
Measurements.jl Documentation

Release 0.4.1-dev

Mose' Giordano

May 31, 2017

1	Installation	3
2	Usage	5
2.1	Correlation Between Variables	6
2.2	Propagate Uncertainty for Arbitrary Functions	6
2.3	Derivative and Gradient	7
2.4	Uncertainty Contribution	7
2.5	Standard Score	7
2.6	Weighted Average	7
2.7	Access Nominal Value and Uncertainty	7
2.8	Error Propagation of Numbers with Units	8
2.9	Printing to TeX and LaTeX MIMEs	8
3	Examples	9
3.1	Measurements from Strings	10
3.2	Correlation Between Variables	10
3.3	@uncertain Macro	12
3.4	Complex Measurements	13
3.5	Arbitrary Precision Calculations	13
3.6	Operations with Arrays and Linear Algebra	14
3.7	Derivative, Gradient and Uncertainty Components	16
3.8	stdscore Function	17
3.9	weightedmean Function	17
3.10	Measurements.value and Measurements.uncertainty Functions	18
3.11	Interplay with Third-Party Packages	18
4	Performance	21
5	Development	25
5.1	How Can I Help?	25
5.2	TODO	25
5.3	History	25
6	Appendix: Technical Details	27
6.1	The Measurement Type	27
6.2	Correlation	28
6.3	Uncertainty Propagation	29

6.4 Defining Methods for Mathematical Operations	30
Bibliography	31

`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 and special functions from [SpecialFunctions.jl](#) package, 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 $x - x \approx 0 \pm 0$, $x/x \approx 1 \pm 0$, $\tan(x) \approx \sin(x)/\cos(x)$, $\text{cis}(x) \approx \exp(ix)$, 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`
- Function 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. This syntactic sugar makes the code more readable and visually appealing
- Combined with external packages allows for error propagation of measurements with their [physical units](#)
- Extensible in combination with external packages: you can propagate errors of measurements with their physical units, perform numerical integration with [QuadGK.jl](#), numerical and automatic differentiation, and much more.

The method used to handle functional correlation is described in this paper:

If you use this package for your research, please cite it.

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

The `Measurements.jl` package is licensed under the MIT “Expat” License. The original author is Mosè Giordano.

CHAPTER 1

Installation

`Measurements.jl` is available for Julia 0.6 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")
```

Older versions are also available for Julia 0.4 and Julia 0.5.

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 \pm 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 analytic expressions of functions' derivatives.

It is also possible to create a `Complex` measurement with

```
complex(measurement(real_part_value, real_part_uncertainty), measurement(imaginary_
↳ part_value, imaginary_part_uncertainty))
```

In addition to making the code prettier, the fact that the \pm sign can be used as infix operator to define new independent `Measurement`s makes the printed representation of these objects valid Julia syntax, so you can quickly copy the output of an operation in the Julia REPL to perform other calculations. Note however that the copied number will not be the *same* object as the original one, because it will be a *new independent* measurement, without memory of the correlations of the original object.

This module extends many methods defined in Julia’s mathematical standard library, and some methods from widespread third-party packages as well. This is the case for most special functions in `SpecialFunctions.jl` package, and the `quadgk` integration routine from `QuadGK.jl` package.

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

```
julia> 7.5±1.2 + 3.9±0.9 # This is wrong!
11.4 ± 1.2 ± 0.9 ± 0.0

julia> (7.5±1.2) + (3.9±0.9) # This is correct
11.4 ± 1.5
```

Correlation Between Variables

The fact that two or more measurements are correlated means that there is some sort of relationship between 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.

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.

Derivative and Gradient

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

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 a convenient function, `Measurements.derivative`, to get the partial derivative and the gradient of an expression with respect to independent measurements.

Uncertainty Contribution

`Measurements.uncertainty_components` ($x::Measurement$)

You may want to inspect which measurement contributes most to the total uncertainty of a derived quantity, in order to minimize it, if possible. The function `Measurements.uncertainty_components` gives you a dictionary whose values are the components of the uncertainty of x .

Standard Score

`stdscore` ($measure::Measurement, expected_value$) \rightarrow `standard_score`

`stdscore` ($measure_1::Measurement, measure_2::Measurement$) \rightarrow `standard_score`

The `stdscore` function is available to calculate the [standard score](#) between a measurement and its expected value (not a `Measurement`). When both arguments are `Measurement` objects, the standard score between their difference and zero is computed, in order to test their compatibility.

Weighted Average

`weightedmean` ($iterable$) \rightarrow `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.

Access Nominal Value and Uncertainty

`Measurements.value` (x)

`Measurements.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 `Measurements.value` and `Measurements.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.

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.

Printing to TeX and LaTeX MIMES

You can print `Measurement` objects to TeX and LaTeX MIMES ("`text/x-tex`" and "`text/x-latex`"), the \pm sign will be rendered with `\pm` command:

