.append(' '.join([ str(self[idxs+(i,)]) for i in range(self.dim) ]))
        return '\n'.join(ss)
    __repr__ = __str__

    def clone(self):
        " deepcopy "
        tensor = self.__class__(self.valence, self.dim)
        keys = self.keys()
        for i, value in enumerate(self.values()):
            tensor[keys[i]] = value
        return tensor

    def __contains__(self, key):
        return key in self.elems

    def keys(self):
        keys = self.elems.keys()
        keys.sort()
        return keys

    def genidx(self):
        " all possible keys "
        if len(self.shape)==0:
            yield ()
        else:
            for idx in range(self.shape[0]):
                for _idx in genidx(self.shape[1:]):
                    yield (idx,)+_idx

    def __neg__(self):
        tensor = self.zero()
        for idx in self.genidx():
            tensor[idx] = -self[idx]
        return tensor

    def __add__(self, other):
        tensor = self.zero()
        for idx in self.genidx():
            tensor[idx] = self[idx]+other[idx]
        return tensor

    def __iadd__(self, other):
        for idx in self.genidx():
            self[idx] += other[idx]
        return self
```

```
def __sub__(self, other):
    tensor = self.zero()
    for idx in self.genidx():
        tensor[idx] = self[idx]-other[idx]
    return tensor

def __isub__(self, other):
    for idx in self.genidx():
        self[idx] -= other[idx]
    return self

def __rmul__(self, other):
    tensor = self.zero()
    for idx in self.genidx():
        tensor[idx] = other * self[idx]
    return tensor

def __radd__(self, other):
    tensor = self.zero()
    for idx in self.genidx():
        tensor[idx] = other + self[idx]
    return tensor

def contract(self, *pairs):
    " pairs: sequence of self index, self index "
    for i, j in pairs:
        assert self.valence[i] != self.valence[j] # pair up with dn, and vice-versa
        assert i not in [j for i,j in pairs]
        assert j not in [i for i,j in pairs]
    valence = []
    left = [i for i,j in pairs]
    right = [j for i,j in pairs]
    contracted = left+right # the contracted indexs
    for idx, v in enumerate(self.valence):
        if idx not in contracted:
            # we are not contracting this idx
            valence.append(v)
    tensor = self.__class__(valence, self.dim)
    assert (self.rank - tensor.rank) % 2 == 0, "internal error"
    kernel = (self.dim,) * ((self.rank - tensor.rank)/2)
    assert len(kernel) == len(pairs)
    for tgt_idx in tensor.genidx():
        res = self.scalar.zero
        for kernel_idx in genidx(kernel):
            # weave _idx into tgt_idx to get src_idx
            src_idx = []
            i = 0 # index into tgt_idx
            for srci in range(self.rank):
                if srci in left:
                    kernel_i = left.index(srci)
                    src_idx.append(kernel_idx[kernel_i])
                elif srci in right:
                    kernel_i = right.index(srci)
                    src_idx.append(kernel_idx[kernel_i])
                else:
                    src_idx.append(tgt_idx[i])
                    i += 1
            assert i == len(tgt_idx)
            assert len(src_idx) == self.rank
            src_idx = tuple(src_idx)
            if src_idx in self:
                res = res + self[src_idx]
        tensor[ tgt_idx ] = res
    return tensor

def outer(self, other):
    assert self.dim == other.dim
    valence = self.valence + other.valence
    tensor = self.__class__(valence, self.dim)
    for idx in self.genidx():
        for _idx in other.genidx():
            tensor[ idx + _idx ] = self[idx]*other[_idx]
    return tensor
__mul__ = outer

def mul(self, other, *pairs):
    " pairs: sequence of self index, other index "
    for i, j in pairs:
        # pair up with dn, and vice-versa
```

```
            assert self.valence[i] != other.valence[j], (self.valence, i, other.valence, j)
        assert self.dim == other.dim
        tensor = self.outer(other) # big
        pairs = [ (i, j+self.rank) for i,j in pairs ]
        tensor = tensor.contract(*pairs)
        return tensor

    def transpose(self, *axes):
        assert len(axes)==self.rank
        valence = [self.valence[i] for i in axes]
        tensor = Tensor(valence, self.dim)
        for idx in self.genidx():
            _idx = tuple([idx[axes[i]] for i in range(self.rank)])
            tensor[idx] = self[_idx] #.clone()
        return tensor

    def transform(self, transform=None, inverse=None):
        tensor = self
        for i, updn in enumerate(self.valence):
            if updn == Tensor.up:
                tensor = tensor.mul(inverse, (i,1))
            elif updn == Tensor.dn:
                tensor = tensor.mul(transform, (i,0))
            else:
                assert 0, "bad valence"
        return tensor

    def apply(self, func):
        for idx in self.genidx():
            self[idx] = func(self[idx])
        return self

    def demote(self):
        "send all our components back to scalar"
        self.apply(lambda a:MJet(0).promote(a)[()])

scalar.Scalar.add_client(Tensor)

def Scalar(value):
    tensor = Tensor()
    tensor[()] = value
    return tensor
```

B.8 field.py

```
from pydx import scalar
from pydx.tensor import Tensor, up, dn
from pydx.mjet import MJet, cross
from pydx.scalar import set_symbolic_scalar, restore_scalar
from pydx.scalar.symbolic import Var, ComputationContext

class TensorField(object):
    """
        Instances are callable, returning a Tensor object.
        Abstract base class.
    """
    scalar_zero = 0.0
    scalar_one = 1.0
    scalar_promote = float
    scalar_type = float
    up = 1; dn = -1;

    def __init__(self, valence = (), dim = 4, g=None):
        """
            @param g: metric
        """
        # for now, all Tensors are "square"
        self.valence = tuple(valence) # sequence of up/dn
        for updn in self.valence:
            assert updn in (self.up,self.dn), updn
        self.rank = len(valence)
        self.shape = (dim,)*self.rank
        self.dim = dim
        assert g is None or isinstance(g, TensorField)
        self.g = g
        if g is not None and 'uu' not in g.__dict__:
```

```
            g.uu = None
        valence_attr = ''.join([{Tensor.up:'u',Tensor.dn:'d'}[v] for v in valence])
        self.__dict__[valence_attr] = self

    def __str__(self):
        return "%s(%s, %s)"%(self.__class__.__name__, self.valence, self.dim)
    __repr__ = __str__

    def __call__(self):
        raise NotImplementedError, "abstract base class"

    def identity(cls, valence, dim):
        tfield = ConcreteTensorField(valence, dim)
        one_func = lambda *xs: cls.scalar_one
        zero_func = lambda *xs: cls.scalar_zero
        for idxs in tfield.genidx():
            on_diag = True
            if idxs:
                i = idxs[0]
            for j in idxs:
                if j!=i:
                    on_diag = False
            if on_diag:
                tfield[idxs] = one_func
            else:
                tfield[idxs] = zero_func
        return tfield
    identity = classmethod(identity)

    def zero(cls, valence, dim):
        tfield = ConcreteTensorField(valence, dim)
        zero_func = lambda *xs: cls.scalar_zero
        for idxs in tfield.genidx():
            tfield[idxs] = zero_func
        return tfield
    zero = classmethod(zero)

    def genidx(self):
        return cross((self.dim,)*self.rank)

    def __getattr__(self, valence_s):
        " For example: riemann.dddd is the fully contravariant riemann "
        # this is perhaps too tricky..
        n = valence_s.count('u')+valence_s.count('d')
        if not n==len(valence_s)==len(self.valence):
            raise AttributeError, valence_s
        assert self.g is not None
        assert self.g.uu is not None
        t = self
        for v,idx in enumerate(valence_s):
            if v=='u' and t.valence[idx]==Tensor.dn:
                t = t.mul(self.g.uu, (idx,0))
            elif v=='d' and t.valence[idx]==Tensor.up:
                t = t.mul(self.g, (idx,0))
        return t

    def view(self, trans):
        return ViewTensorField(self, trans)

    def transform(self, trans):
        return TransformTensorField(self, trans)

    def comma(self):
        return CommaTensorField(self)

    def transpose(self, *perm):
        return TransposeTensorField(self, perm)

    def __pos__(self):
        return self

    def __neg__(self):
        return NegTensorField(self)

    def __add__(self, other):
        return AddTensorField(self, other)

#    def __radd__(self, other):
#        other = self.promote(other)
#        return AddTensorField(other, self)
```

```
    def __sub__(self, other):
        return SubTensorField(self, other)

#    def __rsub__(self, other):
#        other = self.promote(other)
#        return Sub(other, self)

#    def __mul__(self, other):
#        return ScalarMulTensorField(self, other)

    def __rmul__(self, other):
        return ScalarMulTensorField(other, self)

    def mul(self, other, *pairs):
        return MulTensorField(self, other, pairs)
    __mul__ = mul # outer product

    def contract(self, *pairs):
        return ContractTensorField(self, pairs)

    def compile_oldversion(self):
        "the older version of the compile method. slower, but slightly more accurate (why?!?) "
        Float.scalar_zero = Interval(0.0)
        Float.scalar_one = Interval(1.0)
        Float.scalar_promote = Interval
        Float.scalar_type = Interval
        compiledump = open('compiledump.py','w')
        set_symbolic_scalar()
        tfield = ConcreteTensorField(self.valence, self.dim)
        vs = [ Var('x%d'%i) for i in range(self.dim) ]
        tensor = self(*vs)
        for idxs in self.genidx():
            r = tensor[idxs]
            #if not isinstance(r,MJet):
            #  r = MJet(0).promote(r)
            r = r[()]
            #if not isinstance(r,Float):
            #  r = Float.promote(r)
            name = 'func' + ''.join(str(i) for i in idxs)
            func = r.get_func(name, vs, compiledump)
            tfield[idxs] = func
        restore_scalar()
        Float.scalar_zero = 0.0
        Float.scalar_one = 1.0
        Float.scalar_promote = float
        Float.scalar_type = float
        return tfield

