Biblioteca Simplificada de Gás Ideal

Biblioteca básica de gás ideal sob hipóteses:

- Substâncias puras;
- Função cp(T) polinomial cúbica;
- Função cp(T) válida para Tmin ⩽ T ⩽ Tmax;
- sref = s°(Tref) conhecida.

```
using PlutoUI
```

```
using Formatting
```

```
using DataFrames
```

```
using BrowseTables
```

Setup Mínimo

```
# Whether the molar base is the default one
const MOLR = true;
```

```
# Universal gas constant, with an optional precision argument
R̄(ℙ=Float64) = ℙ(8.314472); # ± 0.000015 # kJ/kmol·K
```

```
# Standard Tref - for the internal energy
Tref(ℙ=Float64) = ℙ(298.15); # K
```

```
# Standard Pref - for the entropy
Pref(ℙ=Float64) = ℙ(100.00); # kPa
```

```
# IG (Ideal Gas) structure: values for each gas instance
struct IG
    MW                  # Molecular "Weight", kg/kmol
    CP::NTuple{4}       # Exactly 4 c̄p(T) coefficients
    Tmin                # T_min, K
    Tmax                # T_max, K
    sref                # s̄°ref, kJ/kmol·K
end;
```

Biblioteca Externa

Valores tabelados em Çengel, Y. A., Termodinâmica 7a Ed. ISBN 978-85-8055-200-3 (Tabelas A-1, A-2 e A-26), foram transportados para uma planilha (OpenOffice Spreadsheet) e exportados no formato CSV [Comma Separated Values], os quais podem ser lidos em Julia:

Name	Formula	MW	cp_a	cp_b	cp_c	cp_d	Tmin	Tmax	sref
Nitrogen	N2	28.013	+28.900	-1.571E-03	+8.081E-06	-2.873E-09	273.0	1800.0	191.61
Oxygen	O2	31.999	+25.480	+1.520E-02	-7.155E-06	+1.312E-09	273.0	1800.0	205.04
Hydrogen	H2	2.016	+29.110	-1.916E-03	+4.003E-06	-8.704E-10	273.0	1800.0	130.68
Carbon Monoxide	CO	28.011	+28.160	+1.675E-03	+5.372E-06	-2.222E-09	273.0	1800.0	197.65
Carbon Dioxide	CO2	44.010	+22.260	+5.981E-02	-3.501E-05	+7.469E-09	273.0	1800.0	213.80
Water Vapor	H2O	18.015	+32.240	+1.923E-03	+1.055E-05	-3.595E-09	273.0	1800.0	188.83
Methane	CH4	16.043	+19.890	+5.024E-02	+1.269E-05	-1.101E-08	273.0	1500.0	186.16
Ethane	C2H6	30.070	+6.900	+1.727E-01	-6.406E-05	+7.285E-09	273.0	1500.0	229.49
Propane	C3H8	44.097	-4.040	+3.048E-01	-1.572E-04	+3.174E-08	273.0	1500.0	269.91
N-Butane	C4H10	58.124	+3.960	+3.715E-01	-1.834E-04	+3.500E-08	273.0	1500.0	310.12
Methanol	CH4O	32.042	+19.000	+9.152E-02	-1.220E-05	-8.039E-09	273.0	1000.0	239.70
Ethanol	C2H6O	46.070	+19.900	+2.096E-01	-1.038E-04	+2.005E-08	273.0	1500.0	282.59

```
using CSV
```

