Measurements.jl Documentation

Release 0.2.2

Mose' Giordano

October 17, 2016

Contents

Measurements.jl is a package that allows you to define numbers with uncertainties, perform calculations involving them, and easily get the uncertainty of the result according to linear error propagation theory. This library is written in Julia, a modern high-level, high-performance dynamic programming language designed for technical computing.

When used in the Julia interactive session, it can serve also as an easy-to-use calculator.

The main features of the package are:

- Support for most mathematical operations available in Julia standard library involving real and complex numbers. All existing functions that accept `AbstractFloat` (and `Complex{AbstractFloat}` as well) arguments and internally use already supported functions can in turn perform calculations involving numbers with uncertainties without being redefined. This greatly enhances the power of `Measurements.jl` without effort for the users
- Functional correlation between variables is correctly handled, so $x - x \approx 0 \pm 0$, $x/x \approx 1 \pm 0$, $\tan(x) \approx \sin(x)/\cos(x)$, $\mathrm{cis}(x) \approx \exp(ix)$, etc...
- Support for arbitrary precision (also called multiple precision) numbers with uncertainties. This is useful for measurements with very low relative error
- Define arrays of measurements and perform calculations with them. Some linear algebra functions work out-of-the-box
- Propagate uncertainty for any function of real arguments (including functions based on C/Fortran calls), using `@uncertain` macro
- Functions to get the derivative and the gradient of an expression with respect to one or more independent measurements
- Functions to calculate standard score and weighted mean
- Parse strings to create measurement objects
- Easy way to attach the uncertainty to a number using the $\pm$ sign as infix operator
- Combined with external packages allows for error propagation of measurements with their physical units

Further features are expected to come in the future, see the "How Can I Help?" section and the TODO list.

CHAPTER 1

Installation

`Measurements.jl` is available for Julia 0.4 and later versions, and can be installed with Julia built-in package manager. In a Julia session run the commands

```
julia> Pkg.update()
julia> Pkg.add("Measurements")
```

CHAPTER 2

Usage

After installing the package, you can start using it with

```
using Measurements
```

The module defines a new `Measurement` data type. `Measurement` objects can be created with the two following constructors:

`measurement` (*value*, *uncertainty*)

`value ± uncertainty`

where

- `value` is the nominal value of the measurement
- `uncertainty` is its uncertainty, assumed to be a standard deviation.

They are both subtype of `AbstractFloat`. Some keyboard layouts provide an easy way to type the ± sign, if your does not, remember you can insert it in Julia REPL with `\pm` followed by `TAB` key. You can provide `value` and `uncertainty` of any subtype of `Real` that can be converted to `AbstractFloat`. Thus, `measurement(42, 33//12)` and `pi ± 0.1` are valid.

`measurement(value)` creates a `Measurement` object with zero uncertainty, like mathematical constants. See below for further examples.

Note: Every time you use one of the constructors above you define a *new independent* measurement. Instead, when you perform mathematical operations involving `Measurement` objects you create a quantity that is not independent, but rather depends on really independent measurements.

Most mathematical operations are instructed, by operator overloading, to accept `Measurement` type, and uncertainty is calculated exactly using analityc expressions of functions' derivatives.

In addition, it is possible to create a `Complex` measurement with `complex(measurement(a, b), measurement(c, d))`.

Those interested in the technical details of the package, in order integrate the package in their workflow, can have a look at the technical appendix.

`measurement` (*string*)

`measurement` function has also a method that enables you to create a `Measurement` object from a string. See the **'Examples'_** section for details.

Caution: The ± infix operator is a convenient symbol to define quantities with uncertainty, but can lead to unexpected results if used in elaborate expressions involving many ±s. Use parantheses where appropriate to avoid confusion. See for example the following cases:

```
7.5±1.2 + 3.9±0.9 # This is wrong!
# => 11.4 ± 1.2 ± 0.9 ± 0.0
(7.5±1.2) + (3.9±0.9) # This is correct
# => 11.4 ± 1.5
```

2.1 Correlation Between Variables

The fact that two or more measurements are correlated means that there is some sort of relationship beetween them. In the context of measurements and error propagation theory, the term "correlation" is very broad and can indicate different things. Among others, there may be some dependence between uncertainties of different measurements with different values, or a dependence between the values of two measurements while their uncertainties are different.