    def compile(self):
        compiledump = open('compiledump.py','w')
        set_symbolic_scalar()
        vs = [ Var('x%d'%i) for i in range(self.dim) ]
        name = 'func'
        args = vs + [ 'tensor' ]
        ctx = ComputationContext(name, args, compiledump)
        tensor = self(*vs)
        for idxs in self.genidx():
            r = tensor[idxs]
            #if not isinstance(r,MJet):
            #  r = MJet(0).promote(r)
            r = r[()]
            #if not isinstance(r,Float):
            #  r = Float.promote(r)
            ctx.assign('tensor[%s]'%repr(idxs), r)
        func = ctx.finalize()
        restore_scalar()
        tfield = AdaptedTensorField(self.valence, self.dim, self.g, func)
        return tfield
scalar.Scalar.add_client(TensorField)

class AdaptedTensorField(TensorField):
    """ Used with ComputationContext. Creates tensors via ONE function call.
    """
    def __init__(self, valence = (), dim = 4, g=None, func=None):
        TensorField.__init__(self, valence, dim, g=g)
        self.func = func

    def __call__(self, *xs):
        tensor = Tensor(self.valence, self.dim)
```

```
        args = list(xs)+[tensor]
        self.func(*args)
        return tensor

class ConcreteTensorField(TensorField):
    """ Components are actual (scalar valued) functions
    """
    def __init__(self, valence = (), dim = 4, g=None, elems={}):
        TensorField.__init__(self, valence, dim, g=g)
        self.elems = dict(elems) # map tuple to value
        #for idxs in cross((self.dim,)*self.rank):
        #  self.elems[idxs] = lambda *xs: return scalar_zero
        self.symetry = {}

    def addsymetry(self, src_idxs, tgt_idxs):
        "when src_idxs are requesed, we look-up tgt_idxs"
        assert tgt_idxs not in self.symetry, " chain a symetry ?"
        while tgt_idxs in self.symetry:
            tgt_idxs = self.symetry[tgt_idxs]
        self.symetry[src_idxs] = tgt_idxs

    def __setitem__(self, idxs, func):
        assert callable(func)
        if type(idxs)==int:
            idxs = idxs,
        assert idxs not in self.symetry, "symetric component!"
        self.elems[idxs] = func

    def __getitem__(self, idxs):
        if type(idxs)==int:
            idxs = idxs,
        assert idxs not in self.symetry, "symetric component!"
        func = self.elems[idxs]
        return func

    def __call__(self, *xs):
        assert len(xs)==self.dim
        elems = {}
        for idxs in self.genidx():
            if idxs in self.symetry: # is this any use ??
                tgt_idxs = self.symetry[idxs]
                assert tgt_idxs not in self.symetry
                if tgt_idxs not in elems:
                    elems[tgt_idxs] = self[tgt_idxs](*xs)
                elems[idxs] = elems[tgt_idxs]
            else:
                elems[idxs] = self[idxs](*xs)
        tensor = Tensor(self.valence, self.dim, elems=elems)
        return tensor

class NegTensorField(TensorField):
    def __init__(self, a, g=None):
        TensorField.__init__(self, a.valence, a.dim, g=g)
        self.a = a

    def __call__(self, *xs):
        assert len(xs)==self.dim
        return -self.a(*xs)

class AddTensorField(TensorField):
    def __init__(self, a, b, g=None):
        assert a.valence == b.valence
        assert a.dim == b.dim
        TensorField.__init__(self, a.valence, a.dim, g=g)
        self.a = a
        self.b = b

    def __call__(self, *xs):
        assert len(xs)==self.dim
        return self.a(*xs) + self.b(*xs)

class SubTensorField(TensorField):
    def __init__(self, a, b, g=None):
        assert a.valence == b.valence
        assert a.dim == b.dim
        TensorField.__init__(self, a.valence, a.dim, g=g)
        self.a = a
        self.b = b

    def __call__(self, *xs):
```

```
        assert len(xs)==self.dim
        return self.a(*xs) - self.b(*xs)

class ScalarMulTensorField(TensorField):
    def __init__(self, a, b, g=None):
        TensorField.__init__(self, b.valence, b.dim, g=g)
        self.a = a
        self.b = b

    def __call__(self, *xs):
        assert len(xs)==self.dim
        return self.a * self.b(*xs)

class MulTensorField(TensorField):
    def __init__(self, a, b, pairs, g=None):
        assert a.dim == b.dim
        for i, j in pairs:
            assert a.valence[i] != b.valence[j] # pair up with dn, and vice-versa
        ai = [ aii for aii,_ in pairs ]
        bi = [ bii for _,bii in pairs ]
        valence = []
        for i,v in enumerate(a.valence):
            if i not in ai:
                valence.append(v)
        for i,v in enumerate(b.valence):
            if i not in bi:
                valence.append(v)
        TensorField.__init__(self, tuple(valence), a.dim, g=g)
        self.a = a
        self.b = b
        self.pairs = pairs

    def __call__(self, *xs):
        assert len(xs)==self.dim
        t = self.a(*xs).mul(self.b(*xs), *self.pairs)
        assert t.valence == self.valence
        return t

class ContractTensorField(TensorField):
    def __init__(self, a, pairs, g=None):
        ii = [ aii for aii,_ in pairs ] + [ bii for _,bii in pairs ]
        valence = []
        for i,v in enumerate(a.valence):
            if i not in ii:
                valence.append(v)
        TensorField.__init__(self, tuple(valence), a.dim, g=g)
        self.a = a
        self.pairs = pairs

    def __call__(self, *xs):
        assert len(xs)==self.dim
        return self.a(*xs).contract(*self.pairs)

class TransposeTensorField(TensorField):
    def __init__(self, tfield, perm, g=None):
        self.tfield = tfield
        self.perm = perm
        valence = tuple(tfield.valence[i] for i in perm)
        TensorField.__init__(self, valence, tfield.dim, g=g)

    def __call__(self, *xs):
        t = self.tfield(*xs).transpose(*self.perm)
        assert t.valence == self.valence, (t.valence, self.valence)
        return t

class TransformTensorField(TensorField):
    def __init__(self, tfield, coord_transform, g=None):
        TensorField.__init__(self, tfield.valence, tfield.dim, g=g)
        self.tfield = tfield
        self.coord_transform = coord_transform

    def __call__(self, *xs):
        assert len(xs)==self.dim
        _xs = self.coord_transform(*xs)
        tensor = self.tfield(*_xs)
        if Tensor.up in self.valence:
            inverse_partial = self.coord_transform.inverse.partial(*_xs)
        if Tensor.dn in self.valence:
            partial = self.coord_transform.partial(*xs)
        for i, updn in enumerate(self.valence):
```

```
            if updn == Tensor.up:
                tensor = tensor.mul(inverse_partial, (i,1))
                idxs = range(self.rank)
                idxs = idxs[:i]+[idxs[-1]]+idxs[i:-1]
                tensor = tensor.transpose(*idxs)
                assert self.valence==tensor.valence
            elif updn == Tensor.dn:
                tensor = tensor.mul(partial, (i,0))
                idxs = range(self.rank)
                idxs = idxs[:i]+[idxs[-1]]+idxs[i:-1]
                tensor = tensor.transpose(*idxs)
                assert self.valence==tensor.valence
            else:
                assert 0, "bad valence"
        return tensor

class ViewTensorField(TensorField):
    def __init__(self, tfield, coord_transform, g=None):
        TensorField.__init__(self, tfield.valence, tfield.dim, g=g)
        self.tfield = tfield
        self.coord_transform = coord_transform

    def __call__(self, *xs):
        assert len(xs)==self.dim
        xs = self.coord_transform(*xs)
        tensor = self.tfield(*xs)
        return tensor

class CommaTensorField(TensorField):
    def __init__(self, tfield, g=None):
        valence = tfield.valence + (dn,)
        TensorField.__init__(self, valence, tfield.dim, g=g)
        self.tfield = tfield

    def __call__(self, *xs):
        dim = len(xs)
        assert dim==self.dim
        if not isinstance(xs[0],MJet):
            # must be scalar .. ie. a rank-0 Jet
            xs = [MJet(0).promote(_x) for _x in xs]
        # now we need dim extra ranks
        rank0 = xs[0].rank
        rank = rank0 + dim
        xx = []
        for i,_x in enumerate(xs):
            _x = MJet(rank).promote(_x)
            idxs = [0]*rank
            idxs[rank0 + i] = 1
            _x[tuple(idxs)] = self.scalar_one
            xx.append(_x)

        tensor = self.tfield(*xx)

        comma = Tensor(self.valence, self.dim)

        for _idxs in tensor.genidx():
            g = tensor[_idxs]
            g = MJet(rank).promote(g)
            # now we drop back to rank0
            gs = []
            for i in range(dim):
                idxs = [0]*dim
                idxs[i] = 1
                idxs = tuple([slice(None)]*rank0+idxs)
                g0 = g[idxs]
                g0 = MJet(rank0).promote(g0)
                gs.append(g0)
            comma[_idxs] = gs
        return comma


class ScalarField(ConcreteTensorField):
    def __init__(self, dim, fn, g=None):
        ConcreteTensorField.__init__(self, (), dim, g, {():fn})

class Transform(ConcreteTensorField):
    """Coordinate Transform: a 'vector' of ScalarFields """
    def __init__(self, fns, inverse=None, g=None):
        dim = len(fns)
        self.dim = dim
```

```
        self.inverse = inverse
        if inverse is not None:
            assert dim == inverse.dim
            self.inverse.inverse = self
        elems = [ ((i,),fn) for i,fn in enumerate(fns) ]
        ConcreteTensorField.__init__(self, (Tensor.up,), self.dim, g, elems)
        self.partial = self.comma()

class IdentityTransform(Transform):
    def __init__(self, dim):
        Transform.__init__([lambda x:x]*dim, self)
```

B.9 metric.py

```
from pydx.mjet import MJet, Jet
from pydx.scalar import Scalar, set_symbolic_scalar, set_interval_scalar, restore_scalar
from pydx.scalar.symbolic import Float
from pydx.scalar.mpfi import Interval, DisorderException
from pydx.tensor import Tensor
from pydx.field import TensorField
from pydx.scalar.fmath import exp, log, sin, cos, sqrt, sqr
from pydx.manifold import RManifold
from pydx.geodesic import Geodesic
from pydx.main import options

