ColabでJuliaを使うためのノートブック

このノートブックの内容については再配布・改変・部分的コピーその他すべてを自由に行って構いません.

このノートブックはGoogle Colabで実行できる (https://colab.research.google.com/github/genkuroki/Statistics/blob/master/2022/07-4%20Julia%20notebook%20for%20Google%20Colab.ipynb).

注意警告: すべてのセルを実行する前に少し下の方にある注意警告を参照せよ。

In [1]:

```
# Google Colabと自分のパソコンの両方で使えるようにするための工夫

using Pkg

"""すでにPkg.add済みのパッケージのリスト (高速化のために用意)"""
_packages_added = [info.name for (uuid, info) in Pkg.dependencies() if info.is_direct_de

"""_packages_added内にないパッケージをPkg.addする"""
add_pkg_if_not_added_yet(pkg) = if !(pkg in _packages_added)
    println(stderr, "# $(pkg).jl is not added yet, so let's add it.")
    Pkg.add(pkg)
end

"""expr::Exprからusing内の`.`を含まないモジュール名を抽出"""
function find_using_pkgs(expr::Expr)
    pkgs = String[]
    function traverse(expr::Expr)
        if expr.head == :using
            for arg in expr.args
                if arg.head == :. && length(arg.args) == 1
                    push!(pkgs, string(arg.args[1]))
                elseif arg.head == :(:) && length(arg.args[1].args) == 1
                    push!(pkgs, string(arg.args[1].args[1]))
                end
            end
        else
            for arg in expr.args arg isa Expr && traverse(arg) end
        end
    end
    traverse(expr)
    pkgs
end

"""必要そうなPkg.addを追加するマクロ"""
macro autoadd(expr)
    pkgs = find_using_pkgs(expr)
    :(add_pkg_if_not_added_yet.($(pkgs)); $expr)
end
```

Out[1]: @autoadd

注意警告: 以下のセルの `@usingdocstringtranslation; @switchlang! :ja` をコメントアウトすると(# を行頭に追加すると)、実験的にヘルプの主要な一部分を日本語に翻訳して表示してくれる設定を無効にできる。無効にしない場合にはGoogle Colabで約3分程度時間が取られることになる。HDD上に展開する場合にはそれを超えて大変な時間がかかる可能性がある。

In [2]:

```
macro usingdocstringtranslation()
    quote
        translationurl = "https://github.com/AtelierArith/DocStrBankExperimental.jl/rele
        #translationurl = "https://github.com/AtelierArith/DocStrBankExperimental.jl/re
        scratchspaces_dir = joinpath(DEPOT_PATH[1], "scratchspaces", "d404e13b-1f8e-41a5
        translation_dir = joinpath(scratchspaces_dir, "translation")
        if isdir(translation_dir)
            println(stderr, "Directory $translation_dir already exists.")
            println(stderr, "To download and extract translation.zip again, the directo
        else
            println(stderr, "translation.zip to be downloaded will be extracted into $t
            run(`wget --no-verbose $(translationurl) -P $(scratchspaces_dir)`)
            run(`unzip -q $(joinpath(scratchspaces_dir, "translation.zip")) -d $(scratch
        end
        @autoadd using DocstringTranslation
    end
end

# Google Colabで次の行の実行には3分秒程度かかる。
@usingdocstringtranslation; @switchlang! :ja
```

```
Directory D:\.julia\scratchspaces\d404e13b-1f8e-41a5-a26a-0b758a0c6c97\translation already
exists.
To download and extract translation.zip again, the directory must be deleted.
┌ Warning: The OPENAI_API_KEY has not been set. Please set OPENAI_API_KEY to the environme
nt variable to use the translation function.
└ @ DocstringTranslation D:\.julia\packages\DocstringTranslation\0YgZP\src\switchlang.jl:2
6
```

Out[2]: "ja"

注意: 以下のセルを `using` の行のコメントアウトを全部外してからGoogle Colabで実行すると5分から6分程度かかるようである. その待ち時間に耐え切れないと感じる人は自分のパソコン上にJuliaをJupyter上で実行する環境を作ればよい. コンピュータの取り扱いの初心者のうちはその作業は非常に難しいと感じるかもしれないが, 適当に検索したり, AIに質問したりすればできるはずである.

