An introduction to pyranha

At the present time, pyranha supports two object types:

- polynomials,
- Poisson series.

Each type resides in its own submodule. The module hierarchy looks like this:

- `pyranhapp0x`
    - `polynomial`
    - `poisson_series`
    - `celmec`
    - `math`

For convenience and in order to reduce typing, we will start all notebooks by importing all of pyranha's submodules:

```
In [1]: from pyranhapp0x import *
```

Since we will be working mostly with rational coefficients, the following shortcut is also useful:

```
In [2]: from fractions import Fraction as Frac
```

In some examples we will be using the arbitrary-precision floating point type `mpf`, so let us import that and the configuration class `mp`:

```
In [3]: from mpmath import mpf, mp
```

Defining series types

The first step is to decide which algebraic objects we are going to work with. We need to choose if we want to work with polynomials or with Poisson series, and the coefficient type (or *ring*).

Each series submodule (`polynomial` and `poisson_series`) exposes a function `get_type()` that can be used to define concrete series types passing the coefficient type as argument:

```
In [4]: pt = polynomial.get_type(int)
        print(pt)

        <class 'pyranhapp0x._core._polynomial_1'>
```

Coefficient types can be passed also as strings. The following is equivalent to the previous command:

```
In [5]: pt = polynomial.get_type('integer')
        print(pt)

        <class 'pyranhapp0x._core._polynomial_1'>
```

The `get_cf_types()` function can be used to query the available coefficient types:

```
In [6]: polynomial.get_cf_types()

Out[6]: [('double', builtins.float),
         ('integer', builtins.int),
         ('rational', fractions.Fraction),
         ('real', mpmath.ctx_mp_python.mpf)]
```

```
In [7]: pt = polynomial.get_type('double')
        print(pt)
        pt = polynomial.get_type('real')
        print(pt)
        pt = polynomial.get_type('rational')
        print(pt)

        <class 'pyranhapp0x._core._polynomial_0'>
        <class 'pyranhapp0x._core._polynomial_3'>
        <class 'pyranhapp0x._core._polynomial_2'>
```

We can do the same with the `poisson_series` module:

```
In [8]: poisson_series.get_cf_types()

Out[8]: [('double', builtins.float),
 ('integer', builtins.int),
 ('rational', fractions.Fraction),
 ('real', mpmath.ctx_mp_python.mpf),
 ('polynomial_double', pyranhapp0x._core._polynomial_0),
 ('polynomial_integer', pyranhapp0x._core._polynomial_1),
 ('polynomial_rational', pyranhapp0x._core._polynomial_2),
 ('polynomial_real', pyranhapp0x._core._polynomial_3)]
```

In [9]:
```
pst = poisson_series.get_type('double')
print(pst)
pst = poisson_series.get_type('rational')
print(pst)
pst = poisson_series.get_type('polynomial_rational')
print(pst)
```

```
<class 'pyranhapp0x._core._poisson_series_0'>
<class 'pyranhapp0x._core._poisson_series_2'>
<class 'pyranhapp0x._core._poisson_series_6'>
```

Creating series instances

After defining the object type we want to work with, we can start using it to create series instances:

In [10]:
```
pt = polynomial.get_type('rational')
pt('x')
```

Out[10]: x

In order to construct a series instance, typically you can use:

- a string, which will construct a series mathematically equivalent to a symbol with that name,
- a numerical object,
- another series.

A few examples:

In [11]:
```
pt('y')
```

Out[11]: y

In [12]:
```
pt(pt('z'))
```

Out[12]: z

In [13]:
```
pt(5)
```

Out[13]: 5

In [14]:
```
pt(1.5)
```

Out[14]: $\frac{3}{2}$

Beware of the behaviour of hardware floating-point numbers:

In [15]:
```
pt(1.23)
```

Out[15]: $\frac{2769713770832855}{2251799813685248}$

A better way is to use exact rational numbers:

In [16]:
```
pt(Frac(123,100))
```

Out[16]: $\frac{123}{100}$

Or even arbitrary-precision floats:

In [17]:
```
pt(mpf('1.23'))
```

Out[17]: $\frac{123}{100}$

You can also use $\TeX$ syntax (just remember to specifiy a *raw* string with r):

In [18]:
```
pt(r'\alpha')
```

Out[18]: α

In [19]:
```
pt(r'\mathcal{H}')
```

Out[19]: $\mathcal{H}$

Basic arithmetics

After learning how to define series types and create series instances, we can now start actually doing *something* with them.