```
julia> display("text/x-tex", 5±1)
5.0 \pm 1.0

julia> display("text/x-latex", pi ± 1e-3)
3.141592653589793 \pm 0.001
```

CHAPTER 3

Examples

These are some basic examples of use of the package:

```
julia> using Measurements

julia> a = measurement(4.5, 0.1)
4.5 ± 0.1

julia> b = 3.8 ± 0.4
3.8 ± 0.4

julia> 2a + b
12.8 ± 0.4472135954999579

julia> a - 1.2b
-0.05999999999999961 ± 0.49030602688525043

julia> l = measurement(0.936, 1e-3);

julia> T = 1.942 ± 4e-3;

julia> g = 4pi^2*l/T^2
9.797993213510699 ± 0.041697817535336676

julia> c = measurement(4)
4.0 ± 0.0

julia> a*c
18.0 ± 0.4

julia> sind(94 ± 1.2)
0.9975640502598242 ± 0.0014609761696991563

julia> x = 5.48 ± 0.67;

julia> y = 9.36 ± 1.02;
```



```

julia> x = 8.4 ± 0.7
8.4 ± 0.7

julia> x - x
0.0 ± 0.0
julia> x/x
1.0 ± 0.0

julia> x*x*x - x^3
0.0 ± 0.0

julia> sin(x)/cos(x) - tan(x)
-2.220446049250313e-16 ± 0.0
# They are equal within numerical accuracy

julia> y = -5.9 ± 0.2

julia> 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:

```

julia> u = 2x

julia> (x + x) - u
0.0 ± 0.0

julia> u/2x
1.0 ± 0.0

julia> u^3 - 8x^3
0.0 ± 0.0

julia> 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
julia> v = 16.8 ± 1.4

julia> (x + x) - v
0.0 ± 1.979898987322333

julia> v / 2x
1.0 ± 0.11785113019775792
julia> v^3 - 8x^3
0.0 ± 1676.4200705455657

julia> cos(x)^2 - (1 + cos(v))/2
0.0 ± 0.8786465354843539

```

@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.

```
julia> @uncertain (x -> complex(zeta(x), exp(eta(x)^2)))(2 ± 0.13)
(1.6449340668482273 ± 0.12188127308075564) + (1.9668868646839253 ± 0.
↪042613944993428333)im

julia> @uncertain log(9.4 ± 1.3, 58.8 ± 3.7)
1.8182372640255153 ± 0.11568300475873611

julia> 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

```
julia> using Cuba

julia> cubaerf(x::Real) =
    2x/sqrt(pi)*cuhre((t, f) -> f[1] = exp(-abs2(t[1]*x)))[1][1]
cubaerf (generic function with 1 method)

julia> @uncertain cubaerf(0.5 ± 0.01)
0.5204998778130466 ± 0.008787825789336267

julia> 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. Remember that if you want to numerically integrate a function that returns a `Measurement` object you can use `QuadGK.jl` package, which is written purely in Julia and in addition allows you to set `Measurement` objects as endpoints, see below.

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

```
julia> @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. You can use the `@uncertain` macro on each function separately:

```
julia> @uncertain(zeta(13.4 ± 0.8)) + @uncertain(eta(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), promote it to `Measurement` type:

```
julia> atan2(10, 13.5 ± 0.8)
0.6375487981386927 ± 0.028343666961913202
```



```
julia> @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 polygamma function of order m , `polygamma(m, x)`, is the $m + 1$ -th derivative of the logarithm of gamma function, and 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:

```
julia> @uncertain (x -> polygamma(0, x))(4.8 ± 0.2)
1.4608477407291167 ± 0.046305812845734776

julia> digamma(4.8 ± 0.2) # Exact result
1.4608477407291167 ± 0.04630581284451362
```

Complex Measurements

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