gasRaw =

	Name	Formula	MW	cp_a	cp_b	cp_c	cp_d
1	"Nitrogen"	"N2"	28.013	28.9	-0.001571	8.081e-6	-2.873e-9
2	"Oxygen"	"O2"	31.999	25.48	0.0152	-7.155e-6	1.312e-9
3	"Hydrogen"	"H2"	2.016	29.11	-0.001916	4.003e-6	-8.704e-10
4	"Carbon Monoxide"	"CO"	28.011	28.16	0.001675	5.372e-6	-2.222e-9
5	"Carbon Dioxide"	"CO2"	44.01	22.26	0.05981	-3.501e-5	7.469e-9
6	"Water Vapor"	"H2O"	18.015	32.24	0.001923	1.055e-5	-3.595e-9
7	"Methane"	"CH4"	16.043	19.89	0.05024	1.269e-5	-1.101e-8
8	"Ethane"	"C2H6"	30.07	6.9	0.1727	-6.406e-5	7.285e-9
9	"Propane"	"C3H8"	44.097	-4.04	0.3048	-0.0001572	3.174e-8
10	"N-Butane"	"C4H10"	58.124	3.96	0.3715	-0.0001834	3.5e-8
11	"Methanol"	"CH4O"	32.042	19.0	0.09152	-1.22e-5	-8.039e-9

```
gasRaw = CSV.File("IGTable.csv", normalizenames=true)
```

```
# Transforms a row (from the CSV file) into an IG instance
function rowToIG(row)
    IG( row.MW, (row.cp_a, row.cp_b, row.cp_c, row.cp_d),
        row.Tmin, row.Tmax, row.sref)
end;
```

```
gasLib =
▸ Dict(:O2 ⇒ IG(31.999, (25.48, 0.0152, -7.155e-6, 1.312e-9), 273.0, 1800.0, 205.04), :C2H6
```

```
gasLib = Dict(Symbol(r.Formula) => rowToIG(r) for r in gasRaw)
```

Gás Padrão

Escolha do gás padrão para testes, abaixo:

Nitrogen

```
@bind gas_choice Select([row.Formula => row.Name for row in gasRaw])
```

```
stdGas = ▸ IG(28.013, (28.9, -0.001571, 8.081e-6, -2.873e-9), 273.0, 1800.0, 191.61)
```

```
# Standard test gas
stdGas = gasLib[Symbol(gas_choice)]
```

Funcionalidade – Verificações

```
function inbounds(gas::IG, T)
    𝕡 = typeof(AbstractFloat(T))
    minT, maxT = map(𝕡, (gas.Tmin, gas.Tmax))
    if !(minT <= T <= maxT)
        throw(DomainError(T, "out of bounds $(minT) ⩽ T ⩽ $(maxT)."))
    end
    true
end;
```

▷ Testes:

0

✘ A temperatura de 0 K está **FORA** dos limites para o N2!

Funcionalidade – Constantes

```
R (generic function with 3 methods)
```

```
# "R" can be typed by \bfR<tab>
R(gas::IG, molr=MOLR, 𝕡=Float64) = molr ? R̄(𝕡) : R̄(𝕡) / 𝕡(gas.MW)
```

```
M (generic function with 2 methods)
```

```
# "M" can be typed by \bfM<tab>
M(gas::IG, p=Float64) = p(gas.MW)
```

```
Tmin (generic function with 2 methods)
```

```
Tmin(gas::IG, p=Float64) = p(gas.Tmin)
```

```
Tmax (generic function with 2 methods)
```

```
Tmax(gas::IG, p=Float64) = p(gas.Tmax)
```

```
sref (generic function with 2 methods)
```

```
sref(gas::IG, p=Float64) = p(gas.sref)
```

▷ Testes:

#	gas	R	M	s°
1	N2	0.29681	28.013	191.61
2	O2	0.25984	31.999	205.04
3	H2	4.12424	02.016	130.68
4	CO	0.29683	28.011	197.65
5	CO2	0.18892	44.010	213.80
6	H2O	0.46153	18.015	188.83
7	CH4	0.51826	16.043	186.16
8	C2H6	0.27650	30.070	229.49
9	C3H8	0.18855	44.097	269.91
10	C4H10	0.14305	58.124	310.12
11	CH4O	0.25949	32.042	239.70
12	C2H6O	0.18047	46.070	282.59
#	gas	R	M	s°