In [3]:

```
# 以下は黒木玄がよく使っているパッケージ達
# 例えばQuadGKパッケージ(数値積分のパッケージ)の使い方は
# QuadGK.jl をインターネットで検索すれば得られる.

ENV["LINES"], ENV["COLUMNS"] = 100, 100
using LinearAlgebra
using Printf
using Random
Random.seed!(4649373)

@autoadd begin
using Distributions
using StatsPlots
default(fmt=:png, legendfontsize=12, titlefontsize=12)
#using BenchmarkTools
#using Optim
#using QuadGK
#using RDatasets
#using Roots
#using StatsBase
#using StatsFuns
#using SpecialFunctions
#using SymPy
end
```

Julia言語 (https://julialang.org/)については以下の検索で色々学べる.

- Julia言語のドキュメント: https://docs.julialang.org/en/v1/ (https://docs.julialang.org/en/v1/)
- Julia言語について検索: https://www.google.com/search?q=Julia%E8%A8%80%E8%AA%9E (https://www.google.com/search?q=Julia%E8%A8%80%E8%AA%9E)
- Distributions.jlパッケージについて検索: https://www.google.com/search?q=Distributions.jl (https://www.google.com/search?q=Distributions.jl)
- Plots.jlパッケージについて検索: https://www.google.com/search?q=Plots.jl (https://www.google.com/search?q=Plots.jl)
- StatsPlots.jlパッケージについて検索: https://www.google.com/search?q=StatsPlots.jl (https://www.google.com/search?q=StatsPlots.jl)
- BenchmarkTools.jlパッケージについて検索: https://www.google.com/search?q=BenchmarkTools.jl (https://www.google.com/search?q=BenchmarkTools.jl)
- Optim.jlパッケージについて検索: https://www.google.com/search?q=Optim.jl (https://www.google.com/search?q=Optim.jl)

- QuadGK.jlパッケージについて検索: https://www.google.com/search?q=QuadGK.jl (https://www.google.com/search?q=QuadGK.jl)
- RDatasets.jlパッケージについて検索: https://www.google.com/search?q=RDatasets.jl (https://www.google.com/search?q=RDatasets.jl)
- Roots.jlパッケージについて検索: https://www.google.com/search?q=Roots.jl (https://www.google.com/search?q=Roots.jl)
- StatsBase.jlパッケージについて検索: https://www.google.com/search?q=StatsBase.jl (https://www.google.com/search?q=StatsBase.jl)
- StatsFuns.jlパッケージについて検索: https://www.google.com/search?q=StatsFuns.jl (https://www.google.com/search?q=StatsFuns.jl)
- SpecialFunctions.jlパッケージについて検索: https://www.google.com/search?q=SpecialFunctions.jl (https://www.google.com/search?q=SpecialFunctions.jl)
- SymPy.jlパッケージについて検索: https://www.google.com/search?q=SymPy.jl (https://www.google.com/search?q=SymPy.jl)

@autoadd マクロの使い方

例えば, パッケージA.jlやB.jlをインストール前(Pkg.add前)であるとき,

```
using A
using B: b1, b2
```

を実行しようとするとエラーになってしまう. しかし,

```
@autoadd using A
@autoadd using B: b1, b2
```

または

```
@autoadd begin
using A
using B: b1, b2
end
```

を実行すれば, 自動的にパッケージA.jlやB.jlがインストールされてから, using達が実行される.

以下のように `@macroexpand` を使えば具体的に何が実行されるかを確認できる.

In [4]:
```
(@macroexpand @autoadd using A) |> Base.remove_linenums!
```

Out[4]:
```
quote
    Main.add_pkg_if_not_added_yet.(["A"])
    using A
end
```

In [5]:
```
(@macroexpand @autoadd using A, B, C) |> Base.remove_linenums!
```

Out[5]:
```
quote
    Main.add_pkg_if_not_added_yet.(["A", "B", "C"])
    using A, B, C
end
```

In [6]:
```
(@macroexpand @autoadd using A: a1, a2, @a3) |> Base.remove_linenums!
```

Out[6]:
```
quote
    Main.add_pkg_if_not_added_yet.(["A"])
    using A: a1, a2, @a3
end
```