#
################################################################################
#

class HyperbolicMetric(TensorField):
    def __init__(self):
        TensorField.__init__(self, (Tensor.dn,Tensor.dn), 2)
        self.uu = self.Inverse()
        self.uu.dd = self

    def __call__(self, x, y):
        zero = self.scalar_zero
        one = self.scalar_one
        g = [ [one/(y**2), zero], [zero, one/(y**2)] ]
        g = Tensor(self.valence, 2, elems=g)
        return g

    class Inverse(TensorField):
        def __init__(self):
            TensorField.__init__(self, (Tensor.up,Tensor.up), 2)
        def __call__(self, x, y):
            zero = self.scalar_zero
            one = self.scalar_one
            g = [ [y**2, zero], [zero, y**2] ]
            g = Tensor(self.valence, 2, elems=g)
            return g

class SwartzchildMetric(TensorField):
    def __init__(self, M):
        M = self.scalar_promote(M)
        self.M = M
        self.r = 2.0*M # outer event horizon
        self.uu = SwartzchildMetric.Inverse(M)
        TensorField.__init__(self, (Tensor.dn, Tensor.dn), 4)
        self.uu.dd = self

    def __call__(self, t, r, theta, phi):
        zero = self.scalar_zero
        M = self.M
        dt_dt = (2*M-r)/r
        dr_dr = r/(r-2*M)
        dtheta_dtheta = r**2
        dphi_dphi = (r*sin(theta))**2
        g = [
            [ dt_dt,   zero,   zero,          zero,      ],
            [ zero,    dr_dr,  zero,          zero,      ],
            [ zero,    zero,   dtheta_dtheta, zero,      ],
            [ zero,    zero,   zero,          dphi_dphi, ],
        ]
        g = Tensor(self.valence, self.dim, elems=g)
        return g
```

```
    class Inverse(TensorField):
        def __init__(self, M):
            M = self.scalar_promote(M)
            self.M = M
            self.r = 2.0*M # outer event horizon
            TensorField.__init__(self, (Tensor.up, Tensor.up), 4)

        def __call__(self, t, r, theta, phi):
            zero = self.scalar_zero
            one = self.scalar_one
            M = self.M
            dt_dt = (2*M-r)/r
            dr_dr = r/(r-2*M)
            dtheta_dtheta = r**2
            dphi_dphi = (r*sin(theta))**2
            g = [
                [ one/dt_dt,   zero,   zero,          zero,      ],
                [ zero,    one/dr_dr,  zero,          zero,      ],
                [ zero,    zero,   one/dtheta_dtheta, zero,      ],
                [ zero,    zero,   zero,          one/dphi_dphi, ],
            ]
            g = Tensor(self.valence, self.dim, elems=g)
            return g

class KerrMetric(TensorField):
    def __init__(self, M, a):
        M = self.scalar_promote(M)
        a = self.scalar_promote(a)
        self.M = M
        assert 0.0<abs(a)<=M # why not zero ??
        self.a = a
        self.r_plus = M + sqrt(sqr(M) - sqr(a)) # outer event horizon
        self.uu = self.Inverse(M,a)
        TensorField.__init__(self, (Tensor.dn, Tensor.dn), 4)
        self.uu.dd = self

    def __call__(self, t, r, theta, phi):
        zero = self.scalar_zero
        a = self.a
        M = self.M
        Sigma = sqr(r) + sqr(a*cos(theta))
        Delta = sqr(r) + sqr(a) - 2.0*M*r
        A = sqr(sqr(r) + sqr(a)) - Delta * sqr(a * sin(theta))
        dt_dt = -(1.0 - 2.0*M*r / Sigma)
        dt_dphi = - (2.0 * M * r / Sigma) * a * sqr(sin(theta))
        dr_dr = Sigma / Delta
        dtheta_dtheta = Sigma
        dphi_dphi = A * sqr(sin(theta)) / Sigma
        g = [
            [ dt_dt,   zero,   zero,          dt_dphi,   ],
            [ zero,    dr_dr,  zero,          zero,      ],
            [ zero,    zero,   dtheta_dtheta, zero,      ],
            [ dt_dphi, zero,   zero,          dphi_dphi, ],
        ]
        g = Tensor(self.valence, self.dim, elems=g)
        return g

    class Inverse(TensorField):
        def __init__(self, M, a):
            M = self.scalar_promote(M)
            a = self.scalar_promote(a)
            self.M = M
            assert 0.0<abs(a)<=M # why not zero ??
            self.a = a
            self.r_plus = M + sqrt(sqr(M) - sqr(a)) # outer event horizon
            TensorField.__init__(self, (Tensor.up, Tensor.up), 4)
        def __call__(self, t, r, theta, phi):
            zero = self.scalar_zero
            a = self.a
            M = self.M
            Sigma = sqr(r) + sqr(a*cos(theta))
            Delta = sqr(r) + sqr(a) - 2.0*M*r
            A = sqr(sqr(r) + sqr(a)) - Delta * sqr(a * sin(theta))
            dt_dt = -(1.0 - 2.0*M*r / Sigma)
            dt_dphi = - (2.0 * M * r / Sigma) * a * sqr(sin(theta))
            dr_dr = Sigma / Delta
            dtheta_dtheta = Sigma
            dphi_dphi = A * sqr(sin(theta)) / Sigma
            g = [
```

```
                [ dt_dt,   zero,   zero,          dt_dphi,   ],
                [ zero,    dr_dr,  zero,          zero,      ],
                [ zero,    zero,   dtheta_dtheta, zero,      ],
                [ dt_dphi, zero,   zero,          dphi_dphi, ],
            ]
            Z = 1.0 / (g[0][0]*g[3][3] - g[0][3]*g[3][0])
            g_up = [
                [ g[3][3]*Z, zero, zero, -g[0][3]*Z, ],
                [ zero, 1.0/g[1][1], zero, zero, ],
                [ zero, zero, 1.0/g[2][2], zero, ],
                [ -g[3][0]*Z, zero, zero, g[0][0]*Z, ],
            ]
            g_up = Tensor(self.valence, self.dim, elems=g_up)
            return g_up

class SpatialCurzonMetric(TensorField):
    def __init__(self, m):
        self.m = m
        self.uu = self.Inverse(m)
        TensorField.__init__(self, (Tensor.dn, Tensor.dn), 3)
        self.uu.dd = self

    def __call__(self, r, z, phi):
        zero = self.scalar_zero
        m = self.m
        R2 = sqr(r)+sqr(z)
        nu = -sqr(m)*sqr(r) / (2.0 * sqr(R2))
        lmda = -m/sqrt(R2)
        c = exp(2*(nu-lmda))
        dr_dr = c
        dz_dz = c
        dphi_dphi = sqr(r)*exp(-2.0*lmda)
        g = [
            [ dr_dr,  zero,  zero,      ],
            [ zero,   dz_dz, zero,      ],
            [ zero,   zero,  dphi_dphi, ],
        ]
        g = Tensor(self.valence, self.dim, elems=g)
        return g

    class Inverse(TensorField):
        def __init__(self, m):
            self.m = m
            TensorField.__init__(self, (Tensor.up, Tensor.up), 3)

        def __call__(self, r, z, phi):
            zero = self.scalar_zero
            m = self.m
            R2 = sqr(r)+sqr(z)
            nu = -sqr(m)*sqr(r) / (2.0 * sqr(R2))
            lmda = -m/sqrt(R2)
            # dt_dt = -exp(2*lmda)
            c = exp(2*(nu-lmda))
            dr_dr = c
            dz_dz = c
            dphi_dphi = sqr(r)*exp(-2.0*lmda)
            g = [
                [ dr_dr,  zero,  zero,      ],
                [ zero,   dz_dz, zero,      ],
                [ zero,   zero,  dphi_dphi, ],
            ]

            g_up = [
                [ 1.0/g[0][0], zero, zero, ],
                [ zero, 1.0/g[1][1], zero, ],
                [ zero, zero, 1.0/g[2][2], ],
            ]
            g_up = Tensor(self.valence, self.dim, elems=g_up)
            return g_up

class RZCurzonMetric(TensorField):
    # r and z coords only
    def __init__(self, m):
        self.m = m
        self.uu = self.Inverse(m)
        TensorField.__init__(self, (Tensor.dn, Tensor.dn), 2)
        self.uu.dd = self

    def __call__(self, r, z):
        zero = self.scalar_zero
```

```
        m = self.m
        R2 = sqr(r)+sqr(z)
        self.nu = -sqr(m)*sqr(r) / (2.0 * sqr(R2))
        r = R2[(0,)*R2.rank]
        if type(r) ==Interval:
            assert R2[(0,)*R2.rank]>0.0, R2[(0,)*R2.rank]
        self.lmda = -m/sqrt(R2)
        # dt_dt = -exp(2*lmda)
        c = exp(2*(self.nu-self.lmda))
        dr_dr = c
        dz_dz = c

        g = [
            [ dr_dr,  zero,  ],
            [ zero,   dz_dz, ],
        ]
        g = Tensor(self.valence, self.dim, elems=g)
        return g

    class Inverse(TensorField):
        def __init__(self, m):
            self.m = m
            TensorField.__init__(self, (Tensor.up, Tensor.up), 2)

        def __call__(self, r, z):
            zero = self.scalar_zero
            one = self.scalar_one
            m = self.m
            R2 = sqr(r)+sqr(z)
            self.nu = -sqr(m)*sqr(r) / (2.0 * sqr(R2))
            self.lmda = -m/sqrt(R2)
            # dt_dt = -exp(2*lmda)
            c = exp(2*(self.nu-self.lmda))
            g_up = [
                [ one/c, zero, ],
                [ zero, one/c, ],
            ]
            g_up = Tensor(self.valence, self.dim, elems=g_up)
            return g_up
```

B.10 manifold.py

```
import sys
from random import random

from pydx.scalar import Scalar
from pydx.mjet import MJet
from pydx.tensor import Tensor
from pydx.field import TensorField