Funcionalidade – Comportamento P-T-v

```
prTy (generic function with 1 method)
```

```
# Auxiliary function of promoted types (float types relate to precision bits)!
prTy(A...) = promote_type(map(typeof, AbstractFloat.(A))...)
```

```
P (generic function with 2 methods)
```

```
# "P" can be typed by \bfP<tab>
P(gas::IG, molr=true; T, v) = begin
    p = prTy(T, v)
    R(gas, molr, p) * T / v
end
```

```
T (generic function with 2 methods)
```

```
# "T" can be typed by \bfT<tab>
T(gas::IG, molr=true; P, v) = begin
    p = prTy(P, v)
    P * v / R(gas, molr, p)
end
```

```
v (generic function with 2 methods)
```

```
# "v" can be typed by \bfv<tab>
v(gas::IG, molr=true; P, T) = begin
    p = prTy(P, T)
    R(gas, molr, p) * T / P
end
```

▷ Testes:

P = 300 kPa

T = 300 K

Molar base? ☐

v = 0.2968 m³/kg ; P = 300.0000 kPa ; T = 300.0000 K.

Funcionalidade – Comportamento Calórico

Transformação de coeficientes:

Coeficientes, de $c_p(T)$ ou $c_v(T)$, são retornados como uma *matriz-linha*.

```
# If functions accound for integration factor, then only :cp, :cv are needed here
function coef(gas::IG, kind::Symbol = :cp, molr=MOLR, 𝕡=Float64)
    if kind == :cp      # No coef. transformation
        ret = hcat(𝕡.(gas.CP)...)
    elseif kind == :cv  # Translates first coef.
        ret = hcat(𝕡(gas.CP[1]) - R̄(𝕡), 𝕡.(gas.CP[2:end])...)
    end
    molr ? ret : ret ./ M(gas, 𝕡)
end;
```

Funções dos coeficientes por propriedade:

Estas são as funções para serem aplicadas à temperatura, em três casos distintos. A função `apply` abaixo, faz a aplicação das funções à temperatura, retornando uma *matriz-coluna*.

```
const propF = Dict(
    :c => (x->1     , x->x    , x->x^2  , x->x^3  ),    # Tuple makes it faster
    :h => (x->x     , x->x^2/2, x->x^3/3, x->x^4/4),    # Tuple makes it faster
    :s => (x->log(x), x->x    , x->x^2/2, x->x^3/3),    # Tuple makes it faster
);
```

```
# Generic f(T) function by Symbol key
function apply(p::Symbol, T, rel=false)
    𝕡 = typeof(T)
    rel ?
        apply(p, T, false) - apply(p, Tref(𝕡), false) :
        vcat((f(T) for f in propF[p])...)
end;
```

Propriedades Calóricas Diretas – $c_{p,v}(T)$, $u(T)$, etc:

Estas funções selecionam as matrizes linha e coluna pertinentes, multiplicando-as, extraindo e retornando o único valor da matrix 1x1 resultante, com verificação de limites e somas de eventuais termos constantes.