The common Python operators are overloaded for use with pyranha objects:

In [20]:
```
x,y = pt('x'),pt('y')
x+y
```

Out[20]: $x+y$

In [21]:
```
x*y
```

Out[21]: xy

In [22]:
```
x-y
```

Out[22]: $x-y$

The operators are overloaded also for use with numerical objects:

In [23]:
```
x + y + 3
```

Out[23]: $x+3+y$

In [24]:
```
3*x + Frac(4,3)*y - x / 2
```

Out[24]: $\frac{5}{2}x+\frac{4}{3}y$

Again, be extra careful when operating with hardware floating-point types:

In [25]:
```
1.47 * x
```

Out[25]: $\frac{6620291452234629}{4503599627370496}x$

The exponentiation operator is overloaded too:

In [26]:
```
(x/2 - y/3)**5
```

Out[26]: $\frac{5}{36}x^3y^2+\frac{5}{162}xy^4-\frac{5}{54}x^2y^3-\frac{1}{243}y^5-\frac{5}{48}x^4y+\frac{1}{32}x^5$

In [27]:
```
(x/2 - y/3)**20
```

Out[27]: $-\frac{1615}{4251528}x^7y^{13}+\frac{20995}{4478976}x^{12}y^8-\frac{95}{86093442}x^3y^{17}+\frac{95}{774840978}x^2y^{18}+\frac{1}{1048576}x^{20}-\frac{323}{9565938}x^5y^{15}+\frac{1615}{497664}x^{14}y^6-\frac{10}{116226}$

Series division will generate an error:

In [28]:
```
x/y
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-28-c9175151d842> in <module>()
----> 1 x/y

TypeError: unsupported operand type(s) for /: '_polynomial_2' and '_polynomial_2'
```

In order to represent a symbol raised to a negative power, you can use the ** operator with a negative integer argument:

```
In [29]: x * y**-1
```

Out[29]: $\frac{x}{y}$

Equality and inequality operators are also supported:

```
In [30]: pt('x') == pt('x')

Out[30]: True

In [31]: pt('x') != pt('y')

Out[31]: True

In [32]: pt(1) == 1

Out[32]: True

In [33]: pt(1) == 1.4

Out[33]: False

In [34]: (x+y)**3 == (x+y) * (x+y)**2

Out[34]: True
```

Trigonometric operations

Poisson series types support basic trigonometric operations (sine and cosine). The functions are defined in the pyranha `math` submodule:

```
In [35]: pst = poisson_series.get_type('polynomial_rational')
         x,y,z = pst('x'),pst('y'),pst('z')
         z * math.cos(2*x+3*y)
```

Out[35]: $z\cos\left(2x+3y\right)$

Note that series are always kept in a fully expanded *canonical* form (i.e., there are no `expand` or `reduce` functions):

```
In [36]: (z * math.cos(2*x+3*y) + math.sin(x))**5
```

Out[36]: $\left(\frac{15}{8}z^4+\frac{5}{8}+\frac{15}{4}z^2\right)\sin\left(x\right)-\frac{5}{8}z^2\sin\left(7x+6y\right)+\left(-\frac{15}{8}z^3-\frac{5}{4}z\right)\cos\left(4x+3y\right)+\left(-\frac{5}{4}z^2-\frac{5}{16}\right)\sin\left(3x\right)+\frac{5}{16}z\cos\left(6x+3y\right)+$