__module__ = sys.modules[__name__]
Scalar.clients.append(__module__)

class RManifold(object):
    scalar_one = 1.0
    def __init__( self, g, g_uu=None ):
        if g_uu is not None:
            g.uu = g_uu
        g.uu.dd = g
        dim = g.dim
        g.p = g.comma()
        gamma = (self.scalar_one/2) * g.uu.mul(
            g.p + g.p.transpose(2,1,0) - g.p.transpose(2,0,1), (1,1) ).transpose(0,2,1)
        christoffel = gamma
        gamma.p = gamma.comma()
        # XX write a test for kretschman in SwartzchildMetric XX
        gamma_gamma = gamma.mul(gamma, (0,2))
        riemann = \
                gamma.p.transpose(0,2,3,1) \
            - gamma.p.transpose(0,2,1,3) \
            + gamma_gamma.transpose(2,0,3,1) \
            - gamma_gamma.transpose(2,0,1,3)
        ricci = riemann.contract( (0,2) )
        ricci.ud = ricci.mul( g.uu, (0,0) ) # ricci.ud
        curvature = ricci.ud.contract( (0,1) )
        riemann.dddd = riemann.mul( g, (0,0) ).transpose(3,0,1,2)
```

```
        riemann.uuuu = riemann.mul( g.uu, (1,0) ).transpose(0,3,1,2)
        riemann.uuuu = riemann.uuuu.mul( g.uu, (2,0) ).transpose(0,1,3,2)
        riemann.uuuu = riemann.uuuu.mul( g.uu, (3,0) )
        kretschman = riemann.uuuu.mul( riemann.dddd, (0,0), (1,1), (2,2), (3,3) )
        del gamma_gamma, g_uu
        self.__dict__.update( locals() )
Scalar.clients.append( RManifold )
```

B.11 geodesic.py

```
from random import random

from pydx.mjet import MJet, Jet
from pydx.scalar import Scalar, set_symbolic_scalar, set_interval_scalar, restore_scalar
from pydx.scalar.mpfi import Interval

from pydx.tensor import Tensor
from pydx.field import TensorField, ConcreteTensorField
from pydx.manifold import RManifold
from pydx.transform import MobiusTransform
from pydx.main import options
from pydx.ode import ODE


class Geodesic(ODE):

    def __init__(self, manifold, x0, y0, paramname=None, varnames=None):
        """
        """
        ODE.__init__(self, x0, y0, paramname=paramname, varnames=varnames)
        self.manifold = manifold
        self.gamma = manifold.gamma
        if options.compile:
            self.gamma = self.gamma.compile()

    def __call__(self, t, xs): # XX args should be "x" and "ys" XX
        """
            build Vector of derivatives of each component of Vector xs
        """
        assert xs is not None
        assert len(xs)==self.dim, xs
        dx = xs[self.dim/2:] # these are the derivatives of x
        x = xs[:self.dim/2]
        assert len(x)==self.dim/2
        assert len(dx)==self.dim/2
        result = Tensor((Tensor.up,), self.dim)
        gamma = self.gamma(*x)
        for i in range(self.dim/2):
            result[i] = dx[i]
        for i in range(self.dim/2):
            r = sum([ -gamma[i,j,k]*dx[j]*dx[k]
                        for j in range(self.dim/2)
                        for k in range(self.dim/2) ], self.scalar_zero)
            result[self.dim/2+i] = r
        return result
```

Appendix C: Code Listing - Unit Tests

C.1 test_interval.py

```
import sys
from random import *

#from interval import Interval
#from xinterval import Interval
from pydx.scalar.mpfi import Interval
#from pydx.scalar.optimize import Interval

def test_nocrash():
    x = Interval(0,1)
    print x
    y = Interval(2,3)
    print y
    print x+y
    print x*y
    print x/y

def test_interval():
    " check that interval ops encompass the equivalent float ops "
    def f(x): return x**3 + 3*x + x*x + 99

    x = 1.0
    ix = Interval(x)
    assert f(ix).contains( f(x) )

    ops = [
        lambda a,b:a+b,
        lambda a,b:a-b,
        lambda a,b:a*b,
    ]

    from gmpy import mpf
    seed(1)
    x = [random(), random()]
    for _ in range(100):
        i = 0
        r = 1e-10
        r = 0
        x = [mpf( x[0], 1024 ), mpf( x[1], 1024 )]
#        x = [12345L,654321L] # this works too
#        y = [Interval(float(x[0])-r,float(x[0])+r),Interval(float(x[1])-r,float(x[1])+r)]
        y = [Interval(x[0]-r,x[0]+r),Interval(x[1]-r,x[1]+r)]
#        x = [99*(y[0]/99),99*(y[1]/99)]
#        y = [y[0].hull(x[0]),y[1].hull(x[1])]
        while 1:
            assert y[0].contains(x[0]), "\n  i=%d, \n  y=%s, \n  x=%s" % (i, repr(y), repr(x) )
            assert y[1].contains(x[1]), "\n  i=%d, \n  y=%s, \n  x=%s" % (i, repr(y), repr(x) )
            idx = choice([0,1])
            if abs(x[0])+abs(x[1])>1e10:
                if abs(x[0])>abs(x[1]):
                    op = lambda x,y:abs(x)-abs(y)
                else:
                    op = lambda x,y:abs(y)-abs(x)
#            elif abs(x[0])<1e-5:
#                op = lambda x,y:(x+y+1.0)
            else:
                op = choice(ops)
            x[idx] = op(*x)
            y[idx] = op(*y)
            w = y[0].width()+y[1].width()
```

```
            if w>1.0:
                print "break"
                print "  w =", w
                print "  i =", i
                break
            i += 1


if __name__=="__main__":
    test_interval()
```

C.2 test_mjet.py

```
from random import random, seed

from gmpy import mpf

from pydx.mjet import MJet, Jet, Derivation, multi_unit, multi_range_order, cross
from pydx.scalar import set_interval_scalar, set_symbolic_scalar, restore_scalar, Scalar
from pydx.scalar.mpfi import Interval

def test_jet_expand_1():
    seed(0)

    f = lambda x: 1.0/(x+1.0)
    order = 5
    x0 = random() - 0.5
    x = Jet([x0, 1.0])           # Concrete MJet
    h = 0.01*(random()-0.5)      # small float
    fx = f(x)                    # MJet
    fxh = f(x0+h)                # float
    fxe = fx.expand((h,), order) # float
    error = abs(fxe - fxh)
    relative_error = error / max(1e-4, error)
    assert relative_error < 1e-6

def test_jet_expand_2():
    seed(0)
    f = lambda x: 1.0/(x+1.0)
    order = 5
    set_interval_scalar()
    x0 = Interval(random()-0.5)
    x = Jet([x0,1.0])
    h = Interval(random()-0.5)
    xx0 = x0.hull(x0+h)
    xx = Jet([xx0,1.0])
    err = f(xx)[order+1]
    fx = f(x)
    fxh = f(x0+h)
    fxe = fx.expand_err( (h,), order, (err,) )
    assert fxh.overlapping(fxe)
    restore_scalar()


def test_jet_taylor_univariate():
    seed(0)
    assert Scalar.current.type == float

    fs = []
    fs += [
        lambda x: x,
        lambda x: 1.0/(x+1.0),
        lambda x: x*x-1.0,
        lambda x: x**2-1.0,
#        lambda x: (x+1.0)**0.5, # does not work on float scalars
        lambda x: (x+1.0)*(x-2.0)*x,
        lambda x: (x+1.0)*(x-2.0)/(x+10.0),
        lambda x: (x+1.0)*(x+2.0)*x*(x+1.0)*(x+2.0)*x,
        lambda x: 7.0*x**3-2.0*x**2+1.0,
    ]

    n_evals = 0

    total_err = 0.0
    for f in fs:
        for trials in range(50):
```

```
            x = random()-0.5
            h = 0.01*random()
            order = 5
            dx = Jet([x,1.0])
            y = f(dx)
            r0 = f(x+h)
            r1 = y.expand( (h,), order )
            if abs(r0)+abs(r1)>1e-6:
                n_evals += 1
                err = abs(r0-r1)/(abs(r0)+abs(r1))
                assert err < 1e-6, (r0,r1)
                total_err += err
    print "total_err:", total_err

# These tests need interval scalar's to work:
    fs.append( lambda x: x.exp() )
    fs.append( lambda x: (2.0*x).exp() )
    fs.append( lambda x: (x**3).exp() )
    fs.append( lambda x: (x+1.0).log() )
    fs.append( lambda x: (x**3+1.0).log() )
    fs.append( lambda x: (x+1.0)**0.5 )
    fs.append( lambda x: (x**3+1.0).sin() )
    fs.append( lambda x: (x**3+1.0).cos() )
    fs.append( lambda x: (x**2).tan() )
    fs.append( lambda x: (x**2).asin() )
    fs.append( lambda x: (x**3+1.0).atan() )
    fs.append( lambda x: (x**3+1.0).sqrt() )
    fs.append( lambda x: (x**3+1.0).sinh() )
    fs.append( lambda x: (x**3+1.0).cosh() )
    fs.append( lambda x: (x**3+1.0).tanh() )

    set_interval_scalar()

    for trials in range(4):
        for f in fs:
            x = Interval(random()-0.5)
            h = Interval(random()-0.5)
            xx = x.hull(x+h)
            for order in range(4):
                dx = Jet([x,1.0])
                dxx = Jet([xx,1.0])
                err = f(dxx)[order+1]
                y = f(dx)
                r0 = f(x+h)
                r1 = y.expand_err( (h,), order, (err,) )
                assert r1.overlapping(r0)
                n_evals += 1

    restore_scalar()

    assert Scalar.current.type == float
    print "test_jet_taylor_univariate n_evals:", n_evals

def test_jet_taylor_multivariate():
    seed(0)
    fs = [
#       lambda x: x,
#       lambda x: x*x+1.0,
#       lambda x: x**2-1.0,
        lambda x: (x+1.0)*(x+-2.0)*x,
#       lambda x: (x+1.0)*(x-2.0)/(x*x+10.0),
#       lambda x: (x+1.0)*(x+-2.0)*x*(x+0.5),
#       lambda x: 5.0*x**3-2.0*x**2+1.0,
    ]
    gs =  []
#    gs += [ lambda x, y, f1=f1, f2=f2: f1(x) for f1 in fs for f2 in fs ]
    gs += [ lambda x, y, f1=f1, f2=f2: f1(x)+f2(y) for f1 in fs for f2 in fs ]
    gs += [ lambda x, y, f1=f1, f2=f2: f1(x)*f2(y) for f1 in fs for f2 in fs ]
#    gs += [ lambda x, y, f1=f1, f2=f2: f1(x)+f2(y)*f1(y) for f1 in fs for f2 in fs ]
#    gs += [ lambda x, y, f1=f1, f2=f2: f1(x)*f2(y)+f1(y) for f1 in fs for f2 in fs ]

    max_err = 0.0
    n_evals = 0

    for g in gs:
        for trials in range(30):
            x = random()-0.5, random()-0.5
            h = (0.01*random(),0.01*random())
            dx = Jet({ (0,0):x[0], (1,0):1.0 }), Jet({ (0,0):x[1], (0,1):1.0 })
            y = g(*dx)
```