As funções de uma única letra ASCII tem os nomes em letras negritas (bold-face) para não conflitarem com variáveis de nomes `u` e `h`, por exemplo. Por isso nomes como: `𝐮` e `𝐡`, por exemplo, porém não `γ`, por não ser ASCII.

```
cp (generic function with 2 methods)
```

```
cp(gas::IG, molr=MOLR; T) = begin
    𝕡 = typeof(T)
    inbounds(gas, T) ?
        (coef(gas, :cp, molr, 𝕡) * apply(:c, T))[1] :
        zero(𝕡)
end
```

```
cv (generic function with 2 methods)
```

```
cv(gas::IG, molr=MOLR; T) = begin
    𝕡 = typeof(T)
    inbounds(gas, T) ?
        (coef(gas, :cv, molr, 𝕡) * apply(:c, T))[1] :
        zero(𝕡)
end
```

```
γ (generic function with 1 method)
```

```
# "γ" can be typed by \gamma<tab>
γ(gas::IG; T) = cp(gas, true, T=T) / cv(gas, true, T=T)
```

```
𝐮 (generic function with 2 methods)
```

```
# "𝐮" can be typed by \bfu<tab>
𝐮(gas::IG, molr=MOLR; T) = begin
    𝕡 = typeof(T)
    inbounds(gas, T) ?
        (coef(gas, :cv, molr, 𝕡) * apply(:h, T, true))[1] :
        zero(𝕡)
end
```

```
h (generic function with 2 methods)
```

```
# "h" can be typed by \bfh<tab>
h(gas::IG, molr=MOLR; T) = begin
    ℘ = typeof(T)
    inbounds(gas, T) ?
        (coef(gas, :cp, molr, ℘) * apply(:h, T, true))[1] +
            R(gas, molr, ℘) * Tref(℘) :
        zero(℘)
end
```

```
s° (generic function with 2 methods)
```

```
# "°" can be typed by \degree<tab>
# "Partial" ideal gas entropy
s°(gas::IG, molr=MOLR; T) = begin
    ℘ = typeof(T)
    inbounds(gas, T) ?
        (coef(gas, :cp, molr, ℘) * apply(:s, T, true))[1] + (
            molr ? sref(gas, ℘) : sref(gas, ℘) / M(gas, ℘)
        ) :
        zero(℘)
end
```

```
Pr (generic function with 1 method)
```

```
Pr(gas::IG; T) = begin
    ℘ = typeof(T)
    exp(s°(gas, true, T=T) / R̄(℘)) / exp(sref(gas, ℘) / R̄(℘))
end
```

```
vr (generic function with 1 method)
```

```
vr(gas::IG; T) = T / Pr(gas, T=T)
```

```
s (generic function with 2 methods)
```

```
# "s" can be typed by \bfs<tab>
s(gas::IG, molr=MOLR; T, P) = begin
    ℘ = prTy(P, T)
    inbounds(gas, T) ?
        s°(gas, molr, T=T) - R(gas, molr, ℘) * log(P / Pref(℘)) :
        zero(℘)
end
```

▷ Testes:

#	T	h	Pr	u	vr	s°	cp	cv	γ
1	300.0	90.41	1.02	1.37	293.58	6.85	1.04	0.74	1.4
2	400.0	194.73	2.81	76.0	142.45	7.15	1.05	0.75	1.39
3	500.0	300.29	6.21	151.88	80.54	7.38	1.06	0.77	1.39
4	600.0	407.4	11.99	229.32	50.06	7.58	1.08	0.78	1.38
5	700.0	516.3	21.1	308.53	33.18	7.75	1.1	0.8	1.37
6	800.0	627.17	34.74	389.72	23.03	7.89	1.12	0.82	1.36
7	900.0	740.11	54.38	472.98	16.55	8.03	1.14	0.84	1.35
8	1000.0	855.19	81.81	558.38	12.22	8.15	1.16	0.86	1.34
9	1100.0	972.4	119.19	645.91	9.23	8.26	1.18	0.89	1.34
10	1200.0	1091.66	169.06	735.49	7.1	8.36	1.2	0.91	1.33
11	1300.0	1212.85	234.41	827.0	5.55	8.46	1.22	0.92	1.32
12	1400.0	1335.78	318.61	920.25	4.39	8.55	1.24	0.94	1.32
13	1500.0	1460.18	425.45	1014.97	3.53	8.64	1.25	0.95	1.31
14	1600.0	1585.76	559.01	1110.86	2.86	8.72	1.26	0.96	1.31
15	1700.0	1712.12	723.62	1207.54	2.35	8.79	1.27	0.97	1.31
16	1800.0	1838.83	923.6	1304.57	1.95	8.87	1.27	0.97	1.31
#	T	h	Pr	u	vr	s°	cp	cv	γ

```
using Roots, ForwardDiff
```

Funcionalidade – Funções inversas

Métodos numéricos para **T**(u), **T**(h), **T**(pr), etc.

Definição de Tipos

Como todas as funções inversas acima – $\mathbf{T}(u)$, $\mathbf{T}(h)$, etc. – possuem o mesmo *nome*, a diferenciação entre elas se dará via `Multiple Dispatch`, e assim, cada função `T` será especializada com base nos **tipos** de seus **argumentos**.

O objetivo de saber se o argumento é uma energia interna ou entalpia, etc., é para que se saiba (i) sua forma funcional **e** (ii) a forma funcional de sua derivada, a fim de ajustar o método numérico.