Here, for correlation we mean the most simple case of functional relationship: if $x = \bar{x} \pm \sigma_x$ is an independent measurement, a quantity $y = f(x) = \bar{y} \pm \sigma_y$ that is function of x is not like an independent measurement but is a quantity that depends on x, so we say that y is correlated with x. The package `Measurements.jl` is able to handle this type of correlation when propagating the uncertainty for operations and functions taking two or more arguments. As a result, $x - x = 0 \pm 0$ and $x/x = 1 \pm 0$. If this correlation was not accounted for, you would always get non-zero uncertainties even for these operations that have exact results. Two truly different measurements that only by chance share the same nominal value and uncertainty are not treated as correlated.

2.2 Propagate Uncertainty for Arbitrary Functions

`@uncertain f(x, ...)`

Existing functions implemented exclusively in Julia that accept `AbstractFloat` arguments will work out-of-the-box with `Measurement` objects as long as they internally use functions already supported by this package. However, there are functions that take arguments that are specific subtypes of `AbstractFloat`, or are implemented in such a way that does not play nicely with `Measurement` variables.

The package provides the `@uncertain` macro that overcomes this limitation and further extends the power of `Measurements.jl`.

This macro allows you to propagate uncertainty in arbitrary functions, including those based on C/Fortran calls, that accept any number of real arguments. The macro exploits `derivative` and `gradient` functions from Calculus package in order to perform numerical differentiation.

2.3 Derivative and Gradient

`Measurements.`**`derivative`**(*y::Measurement*, *x::Measurement*)

`Measurements.`**`gradient`**(*y::Measurement*, *x::AbstractArray{Measurement}*)

In order to propagate the uncertainties, `Measurements.jl` keeps track of the partial derivative of an expression with respect to all independent measurements from which the expression comes. For this reason, the package provides two convenient functions, `Measurements.derivative` and `Measurements.gradient`, to get the partial derivative and the gradient of an expression with respect to independent measurements.

2.4 Standard Score

stdscore(*measure::Measurement*, *expected_value::Real*) → standard_score

The `stdscore` function is available to calculate the standard score between a measurement and its expected value.

2.5 Weighted Average

weightedmean(*iterable*) → weighted_mean

`weightedmean` function gives the weighted mean of a set of measurements using inverses of variances as weights. Use `mean` for the simple arithmetic mean.

2.6 Access Nominal Value and Uncertainty

value(*x*)

uncertainty(*x*)

As explained in the technical appendix, the nominal value and the uncertainty of `Measurement` objects are stored in `val` and `err` fields respectively, but you do not need to use those field directly to access this information. Functions `value` and `uncertainty` allow you to get the nominal value and the uncertainty of `x`, be it a single measurement or an array of measurements. They are particularly useful in the case of complex measurements or arrays of measurements.

2.7 Error Propagation of Numbers with Units

`Measurements.jl` does not know about units of measurements, but can be easily employed in combination with other Julia packages providing this feature. Thanks to the type system of Julia programming language this integration is seamless and comes for free, no specific work has been done by the developer of the present package nor by the developers of the above mentioned packages in order to support their interplay. They all work equally good with `Measurements.jl`, you can choose the library you prefer and use it. Note that only algebraic functions are allowed to operate with numbers with units of measurement, because transcendental functions operate on dimensionless quantities. In the Examples section you will find how this feature works with a couple of packages.

CHAPTER 3

Examples

These are some basic examples of use of the package:

```
using Measurements
a = measurement(4.5, 0.1)
# => 4.5 ± 0.1
b = 3.8 ± 0.4
# => 3.8 ± 0.4
2a + b
# => 12.8 ± 0.4472135954999579
a - 1.2b
# => -0.05999999999999961 ± 0.49030602688525043
l = measurement(0.936, 1e-3);
T = 1.942 ± 4e-3;
g = 4pi^2*l/T^2
# => 9.797993213510699 ± 0.041697817535336676
c = measurement(4)
# => 4.0 ± 0.0
a*c
# => 18.0 ± 0.4
sind(94 ± 1.2)
# => 0.9975640502598242 ± 0.0014609761696991563
x = 5.48 ± 0.67;
y = 9.36 ± 1.02;
log(2x^2 - 3.4y)
# =>  3.3406260917568824 ± 0.5344198747546611
atan2(y, x)
# => 1.0411291003154137 ± 0.07141014208254456
```