```
            r0 = g( x[0]+h[0], x[1]+h[1] )
#            r2 = y.expand( h, 2 )
            r3 = y.expand( h, 3 )
            r = abs(r0)+abs(r3)
            if r>1e-6:
                err = abs(r0-r3)/r
                assert err < 1e-5, err
                n_evals += 1
    print "test_jet_taylor_multivariate n_evals:",n_evals

    # The following ops are only defined on intervals
    for g in gs[:]:
        gs.append( lambda x, y, g=g: g(x,y).exp() )
        gs.append( lambda x, y, g=g: (g(x,y)**2+0.1).log() )
        gs.append( lambda x, y, g=g: (g(x,y)**2+0.1)**1.1 )
        gs.append( lambda x, y, g=g: g(x,y).sin() )
        gs.append( lambda x, y, g=g: g(x,y).cos() )
        gs.append( lambda x, y, g=g: g(x,y).tan() )
##      gs.append( lambda x, y, g=g: g(x,y).acos() ) # not implemented
        gs.append( lambda x, y, g=g: g(x,y).atan() )
        gs.append( lambda x, y, g=g: g(x,y).sinh() )
        gs.append( lambda x, y, g=g: g(x,y).cosh() )
        gs.append( lambda x, y, g=g: g(x,y).tanh() )

        gs.append( lambda x, y, g=g: g(x,y)**3 )

    gs.append( lambda x, y, g=g: ((x+y)/2.0).asin() )
    gs.append( lambda x, y, g=g: (2.0+(x+y)).sqrt() )

    set_interval_scalar(128)

    total_width = mpf(0.0)
    for trials in range(30):
        for g in gs:
            x = [ Interval(random()-0.5) for i in (0,1) ]
            h = [ Interval(random()-0.5)*0.01 for i in (0,1) ]
            xx = [ x[i].hull(x[i]+h[i]) for i in (0,1) ]
            for order in (0,1,2):
                r0 = g(x[0]+h[0],x[1]+h[1])
                dx = [ Jet( {(0,0):x[i],multi_unit(2,i):1.0} ) for i in (0,1) ]
                y = g(*dx)
                dxx = [ Jet( {(0,0):xx[i],multi_unit(2,i):1.0} ) for i in (0,1) ]
                yy = g(*dxx)
                err = [ yy[j] for j in multi_range_order(2,order+1,order+1) ]
                r1 = y.expand( h, order )
                r1 = y.expand_err( h, order, err )
                total_width += r1.width()
                assert r1.overlapping(r0), (x,h)
                n_evals += 1
    print "total_width:", total_width

    restore_scalar()

    print "test_jet_taylor_multivariate n_evals:",n_evals


#
################################################################################
#

def test_slice():
    r = 1.234
    x = Jet( {(0,0):r, (1,0):1.0} )
    x0 = Jet( {(0,):r, (1,):1.0} )
    x1 = Jet( {(0,):0.0, (1,):0.0} )
    assert x[:,0] == x0
    assert x[:,1] == x1
    assert x0 != x1

def derive( f, *y ):
    return Derivation(f)(*y)

def test_derive():
    seed(0)

    tests = [
    # list of (f, f')
        ( lambda x: x, lambda x: 1.0 ),
        ( lambda x: x*x, lambda x: 2.0*x ),
        ( lambda x: x*x+7.0, lambda x: 2.0*x ),
```

```
        ( lambda x: 3.0*x*x-9.0*x, lambda x: 6.0*x-9.0 ),
        ( lambda x: 2.0*x**3+1.0, lambda x: 6.0*x*x ),
    ]

    n_evals = 0
    for f, df in tests:
        for i in range(5):
            # scalar
            x = random()-0.5
            y0 = derive( f, x )[0]
            y0 = y0[()] # XX should this be a scalar already ?
            y1 = df( x )
            assert abs(y0-y1)<1e-10, abs(y0-y1)
            n_evals += 1

            # rank 1
            rank = 1
            idxs = (0,)
            x = Jet( {(0,):x,(1,):1.0} )
            y0 = derive( f, x )[0]
            y0 = y0[idxs]
            y1 = df( x )
            y1 = MJet(rank).promote(y1)
            y1 = y1[idxs]
            assert abs(y0-y1)<1e-10, abs(y0-y1)
            n_evals += 1

            # rank 1 to 4
            for rank in range(1,5):
                x = Jet( rank = rank )
                for idxs in cross( (3,)*rank ):
                    x[idxs] = random()-0.5
                y0 = derive( f, x )[0]
                y1 = df( x )
                y1 = MJet(rank).promote(y1)
                for idxs in cross( (3,)*rank ):
                    _y0 = y0[idxs]
                    _y1 = y1[idxs]
                    assert abs( _y0 - _y1 ) < 1e-10, abs(_y0-_y1)
                    n_evals += 1
    print "test_derive: n_evals:", n_evals

def test_derive_multivariate():
    seed(0)
    _tests = [
    # list of (f, f')
        ( lambda x: x, lambda x: 1.0 ),
#        ( lambda x: x*x, lambda x: 2.0*x ),
        ( lambda x: x*x+7.0, lambda x: 2.0*x ),
        ( lambda x: 3.0*x*x-9.0*x, lambda x: 6.0*x-9.0 ),
        ( lambda x: 2.0*x**3+1.0, lambda x: 6.0*x*x ),
    ]
    tests = []
    for f1, df1 in _tests:
        for f2, df2 in _tests:
#            tests.append(
#                ( lambda x, y: f1(x)+f2(x), lambda x, y: df1(x)+df2(x), lambda x, y: 0.0 ))
#            tests.append(
#                ( lambda x, y: f1(y)+f2(y), lambda x, y: 0.0, lambda x, y: df1(y)+df2(y) ))
            tests.append(
                ( lambda x, y: f1(x)+f2(y), lambda x, y: df1(x), lambda x, y: df2(y) ))
            tests.append(
                ( lambda x, y: (f1(x)+f2(y))**2,
                    lambda x, y: 2.0*(f1(x)+f2(y))*df1(x),
                    lambda x, y: 2.0*(f1(x)+f2(y))*df2(y), ))
#            tests.append(
#                ( lambda x, y: f1(x)*f2(y), lambda x, y: df1(x)*f2(y), lambda x, y: f1(x)*df2(y) ))
    n_evals = 0
    for f, fx, fy in tests:
#        for _ in range(3):
            # test scalar args
            x = [ random()-0.5 for _ in (0,1) ]
            df = derive( f, *x )
            for i in (0,1):
                y0 = df[i][()]
                y1 = (fx,fy)[i]( *x )
                assert abs(y0-y1)<1e-10
                n_evals += 1

            # test rank-1 arg
```

```
            x = [ random()-0.5 for _ in (0,1) ]
            rank = 1
            idxs = (0,)
            x = [ Jet( {(0,):xi,(1,):1.0} ) for xi in x ]
            df = derive( f, *x )

            for i in (0,1):
                y0 = df[i][(0,)]
                y1 = (fx,fy)[i]( *x )
                y1 = MJet(rank).promote(y1)
                y1 = y1[idxs]
                assert abs(y0-y1)<1e-10
                n_evals += 1

            # test rank-n args
            for rank in range(1,3):
                x = [ Jet( rank = rank ) for _ in (0,1) ]
                for xi in x:
                    for idxs in cross( (3,)*rank ):
                        xi[idxs] = random()-0.5
                y0 = derive( f, *x )
                for i in (0,1):
                    y1 = (fx,fy)[i]( *x )
                    y1 = MJet(rank).promote(y1)
                    for idxs in cross( (3,)*rank ):
                        _y0 = y0[i][idxs]
                        _y1 = y1[idxs]
                        assert abs( _y0 - _y1 ) < 1e-10
                        n_evals += 1

    print "test_derive_multivariate: n_evals:", n_evals

if __name__=="__main__":
    pass
```

C.3 test_nops.py

```
from random import random, seed, choice, randint

from gmpy import mpf

from pydx.mjet import MJet, Jet, Derivation, multi_unit, multi_range_order, cross
from pydx.scalar import set_interval_scalar, set_symbolic_scalar, restore_scalar, Scalar
from pydx.scalar.mpfi import Interval
from pydx.scalar.symbolic import Var, OneFloat, ComputationContext, Float

def mk_expr(rank, nops, depth=0):
    vars = ['x%d'%i for i in range(rank)]
    if nops == 0:
        expr = choice(vars)
    elif nops == 1:
        op = choice('+-*/')
        var0 = choice(vars)
        var1 = choice(vars)
        expr = "(%s %s %s)" % (var0, op, var1)
    else:
        op = choice('+-*/')
        nops0 = randint(0, nops-1)
        nops1 = nops - nops0 - 1
        assert nops0+nops1+1==nops
        expr0 = mk_expr(rank, nops0, depth+1)
        expr1 = mk_expr(rank, nops1, depth+1)
        expr = "(%s %s %s)" % (expr0, op, expr1)
    if depth == 0:
        expr = '(lambda %s: %s)' % (', '.join(vars), expr)
    return expr