```
julia> u = complex(32.7 ± 1.1, -3.1 ± 0.2)

julia> v = complex(7.6 ± 0.9, 53.2 ± 3.4)

julia> 2u + v
(73.0 ± 2.3769728648009427) + (47.0 ± 3.4234485537247377)im

julia> sqrt(u * v)
(33.004702573592 ± 1.0831254428098636) + (25.997507418428984 ± 1.1082833691607152)im
```

You can also verify the Euler's formula

```
julia> cis(u)
(6.27781144696534 ± 23.454542573739754) + (21.291738410228678 ± 8.112997844397572)im

julia> cos(u) + sin(u)*im
(6.277811446965339 ± 23.454542573739754) + (21.291738410228678 ± 8.112997844397572)im
```

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 lose 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 $b = 4.0000000100000001 \pm (1 \times 10^{-17})$

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

```
julia> a = big"3.00000001" ± big"1e-17"

julia> b = big"4.0000000100000001" ± big"1e-17"

julia> stdscore(a * b, big"12.00000007")
7.999999997599999878080000420160000093695993825308195353920411656927305928530607
```

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:

```
julia> stdscore((3.00000001 ± 1e-17)*(4.0000000100000001 ± 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:

```
julia> hypot(a, b)
5.000000014000000080399999974880000423919999216953595312794907845334503498479533 ± 1.
↪0000000000000000000000000000000000000000000000000000000000000000000000000009e-17

julia> log(2a) ^ b
1.030668110995484998145373137400169442058573718746529435800255440973153647087416e+01 ↪
↪± 9.744450581349822034766870718391736028419817951565653507621645979913795265663606e-
↪17
```

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:

```
julia> A = [1.03 ± 0.14, 2.88 ± 0.35, 5.46 ± 0.97]
3-element Array{Measurements.Measurement{Float64},1}:
 1.03±0.14
 2.88±0.35
 5.46±0.97

julia> B = [0.92 ± 0.11, 3.14 ± 0.42, 4.67 ± 0.58]
3-element Array{Measurements.Measurement{Float64},1}:
 0.92±0.11
 3.14±0.42
 4.67±0.58

julia> exp.(sqrt.(B)) .- log.(A)
3-element Array{Measurements.Measurement{Float64},1}:
 2.57996±0.202151
 4.82484±0.707663
 6.98252±1.17829

julia> @. 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 using `measurement` or `±` with the `dot` syntax for vectorizing functions:

```
julia> C = measurement.([174.9, 253.8, 626.3], [12.2, 19.4, 38.5])
3-element Array{Measurements.Measurement{Float64},1}:
 174.9±12.2
 253.8±19.4
 626.3±38.5

julia> sum(C)
1055.0 ± 44.80457565918909

julia> D = [549.4, 672.3, 528.5] .± [7.4, 9.6, 5.2]
3-element Array{Measurements.Measurement{Float64},1}:
 549.4±7.4
 672.3±9.6
 528.5±5.2

julia> mean(D)
583.4 ± 4.396463225012679
```

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 `Measurements` 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 LU and QR factorization, etc.

```
julia> A = [(14 ± 0.1) (23 ± 0.2); (-12 ± 0.3) (24 ± 0.4)]
2×2 Array{Measurements.Measurement{Float64},2}:
 14.0±0.1  23.0±0.2
-12.0±0.3  24.0±0.4

julia> b = [(7 ± 0.5), (-13 ± 0.6)]
2-element Array{Measurements.Measurement{Float64},1}:
 7.0±0.5
-13.0±0.6

# Solve the linear system Ax = b
julia> 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
julia> A * x b
true
```

```

julia> dot(x, b)
7.423202614379084 ± 0.5981875954418516

julia> det(A)
611.9999999999999 ± 9.51262319236918

julia> trace(A)
38.0 ± 0.4123105625617661

julia> A * inv(A) eye(A)
true

julia> lufact(A)
Base.LinAlg.LU{Measurements.Measurement{Float64},Array{Measurements.Measurement
↪{Float64},2}} with factors L and U:
Measurements.Measurement{Float64}[1.0±0.0 0.0±0.0; -0.857143±0.0222861 1.0±0.0]
Measurements.Measurement{Float64}[14.0±0.1 23.0±0.2; 0.0±0.0 43.7143±0.672403]

julia> qrfact(A)
Base.LinAlg.QR{Measurements.Measurement{Float64},Array{Measurements.Measurement
↪{Float64},2}}(Measurements.Measurement{Float64}[-18.4391±0.209481 -1.84391±0.522154;
↪ -0.369924±0.00730266 33.1904±0.331267],Measurements.Measurement{Float64}[1.
↪75926±0.00836088,0.0±0.0])

```

Derivative, Gradient and Uncertainty Components

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 a convenient function, `Measurements.derivative()`, that returns the partial derivative of an expression with respect to independent measurements. Its vectorized version can be used to compute the gradient of an expression with respect to multiple independent measurements.