In [7]:

```
(@macroexpand @autoadd begin
using A: a1
using A.B
using A.C: c1, c2
#using D
using E, A.F, G
using H: h1, h2
using I
end) ▷ Base.remove_linenums!
```

Out[7]:

```
quote
    Main.add_pkg_if_not_added_yet.(["A", "E", "G", "H", "I"])
    begin
        using A: a1
        using A.B
        using A.C: c1, c2
        using E, A.F, G
        using H: h1, h2
        using I
    end
end
```

ランダムウォーク

期待値が μ で標準偏差が σ の確率分布の独立同分布確率変数列 $X_1, X_2, X_3, \ldots$について,

$$W_n = (X_1 - \mu) + (X_2 - \mu) + \cdots + (X_n - \mu), \quad n = 1, 2, 3, \ldots$$

の様子がどうなるかを見てみよう.

まずはガンマ分布の説明を見てみよう。セルの先頭に ？ と書き、その後に説明してもらいたいものの名前を書けば説明が表示される。

In [8]:

```
?Gamma
```

```
search: Gamma gamutmax
```

Out[8]:

```
Gamma(α,θ)
```

ガンマ分布は、形状パラメータ α とスケール θ を持ち、確率密度関数は次のようになります。

$$f(x; \alpha, \theta) = \frac{x^{\alpha-1} e^{-x/\theta}}{\Gamma(\alpha)\theta^{\alpha}}, \quad x > 0$$

```
Gamma()            # 形状が単位、スケールが単位のガンマ分布、すなわちGamma(1, 1)
Gamma(α)           # 形状がα、スケールが単位のガンマ分布、すなわちGamma(α, 1)
Gamma(α, θ)        # 形状がα、スケールがθのガンマ分布

params(d)          # パラメータを取得、すなわち(α, θ)
shape(d)           # 形状パラメータを取得、すなわちα
scale(d)           # スケールパラメータを取得、すなわちθ
```

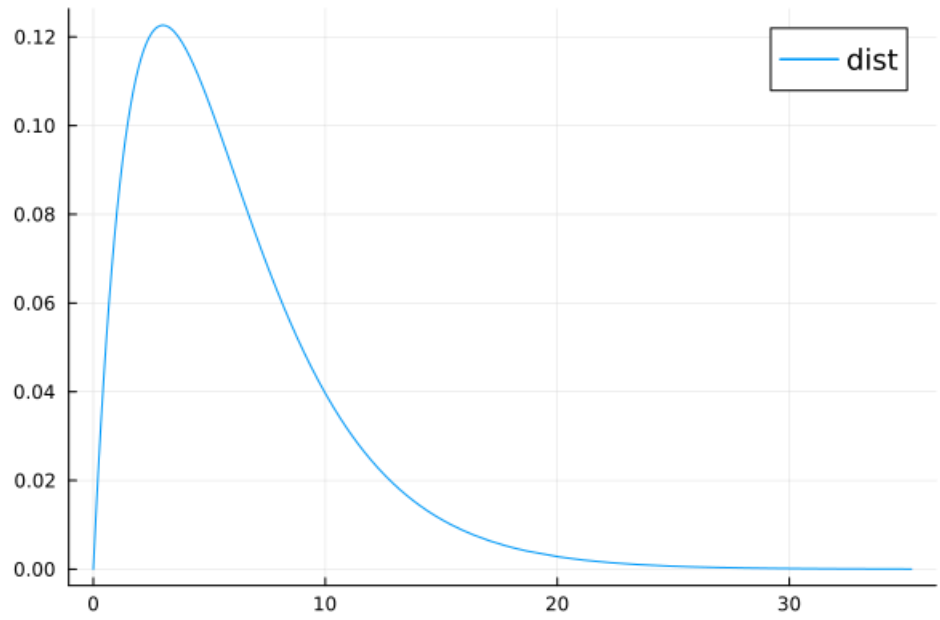
外部リンク

- ガンマ分布 - Wikipedia (http://en.wikipedia.org/wiki/Gamma_distribution)

In [9]:
```
dist = Gamma(2, 3)
@show mu, sigma = mean(dist), std(dist)
plot(dist; label="dist")
```

```
(mu, sigma) = (mean(dist), std(dist)) = (6.0, 4.242640687119285)
```

Out[9]:



In [10]:
```
X_minus_mu = rand(dist - mu, 10) # X_1 - mu, X_2 - mu, ..., X_10 - mu を生成
```

Out[10]:
```
10-element Vector{Float64}:
  3.0311267478769928
 -1.4409032773374868
 -4.693642987661233
 -1.7502325815689757
 10.650497896417825
 -2.563141099889714
 -3.189505348153571
 -1.8701849265462256
 -3.0119934886403863
 -4.283202704007239
```

In [11]:
```
cumsum(X_minus_mu) # W_1, W_2, ..., W_10 を作成
```

Out[11]:
```
10-element Vector{Float64}:
  3.0311267478769928
  1.590223470539506
 -3.1034195171217274
 -4.853652098690703
  5.796845797727122
  3.2337046978374078
  0.044199349683836875
 -1.8259855768623883
 -4.837979065502775
 -9.121181769510013
```

上で使った cumsum の説明を見てみよう。

In [12]:

```
1 ?cumsum
```

search: **cumsum cumsum!** sum

Out[12]: `cumsum(A; dims::Integer)`

次元 `dims` に沿った累積和。パフォーマンスのために事前に割り当てられた出力配列を使用するには、 `cumsum!` (@ref) も参照してください。また、出力の精度を制御するためにも使用できます（例：オーバーフローを避けるため）。

例

```
julia> a = [1 2 3; 4 5 6]
2×3 Matrix{Int64}:
 1  2  3
 4  5  6

julia> cumsum(a, dims=1)
2×3 Matrix{Int64}:
 1  2  3
 5  7  9

julia> cumsum(a, dims=2)
2×3 Matrix{Int64}:
 1  3   6
 4  9  15
```

!!! note 戻り値の配列の `eltype` は、システムのワードサイズ未満の符号付き整数の場合は `Int` 、システムのワードサイズ未満の符号なし整数の場合は `UInt` です。小さな符号付きまたは符号なし整数の配列の `eltype` を保持するには、 `accumulate(+, A)` を使用する必要があります。

```jldoctest
julia> cumsum(Int8[100, 28])
2-element Vector{Int64}:
 100
 128

julia> accumulate(+,Int8[100, 28])
2-element Vector{Int8}:
  100
 -128
```

前者の場合、整数はシステムのワードサイズに拡張され、その結果は `Int64[100, 128]` になります。後者の場合、そのような拡張は行われず、整数のオーバーフローが `Int8[100, -128]` になります。

`cumsum(itr)`

イテレータの累積和。

他の関数を適用するには、 `accumulate` (@ref)を参照してください。

!!! compat "Julia 1.5" 非配列イテレータに対する `cumsum` は、少なくともJulia 1.5が必要です。

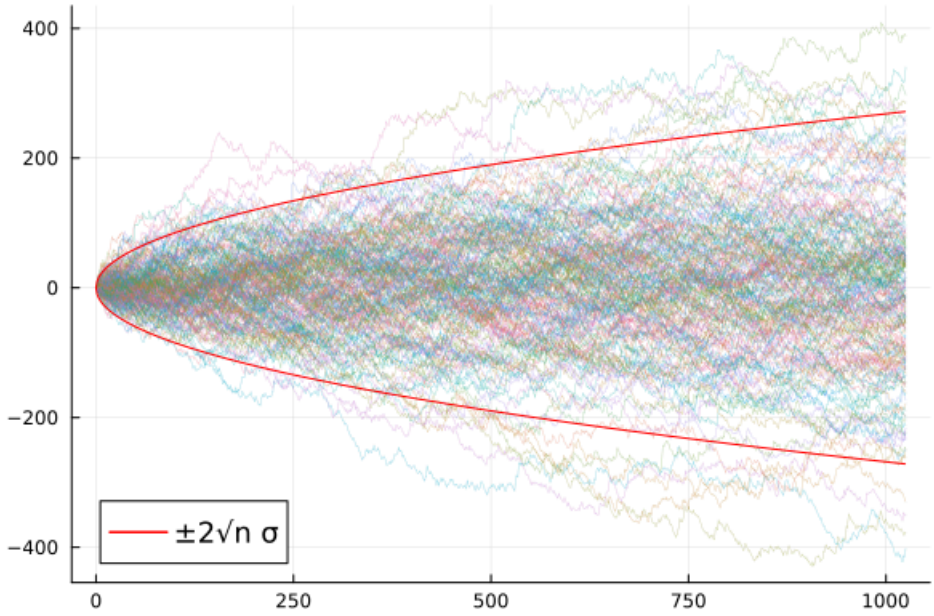
例

```
julia> cumsum(1:3)
3-element Vector{Int64}:
 1
 3
 6

julia> cumsum((true, false, true, false, true))
(1, 1, 2, 2, 3)

julia> cumsum(fill(1, 2) for i in 1:3)
3-element Vector{Vector{Int64}}:
 [1, 1]
 [2, 2]
 [3, 3]
```