def test_nops():
    #seed(4)
    set_symbolic_scalar()
    one = OneFloat()

    trials = range(1)
    ranks = range(1, 5)
    ranks = [1]
    n = 12
```

```
    nopss = [6]

    for rank in ranks:
        zero_idx = (0,)*rank

        vars = [Var('x%d'%i) for i in range(rank)]
        xs = []
        for i, var in enumerate(vars[:]):
            unit_idx = multi_unit(rank, i)
            x = Jet({zero_idx:var, unit_idx:one})
            xs.append(x)

        idxss = [idxs for idxs in cross((n,)*rank)]

        for nops in nopss:
            for trial in trials:
                print
                _count = 0
                expr = mk_expr(rank, nops)
                func = eval(expr)
                # this is the example in the text
                #func = lambda x: (x**2).exp()
                for idxs in idxss:
                    file = open('temp.py', 'w')
                    ctx = ComputationContext('func', vars, file)
                    rval = func(*xs) # symbolic value
                    rval = rval[idxs]
                    ctx.assign('rval', rval)
                    _ = ctx.finalize('rval')
                    file.close()
                    count = len(open('temp.py').readlines()) - 4
                    ucount = rval.uniqlen()
                    print '%6d'%idxs[0],
                    print '%6d'%rval.deeplen(),
                    print '%6d'%count,
                    print '%6d'%ucount,
                    print '%4d'%nops, idxs
                    _count = count
                    Float.cache.clear()

    restore_scalar()

if __name__ == "__main__":
    test_nops()
```

C.4 test_ode.py

```
from pydx.ode import ODE
from pydx.scalar import set_interval_scalar, restore_scalar, Scalar
from pydx.scalar.mpfi import Interval
from pydx.tensor import Tensor

def test_ode():
    set_interval_scalar()

    scalar_promote = ODE.scalar_promote
    scalar_one = ODE.scalar_one
    scalrand = lambda : ODE.scalar_promote(random())

    steps = 1000
    h = ODE.scalar_promote(0.001)

    n_evals = 0

    class Pendulum(ODE):
        def __init__(self, x0, y0):
            """
            """
            ODE.__init__(self, x0, y0)
            assert len(y0)==2
        def __call__(self, x, y):
            y, dy = y
            result = Tensor((Tensor.up,), self.dim)
            result[0] = dy
            result[1] = -y
            return result
```

```
    # test first order interval method
    # y''(x) = -y(x), y(0) = 1, y'(0) = 0
    # solution: y(x) = cos(x)
    x = ODE.scalar_zero
    y, dy = ODE.scalar_one, ODE.scalar_zero
    ode = Pendulum(x, [y,dy])
    for i in range(steps):
        x = Interval(ode.x).lower
        Y0 = ode.contract1(Interval(x,(x+h).upper), ode.y)
        ode.istep1(h, Y0)
        y, dy = ode.y
        assert ode.y[0].overlapping(ode.x.cos())
        n_evals += 1
    print "istep1 x",ode.x," width:", ode.y[0].width(), "centre:", ode.y[0].centre()

    # test second order interval method
    # y''(x) = -y(x), y(0) = 1, y'(0) = 0
    # solution: y(x) = cos(x)
    x = ODE.scalar_zero
    y, dy = ODE.scalar_one, ODE.scalar_zero
    ode = Pendulum(x, [y,dy])
    for i in range(steps):
        x = Interval(ode.x).lower
        Y0 = ode.contract1(Interval(x,(x+h).upper), ode.y)
        ode.istep2(h, Y0)
        y, dy = ode.y
        assert ode.y[0].overlapping(ode.x.cos())
        n_evals += 1
    print "istep2 x",ode.x," width:", ode.y[0].width(), "centre:", ode.y[0].centre()

    restore_scalar()
    print "test_ode: n_evals:", n_evals

if __name__ == "__main__":
    test_ode()
```

C.5 test_field.py

```
from random import random
from time import time

from pydx import scalar
from pydx.scalar import set_symbolic_scalar, restore_scalar
from pydx.scalar.symbolic import Var
from pydx.tensor import Tensor, up, dn
from pydx.field import TensorField, ScalarField, Transform
from pydx.transform import Complex, LinearTransform, ComplexTransform, MobiusTransform

#
################################################################################
#

def test_tensor():
    t = Tensor( (dn,), 4 )
    t[0,] = 1.0

    n_evals = 0
    for _ in range(10):
        a, b, c, d = [(random()-0.5,random()-0.5) for _ in range(4)]
        for T in (
            LinearTransform(*[random()-0.5 for _ in range(4)]),
            ComplexTransform(a, b),
            MobiusTransform(a, b, c, d),
        ):

            # test the inverse of the transform
            x, y = (random()-0.5)*10.0, (random()-0.5)*10.0
            _x, _y = T( x, y )
            _x, _y = T.inverse( _x, _y )
            assert abs(x-_x)+abs(y-_y) < 1e-6
            n_evals += 1

            # test .view
            _x, _y = T.inverse.view( T )( x, y )
```

```
            assert abs(x-_x)+abs(y-_y) < 1e-6
            n_evals += 1

            # test the inverse of the partial matrix
            # is the partial of the inverse transform
            Tx, Ty = T(x, y)
            p1 = T.partial(x,y)  # 2-2 matrix of partials
            p2 = T.inverse.partial(Tx, Ty)  # 2-2 matrix of partials of the inverse transform
            p = p1.mul( p2, (1,0) ) # sum over second index of p1 and first index of p2
            p.apply( lambda x:x[()] ) # get the scalar values out of the 0-Jet's
            # test identity matrix
            for r in p[1,1], p[1,1]:
                assert abs(r-1)<1e-10
            for r in p[0,1], p[1,0]:
                assert abs(r)<1e-10
            n_evals += 1

            # test ScalarField's are invarient
            a, b, c = [random()-0.5 for _ in range(3)]
            f = lambda x,y: a*x*x+b*y+c
            sf = ScalarField( 2, f )
            s1 = sf( *T(x,y) )
            s2 = sf.transform(T)( x, y )
            assert abs((s1-s2)[()])<1e-6
            n_evals += 1

            # test vector fields transform back
            vf0 = sf.comma()
            vf1 = vf0.transform(T)
            vf2 = vf1.transform(T.inverse)
            assert vf0( x, y ).is_close( vf2(x, y) )
            n_evals += 1

            # test some tensor fields transform back
            for valence in [
                    (Tensor.up,Tensor.up),
                    (Tensor.up,Tensor.dn),
                    (Tensor.dn,Tensor.dn),
                    (Tensor.dn,Tensor.up), ]:
                k0 = TensorField.identity( valence, 2 )
                k1 = k0.transform(T)
                k2 = k1.transform(T.inverse)
                assert k0.valence == k1.valence == k2.valence
                assert k0( x, y ).is_close( k2(x, y) )
                n_evals += 1

            # test that vector fields transform correctly
            sf0 = sf
            vf0 = sf0.comma()
            sf1 = sf0.transform(T)
            vf10 = sf1.comma()
            vf01 = vf0.transform(T)
            assert vf10( x, y ).is_close( vf01(x, y) )
            n_evals += 1



        print "test_tensor: n_evals:", n_evals


    def test_get_func():
        scalar_zero = 0.0
        scalar_one = 1.0
        scalar_promote = float

        xys = [(random()-0.5,random()-0.5) for _ in range(100)]

        metric = TensorField.identity( (Tensor.dn,Tensor.dn), 2 )
        metric_up = TensorField.identity( (Tensor.up,Tensor.up), 2 )

        n_evals = 0
        T=MobiusTransform(*[(scalar_promote(1.2+i),scalar_promote(3.3*i)) for i in (-2,3,1,7)])
        g = metric.transform(T)
        g_uu = metric_up.transform(T)
        gp = g.comma()
        gamma = (1.0/2)*g_uu.mul( gp + gp.transpose(0,2,1) - gp.transpose(1,2,0), (1,0) )

        rs = []
        t0 = time()
        for x, y in xys:
```

```
#        r = g(x,y)[0,0][()]
        r = gamma(x,y)[0,0,0][()]
        rs.append(r)
    print "time:", time()-t0

    set_symbolic_scalar()

    metric = TensorField.identity( (Tensor.dn,Tensor.dn), 2 )
    metric_up = TensorField.identity( (Tensor.up,Tensor.up), 2 )

    n_evals = 0
    T=MobiusTransform(*[(scalar_promote(1.2+i),scalar_promote(3.3*i)) for i in (-2,3,1,7)])
    g = metric.transform(T)
    g_uu = metric_up.transform(T)
    gp = g.comma()
    gamma = (1.0/2)*g_uu.mul( gp + gp.transpose(0,2,1) - gp.transpose(1,2,0), (1,0) )
#    expr = g(Var('x'),Var('y'))[0,0][()]
    expr = gamma(Var('x'),Var('y'))[0,0,0][()]
    print "deeplen:", expr.deeplen()
    print "uniqlen:", expr.uniqlen()
    s_expr = expr.expr()
    print "strlen:", len(s_expr)

#    this breaks down on big expressions
#    t0 = time()
#    for i, (x, y) in enumerate(xys):
#        x = scalar_promote(x)
#        y = scalar_promote(y)
#        r = eval(s_expr, {'x':eval(str(x)),'y':eval(str(y))})

    func = expr.get_func( 'func', ['x','y'] )
    t0 = time()
    for i, (x, y) in enumerate(xys):
        r = func(x,y)
        assert abs(r-rs[i])<1e-10
    print "time:", time()-t0

    restore_scalar()

if __name__=="__main__":
    pass
```

C.6 test_geodesic.py

```
from random import random

from pydx.scalar import set_mpf_scalar, set_interval_scalar, set_symbolic_scalar, restore_scalar
from pydx.scalar.mpfi import Interval
from pydx.mjet import MJet, Jet
from pydx.manifold import RManifold
from pydx.tensor import Tensor
from pydx.field import TensorField, ConcreteTensorField
from pydx.ode import ODE
from pydx.geodesic import Geodesic
from pydx.transform import MobiusTransform, LinearTransform

def test_geodesic_snell():
    # from "Reflections on Relativity":
    # http://www.mathpages.com/rr/s8-04/8-04.htm