Only integer trigonometric multipliers are supported:

```
In [37]: math.cos(x/2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-37-ceaf93ad01df> in <module>()
----> 1 math.cos(x/2)

/usr/local/lib/python3.2/site-packages/pyranhapp0x/math.py in cos(arg)
     58                 pass
     59         from ._core import _cos
---> 60         return _cpp_type_catcher(_cos,arg)
     61 
     62 def sin(arg):

/usr/local/lib/python3.2/site-packages/pyranhapp0x/_common.py in _cpp_type_catcher(func, *args)
     26 def _cpp_type_catcher(func,*args):
     27         try:
---> 28                 return func(*args)
     29         except TypeError:
     30                 raise TypeError("invalid argument type(s)")

ValueError:
function: call_impl
where: /home/yardbird/repos/piranhapp0x/pyranha/../src/series.hpp, 1920
what: series is unsuitable for the calculation of cosine
```

Notice that the first trigonometric variable in alphabetical order always has the multiplier set to a positive value:

```
In [38]: math.cos(x)
```

Out[38]: $\cos(x)$

```
In [39]: math.cos(-x)
```

Out[39]: $\cos(x)$

```
In [40]: math.sin(-x)
```

Out[40]: $-\sin(x)$

Pyranha's mathematical functions are overloaded to accept also other types:

```
In [41]: math.cos(1.23)

Out[41]: 0.3342377271245026
```

```
In [42]: math.cos(mpf('1.2345'))

Out[42]: mpf('0.32999315767856785')
```

```
In [43]: mp.dps = 50
         math.cos(mpf('1.2345'))

Out[43]: mpf('0.32999315767856777128701503207778413231195886790641109')
```

Series attributes

Both polynomials and Poisson series expose a set of methods to query their properties. Similarly to `lists` and other Python containers, the `len` function will return the number of terms in the series:

```
In [44]: pt = polynomial.get_type('rational')
         x,y = pt('x'),pt('y')
         len((x+y)**10)

Out[44]: 11
```

```
In [45]: pst = poisson_series.get_type('polynomial_rational')
         a,b,c = pst('a'),pst('b'),pst('c')
         len((a+3*math.cos(b+c))**10)

Out[45]: 11
```

Something looks a bit suspicious. Shouldn't the Poisson series have more terms than that? Let's take a closer look:

```
In [46]: (x+y)**10
```

Out[46]: $x^{10}+45x^2y^8+120x^3y^7+10xy^9+y^{10}+210x^4y^6+10x^9y+45x^8y^2+210x^6y^4+120x^7y^3+252x^5y^5$

```
In [47]: (a+3*math.cos(b+c))**10
```

Out[47]: $\left(\frac{405}{2}a^8+\frac{3720087}{256}+a^{10}+\frac{10333575}{128}a^2+\frac{382725}{8}a^4+\frac{25515}{4}a^6\right)+\frac{59049}{512}\cos(10b+10c)+\left(\frac{885735}{128}a+\frac{32805}{8}a^3\right)\cos(7b+$

The `len` function returns the length of the *top-level* series. The lengths of the polynomials that constitute the series' coefficients are not taken into account. We will see later how it is possible to interact with coefficient series.

Polynomials and Poisson series have *degree*-like properties, which can be queried with a set of *methods*:

```
In [48]: x.degree()

Out[48]: 1
```

```
In [49]: ((x+1)**4).degree()

Out[49]: 4
```

```
Out[49]: 4
```

Low degree:

```
In [50]: ((x+1)**4).ldegree()
```

```
Out[50]: 0
```

Often it is useful to query the *partial* (low) degree:

```
In [51]: alpha = pt(r'\alpha')
         (1+alpha*x)**4
```

Out[51]: $6\alpha^2x^2 + 1 + \alpha^4x^4 + 4\alpha x + 4\alpha^3x^3$

```
In [52]: ((1+alpha*x)**4).degree()
```

```
Out[52]: 8
```

```
In [53]: ((1+alpha*x)**4).degree(['x'])
```

```
Out[53]: 4
```

```
In [54]: ((x+y)**4)
```