3.1 Measurements from Strings

You can construct `Measurement` objects from strings. Within parentheses there is the uncertainty referred to the corresponding last digits.

```
measurement("-123.4(56)")
# => -123.4 ± 5.6
measurement("+1234(56)e-1")
# => ->  123.4 ± 5.6
measurement("12.34e-1 +- 0.56e1")
# => 123.4 ± 5.6
measurement("(-1.234 ± 0.056)e2")
# => -123.4 ± 5.6
```

```
measurement("1234e-1 +/- 5.6e0")
# => 123.4 ± 5.6
measurement("-1234e-1")
# => -123.4 ± 0.0
```

3.2 Correlation Between Variables

Here you can see examples of how functionally correlated variables are treated within the package:

```
x = 8.4 ± 0.7
x - x
# => 0.0 ± 0.0
x/x
# => 1.0 ± 0.0
x*x*x - x^3
# => 0.0 ± 0.0
sin(x)/cos(x) - tan(x)
# => -2.220446049250313e-16 ± 0.0
# They are equal within numerical accuracy
y = -5.9 ± 0.2
beta(x, y) - gamma(x)*gamma(y)/gamma(x + y)
# => 0.0 ± 3.979039320256561e-14
```

You will get similar results for a variable that is a function of an already existing `Measurement` object:

```
u = 2x
(x + x) - u
# => 0.0 ± 0.0
u/2x
# => 1.0 ± 0.0
u^3 - 8x^3
# => 0.0 ± 0.0
cos(x)^2 - (1 + cos(u))/2
# => 0.0 ± 0.0
```

A variable that has the same nominal value and uncertainty as `u` above but is not functionally correlated with `x` will give different outcomes:

```
# Define a new measurement but with same nominal value
# and uncertainty as u, so v is not correlated with x
v = 16.8 ± 1.4
(x + x) - v
# => 0.0 ± 1.979898987322333
v/2x
# => 1.0 ± 0.11785113019775792
v^3 - 8x^3
# => 0.0 ± 1676.4200705455657
cos(x)^2 - (1 + cos(v))/2
# => 0.0 ± 0.8786465354843539
```

3.3 `@uncertain` Macro

Macro `@uncertain` can be used to propagate uncertainty in arbitrary real or complex functions of real arguments, including functions not natively supported by this package.

```
@uncertain (x -> complex(zeta(x), exp(eta(x)^2)))(2 ± 0.13)
# => (1.6449340668482273 ± 0.12188127308075564) + (1.9668868646839253 ± 0.042613944993428333)im
@uncertain log(9.4 ± 1.3, 58.8 ± 3.7)
# => 1.8182372640255153 ± 0.11568300475873611
log(9.4 ± 1.3, 58.8 ± 3.7)
# => 1.8182372640255153 ± 0.11568300475593848
```

You usually do not need to define a wrapping function before using it. In the case where you have to define a function, like in the first line of previous examples, anonymous functions allow you to do it in a very concise way.

The macro works with functions calling C/Fortran functions as well. For example, Cuba.jl package performs numerical integration by wrapping the C Cuba library. You can define a function to numerically compute with `Cuba.jl` the integral defining the error function and pass it to `@uncertain` macro. Compare the result with that of the `erf` function, natively supported in `Measurements.jl` package

```
using Cuba
cubaerf(x::Real) =
    2x/sqrt(pi)*cuhre((t, f) -> f[1] = exp(-abs2(t[1]*x)), 1, 1)[1][1]
@uncertain cubaerf(0.5 ± 0.01)
# => 0.5204998778130466 ± 0.008787825789336267
erf(0.5 ± 0.01)
# => 0.5204998778130465 ± 0.008787825789354449
```

Also here you can use an anonymous function instead of defining the `cubaerf` function, do it as an exercise.

Tip: Note that the argument of `@uncertain` macro must be a function call whose arguments are `Measurement` objects. Thus,

```
@uncertain zeta(13.4 ± 0.8) + eta(8.51 ± 0.67)
```

will not work because here the outermost function is +, whose arguments are `zeta(13.4 ± 0.8)` and `eta(8.51 ± 0.67)`, that however cannot be calculated. Once more, wrap this expression in an (anonymous) function:

```
@uncertain ((x, y) -> zeta(x) + eta(y))(13.4 ± 0.8, 8.51 ± 0.67)
# => 1.9974303172187315 ± 0.0012169293212062773
```

The type of *all* the arguments provided must be `Measurement`. If one of the arguments is actually an exact number (so without uncertainty), convert it to `Measurement` type:

```
atan2(10, 13.5 ± 0.8)
# => 0.6375487981386927 ± 0.028343666961913202
@uncertain atan2(10 ± 0, 13.5 ± 0.8)
# => 0.6375487981386927 ± 0.028343666962347438
```