```

julia> x = 98.1 ± 12.7
98.1 ± 12.7

julia> y = 105.4 ± 25.6
105.4 ± 25.6

julia> z = 78.3 ± 14.1
78.3 ± 14.1

julia> Measurements.derivative(2x - 4y, x)
2.0

julia> Measurements.derivative(2x - 4y, y)
-4.0

julia> Measurements.derivative.(log1p(x) + y^2 - cos(x/y), [x, y, z])
3-element Array{Float64,1}:
 0.0177005
210.793
 0.0      # The expression does not depend on z

```

Tip: The vectorized version of `Measurements.derivative()` 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:

```
julia> w = y^(3//4)*log(y) + 3x - cos(y/x)
447.0410543780643 ± 52.41813324207829

julia> abs.(Measurements.derivative.(w, [x, y]) .* Measurements.uncertainty.([x, y]))
2-element Array{Float64,1}:
 37.9777
 36.1297
```

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:

```
julia> vecnorm(Measurements.derivative.(w, [x, y]) .* Measurements.uncertainty.([x, y]))
52.41813324207829
```

The `Measurements.uncertainty_components()` function simplifies calculation of all uncertainty components of a derived quantity:

```
julia> Measurements.uncertainty_components(w)
Dict{Tuple{Float64,Float64,Float64},Float64} with 2 entries:
 (98.1, 12.7, 0.303638) => 37.9777
 (105.4, 25.6, 0.465695) => 36.1297

julia> vecnorm(collect(values(Measurements.uncertainty_components(w))))
52.41813324207829
```

stdscore Function

You can get the distance in number of standard deviations between a measurement and its expected value (not a Measurement) using `stdscore()`:

```
julia> stdscore(1.3 ± 0.12, 1)
2.5000000000000004
```

You can use the same function also to test the consistency of two measurements by computing the standard score between their difference and zero. This is what `stdscore()` does when both arguments are Measurement objects:

```
julia> stdscore((4.7 ± 0.58) - (5 ± 0.01), 0)
-0.5171645175253433

julia> stdscore(4.7 ± 0.58, 5 ± 0.01)
-0.5171645175253433
```

weightedmean Function

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

```
julia> weightedmean((3.1±0.32, 3.2±0.38, 3.5±0.61, 3.8±0.25))
3.4665384454054498 ± 0.16812474090663868

julia> mean((3.1±0.32, 3.2±0.38, 3.5±0.61, 3.8±0.25))
3.4000000000000004 ± 0.2063673908348894
```

Measurements.value and Measurements.uncertainty Functions

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

```
julia> Measurements.value(94.5 ± 1.6)
94.5

julia> Measurements.uncertainty(94.5 ± 1.6)
1.6

julia> Measurements.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

julia> Measurements.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
```

Interplay with Third-Party Packages

`Measurements.jl` works out-of-the-box with any function taking arguments no more specific than `AbstractFloat`. This makes this library particularly suitable for cooperating with well-designed third-party packages in order to perform complicated calculations always accurately taking care of uncertainties and their correlations, with no effort for the developers nor users.

The following sections present a sample of packages that are known to work with `Measurements.jl`, but many others will interplay with this package as well as them.

Numerical Integration with `QuadGK.jl`

The powerful integration routine `quadgk` from `QuadGK.jl` package is smart enough to support out-of-the-box integrand functions that return arbitrary types, including `Measurement`:

```
julia> QuadGK.quadgk(x -> exp(x / (4.73 ± 0.01)), 1, 7)
(14.933307243306032 ± 0.009999988180463411, 0.0 ± 0.010017961523508253)
```

`Measurements.jl` pushes the capabilities of `quadgk` further by supporting also `Measurement` objects as endpoints:

```
julia> QuadGK.quadgk(cos, 1.19 ± 0.02, 8.37 ± 0.05)
(-0.05857827689796702 ± 0.02576650561689427, 2.547162480937004e-11)
```

Compare this with the expected result:

```
julia> sin(8.37 ± 0.05) - sin(1.19 ± 0.02)
-0.058578276897966686 ± 0.02576650561689427
```

Also with `quadgk` correlation is properly taken into account:

```
julia> a = 6.42 ± 0.03
6.42 ± 0.03

julia> QuadGK.quadgk(sin, -a, a)
(2.484178227707412e-17 ± 0.0, 0.0)
```

If instead the two endpoints have, by chance, the same nominal value and uncertainty but are not correlated:

```
julia> QuadGK.quadgk(sin, -6.42 ± 0.03, 6.42 ± 0.03)
(2.484178227707412e-17 ± 0.005786464233000303, 0.0)
```

Numerical and Automatic Differentiation

With `Calculus.jl` package it is possible to perform numerical differentiation using finite differencing. You can pass in to the `Calculus.derivative` function both functions returning `Measurement` objects and a `Measurement` as the point in which to calculate the derivative.

```
julia> using Measurements, Calculus

julia> a = -45.7 ± 1.6
-45.7 ± 1.6

julia> b = 36.5 ± 6.0
36.5 ± 6.0

julia> Calculus.derivative(exp, a) exp(a)
true

julia> Calculus.derivative(cos, b) -sin(b)
true

julia> Calculus.derivative(t -> log(-t * b)^2, a) 2log(-a * b)/a
true
```

Other packages provide automatic differentiation methods. Here is an example with `AutoGrad.jl`, just one of the packages available:

```
julia> using AutoGrad

julia> grad(exp)(a) exp(a)
true

julia> grad(cos)(b) -sin(b)
true
```

```
julia> grad(t -> log(-t * b)^2)(a) 2log(-a * b)/a
true
```

However remember that you can always use `Measurements.derivative()` to compute the value (without uncertainty) of the derivative of a `Measurement` object.

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.

```
julia> using Measurements, SIUnits, SIUnits.ShortUnits

julia> hypot((3 ± 1)*m, (4 ± 2)*m) # Pythagorean theorem
5.0 ± 1.7088007490635064 m

julia> (50 ± 1)Ω * (13 ± 2.4)*1e-2*A # Ohm's Law
6.5 ± 1.20702112657567 kg m2s3A1

julia> 2pi*sqrt((5.4 ± 0.3)*m / ((9.81 ± 0.01)*m/s2)) # Pendulum's period
4.661677707464357 ± 0.1295128435999655 s

julia> using Measurements, Unitful

julia> hypot((3 ± 1)*u"m", (4 ± 2)*u"m") # Pythagorean theorem
5.0 ± 1.7088007490635064 m

julia> (50 ± 1)*u"Ω" * (13 ± 2.4)*1e-2*u"A" # Ohm's Law
6.5 ± 1.20702112657567 A Ω

julia> 2pi*sqrt((5.4 ± 0.3)*u"m" / ((9.81 ± 0.01)*u"m/s2")) # Pendulum's period
4.661677707464357 ± 0.12951284359996548 s
```


CHAPTER 4

Performance

`Measurements.jl` strives to be as fast as possible. These are the benchmark results obtained with the `BenchmarkTools.jl` suite on a system equipped with an Intel(R) Core(TM) i7-4700MQ CPU running Julia 0.6.0-pre.beta.314 (commit 7cd0324e03).

```
julia> using Measurements, BenchmarkTools

# Creation of a `Measurement` object
julia> @benchmark 4.7 ± 0.3
BenchmarkTools.Trial:
  memory estimate: 96 bytes
  allocs estimate: 2
  -----
  minimum time:      18.760 ns (0.00% GC)
  median time:       22.548 ns (0.00% GC)
  mean time:         29.324 ns (16.03% GC)
  maximum time:      1.353 μs (93.61% GC)
  -----
  samples:           10000
  evals/sample:      997

julia> a = 12.3 ± 4.5; b = 67.8 ± 9.0;

# Sum of two `Measurement` objects
julia> @benchmark $a + $b
BenchmarkTools.Trial:
  memory estimate: 176 bytes
  allocs estimate: 4
  -----
  minimum time:      76.605 ns (0.00% GC)
  median time:       83.412 ns (0.00% GC)
  mean time:         102.682 ns (13.96% GC)
  maximum time:      2.339 μs (92.91% GC)
  -----
  samples:           10000
  evals/sample:      970
```