#    set_mpf_scalar()
    set_interval_scalar()

    scalar_promote = MJet.scalar_promote
    scalar_one = MJet.scalar_one
    scalrand = lambda : scalar_promote(random())

#    set_symbolic_scalar()
#    t_uu = Tensor((Tensor.up,Tensor.up), 2)
#    for idxs in t_uu.genidx():
#        t_uu[idxs] = Var('t^%s' % (''.join(str(i) for i in idxs)))
#    t_ddd = Tensor((Tensor.dn,Tensor.dn,Tensor.dn), 2)
#    for idxs in t_ddd.genidx():
#        t_ddd[idxs] = Var('t_%s' % (''.join(str(i) for i in idxs)))
#    gamma = t_uu.mul(t_ddd + t_ddd.transpose(2,1,0) - t_ddd.transpose(2,0,1), (1,1)).transpose(0,2,1)
#    for idxs in gamma.genidx():
```

```
#    return

#    restore_scalar()

    class ODE_for_func1(ODE):
        def __init__(self, x0, y0, A, B):
            ODE.__init__(self, x0, y0)
            self.A = A
            self.B = B
        def __call__(self, t, x):
            assert len(x)==self.dim, x
            dx,dy = x[self.dim/2:] # these are the derivatives of x
            x,y = x[:self.dim/2]
            C = A/(A*x+B)
            result = Tensor((Tensor.up,), self.dim)
            result[0] = dx
            result[1] = dy
            result[2] = -C * dx*dx + C * dy*dy
            result[3] = -2.0 * C * dx * dy
            return result


    A = scalar_promote(5.0)
    B = 1.0/A
    n_func0 = lambda x,y: scalar_one
    n_func1 = lambda x,y: A*x+B
    n_func2 = lambda x,y: scalar_one/y
    n_evals = 0
    for n_func in (n_func0, n_func1, n_func2):
            print '--'*30
            g = ConcreteTensorField.zero((Tensor.dn,Tensor.dn), 2)
            g.uu = ConcreteTensorField.zero((Tensor.up,Tensor.up), 2)
            for i in (0,1):
                g[i,i] = lambda x,y: n_func(x,y)*n_func(x,y)
                g.uu[i,i] = lambda x,y: scalar_one / (n_func(x,y)*n_func(x,y))
            manifold = RManifold(g)
            gamma = manifold.gamma

#           x, y = x0, y0 = (scalrand()-0.5)*10.0, (scalrand()-0.5)*10.0
            x, y = x0, y0 = scalar_promote(1.0), scalar_promote(1.0)
            dx, dy = dx0, dy0 = scalrand(), scalrand()-0.5
            print "x, y, dx, dy", x, y, dx, dy
            t = t0 = scalar_promote(0.0)

            if 1:
                n = n_func(x,y)
                gp_xy = g.p(x,y).apply(lambda r:r[()])
                g00_func = lambda x,y: n_func(x,y)*n_func(x,y)
                gp_000 = MJet(1).promote(g00_func(Jet([x,scalar_one]), y))[1]
                gp_001 = MJet(1).promote(g00_func(x, Jet([y,scalar_one])))[1]
                assert abs(gp_xy[0,0,0] - gp_000) < 1e-10
                assert abs(gp_xy[0,0,1] - gp_001) < 1e-10
                assert abs(gp_xy[1,0,0]) < 1e-10
                assert abs(gp_xy[0,1,0]) < 1e-10
                assert abs(gp_xy[1,0,1]) < 1e-10
                assert abs(gp_xy[0,1,1]) < 1e-10
                assert abs(gp_xy[1,1,1] - gp_001) < 1e-10
                assert abs(gp_xy[1,1,0] - gp_000) < 1e-10

    #       for idxs in g.uu.genidx():
    #       for idxs in g.p.genidx():
    #       for idxs in gamma.genidx():
                gamma_000 = (scalar_one/n) * MJet(1).promote(n_func(Jet([x,scalar_one]),y))[1]
                gamma_001 = (scalar_one/n) * MJet(1).promote(n_func(x,Jet([y,scalar_one])))[1]

                assert abs(gamma(x,y)[0,0,0][()] - gamma_000) < 1e-10
                assert abs(gamma(x,y)[0,1,1][()] - -gamma_000) < 1e-10
                assert abs(gamma(x,y)[1,0,1][()] - gamma_000) < 1e-10
                assert abs(gamma(x,y)[1,1,0][()] - gamma_000) < 1e-10
                assert abs(gamma(x,y)[0,0,1][()] - gamma_001) < 1e-10
                assert abs(gamma(x,y)[0,1,0][()] - gamma_001) < 1e-10
                assert abs(gamma(x,y)[1,0,0][()] - -gamma_001) < 1e-10
                assert abs(gamma(x,y)[1,1,1][()] - gamma_001) < 1e-10

            STEPS = 10
            steps1 = []
            invariant = None

            ode = Geodesic(manifold, t0, [x,y,dx,dy])
```

```
            h = scalar_promote(0.01)
            for i in range(STEPS):
                n = n_func(x,y)
                if n_func in (n_func0,n_func1):
                    q = dy/(dx**2+dy**2).sqrt()
                else:
                    q = dx/(dx**2+dy**2).sqrt()
                _invariant = n*q
                print invariant, _invariant
                if invariant is None:
                    invariant = _invariant
                assert invariant.overlapping(_invariant)
                n_evals += 1

                x = Interval(x).lower
                Y0 = ode.contract1(Interval(x,(x+h).upper), ode.y)
                ode.istep2(h, Y0)
                x, y, dx, dy = ode.y
                steps1.append((x,y,dx,dy))

            if n_func == n_func1:
                x, y = x0, y0
                dx, dy = dx0, dy0
                print "x, y, dx, dy", x, y, dx, dy
                t = t0 = scalar_promote(0.0)
                steps2 = []

                ode = ODE_for_func1(t0, [x,y,dx,dy], A, B)
                for i in range(STEPS):
                    x = Interval(x).lower
                    Y0 = ode.contract1(Interval(x,(x+h).upper), ode.y)
                    ode.istep2(h, Y0)
                    x, y, dx, dy = ode.y
                    steps2.append((x,y,dx,dy))
                    n = n_func(x,y)
                    if n_func in (n_func0,n_func1):
                        q = dy/(dx**2+dy**2).sqrt()
                    else:
                        q = dx/(dx**2+dy**2).sqrt()
                    _invariant = n*q
                    print invariant, _invariant
                    assert invariant.overlapping(_invariant)
                    n_evals += 1

                for i in range(STEPS):
                    for x0, x1 in zip(steps1[i], steps2[i]):
                        assert x0.overlapping(x1)
                        n_evals += 1

    restore_scalar()
    print "test_geodesic_snell: n_evals:", n_evals



def test_geodesic_flatspace():
#    set_mpf_scalar()
    set_interval_scalar()

    scalar_promote = MJet.scalar_promote
    scalar_one = MJet.scalar_one
    scalrand = lambda : scalar_promote(random())

    metric = TensorField.identity((Tensor.dn,Tensor.dn), 2)
    metric_up = TensorField.identity((Tensor.up,Tensor.up), 2)

    n_evals = 0
    for _ in range(1):
#        a, b, c, d = [(scalrand()-0.5,scalrand()-0.5) for _ in range(4)]
        a, b, c, d = ((0.0,-1.0), (0.0,1.0), (1.0,0.0), (1.0,0.0),)
        for T in (
#            LinearTransform(*[scalrand()-0.5 for _ in range(4)]),
#            ComplexTransform(a, b),
            MobiusTransform(a, b, c, d),
        ):
            print T

            g = metric.transform(T)
            g_uu = metric_up.transform(T)
```

```
#            x, y = x0, y0 = (scalrand()-0.5)*10.0, (scalrand()-0.5)*10.0
            x, y = x0, y0 = T(0.0,0.0)
#            dx, dy = dx0, dy0 = scalrand(), scalrand()
            dx, dy = dx0, dy0 = (0.0,1.0)
            print x, y, dx, dy
            t = t0 = scalar_promote(0.0)

            manifold = RManifold(g, g_uu)
            ode = Geodesic(manifold, t0, [x,y,dx,dy])
            h = scalar_promote(0.0001)
            for i in range(5):
                x = Interval(x).lower
                Y0 = ode.contract1(Interval(x,(x+h).upper), ode.y)
                ode.istep1(h, Y0)
                tensor = ConcreteTensorField((Tensor.dn,), 2)
                tensor[0] = lambda x,y: dx
                tensor[1] = lambda x,y: dy
                x, y, dx, dy = ode.y

                print x, y

                for _tensor in (
#                    tensor(x,y),
#                    tensor.transform(T)(x,y),
#                    tensor.transform(T)(*T(x,y)),
#                    tensor.transform(T)(*T.inverse(x,y)),
#                    tensor.transform(T.inverse)(x,y),
#                    tensor.transform(T.inverse)(*T(x,y)),
#                    tensor.transform(T.inverse)(*T.inverse(x,y)),
                ):
                    print MJet(0).promote(_tensor[0])[()],
                print

                n_evals += 1


    restore_scalar()
    print "test_geodesic_flatspace: n_evals:", n_evals


if __name__=="__main__":
    test_geodesic_snell()
    test_geodesic_flatspace()
```

C.7 test_curzon.py

```
import sys
from random import random

from pydx.mjet import MJet
from pydx.scalar import Scalar, set_interval_scalar, restore_scalar, set_symbolic_scalar
from pydx.scalar.symbolic import Var
from pydx.scalar.mpfi import Interval, DisorderException
from pydx.metric import RZCurzonMetric, SpatialCurzonMetric
from pydx.manifold import RManifold
from pydx.main import options, main


def test_rz_in_spatial_curzon():

    m = 1.0
    g0 = RZCurzonMetric(m)
    g1 = SpatialCurzonMetric(m)

    pg0 = g0.comma()
    pg1 = g1.comma()

    n_evals = 0
    for i in range(10):
        r, z = 1+random(), 1+random()
        phi = 1+random()

        r = MJet(0).promote(r)
        z = MJet(0).promote(z)
        phi = MJet(0).promote(phi)
```