In addition, the function must be differentiable in all its arguments. For example, the scaled first derivative of the Airy Ai function $\mathrm{airyx}(1, z) = \exp((2/3)z\sqrt{z})\mathrm{Ai}'(z)$ is not differentiable in the first argument, not even the trick of passing an exact measurement would work because the first argument must be an integer. You can easily work around this limitation by wrapping the function in a single-argument function

```
@uncertain (x -> airyx(1, x))(4.8 ± 0.2)
# => -0.42300740589773583 ± 0.004083414330362105
```

3.4 Complex Measurements

Here are a few examples about uncertainty propagation of complex-valued measurements.

```
u = complex(32.7 ± 1.1, -3.1 ± 0.2)
v = complex(7.6 ± 0.9, 53.2 ± 3.4)
2u+v
# => (73.0 ± 2.3769728648009427) + (47.0 ± 3.4234485537247377)im
sqrt(u*v)
# => (33.004702573592 ± 1.0831254428098636) + (25.997507418428984 ± 1.1082833691607152)im
gamma(u/v)
# => (-0.25050193836584694 ± 0.011473098558745594) + (1.2079738483289788 ± 0.133606565257322)im
```

You can also verify the Euler's formula

```
cis(u)
# => (6.27781144696534 ± 23.454542573739754) + (21.291738410228678 ± 8.112997844397572)im
cos(u) + sin(u)*im
# => (6.277811446965339 ± 23.454542573739754) + (21.291738410228678 ± 8.112997844397572)im
```

3.5 Arbitrary Precision Calculations

If you performed an exceptionally good experiment that gave you extremely precise results (that is, with very low relative error), you may want to use arbitrary precision (or multiple precision) calculations, in order not to loose significance of the experimental results. Luckily, Julia natively supports this type of arithmetic and so `Measurements.jl` does. You only have to create `Measurement` objects with nominal value and uncertainty of type `BigFloat`.

Tip: As explained in the Julia documentation, it is better to use the `big` string literal to initialize an arbitrary precision floating point constant, instead of the `BigFloat` and `big` functions. See examples below.

For example, you want to measure a quantity that is the product of two observables a and b, and the expected value of the product is 12.00000007. You measure $a = 3.00000001 \pm (1 \times 10^{-17})$ and $b = 4.00000001 \pm (1 \times 10^{-17})$ and want to compute the standard score of the product with *`stdscore()`*. Using the ability of `Measurements.jl` to perform arbitrary precision calculations you discover that

```
a = big"3.00000001" ± big"1e-17"
b = big"4.00000001" ± big"1e-17"
stdscore(a*b, 12.00000007)
# => -7.255109014397189800954688841706490473843234068878544115810990031483656163515 48
```

the measurement significantly differs from the expected value and you make a great discovery. Instead, if you used double precision accuracy, you would have wrongly found that your measurement is consistent with the expected value:

```
stdscore((3.00000001 ± 1e-17)*(4.00000001 ± 1e-17), 12.00000007)
# => 0.0
```

and you would have missed an important prize due to the use of an incorrect arithmetic.

Of course, you can perform any mathematical operation supported in `Measurements.jl` using arbitrary precision arithmetic:

```
hypot(a, b)
# => 5.000000014000000000399999998880000003119999991353600023834879934652928178154746 ± 9.9999999999
log(2a)^b
# => 1.030668110995484938037006520012324656386442805506891265153048683619922226691323e+01 ± 9.744450
```

3.6 Operations with Arrays and Linear Algebra

You can create arrays of `Measurement` objects and perform mathematical operations on them in the most natural way possible:

```
A = [1.03 ± 0.14, 2.88 ± 0.35, 5.46 ± 0.97]
B = [0.92 ± 0.11, 3.14 ± 0.42, 4.67 ± 0.58]
exp(sqrt(B)) - log(A)
# => 3-element Array{Measurements.Measurement{Float64},1}:
#      2.5799612193837493 ± 0.20215123893809778
#      4.824843081566397 ± 0.7076631767039828
#      6.982522998771525 ± 1.178287422979362
cos(A).^2 + sin(A).^2
# 3-element Array{Measurements.Measurement{Float64},1}:
#      1.0 ± 0.0
#      1.0 ± 0.0
#      1.0 ± 0.0
```