Para tanto, é necessário a criação de novos **tipos**, que **rotulem** seus valores como "energia interna", "entalpia", etc.:

```
# A Thermodynamic abstract type to hook all concrete property value types under it
abstract type THERM end
```

```
begin
    # A type to LABEL values as internal energy ones:
    struct uType <: THERM
        val
    end
    # Functor to extract the stored value 'val'...
    # ... thus avoiding further implementing the type:
    (_u::uType)() = _u.val
end
```

```
begin
    struct hType <: THERM; val; end
    (_h::hType)() = _h.val
end
```

```
begin
    struct prType <: THERM; val; end
    (_p::prType)() = _p.val
end
```

```
begin
    struct vrType <: THERM; val; end
    (_v::vrType)() = _v.val
end
```

▷ Ilustração do conceito:

```
["ū = 314.15 kJ/kmol", "h = 314.15 kJ/kg", true]
```

```
begin
    # First METHOD definition for the function "example":
    function example(x::uType, molr=MOLR)
        molr ?
            "ū = $(x()) kJ/kmol" :
            "u = $(x()) kJ/kg"
    end
    # Second METHOD definition for the function "example":
    function example(x::hType, molr=MOLR)
        molr ?
            "h̄ = $(x()) kJ/kmol" :
            "h = $(x()) kJ/kg"
    end
    # Same function name "example" called: specialize based on argument(s) TYPE(s):
    vcat(
        example(uType(314.15)),         # uType argument
        example(hType(314.15), false),  # htype argument
        uType(3.14)() == hType(3.14)()  # Their _values_ are the same!
    )
end
```

