

pariman

v0.2.1

<https://github.com/pacaunt/pariman>

Calculations with Units in Typst

Contents

1	Installation	1
2	Usage	1
2.1	The <code>quantity</code> function	1
2.2	The calculation module	2
2.3	<code>set-quantity</code>	3
2.4	Unit conversions	3
2.5	In-Text Quantity Declaration (The <code>qt</code> Module)	4
3	References	5
3.1	The Constructors	5
3.2	Available Calculation Methods	9

1 Installation

Import the package by

typst

```
1 #import "@preview/pariman:0.2.1": *
```

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 dictionary-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

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

Display value and unit: $1.0 \times 10^5 \text{ m s}^{-2}$
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

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

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

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`.

```

1 #let pi = exact(calc.pi)
2 The value: #pi.display \
3 Significant figures: #pi.figures \
4 Decimal places: #pi.places
5
6 // The shorter version
7 #let s-pi = exact(calc.pi, display-figures: 4)
8 The displayed value: #s-pi.display \
9 // does not effect the real significant figures
10 Significant figures: #s-pi.figures \
11 Decimal places: #s-pi.places

```

typst

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 quantity.

2.2 The calculation module

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

```

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

```

typst

The velocity is given by 0.20 m s⁻¹.

The unit is combined!

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

```

1 #let V = quantity("2.0", "cm^3")
2 #let d = quantity("0.89", "g/cm^3")
3 #let m = calculation.mul(d, V)
4 From $V = #V.display$, and density $d =
  #d.display$, we have
5 $m = d V = #m.method = #m.display. $

```

typst

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

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

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

```

1 #let A = quantity("1.50e4", "1/s")
2 #let Ea = quantity("50e3", "J/mol")
3 #let R = quantity("8.314", "J/mol K")
4 #let T = quantity("298", "K")
5
6 Arrhenius equation is given by
7 $k = A e^{(-E_a/(R T))}$
8 This $k$, at $A = #A.display$, $E_a =
  #Ea.display$, and $T = #T.display$, we have
9 #let k = {
10   import calculation: *
11   mul(A, exp(
12     div(
13       neg(Ea),
14       mul(R, T)
15     )
16   ))
17 }
18 $
19 k &= #k.method \
20   &= #k.display
21 $

```

typst

Arrhenius equation is given by

$$k = A e^{-\frac{E_a}{RT}}$$

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

$$\begin{aligned}
 k &= 1.50 \times 10^4 \text{ s}^{-1} \times \exp\left(\frac{-5.0 \times 10^4 \text{ J mol}^{-1}}{8.314 \text{ J mol}^{-1} \text{ K}^{-1} \times 298 \text{ K}}\right) \\
 &= 2.6 \times 10^{-5} \text{ s}^{-1}
 \end{aligned}$$

Sometimes, you may want to display your calculations with units omitted. The `quantity` function has a parameter `explicit` which is `true` by default. If this parameter is set to `false`, the unit of that quantity is omitted for all of its calculation.

```

1 #let r = quantity("2.0e-5", "cm")
2 // hide the unit in `method`.
3 #let T = quantity(
4   "373", "K", explicit-method: false
5 )
6 #let M = quantity("28", "g/mol")
7
8 Knudsen Diffusivity of nitrogen gas ($M_w =
9   #M.display$) at $T = #T.display$ and pore radius
10  $r = #r.display$ is
11
12 #let D-K = {
13   import calculation: *
14   mul(
15     exact(9700), r, root(div(T, M), 2),
16     unit: "cm^2/s"
17   )
18 }
19
20 $ D_k &= #D-K.method \ &= #D-K.display $

```

Knudsen Diffusivity of nitrogen gas ($M_w = 28 \text{ g mol}^{-1}$) at $T = 373 \text{ K}$ and pore radius $r = 2.0 \times 10^{-5} \text{ cm}$ is

$$D_k = 9700 \times 2.0 \times 10^{-5} \text{ cm} \times \left(\frac{373}{28 \text{ g mol}^{-1}} \right)^{\frac{1}{2}} = 0.71 \text{ cm}^2 \text{ s}^{-1}$$

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.

```

1 #let R = quantity("8.314", "J/mol K")
2 #let T = quantity("298.15", "K")
3
4 #calculation.mul(R, T).display
5 // 4 figures, follows R (the least).
6
7 // reset the significant figures
8 #let R = set-quantity(R, figures: 8)
9 #calculation.mul(R, T).display
10 // 5 figures, follows the T.