If you originally have separate arrays of values and uncertainties, you can create an array of `Measurement` objects by providing `measurement` with those arrays:

```
C = measurement([174.9, 253.8, 626.1], [12.2, 19.4, 38.5])
# => 3-element Array{Measurements.Measurement{Float64},1}:
#      174.9 ± 12.2
#      253.8 ± 19.4
#      626.1 ± 38.5
sum(C)
# => 1054.8000000000002 ± 44.80457565918909
mean(C)
# => 351.6000000000001 ± 14.93485855306303
```

Tip: `prod` and `sum` (and `mean`, which relies on `sum`) functions work out-of-the-box with any iterable of `Measurement` objects, like arrays or tuples. However, these functions have faster methods (quadratic in the number of elements) when operating on an array of "Measurement"s than on a tuple (in this case the computational complexity is cubic in the number of elements), so you should use an array if performance is crucial for you, in particular for large collections of measurements.

Some linear algebra functions work out-of-the-box, without defining specific methods for them. For example, you can solve linear systems, do matrix multiplication and dot product between vectors, find inverse, determinant, and trace of a matrix, do QR factorization, etc.

```
A = [(14 ± 0.1) (23 ± 0.2); (-12 ± 0.3) (24 ± 0.4)]
b = [(7 ± 0.5), (-13 ± 0.6)]
# Solve the linear system Ax = b
x = A \ b
# => 2-element Array{Measurements.Measurement{Float64},1}:
#       0.763072 ± 0.0313571
#      -0.160131 ± 0.0177963
# Verify this is the correct solution of the system
A * x # This should be equal to `b`
# => 2-element Array{Measurements.Measurement{Float64},1}:
#        7.0 ± 0.5
#      -13.0 ± 0.6
dot(x, b)
# 7.423202614379084 ± 0.5981875954418516
det(A)
```

```
# => 611.9999999999999 ± 9.51262319236918
trace(A)
# => 38.0 ± 0.4123105625617661
A * inv(A)  eye(A)
# => true
qrfact(A)
# => Base.LinAlg.QR{Measurements.Measurement{Float64},Array{Measurements.Measurement{Float64},2}}(Mea
```

3.7 Derivative and Gradient

In order to propagate the uncertainties, `Measurements.jl` keeps track of the partial derivative of an expression with respect to all independent measurements from which the expression comes. The package provides two convenient functions, `Measurements.derivative` and `Measurements.gradient`, that return the partial derivative and the gradient of an expression with respect to independent measurements.

```
x = 98.1 ± 12.7
y = 105.4 ± 25.6
z = 78.3 ± 14.1
Measurements.derivative(2x - 4y, x)
# => 2.0
Measurements.derivative(2x - 4y, y)
# => -4.0
Measurements.gradient(2x - 4y, [x, y, z])
# => 3-element Array{Float64,1}:
#       2.0
#      -4.0
#       0.0  # The expression does not depend on z
```

Tip: The `Measurements.gradient` function is useful in order to discover which variable contributes most to the total uncertainty of a given expression, if you want to minimize it. This can be calculated as the Hadamard (element-wise) product between the gradient of the expression with respect to the set of variables and the vector of uncertainties of the same variables in the same order. For example:

```
w = y^(3//4)*log(y) + 3x - cos(y/x)
# => 447.0410543780643 ± 52.41813324207829
(Measurements.gradient(w, [x, y]) .* uncertainty([x, y])).^2
# => 2-element Array{Any,1}:
#      1442.31
#      1305.36
```

In this case, the `x` variable contributes most to the uncertainty of `w`. In addition, note that the Euclidean norm of the Hadamard product above is exactly the total uncertainty of the expression:

```
vecnorm(Measurements.gradient(w, [x, y]) .* uncertainty([x, y]))
# => 52.41813324207829
```

3.8 `stdscore` Function

You can get the distance in number of standard deviations between a measurement and its expected value (this can be with or without uncertainty) using `stdscore`:

```
stdscore(1.3 ± 0.12, 1)
# => 2.5000000000000004
stdscore(4.7 ± 0.58, 5 ± 0.01)
# => -0.5172413793103445 ± 0.017241379310344827
```

3.9 `weightedmean` Function

Calculate the weighted and arithmetic means of your set of measurements with `weightedmean` and `mean` respectively:

```
weightedmean((3.1±0.32, 3.2±0.38, 3.5±0.61, 3.8±0.25))
# => 3.4665384454054498 ± 0.16812474090663868
mean((3.1±0.32, 3.2±0.38, 3.5±0.61, 3.8±0.25))
# => 3.4000000000000004 ± 0.2063673908348894
```

