pariman

v0.2.0
https://github.com/pacaunt/pariman
Calculations with Units in Typst

Contents

1 Installation

Import the package by

typst
```typst
#import "@preview/pariman:0.1.0": *
```

Or install the package locally by cloning this package into your local package location.

2 Usage

2.1 The `quantity` function

The package provides a dictonary-based element called `quantity`. This `quantity` can be used as a number to all of the calculation functions in Pariman's framework. The quantity is declared by specify its value and unit.

typst
```typst
#let a = quantity("1.0e5", "m/s^2")
Display value and unit: #a.display \
Display only the formatted value: #a.show \
Display the raw value verbatim: #a.text \
Significant figures: #a.figures \
Decimal places: #a.places
```

Display value and unit: $1.0 \times 10^{5}\ \mathrm{m\,s^{-2}}$
Display only the formatted value: 1.0×10^{5}
Display the raw value verbatim: 1e5
Significant figures: 2
Decimal places: 0

Pariman's `quantity` takes care of the significant figure calculations and unit formatting automatically. The unit formatting functionality is provided by the zero package. Therefore, the format options for the unit can be used.

typst
```typst
#let b = quantity("1234.56", "kg m/s^2")
The formatted value and unit: #b.display  \
#zero.set-unit(fraction: "fraction")
After new fraction mode: #b.display
```

The formatted value and unit: $1234.56\ \mathrm{kg\,m\,s^{-2}}$
After new fraction mode: $1234.56\ \frac{\mathrm{kg\,m}}{\mathrm{s^2}}$

Pariman loads the `zero` package automatically, so the the unit formatting options can be modified by `zero.set-xxx` functions.

For exact values like integers, pi, or other constants, that should not be counted as significant figures, Pariman have the `#exact` function for exact number quantities. The `#exact` function has 99 significant figures and 99 decimal places, but the displayed figures and decimal places can be set by using the option `display-figures` and `display-places`.

typst

```
#let pi = exact(calc.pi)
The value: #pi.display \
Significant figures: #pi.figures \
Decimal places: #pi.places

// The shorter version
#let s-pi = exact(calc.pi, display-figures: 4)
The displayed value: #s-pi.display \
// does not effect the real significant figures
Significant figures: #s-pi.figures \
Decimal places: #s-pi.places
```

The value: 3.141 592 653 589 793
Significant figures: 99
Decimal places: 99

The displayed value: 3.142
Significant figures: 99
Decimal places: 99

Note that the `quantity` function can accept only the value for the unitless quantoity.

2.2 The `calculation` module

The `calculation` module provides a framework for calculations involving units. Every function will modify the input `quantitys` into a new value with a new unit corresponding to the law of unit relationships.

typst

```
#let s = quantity("1.0", "m")
#let t = quantity("5.0", "s")
#let v = calculation.div(s, t) // division
The velocity is given by #v.display. \
The unit is combined!
```

The velocity is given by $0.20\,\mathrm{m\,s^{-1}}$.
The unit is combined!

Moreover, each quantity also have a `method` property that can show its previous calculation.

typst

```
#let V = quantity("2.0", "cm^3")
#let d = quantity("0.89", "g/cm^3")
#let m = calculation.mul(d, V)
From $V = #V.display$, and density $d = #d.display$, we have
$ m = d V = #m.method = #m.display. $
```

From $V = 2.0\,\mathrm{cm^3}$, and density $d = 0.89\,\mathrm{g\,cm^{-3}}$, we have

$$m = dV = 0.89\,\mathrm{g\,cm^{-3}} \times 2.0\,\mathrm{cm^3} = 1.8\,\mathrm{g}.$$

The `method` property is recursive, meaning that it is accumulated if your calculation is complicated. Initially, `method` is set to `auto`.