Implementação

```
T (generic function with 8 methods)
```

```
begin

    #-------------------------------------------------------------------------#
    #                              T(u) inverse                               #
    #-------------------------------------------------------------------------#
    # "T" can be typed by \bfT<tab>
    function T(
            gas::IG, uVal::uType, molr=true;
            maxIt::Integer=0, epsTol::Integer=4
        )
        # Auxiliary function of whether to break due to iterations
        breakIt(i) = maxIt > 0 ? i >= maxIt || i >= 128 : false
        # Set functions f(x) and g(x) ≡ df/dx
        f = x -> u(gas, molr, T=x)
        g = x -> cv(gas, molr, T=x)
        thef, symb = (uVal)(), "u"
        ε, ℙ = eps(thef), typeof(thef)
        # Get f bounds and check
        TMin, TMax = Tmin(gas, ℙ), Tmax(gas, ℙ)
        fMin, fMax = f(TMin), f(TMax)
        if !(fMin <= thef <= fMax)
            throw(DomainError(thef, "out of bounds $(fMin) ⩽ $(symb) ⩽ $(fMax)."))
        end
        # Linear initial estimate and initializations
        r = (thef - fMin) / (fMax - fMin)
        T = [ TMin + r * (TMax - TMin) ] # Iterations are length(T)-1
        f = [ f(T[end]) ]
        why = :because
        # Main loop
        while true
            append!(T, T[end] + (thef - f[end]) / g(T[end]))
            append!(f, f(T[end]))
            if breakIt(length(T)-1)
                why = :it; break
            elseif abs(f[end] - thef) <= epsTol * ε
                why = :Δf; break
            end
        end
        return Dict(
            :sol => T[end],
            :why => why,
            :it  => length(T)-1,
            :Δf  => f .- thef,
            :Ts  => T,
            :fs  => f
        )
    end

    #-------------------------------------------------------------------------#
    #                              T(h) inverse                               #
    #-------------------------------------------------------------------------#
    # "T" can be typed by \bfT<tab>
    function T(
            gas::IG, hVal::hType, molr=true;
            maxIt::Integer=0, epsTol::Integer=4
        )
        # Auxiliary function of whether to break due to iterations
        breakIt(i) = maxIt > 0 ? i >= maxIt || i >= 128 : false
        # Set functions f(x) and g(x) ≡ df/dx
        f = x -> h(gas, molr, T=x)
        g = x -> cp(gas, molr, T=x)
        thef, symb = (hVal)(), "h"
        ε, ℙ = eps(thef), typeof(thef)
        # Get f bounds and check
        TMin, TMax = Tmin(gas, ℙ), Tmax(gas, ℙ)
        fMin, fMax = f(TMin), f(TMax)
        if !(fMin <= thef <= fMax)
            throw(DomainError(thef, "out of bounds $(fMin) ⩽ $(symb) ⩽ $(fMax)."))
        end
        # Linear initial estimate and initializations
        r = (thef - fMin) / (fMax - fMin)
        T = [ TMin + r * (TMax - TMin) ] # Iterations are length(T)-1
        f = [ f(T[end]) ]
        why = :because
        # Main loop
        while true
            append!(T, T[end] + (thef - f[end]) / g(T[end]))
            append!(f, f(T[end]))
            if breakIt(length(T)-1)
                why = :it; break
            elseif abs(f[end] - thef) <= epsTol * ε
                why = :Δf; break
            end
        end
        return Dict(
            :sol => T[end],
            :why => why,
            :it  => length(T)-1,
            :Δf  => f .- thef,
            :Ts  => T,
            :fs  => f
        )
    end

    #-------------------------------------------------------------------------#
    #                              T(pr) inverse                              #
    #-------------------------------------------------------------------------#
    # "T" can be typed by \bfT<tab>
    function T(
```

```
            gas::IG, pVal::prType;
            maxIt::Integer=0, epsTol::Integer=4
        )
        # Auxiliary function of whether to break due to iterations
        breakIt(i) = maxIt > 0 ? i >= maxIt || i >= 128 : false
        # Set functions f(x) and g(x) ≡ df/dx
        f = x -> Pr(gas, T=x)
        g = x -> ForwardDiff.derivative(f,float(x))
        thef, symb = (pVal)(), "Pr"
        ε, ℙ = eps(thef), typeof(thef)
        # Get f bounds and check
        TMin, TMax = Tmin(gas, ℙ), Tmax(gas, ℙ)
        fMin, fMax = f(TMin), f(TMax)
        if !(fMin <= thef <= fMax)
            throw(DomainError(thef, "out of bounds $(fMin) ⩽ $(symb) ⩽ $(fMax)."))
        end
        # Linear initial estimate and initializations
        r = (thef - fMin) / (fMax - fMin)
        T = [ TMin + r * (TMax - TMin) ] # Iterations are length(T)-1
        f = [ f(T[end]) ]
        why = :because
        # Main loop
        while true