Out[54]: $4x^3y + 6x^2y^2 + y^4 + 4xy^3 + x^4$

```
In [55]: ((x+y)**4).ldegree(['y'])
```

```
Out[55]: 0
```

Poisson series also have degree and low degree properties:

```
In [56]: ((a+3*math.cos(b+c))**10).degree()
```

```
Out[56]: 10
```

```
In [57]: ((a+3*math.cos(b+c))**10).ldegree()
```

```
Out[57]: 0
```

```
In [58]: ((a+3*math.cos(b+c))**10).degree(['b','c'])
```

```
Out[58]: 0
```

```
In [59]: ((a+3*math.cos(b+c))**10).degree(['a'])
```

```
Out[59]: 10
```

Additionally, Poisson series have *trigonometric* degree and *order*:

```
In [60]: math.cos(a).t_degree()
```

```
Out[60]: 1
```

```
In [61]: math.cos(a-b).t_degree()
```

```
Out[61]: 0
```

```
In [62]: math.cos(a-b).t_degree(['a'])
```

```
Out[62]: 1
```

```
In [63]: math.cos(a-b).t_ldegree(['b'])
```

```
Out[63]: -1
```

The trigonometric order is calculated similarly to the degree, but taking into account the *absolute values* of the trigonometric multipliers:

```
In [64]: math.cos(a).t_order()
```

```
Out[64]: 1
```

In [65]:
```
math.cos(a-b).t_order()
```
Out[65]:
```
2
```

In [66]:
```
math.cos(a-b).t_order(['b'])
```
Out[66]:
```
1
```

In [67]:
```
(math.cos(a-b) + math.cos(a)).t_lorder(['b'])
```
Out[67]:
```
0
```

Differential operators

Poisson series and polynomials support partial differentiation and integration. These capabilities are available both as methods and as functions in the `math` submodule.

Partial differentiation is performed via `partial()`:

In [68]:
```
x.partial('x')
```
Out[68]: 1

In [69]:
```
math.partial(x,'x')
```
Out[69]: 1

In [70]:
```
((x+1)**10).partial('x')
```
Out[70]: $10+90x+360x^2+840x^3+1260x^4+1260x^5+840x^6+360x^7+90x^8+10x^9$

In [71]:
```
((x+1)**10).partial('y')
```
Out[71]: 0

In [72]:
```
((a/11+math.cos(5*a-3*b))**4).partial('a')
```
Out[72]: $\left(\frac{4}{14641}a^3+\frac{6}{121}a\right)-\frac{5}{2}\sin\left(20a-12b\right)-\frac{15}{11}a\sin\left(15a-9b\right)+\frac{1}{11}\cos\left(15a-9b\right)+\left(-\frac{20}{1331}a^3-\frac{15}{11}a\right)\sin\left(5a-3b\right)+\frac{6}{121}a\,\mathrm{co}$

In [73]:
```
((a/11+math.cos(5*a-3*b))**4).partial('b')
```
Out[73]: $\left(3+\frac{18}{121}a^2\right)\sin\left(10a-6b\right)+\frac{3}{2}\sin\left(20a-12b\right)+\frac{9}{11}a\sin\left(15a-9b\right)+\left(\frac{12}{1331}a^3+\frac{9}{11}a\right)\sin\left(5a-3b\right)$

Integration is performed by `integrate()`:

In [74]:
```
x.integrate('x')
```
Out[74]: $\frac{1}{2}x^2$

In [75]:
```
math.integrate(x,'x')
```
Out[75]: $\frac{1}{2}x^2$

In [76]:
```
math.integrate(x,'y')
```
Out[76]: xy

In [77]:
```
((x+y*x)**10).partial('x').integrate('x') - (x+y*x)**10
```
Out[77]: 0

Note that you can integrate even when the variable is both a polynomial and a trigonometric variable:

In [78]:
```
((a/11+a*math.cos(5*a-2*b))**3).partial('a').integrate('a') - (a/11+a*math.cos(5*a-2*b))**3
```
Out[78]: 0

The polynomial and Poisson series rings are **not** closed under integration:

```
In [79]: (x**-1).integrate('x')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-79-3a0e3dd1fb1d> in <module>()
----> 1 (x**-1).integrate('x')

ValueError:
function: integrate
where: /home/yardbird/repos/piranhapp0x/pyranha/../src/monomial.hpp, 410
what: unable to perform monomial integration: negative unitary exponent
```

Evaluation

Evaluation is the substitution of symbolic quantities for numerical values. We can map symbolic quantities to numerical values using the *dictionary* data structure:

```
In [80]: d = {'x':1,'y':2,'z':3}
```

We can then pass the mapping dictionary to the `evaluate()` method:

```
In [81]: pt = polynomial.get_type('rational')
         x,y,z = pt('x'),pt('y'),pt('z')
         (x + y + z).evaluate(d)

Out[81]: Fraction(6, 1)
```

In this case, the mapping dictionary d contains integer numerical values, which, combined with the rational coefficients of the polynomial, will produce a rational result.

If we define d in terms of double-precision floats, the result will be different:

```
In [82]: d = {'x':1.,'y':2.,'z':3.}
         (x + y + z).evaluate(d)

Out[82]: 6.0
```

Similarly, we can use rationals and arbitrary-precision floats for evaluation:

```
In [83]: d = {'x':Frac(1,2),'y':Frac(3,4),'z':Frac(5,6)}
         (x + y + z).evaluate(d)

Out[83]: Fraction(25, 12)
```

```
In [84]: d = {'x':mpf(1.2),'y':mpf(3.4),'z':mpf(5.6)}
         (x + y + z).evaluate(d)

Out[84]: mpf('10.199999999999999511501869164931122213602066040039063')
```

Note that:

- you **cannot mix** different types in the evaluation dictionary,
- all symbols must be present in the evaluation dictionary.

```
In [85]: d = {'x':1,'y':2.3,'z':mpf(5.6)}
         (x + y + z).evaluate(d)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-85-55782837e9e5> in <module>()
      1 d = {'x':1,'y':2.3,'z':mpf(5.6)}
----> 2 (x + y + z).evaluate(d)

/usr/local/lib/python3.2/site-packages/pyranhapp0x/_common.py in _evaluate_wrapper(self, d)
     76         t_set = set([type(d[k]) for k in d])
     77         if not len(t_set) == 1:
---> 78                 raise TypeError('all values in the evaluation dictionary must be of the same type')
     79         return _cpp_type_catcher(self._evaluate,d,d[list(d.keys())[0]])
     80
```

```
TypeError: all values in the evaluation dictionary must be of the same type
```

In [86]:
```
d = {'x':1,'y':2}
(x + y + z).evaluate(d)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-86-7299081f12ec> in <module>()
      1 d = {'x':1,'y':2}
----> 2 (x + y + z).evaluate(d)

/usr/local/lib/python3.2/site-packages/pyranhapp0x/_common.py in _evaluate_wrapper(self, d)
     77         if not len(t_set) == 1:
     78                 raise TypeError('all values in the evaluation dictionary must be of the same type')
---> 79         return _cpp_type_catcher(self._evaluate,d,d[list(d.keys())[0]])
     80
     81 # Register the evaluate wrapper for a particular series.

/usr/local/lib/python3.2/site-packages/pyranhapp0x/_common.py in _cpp_type_catcher(func, *args)
     26 def _cpp_type_catcher(func,*args):
     27         try:
---> 28                 return func(*args)
     29         except TypeError:
     30                 raise TypeError("invalid argument type(s)")

ValueError:
function: evaluate
where: /home/yardbird/repos/piranhapp0x/pyranha/../src/monomial.hpp, 526
what: cannot evaluate monomial: symbol 'z' does not appear in dictionary
```