```
        a0 = g0(r, z)
        a1 = g1(r, z, phi)

        a0.demote()
        a1.demote()

        for i in range(2):
            for j in range(2):
                assert abs( a0[i,j] - a1[i,j] ) < 1e-6
                n_evals += 1

        a0 = pg0(r, z)
        a1 = pg1(r, z, phi)

        a0.demote()
        a1.demote()

        for i in range(2):
            for j in range(2):
                for k in range(2):
                    assert abs( a0[i,j,k] - a1[i,j,k] ) < 1e-6
                    n_evals += 1


    print n_evals
```

C.8 test_manifold.py

```
from random import random

from pydx.scalar import Scalar
from pydx.mjet import MJet
from pydx.manifold import RManifold
from pydx.tensor import Tensor
from pydx.field import TensorField
from pydx.test.test_field import LinearTransform, MobiusTransform


def test_rmanifold():
    assert Scalar.current.type == float

    n_evals = 0

    id_dd = TensorField.identity((Tensor.dn,Tensor.dn), 2)
    id_uu = TensorField.identity((Tensor.up,Tensor.up), 2)

#    T = MobiusTransform(*[((1.2+i),(3.3*i)) for i in (-2,3,1,7)])
    T = LinearTransform(*[random()-0.5 for _ in range(4)])
    g = id_dd.transform(T)
    g_uu = id_uu.transform(T)

    manifold = RManifold(g, g_uu)

    assert manifold.gamma.valence == (Tensor.up,Tensor.dn,Tensor.dn)

    def test_equ(a, b, n=10):
        assert a.valence == b.valence
        xys = [(random()-0.5,random()-0.5) for _ in range(n)]
        n_evals = 0
        for x,y in xys:
            ta = a(x,y)
            tb = b(x,y)
            for idxs in ta.genidx():
                r1 = ta[idxs]
                r1 = MJet(0).promote(r1)[()]
                r2 = tb[idxs]
                r2 = MJet(0).promote(r2)[()]
                assert abs(r1-r2)<1e-10, (r1,r2)
                n_evals += 1
        return n_evals

    t_id = Tensor.identity((Tensor.up,Tensor.dn), 2)

    # XX This is in test_tensor
    xys = [(random()-0.5,random()-0.5) for _ in range(10)]
    for x,y in xys:
```

```
        p1 = T.partial(x,y)
        Tx,Ty=T(x,y)
        p2 = T.inverse.partial(Tx,Ty)
        p = p1.mul(p2, (1,0))
        assert p.is_close(t_id)
        n_evals += 1

    t_id = Tensor.identity((Tensor.up,Tensor.up), 2)

    for x,y in xys:
        assert t_id.is_close(id_uu(x,y))
        n_evals += 1

    g1 = g.uu.transform(T.inverse)
    n_evals += test_equ(g1, id_uu)

    n_evals += test_equ(
        TensorField.identity((Tensor.up,Tensor.dn), 2),
        g.uu.mul(g.dd, (0,0)))

    def test_sym(tfield, sym, anti=1.0, n=10):
        xys = [(random()-0.5,random()-0.5) for _ in range(n)]
        n_evals = 0
        for x,y in xys:
            tensor = tfield(x,y)
            tensor_t = tensor.transpose(*sym)
            for idxs in tensor.genidx():
                r1 = tensor[idxs][()]
                r2 = anti * tensor_t[idxs][()]
                assert abs(r1-r2)<1e-10, (r1,r2)
                n_evals += 1
        return n_evals

    # True for any manifold
    n_evals += test_sym(g.uu, (1,0))
    n_evals += test_sym(g.dd, (1,0))
    n_evals += test_sym(manifold.gamma, (0,2,1))
    n_evals += test_sym(manifold.riemann, (0,1,3,2), -1.0)

    # True in flat space
    n_evals = test_equ(manifold.riemann, TensorField.zero(manifold.riemann.valence, 2))

    # test for zero scalar curvature and zero kretschman
    for x, y in xys:
        assert abs(manifold.curvature(x,y)[()][()]) < 1e-10
        assert abs(manifold.kretschman(x,y)[()][()]) < 1e-10
        n_evals += 1

    print "test_rmanifold: n_evals:", n_evals

if __name__=="__main__":
    pass
```

C.9 test_metric.py

```
import sys

from pydx.mjet import MJet
from pydx.scalar import Scalar, set_interval_scalar, restore_scalar, set_symbolic_scalar
from pydx.scalar.symbolic import Var
from pydx.scalar.mpfi import Interval, DisorderException
from pydx.metric import HyperbolicMetric, SwartzchildMetric, RZCurzonMetric
from pydx.manifold import RManifold
from pydx.geodesic import Geodesic
from pydx.main import options, main

def test_hyperbolic_geodesic():
    set_interval_scalar()

    scalar_promote = MJet.scalar_promote
    scalar_one = MJet.scalar_one
    scalrand = lambda : scalar_promote(random())

    n_evals = 0

    # Test these geodesics are circles centred
```

```
    # on a point on the x-axis.
    g = HyperbolicMetric()
    manifold = RManifold(g)

    x, y = scalar_promote(0.0), scalar_promote(1.0)
    dx, dy = scalar_promote(1.0), scalar_promote(0.0)
    t0 = scalar_promote(0.0)

    ode = Geodesic(manifold, t0, [x,y,dx,dy])
    h = scalar_promote(0.01)
    for i in range(20):
        x = ode.x
        Y0 = ode.contract1(x.hull(x+h), ode.y)
        ode.istep2(h, Y0)
        x, y, dx, dy = ode.y
        print (x**2+y**2)
        assert (x**2+y**2).contains(Interval(1.0))
        n_evals += 1

    restore_scalar()
    print "test_hyperbolic_geodesic: n_evals:", n_evals

def test_swartzchild_geodesic():
    set_interval_scalar(options.prec)

    scalar_promote = MJet.scalar_promote
    scalar_one = MJet.scalar_one
    scalrand = lambda : scalar_promote(random())

    n_evals = 0

    M = 1.0 # mass
    g = SwartzchildMetric(M)
    manifold = RManifold(g)

    t = Interval(0.0)
    a = 1.0 # arbitrary
    dt = Interval(a)
    r = Interval(3.0)
    dr = Interval(0.0)
    theta = Interval(0.0).get_pi() / 2 # blows at sin(theta)=0
    dtheta = Interval(0.0)
    phi = Interval(0.0)
    dphi = a / (M * Interval(27.0).sqrt())
    x = [t, r, theta, phi, dt, dr, dtheta, dphi]

    t0 = scalar_promote(0.0)

    ode = Geodesic(manifold, t0, x)
#    h = scalar_promote(0.2)
    h = scalar_promote(options.step_size)
    for i in range(options.steps):
#    while phi.lower < (3.1416*2):
        x = ode.x
        Y0 = ode.contract1(x.hull(x+h), ode.y)
        ode.istepn(h, Y0, options.order)

        t, r, theta, phi, dt, dr, dtheta, dphi = ode.y
        w = sum(yi.width() for yi in ode.y)
        print t, r, phi, "width=", w
        assert r.contains(3.0)
        n_evals += 1

    restore_scalar()
    print "test_swartzchild_geodesic: n_evals:", n_evals

def test_rzcurzon_geodesic():
    set_interval_scalar(options.prec)

    scalar_promote = MJet.scalar_promote
    scalar_one = MJet.scalar_one
    scalrand = lambda : scalar_promote(random())

    n_evals = 0

    m = scalar_promote(1.0)
    g = RZCurzonMetric(m)
    manifold = RManifold(g)

    # set from options, or set from default starting values
```

```
    r = options.get('r', -0.01)
    z = options.get('z', 0.1)
    dr = options.get('dr', -1.0)
    dz = options.get('dz', -0.2)
    t0 = options.get('t0', 0.0)
    h = options.get('step_size', 0.0001)
    r,z,dr,dz,t0,h = [scalar_promote(_) for _ in r,z,dr,dz,t0,h ]
    # XX should we just keep all of those as params in options ? XX

    # build the ode object:
    ode = Geodesic(manifold, t0, [r,z,dr,dz], paramname='t', varnames='r z dr dz'.split())

    if options.filename is not None:
        # replay file
        #info = Info(begin=False)
        file = open(options.filename)
        count = 0
        try:
            while 1:
                ode.loadstate(file)
                width = sum(yi.width() for yi in ode.y)
                if options.session is not None:
        #            ode.commit(options.session, width=width, h=h)
                    ode.commit(options.session, width=width)
                count += 1
        except SyntaxError:
            pass
        print "done loadstate, count =", count
        file.close()
        return
        #file = open(options.filename, 'a')
#    else:
#        ode.dumpstate(file)

    file = sys.stdout

    try:

        while ode.i<options.steps or options.steps==0:

            x = ode.x
            Y0 = ode.contract1(x.hull(x+h), ode.y) #, max_iter=5)
            # ode.istep2(h, Y0)
            ode.istepn(h, Y0, options.order)

#            ode.dumpstate(file)

            width = sum(yi.width() for yi in ode.y)
            abs_size = sum(abs(yi).upper for yi in ode.y)
            abs_size = max(abs_size, 1e-12)
            relwidth = width / abs_size

            if options.session is not None:
    #            ode.commit(options.session, width=width, h=h)
                ode.commit(options.session, width=width)

            print "step", ode.i
            if relwidth > 1e5 or width > 1 or width==0:
                break

    except DisorderException, e:
        print >>file, 'DisorderException, e="%s"' % repr(e)
    print "Finished"

    restore_scalar()

#
#
# XX another test to write: that as we increase the order,   XX
# XX solutions overlap each other                            XX
#

if __name__=="__main__":
    main()
```