typst

```
#let A = quantity("1.50e4", "1/s")
#let Ea = quantity("50e3", "J/mol")
#let R = quantity("8.314", "J/mol K")
#let T = quantity("298", "K")

Arrhenius equation is given by
$ k = A e^(-E_a/(R T)) $
This $k$, at $A = #A.display$, $E_a = #Ea.display$, and $T = #T.display$, we have
#let k = {
  import calculation: *
  mul(A, exp(
    div(
      neg(Ea),
      mul(R, T)
    )
  ))
}
$
  k &= #k.method \
    &= #k.display
$
```

Arrhenius equation is given by

$$k = Ae^{-\frac{E_a}{RT}}$$

This k, at $A = 1.50 \times 10^4\,\mathrm{s^{-1}}$, $E_a = 5.0 \times 10^4\,\mathrm{J\,mol^{-1}}$, and $T = 298\,\mathrm{K}$, we have

$$\begin{aligned} k &= 1.50 \times 10^4\,\mathrm{s^{-1}} \times \exp\left(\frac{-5.0 \times 10^4\,\mathrm{J\,mol^{-1}}}{8.314\,\mathrm{J\,mol^{-1}\,K^{-1}} \times 298\,\mathrm{K}}\right) \\ &= 2.58 \times 10^{-5}\,\mathrm{s^{-1}} \end{aligned}$$

2.3 `set-quantity`

If you want to manually set the formatting unit and numbers in the `quantity`, you can use the `set-quantity` function.

typst

```
#let R = quantity("8.314", "J/mol K")
#let T = quantity("298.15", "K")

#calculation.mul(R, T).display
// 4 figures, follows R (the least).

// reset the significant figures
#let R = set-quantity(R, figures: 8)
#calculation.mul(R, T).display
// 5 figures, follows the T.
```

$2479\,\mathrm{J\,mol^{-1}}$

$2478.8\,\mathrm{J\,mol^{-1}}$

Moreover, if you want to reset the `method` property of a quantity, you can use `set-quantity(q, method: auto)` as

typst

```
#let R = quantity("8.314", "J/mol K")
#let T = quantity("298.15", "K")

#let prod = calculation.mul(R, T)
Before reset:
$ prod.method = prod.display $
// reset
#let prod = set-quantity(prod, method: auto)
After reset:
$ prod.method = prod.display $
```

Before reset:

$$8.314\,\mathrm{J\,mol^{-1}\,K^{-1}} \times 298.15\,\mathrm{K} = 2479\,\mathrm{J\,mol^{-1}}$$

After reset:

$$2479\,\mathrm{J\,mol^{-1}} = 2479\,\mathrm{J\,mol^{-1}}$$

2.4 Unit conversions

The `new-factor` function creates a new quantity that can be used as a conversion factor. This conversion factor have the following characteristics:

1. It has, by default, 10 significant figures.
2. It have a method called `inv` for inverting the numerator and denominator units.

typst

```
#let v0 = quantity("60.0", "km/hr")
#let km-m = new-factor(
  quantity("1", "km"),
  quantity("1000", "m")
) // km → m

#let hr-s = new-factor(
  quantity("1", "hr"),
  quantity("3600", "s"),
) // s → hr

#let v1 = calculation.mul(v0, km-m)
First conversion, from km to m
$ v1.method = v1.display $

// change from hr → s, use `hr-s.inv` because hr is in the denominator
#let v2 = calculation.mul(v1, hr-s.inv)
Second conversion:
$ v2.method = v2.display $
```

First conversion, from km to m

$$60.0\,\mathrm{km\,hr^{-1}} \times \frac{1000\,\mathrm{m}}{1\,\mathrm{km}} = 6.00 \times 10^{4}\,\mathrm{m\,hr^{-1}}$$

Second conversion:

$$60.0\,\mathrm{km\,hr^{-1}} \times \frac{1000\,\mathrm{m}}{1\,\mathrm{km}} \times \frac{1\,\mathrm{hr}}{3600\,\mathrm{s}} = 16.7\,\mathrm{m\,s^{-1}}$$