3.10 `value` and `uncertainty` Functions

Use `value` and `uncertainty` to get the values and uncertainties of measurements. They work with real and complex measurements, scalars or arrays:

```
value(94.5 ± 1.6)
# => 94.5
uncertainty(94.5 ± 1.6)
# => 1.6
value([complex(87.3 ± 2.9, 64.3 ± 3.0), complex(55.1 ± 2.8, -19.1 ± 4.6)])
# => 2-element Array{Complex{Float64},1}:
#      87.3+64.3im
#      55.1-19.1im
uncertainty([complex(87.3 ± 2.9, 64.3 ± 3.0), complex(55.1 ± 2.8, -19.1 ± 4.6)])
# => 2-element Array{Complex{Float64},1}:
#      2.9+3.0im
#      2.8+4.6im
```

3.11 Use with `SIUnits.jl` and `Unitful.jl`

You can use `Measurements.jl` in combination with a third-party package in order to perform calculations involving physical measurements, i.e. numbers with uncertainty and physical unit. The details depend on the specific package adopted. Such packages are, for instance, SIUnits.jl and Unitful.jl. You only have to use the `Measurement` object as the value of the `SIQuantity` object (for `SIUnits.jl`) or of the `Quantity` object (for `Unitful.jl`). Here are a few examples.

```
using Measurements, SIUnits, SIUnits.ShortUnits
hypot((3 ± 1)*m, (4 ± 2)*m) # Pythagorean theorem
# => 5.0 ± 1.7088007490635064 m
(50 ± 1)Ω * (13 ± 2.4)*1e-2*A # Ohm's Law
# => 6.5 ± 1.20702112657567 kg m²s³A¹
2pi*sqrt((5.4 ± 0.3)*m / ((9.81 ± 0.01)*m/s^2)) # Pendulum's  period
# => 4.661677707464357 ± 0.1295128435999655 s

using Measurements, Unitful
hypot((3 ± 1)*u"m", (4 ± 2)*u"m") # Pythagorean theorem
```

```
# => 5.0 ± 1.7088007490635064 m
(50 ± 1)*u"Ω" * (13 ± 2.4)*1e-2*u"A" # Ohm's Law
# => 6.5 ± 1.20702112657567 A Ω
2pi*sqrt((5.4 ± 0.3)*u"m" / ((9.81 ± 0.01)*u"m/s^2")) # Pendulum's period
# => 4.661677707464357 ± 0.12951284359996548 s
```

CHAPTER 4

Development

The package is developed at https://github.com/giordano/Measurements.jl. There you can submit bug reports, make suggestions, and propose pull requests.

4.1 How Can I Help?

Have a look at the *TODO* list below and the bug list at https://github.com/giordano/Measurements.jl/issues, pick-up a task, write great code to accomplish it and send a pull request. In addition, you can instruct more mathematical functions to accept `Measurement` type arguments. Please, read the technical appendix in order to understand the design of this package. Bug reports and wishlists are welcome as well.

4.2 TODO

- Add pretty printing: optionally print only the relevant significant digits (issue #5)
- Other suggestions welcome `:-)`

4.3 History

The ChangeLog of the package is available in NEWS.md file in top directory. There have been some breaking changes from time to time, beware of them when upgrading the package.

CHAPTER 5

Appendix: Technical Details

This technical appendix explains the design of `Measurements.jl` package, how it propagates the uncertainties when performing calculations, and how you can contribute by providing new methods for mathematical operations.

5.1 The `Measurement` Type

`Measurement` is a composite parametric type, whose parameter is the `AbstractFloat` subtype of the nominal value and the uncertainty of the measurement. `Measurement` type itself is subtype of `AbstractFloat`, thus `Measurement` objects can be used in any function taking `AbstractFloat` arguments without redefining it, and calculation of uncertainty will be exact.

In detail, this is the definition of the type:

```
immutable Measurement{T<:AbstractFloat} <: AbstractFloat
    val::T
    err::T
    tag::Float64
    der::Derivatives{Tuple{T, T, Float64}, T}
end
```

The fields represent:

- `val`: the nominal value of the measurement
- `err`: the uncertainty, assumed to be standard deviation
- `tag`: a unique identifier, it is used to identify a specific measurement in the list of derivatives. This is automatically created with `rand`. The result of mathematical operation will have this field set to `NaN` because it is not relevant for non independent measurements.
- `der`: the list of derivates with respect to the independent variables from which the expression comes. `Derivatives` is a lightweight dictionary type. The keys are the tuples `(val, err, tag)` of all independent variables from which the object has been derived, while the corresponding value is the partial derivative of the object with respect to that independent variable.