Note that, when working with Poisson series, you might be forced to use floating-point numbers for evaluation. For instance:

In [87]:
```
pst = poisson_series.get_type('polynomial_rational')
a,b,c = pst('a'),pst('b'),pst('c')
d = {'a':1,'b':2}
(a+b).evaluate(d)
```

Out[87]:
```
Fraction(3, 1)
```

But:

In [88]:
```
(math.cos(a+b)).evaluate(d)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-88-484ca04c587e> in <module>()
----> 1 (math.cos(a+b)).evaluate(d)

/usr/local/lib/python3.2/site-packages/pyranhapp0x/_common.py in _evaluate_wrapper(self, d)
     77         if not len(t_set) == 1:
     78                 raise TypeError('all values in the evaluation dictionary must be of the same type')
---> 79         return _cpp_type_catcher(self._evaluate,d,d[list(d.keys())[0]])
     80
     81 # Register the evaluate wrapper for a particular series.

/usr/local/lib/python3.2/site-packages/pyranhapp0x/_common.py in _cpp_type_catcher(func, *args)
     26 def _cpp_type_catcher(func,*args):
     27         try:
---> 28                 return func(*args)
     29         except TypeError:
     30                 raise TypeError("invalid argument type(s)")

ValueError:
function: operator()
where: /home/yardbird/repos/piranhapp0x/pyranha/../src/integer.hpp, 2033
what: cannot calculate the cosine of a nonzero integer
```

In [89]:
```
d = {'a':1.,'b':2.}
(math.cos(a+b)).evaluate(d)
```

Out[89]:
```
-0.9899924966004454
```

Substitution

Substitution is conceptually similar to evaluation, but it involves the replacement of *one* symbol for a series object. It is achieved through the `subs()` method:

```
In [90]: pt = polynomial.get_type('rational')
         x,y,z = pt('x'),pt('y'),pt('z')
         (x + y + z).subs('z',x)
```

Out[90]: $y+2x$

```
In [91]: (x + y + z).subs('z',x).subs('y',x)
```

Out[91]: $3x$

```
In [92]: (x + y + z).subs('z',(x/2 + y/3)**3)
```

Out[92]: $\frac{1}{4}x^2y+x+\frac{1}{8}x^3+y+\frac{1}{6}xy^2+\frac{1}{27}y^3$

Substitution works also on Poisson series, as long as the result is still a Poisson series:

```
In [93]: pst = poisson_series.get_type('polynomial_rational')
         a,b,c = pst('a'),pst('b'),pst('c')
         a*math.cos(a+b)
```

Out[93]: $a\cos\left(a+b\right)$

```
In [94]: (a*math.cos(a+b)).subs('a',b+3*c)
```

Out[94]: $\left(b+3c\right)\cos\left(2b+3c\right)$

```
In [95]: (a*math.cos(a+b)).subs('a',b+c/2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-95-0486e380f78f> in <module>()
----> 1 (a*math.cos(a+b)).subs('a',b+c/2)

ValueError:
function: call_impl
where: /home/yardbird/repos/piranhapp0x/pyranha/../src/series.hpp, 1920
what: series is unsuitable for the calculation of cosine
```

There are two other types of substitution. The first one, `ipow_subs()`, will replace *integral powers* of a variable:

```
In [96]: x+x**2+x**3+x**4
```

Out[96]: $x^4+x+x^2+x^3$

```
In [97]: (x+x**2+x**3+x**4).ipow_subs('x',2,y)
```

Out[97]: $x+y+y^2+xy$

The second one, `t_subs()`, is available only for Poisson series, and it will replace *cosine* and *sine* of a variable:

```
In [98]: math.cos(a+b)
```

Out[98]: $\cos\left(a+b\right)$

```
In [99]: s = pst('s')
         math.cos(a+b).t_subs('a',c,s)
```

Out[99]: $c\cos\left(b\right)-s\sin\left(b\right)$