```
# One-argument functions, where functional
# correlation is not a concern, are even faster
julia> @benchmark sqrt($b)
BenchmarkTools.Trial:
  memory estimate: 96 bytes
  allocs estimate: 2
  -----
  minimum time:      31.226 ns (0.00% GC)
  median time:       33.805 ns (0.00% GC)
  mean time:         41.899 ns (13.09% GC)
  maximum time:      1.656 μs (91.62% GC)
  -----
  samples:           10000
  evals/sample:      994

julia> @benchmark sin($a)
BenchmarkTools.Trial:
  memory estimate: 96 bytes
  allocs estimate: 2
  -----
  minimum time:      56.797 ns (0.00% GC)
  median time:       58.526 ns (0.00% GC)
  mean time:         67.799 ns (8.59% GC)
  maximum time:      1.860 μs (93.77% GC)
  -----
  samples:           10000
  evals/sample:      984

julia> @benchmark gamma($a)
BenchmarkTools.Trial:
  memory estimate: 96 bytes
  allocs estimate: 2
  -----
  minimum time:      136.277 ns (0.00% GC)
  median time:       140.353 ns (0.00% GC)
  mean time:         151.901 ns (4.11% GC)
  maximum time:      2.164 μs (90.13% GC)
  -----
  samples:           10000
  evals/sample:      867

# Vectorial functions take a linear time
julia> vector = [1 ± 0.1 for _ in 1:10000];

julia> @benchmark sqrt.($vector)
BenchmarkTools.Trial:
  memory estimate: 1015.70 KiB
  allocs estimate: 20002
  -----
  minimum time:      330.481 μs (0.00% GC)
  median time:       352.413 μs (0.00% GC)
  mean time:         457.011 μs (20.01% GC)
  maximum time:      2.970 ms (85.16% GC)
  -----
  samples:           10000
  evals/sample:      1
```

```
julia> @benchmark sin.($vector)
BenchmarkTools.Trial:
  memory estimate: 1015.70 KiB
  allocs estimate: 20002
  -----
  minimum time:      535.720 μs (0.00% GC)
  median time:      556.428 μs (0.00% GC)
  mean time:        669.928 μs (14.53% GC)
  maximum time:     3.263 ms (80.73% GC)
  -----
  samples:          7440
  evals/sample:     1

julia> @benchmark gamma.($vector)
BenchmarkTools.Trial:
  memory estimate: 1015.70 KiB
  allocs estimate: 20002
  -----
  minimum time:      1.097 ms (0.00% GC)
  median time:      1.145 ms (0.00% GC)
  mean time:        1.270 ms (8.19% GC)
  maximum time:     4.137 ms (67.33% GC)
  -----
  samples:          3928
  evals/sample:     1

julia> @benchmark cos.($vector) .^ 2 .+ sin.($vector) .^ 2
BenchmarkTools.Trial:
  memory estimate: 4.50 MiB
  allocs estimate: 100002
  -----
  minimum time:      2.484 ms (0.00% GC)
  median time:      2.641 ms (0.00% GC)
  mean time:        3.272 ms (19.46% GC)
  maximum time:     7.793 ms (55.38% GC)
  -----
  samples:          1526
  evals/sample:     1
```


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

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.

TODO

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

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.

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.

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{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 derivatives 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 constructors 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,

```
julia> x = 24.3 ± 2.7
24.3 ± 2.7

julia> y = 24.3 ± 2.7
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`

```
julia> x === y
false
```

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

```
julia> x = y = 24.3 ± 2.7
24.3 ± 2.7

julia> 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.

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. 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.

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:

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

where the σ_{ab} factors are the [covariances](#) defined as

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

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

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

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

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

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, \dots , it is possible to determine the set of independent variables. If other types of correlation (not functional) between x, y, z, \dots , 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

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

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 calculate 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.

This method is also described in [\[GIO16\]](#).

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, der, a)
```

for functions of two or more arguments, in which `der` and `a` are the collections (tuples, arrays, etc...) of the same length. 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, -1), (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 function with respect to all arguments and then pass this information to `Measurements.result` function.

Bibliography

[GIO16] M. Giordano, 2016, “Uncertainty propagation with functionally correlated quantities”, [arXiv:1610.08716](https://arxiv.org/abs/1610.08716)
(Bibcode: 2016arXiv161008716G)

M

measurement() (built-in function), 5, 6

Measurements.derivative() (built-in function), 7

Measurements.uncertainty() (built-in function), 7

Measurements.uncertainty_components() (built-in function), 7

Measurements.value() (built-in function), 7

S

stdscore() (built-in function), 7

W

weightedmean() (built-in function), 7