In [13]:

```
nmax = 2^10 # maximum sample size
niters = 200 # number of iterations
Ws = [cumsum(rand(dist - mu, nmax)) for _ in 1:niters] # [W_1, W_2, ..., W_nmax] をniter

plot()
for W in Ws
    plot!([0; W]; label="", lw=0.3, alpha=0.5)
end
plot!(n -> +2sqrt(n)*sigma, 0, nmax; label="±2√n σ", c=:red)
plot!(n -> -2sqrt(n)*sigma, 0, nmax; label="", c=:red)
```

Out[13]:



期待値が 0 のギャンブルを n 回繰り返すと, **トータルでの勝ち負けの金額**はおおよそ $\pm 2\sqrt{n}\,\sigma$ の範囲におさまる(ランダムウォークの偏差).

大数の法則

期待値が 0 で標準偏差が σ の確率分布の独立同分布確率変数列$X_1, X_2, X_3, \ldots$について, サイズ n の標本平均

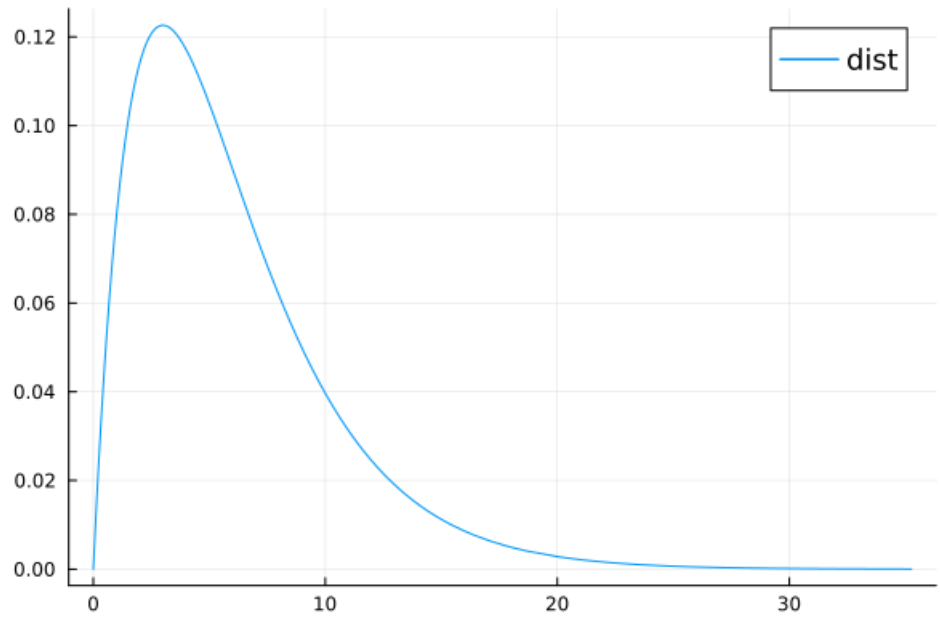
$$\bar{X}_n = \frac{X_1 + X_2 + \cdots + X_n}{n}$$

の様子がどうなるかを見てみよう.