As already explained in the "Usage" section, every time you use one of the constructors

```
measurement(value, uncertainty)
value ± uncertainty
```

you define a *new independent* measurement. This happens because these contructors generate a new random and (hopefully) unique `tag` field, that is used to distinguish between really equal objects and measurements that only by chance share the same nominal value and uncertainty. For these reasons,

```
x = 24.3 ± 2.7
y = 24.3 ± 2.7
```

will produce two independent measurements and they will be treated as such when performing mathematical operations. In particular, you can also notice that they are not egal

```
x === y
# => false
```

If you instead intend to make `y` really the same thing as `x` you have to use assignment:

```
x = y = 24.3 ± 2.7
x === y
# => true
```

Thanks to how the Julia language is designed, support for complex measurements, arbitrary precision calculations and array operations came with practically no effort during the development of the package. As explained by Steven G. Johnson, since in Julia a lot of nonlinear functions are internally implemented in terms of elementary operations on the real and imaginary parts it was natural to make the type subtype of `Real` in order to easily work with complex measurements. In particular, it was then chosen to select the `AbstractFloat` type because some functions of complex arguments (like `sqrt` and `log`) take `Complex{AbstractFloat}` arguments instead of generic `Complex{Real}`, and any operation on a `Measurement{R}` object, with `R` subtype of `Real` different from `AbstractFloat`, would turn it into `Measurement{F}`, with `F` subtype of `AbstractFloat`, anyway.

5.2 Correlation

One must carefully take care of correlation between two measurements when propagating the uncertainty for an operation. Actually, the term "correlation" may refer to different kind of dependences between two or more quantities, what we mean by this term in `Measurements.jl` is explained in the "Usage" section of this manual.

Dealing with functional correlation between `Measurement` objects, when using functions with arity larger than one, is an important feature of this package. This is accomplished by keeping inside each `Measurement` object the list of its derivatives with respect to the independent variables from which the quantity comes, and this list is updated every time a new mathematical operation is performed. This role is played by the `der` field. This dictionary is useful in order to trace the contribution of each measurement and propagate the uncertainty in the case of functions with more than one argument.

The use of the list of derivatives has been inspired by Python package uncertainties, but the rest of the implementation of `Measurements.jl` is completely independent from that of `uncertainties` package, even though it may happen to be similar.

5.3 Uncertainty Propagation

For a function $G(a, b, c, \dots)$ of real arguments with uncertainties $a = \bar{a} \pm \sigma_a$, $b = \bar{b} \pm \sigma_b$, and $c = \bar{c} \pm \sigma_c$, ..., the linear error propagation theory prescribes that uncertainty is propagated as follows:

$$\begin{aligned}\sigma_G^2 = &\left(\left.\frac{\partial G}{\partial a}\right|_{a=\bar{a}} \sigma_a\right)^2 + \left(\left.\frac{\partial G}{\partial b}\right|_{b=\bar{b}} \sigma_b\right)^2 + \left(\left.\frac{\partial G}{\partial c}\right|_{c=\bar{c}} \sigma_c\right)^2 + \cdots \\ &+2\left(\frac{\partial G}{\partial a}\right)_{a=\bar{a}} \left(\frac{\partial G}{\partial b}\right)_{b=\bar{b}} \sigma_{ab} + 2\left(\frac{\partial G}{\partial a}\right)_{a=\bar{a}} \left(\frac{\partial G}{\partial c}\right)_{c=\bar{c}} \sigma_{ac} \\ &+2\left(\frac{\partial G}{\partial b}\right)_{b=\bar{b}} \left(\frac{\partial G}{\partial c}\right)_{c=\bar{c}} \sigma_{bc} + \dots\end{aligned}$$

where the σ_{ab} factors are the covariances defined as

$$\sigma_{ab} = \mathrm{E}[(a - \mathrm{E}[a])(b - \mathrm{E}[b])]$$

$E[a]$ is the expected value, or mean, of a. If uncertainties of the quantities a, b, c, ..., are independent and normally distributed, the covariances are null and the above formula for uncertainty propagation simplifies to

$$\sigma_G^2 = \left(\left.\frac{\partial G}{\partial a}\right|_{a=\bar{a}} \sigma_a \right)^2 + \left(\left.\frac{\partial G}{\partial b}\right|_{b=\bar{b}} \sigma_b \right)^2 + \left(\left.\frac{\partial G}{\partial c}\right|_{c=\bar{c}} \sigma_c \right)^2 + \cdots$$