```

2479 J mol⁻¹

2478.8 J mol⁻¹

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

```

1 #let R = quantity("8.314", "J/mol K")
2 #let T = quantity("298.15", "K")
3
4 #let prod = calculation.mul(R, T)
5 Before reset:
6 $ prod.method = prod.display $
7 // reset
8 #let prod = set-quantity(prod, method: auto)
9 After reset:
10 $ prod.method = prod.display $

```

Before reset:

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

After reset:

$$2479 \text{ J mol}^{-1} = 2479 \text{ J mol}^{-1}$$

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.

```

1 #let v0 = quantity("60.0", "km/hr")
2 #let km-m = new-factor(
3   quantity("1", "km"),
4   quantity("1000", "m")
5 ) // km → m
6
7 #let hr-s = new-factor(
8   quantity("1", "hr"),
9   quantity("3600", "s"),
10 ) // s → hr
11
12 #let v1 = calculation.mul(v0, km-m)
13 First conversion, from km to m
14 $ v1.method = v1.display $
15
16 // change from hr → s, use `hr-s.inv` because
   hr is in the denominator
17 #let v2 = calculation.mul(v1, hr-s.inv)
18 Second conversion:
19 $ v2.method = v2.display $

```

typst

First conversion, from km to m

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

Second conversion:

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

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.

```

1 // Syntax: #qt.new(name, value, ..units)
2 A chemist added #qt.new("mA", "1.050", "g")
3 of A into a beaker filled with
4 #qt.new("Vw", "100", "mL") of water.

```

typst

A chemist added 1.050 g of A into a beaker filled with 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,

```

1 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
2 // import the division function
3 #import calculation: div
4 // An update to calculate the concentration!
5 #qt.update("conc", q => div(q.ms, q.V))
6 // Show the result!
7 $ #qt.method("conc") = #qt.display("conc") $

```

typst

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

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

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.

```

1 #context qt.get(key: "ms")
(
  value: 30.0,
  display-value: 30.0,
  unit: ("g", 1),
  places: 1,
  figures: 3,
  show: context(),
  text: "30",
  display: context(),
  method: context(),
  source: none,
  round-mode: "figures",
  is-exact: false,
)

```

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

```

1 What is the value of $pi$? \
2 // too long number!
3 It is #qt.new("pi", calc.pi, is-exact: true) \
4 Oh, too long, \
5 // set the displayed figure number
6 #qt.set-property("pi", display-figures: 4)
7 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 References

3.1 The Constructors

- [exact\(\)](#)
- [quantity\(\)](#)
- [set-quantity\(\)](#)

exact

Exact value quantities

Parameters

```

exact(
  value: float str,
  ..args: any,
  figures: int,
  places: int,
  display-figures: auto int,
  display-places: auto int
)

```

value float or str

The value passed to the `quantity` function.

..args any

The units and format options for displaying the quantity. Same as `quantity`'s parameters.

figures `int`

Default significant figures.

Default: `99`

places `int`

Default decimal places

Default: `99`

display-figures `auto` or `int`

The displaying significant figures.

Default: `auto`

display-places `auto` or `int`

The displaying decimal places.

Default: `auto`

quantity

Constructor for quantities.

Parameters

```
quantity(  
  raw-value: str float ,  
  ..args: any ,  
  unit-separator: str ,  
  places: auto int ,  
  figures: auto int ,  
  magnitude-limit: int ,  
  round-mode: str ,  
  method: any ,  
  display: any ,  
  precision: int ,  
  explicit-method: bool ,  
  source,  
  is-exact  
) -> dictionary
```

raw-value `str` or `float`

The numerical value of the quantity. It is recommended to specify by `str` representation of the value to retain most of the information, like significant figures, number of decimal places or scientific notations.

..args `any`

The units (positional arguments) or format options (named arguments) to the units. The unit must be a string, like "km/s^2 N", or an array of the unit representation and its exponent, like (`$angstrom$`, 1).

The format options will be passed to `zero` package directly.

unit-separator `str`

The separator between units for string of unit input

Default: " "

places `auto` or `int`

Number of decimal places to display. If this is `auto`, it will determine from the input value.

Default: `auto`

figures `auto` or `int`

Number of significant figures to display. If this is `auto`, it will determine from the input value.

Default: `auto`

magnitude-limit `int`

For very large or small value whose magnitude are exceeded the absolute value of this option, `pariman` will automatically convert it to be in a scientific notation.

Default: 4

round-mode `str`

Specify which way to round the numerical value. The possible options are "figures" for significant figures, "places" for number of decimal places, `auto` for a smart default.

If this is set to `auto`, it will determine whether `figures` or `places` are specified.

- If `figures` is an integer, this will resolve to "figures".
- If `places` is an integer, this will resolve to "places".
- If both `figures` and `places` are specified, the default behavior is "places".

Default: `auto`

method `any`

Way to display this value when using `quantity.method` or `qt.metod(..)`.

Default: `auto`

display `any`

Way to display this value when using `quantity.display` or `qt.display(..)`.

Default: `auto`

precision `int`

The additional number of significant figures or decimal places to keep on for more precise calculation. This number will be added to `figures` or `places` when rounding the input value.

Default: `10`

explicit-method `bool`

Whether to show the unit when accessing the `method` property. This will effect only when setting BEFORE the calculation.

Default: `true`

set-quantity

Reset or modify the behavior of `exact` or `quantity` functions.

Parameters

```
set-quantity(  
    qty: dictionary,  
    unit: str array,  
    value: str float,  
    ..formatting  
) -> dictionary
```

qty dictionary

The quantity.

unit str or array

New units.

Default: **auto**

value str or float

New value

Default: **auto**

..formatting

Format options. Same as **quantity**'s or **exact**'s.

3.2 Available Calculation Methods

All functions in calculation module also accept the same format options in the **quantity** and **set-quantity** function for formatting the result quantity.

- **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 **n**th 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.