2.5 In-Text Quantity Declaration (The `qt` Module)

This module provides a top-layer functions that makes declaration of the quantities can be done at the same time as showing the formatted quantities. Declaration can be done by `qt.new()` function, which receives the same argument set as the `quantity` constructor, but with an additional, positional argument: its key/name. This name is important because it will be used to retrieve the value declared for further calculations or updates.

typst

```
// Syntax: #qt.new(name, value, ..units)
A chemist added #qt.new("mA", "1.050", "g")
```

A chemist added $1.050\,\mathrm{g}$ of A into a beaker filled with $100\,\mathrm{mL}$ of water.

```typst
of A into a beaker filled with
#qt.new("Vw", "100", "mL") of water.
```

Moreover, this `#qt.new` function also receives the following named options:

- `displayed` (bool, default: `true`) Whether to display the declared quantity immediately.
- `is-exact` (bool, default: `false`) Whether to set the specified quantity as an exact value (like declaring by `exact` function).

To manipulate the quantities declared, we can use `#qt.update(key, function)` to update the variable that has a named `key` (same as the name specified by `#qt.new`), or create a new quantity named `key` by using a function `function`. For example,

```typst
I put a #qt.new("ms", "30.0", "g") of sugar into a #qt.new("V", "105", "mL") of water in a cup. After being stirred thoroughly, the sugar solution will have a concentration of
// import the division function
#import calculation: div
// An update to calculate the concentration!
#qt.update("conc", q => div(q.ms, q.V))
// Show the result!
$ #qt.method("conc") = #qt.display("conc") $
```

I put a 30.0 g of sugar into a 105 mL of water in a cup. After being stirred thoroughly, the sugar solution will have a concentration of

$$\frac{30.0\ \mathrm{g}}{105\ \mathrm{mL}} = 0.286\ \mathrm{g\ mL^{-1}}$$

Note that `#qt.display(key)` and `#qt.method(key)` are used as shortcut for accessing the `display` and `method` properties of the quantity identified by the name `key`. For other properties, you can access by `#qt.get(key: name)` as the following. Highlight the `context`.

```typst
#context qt.get(key: "ms")
```

```
(
  value: 30,
  unit: (("g", 1),),
  places: 1,
  figures: 3,
  show: context(),
  text: "30",
  display: context(),
  method: context(),
  source: none,
  round-mode: "places",
  is-exact: false,
)
```

Lastly, you can set the property like `set-quantity` function by using the analogous `#qt.set-property(key, ..properties)`, such as

```typst
What is the value of $pi$? \
// too long number!
It is #qt.new("pi", calc.pi, is-exact: true)  \
Oh, too long,  \
// set the displayed figure number
#qt.set-property("pi", display-figures: 4)
It is now only #qt.display("pi")
```

What is the value of π?
It is 3.141 592 653 589 793
Oh, too long,
It is now only 3.142

3 Available Calculation Methods

- `neg(a)` negate a number, returns negative value of `a`.
- `add(..q)` addition. Error if the unit of each added quantity has different units. Returns the sum of all `q`.
- `sub(a, b)` subtraction. Error if the unit of each quantity is not the same. Returns the quantity of `a - b`.
- `mul(..q)` multiplication, returns the product of all `q`.
- `div(a, b)` division, returns the quantity of `a/b`.

- `inv(a)` returns the reciprocal of `a`.
- `exp(a)` exponentiation on based e. Error if the argumenrt of e has any leftover unit. Returns a unitless `exp(a)`.
- `pow(a, n)` returns a^n. If n is not an integer, `a` must be unitless.
- `ln(a)` returns the natural log of `a`. The quantity `a` must be unitless.
- `log(a, base: 10)` returns the logarithm of `a` on base `base`. Error if `a` is not unitless.
- `root(a, n)` returns the nth root of `a`. If `n` is not an integer, then `a` must be unitless.
- `solver(func, init: none)` solves the function that is written in the form $f(x) = 0$. It returns another quantity that has the same dimension as the `init` value.