In general, calculating the covariances is not an easy task. The trick adopted in `Measurements.jl` in order to deal with simple functional correlation is to propagate the uncertainty always using really independent variables. Thus, dealing with functional correlation boils down to finding the set of all the independent measurements on which an expression depends. If this set is made up of $\{x, y, z, \dots\}$, it is possible to calculate the uncertainty of $G(a, b, c, \dots)$ with

$$\sigma_G^2 = \left(\left.\frac{\partial G}{\partial x}\right|_{x=\bar{x}} \sigma_x \right)^2 + \left(\left.\frac{\partial G}{\partial y}\right|_{y=\bar{y}} \sigma_y \right)^2 + \left(\left.\frac{\partial G}{\partial z}\right|_{z=\bar{z}} \sigma_z \right)^2 + \cdots$$

where all covariances due to functional correlation are null. This explains the purpose of keeping the list of derivatives with respect to independent variables in `Measurement` objects: by looking at the `der` fields of a, b, c, ..., it is possible to determine the set of independent variables. If other types of correlation (not functional) between x, y, z, ..., are present, they should be treated by calculating the covariances as shown above.

For a function of only one argument, $G = G(a)$, there is no problem of correlation and the uncertainty propagation formula in the linear approximation simply reads

$$\sigma_G = \left| \left.\frac{\partial G}{\partial a}\right|_{a=\bar{a}} \right| \sigma_a$$

even if a is not an independent variable and comes from operations on really independent measurements.

For example, suppose you want to calculate the function $G = G(a, b)$ of two arguments, and a and b are functionally correlated, because they come from some mathematical operations on really independent variables x, y, z, say $a = a(x, y)$, $b = b(x, z)$. By using the chain rule, the uncertainty on $G(a, b)$ is calculated as follows:

$$\sigma_G^2 = \left(\left(\frac{\partial G}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial G}{\partial b}\frac{\partial b}{\partial x} \right)_{x=\bar{x}} \sigma_x \right)^2 + \left(\left(\frac{\partial G}{\partial a}\frac{\partial a}{\partial y} \right)_{y=\bar{y}} \sigma_y \right)^2 + \left(\left(\frac{\partial G}{\partial b}\frac{\partial b}{\partial z} \right)_{z=\bar{z}} \sigma_z \right)^2$$

What `Measurements.jl` really does is to calulate the derivatives like $\partial a/\partial x$ and $\partial G/\partial x = (\partial G/\partial a)(\partial a/\partial x) + (\partial G/\partial b)(\partial b/\partial x)$, and store them in the `der` field of a and G respectively in order to be able to perform further operations involving these quantities.

5.4 Defining Methods for Mathematical Operations

`Measurements.jl` defines new methods for mathematical operations in order to make them accept `Measurement` arguments. The single most important thing to know about how to define new methods in the package is the `Measurements.result`. This function, not exported because it is intended to be used only within the package, takes care of propagating the uncertainty as described in the section above. It has two methods: one for functions with arity equal to one, and the other for any other case. This is its syntax:

```
result(val::Real, der::Real, a::Measurement)
```

for functions of one argument, and

```
result(val::Real, der::Tuple{Vararg{Real}},
       a::Tuple{Vararg{Measurement}})
```

for functions of two or more arguments. The arguments are:

- `val`: the nominal result of the operation $G(a, \dots)$;
- `der`: the partial derivative $\partial G/\partial a$ of a function $G = G(a)$ with respect to the argument a for one-argument functions or the tuple of partial derivatives with respect to each argument in other cases;
- `a`: the argument(s) of G, in the same order as the corresponding derivatives in `der` argument.

In the case of functions with arity larger than one, `der` and `a` tuples must have the same length.

For example, for a one-argument function like $\cos$ we have

```
cos(a::Measurement) = result(cos(a.val), -sin(a.val), a)
```

Instead, the method for subtraction operation is defined as follows:

```
-(a::Measurement, b::Measurement) =
    result(a.val - b.val, (1.0, -1.0), (a, b))
```

Thus, in order to support `Measurement` argument(s) for a new mathematical operation you have to calculate the result of the operation, the partial derivatives of the functon with respect to all arguments and then pass this information to `Measurements.result` function.

M

S

U

V

W