            append!(T, T[end] + (thef - f[end]) / g(T[end]))
            append!(f, f(T[end]))
            if breakIt(length(T)-1)
                why = :it; break
            elseif abs(f[end] - thef) <= epsTol * ε
                why = :Δf; break
            end
        end
        return Dict(
            :sol => T[end],
            :why => why,
            :it  => length(T)-1,
            :Δf  => f .- thef,
            :Ts  => T,
            :fs  => f
        )
    end

    #-------------------------------------------------------------------------#
    #                            T(vr) inverse                                #
    #-------------------------------------------------------------------------#
    # "T" can be typed by \bfT<tab>
    function T(
            gas::IG, vVal::vrType;
            maxIt::Integer=0, epsTol::Integer=4
        )
        # Auxiliary function of whether to break due to iterations
        breakIt(i) = maxIt > 0 ? i >= maxIt || i >= 128 : false
        # Set f(x) function
        thef, symb = (vVal)(), "vr"
        f = x -> vr(gas, T=x) - thef
        ε, ℙ = eps(thef), typeof(thef)
        # Get f bounds and check
        TMin, TMax = Tmin(gas, ℙ), Tmax(gas, ℙ)
        fMin, fMax = f(TMax), f(TMin)
        if !(fMin <= zero(ℙ) <= fMax)
            throw(
                DomainError(
                    thef,
                    "out of bounds $(fMin+thef) ⩽ $(symb) ⩽ $(fMax+thef)."
                )
            )
        end
        # Bisection method initializations
        TB = [ TMin, TMax ] # T bounds
        FB = map(f, TB)     # f bounds
        T = ℙ[ ] # Iterations are length(T)
        f = ℙ[ ]
        s = map(signbit, FB)
        why = :unbracketed
        while !reduce(==, s)
            # Main loop
            append!(T, reduce(+, TB) / 2)
            append!(f, f(T[end]))
            sMid = signbit(f[end])
            if sMid == s[1]
                TB[1], FB[1] = T[end], f[end]
            else
                TB[2], FB[2] = T[end], f[end]
            end
            if breakIt(length(T))
                why = :it; break
            elseif abs(f[end]) <= epsTol * ε
                why = :Δf; break
            end
        end
        return Dict(
            :sol => T[end],
            :why => why,
            :it  => length(T),
            :Δf  => f,
            :Ts  => T,
            :fs  => f .+ thef,
            :TB  => TB,
            :FB  => FB
        )
    end
```

```
end
```

```
T (generic function with 8 methods)
```

```
T
```

```
Tu =
▸ Dict(:it ⇒ 2, :Δf ⇒ [-2.91995, 0.000685624, 3.82154e-11], :sol ⇒ 300.0, :Ts ⇒ [296.06,
```

```
Tu = T(
    stdGas,
    uType(
        u(
            stdGas,
            false,
            T=300.0
        )
    ),
    false,
    epsTol=2^26 # 2²⁶ = 67108864: don't care about the last 26 bits
    #epsTol=2^16 # 2¹⁶ = 65536: don't care about the last 16 bits
)
```

```
▸ ["296.0597138486446", "300.0009249874004", "300.0000000000516"]
```

```
collect(sprintf1("%.$(16-3)f", i) for i in Tu[:Ts])
```

```
Tu₃₂ =
▸ Dict(:it ⇒ 2, :Δf ⇒ [-2.91994, 0.000678539, 0.0], :sol ⇒ 300.0, :Ts ⇒ [296.06, 300.001
```

```
Tu₃₂ = T(
    stdGas,
    uType(
        u(
            stdGas,
            false,
            T=300.0f0 # literal floats with "f0" are 32-bit, single-precision
        )
    ),
    false,
    epsTol=1  # 2⁰ = 1: care about all bits
)
```

```
▸ ["296.0597", "300.0009", "300.0000"]
```

```
collect(sprintf1("%.$(7-3)f", i) for i in Tu₃₂[:Ts])
```

```
Th =
▸ Dict(:it ⇒ 5, :Δf ⇒ [-3.04622, 0.000381454, 6.03033e-12, 1.50709e-27, 9.41309e-59, 0.0],
```

```
Th = T(
    stdGas,
    hType(
        h(
            stdGas,
            false,
            T=BigFloat(300.0)
        )
    ),
    false
)
```

```
▸ ["297.065020802418806975208246766375870391668061284181485246454206601924027496952", "300.
```

```
collect(sprintf1("%.$(78-3)f", i) for i in Th[:Ts])
```

```
Tp =
▸ Dict(:it ⇒ 6, :Δf ⇒ [-0.282358, 0.0410276, 0.000568156, 1.13878e-7, 3.55271e-15, 3.55271
```

```
Tp = T(
    stdGas,
    prType(
        Pr(
            stdGas,
            T=300.0
        )
    ),
    epsTol=1
)
```

```
▸ ["+273.4745933292123", "+303.3952611692405", "+300.0476837891530", "+300.0000095593841",
```

```
collect(sprintf1("%+.$(16-3)f", i) for i in Tp[:Ts])
```

```
▸ ["-2.823582329270165e-01", "+4.102760579279985e-02", "+5.681556327807868e-04", "+1.138778
```

```
collect(sprintf1("%+.$(16-1)e", i) for i in Tp[:Δf])
```

```
Tv =
▸ Dict(:TB ⇒ [300.0, 300.0], :it ⇒ 51, :Δf ⇒ [-282.574, -253.876, -195.909, -118.238, -45
```

```
Tv = T(
    stdGas,
    vrType(
        vr(
            stdGas,
            T=300.0
        )
    ),
    epsTol=1
)
```