In [14]:
```
dist = Gamma(2, 3)
@show mu, sigma = mean(dist), std(dist)
plot(dist; label="dist")
```

```
(mu, sigma) = (mean(dist), std(dist)) = (6.0, 4.242640687119285)
```

Out[14]:



In [15]:
```
X = rand(dist, 10) # X_1, X_2, ..., X_10 を生成
```

Out[15]:
```
10-element Vector{Float64}:
  0.40805136957750254
 11.35246443572607
  1.7750298759291916
  2.871320442475634
  5.377988900748953
  1.6866822858975898
  4.050010771853904
  4.8982949336627355
  1.492397480850824
  1.1322589811898953
```

In [16]:
```
Xbar = cumsum(X) ./ (1:10) # Xbar_1, Xbar_2, ..., Xbar_10 を作成
```

Out[16]:
```
10-element Vector{Float64}:
 0.40805136957750254
 5.8802579026517865
 4.511848560410922
 4.101716530927099
 4.356971004891471
 3.911922885059157
 3.9316497260298355
 4.052480376983947
 3.7680267218580448
 3.5044499477912296
```

上の行の `./` で使っているドット記法については

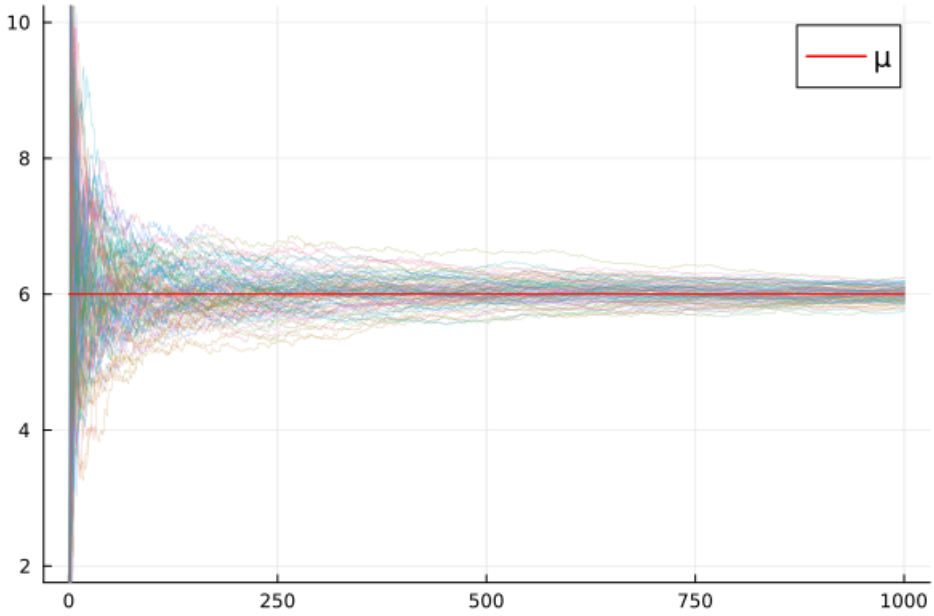
- https://docs.julialang.org/en/v1/manual/functions/#man-vectorized (https://docs.julialang.org/en/v1/manual/functions/#man-vectorized)

を参照せよ。例えば長さ n の配列 A について `A ./ (1:n)` の結果は A の第k成分をkで割って得られる配列になる。ドット記法は各成分ごとに演算子や関数を作用させるために使われる。

In [17]:

```
nmax = 1000 # maximum sample size
niters = 100 # number of iterations
Xbars = [cumsum(rand(dist, nmax)) ./ (1:nmax) for _ in 1:niters] # [Xbar_1, ..., Xbar_nm

plot()
for Xbar in Xbars
    plot!([0; Xbar]; label="", lw=0.3, alpha=0.5)
end
plot!(x -> mu, 0, nmax; label="μ", c=:red)
plot!(ylim=(mu-sigma, mu+sigma))
```

Out[17]:



期待値が 0 のギャンブルを n 回繰り返すと, 1回ごとの勝ち負けの平均値は μ に近付く(大数の法則).

ランダムウォーク(トータルでの勝ち負けの金額の話)と大数の法則(トータルの勝ち負けの金額を繰り返した回数の n で割って得られる1回ごとの平均値の話)を混同するとひどい目にあうだろう!

中心極限定理の素朴な確認の仕方

期待値が μ で標準偏差が σ の確率分布の独立同分布確率変数列$X_1, X_2, X_3, \ldots$について, 標本平均 $\bar{X}_n = (X_1 + \cdots + X_n)/n$ が従う分布は n が大きなとき, 期待値 μ と標準偏差 $\sigma/\sqrt{n}$ を持つ正規分布で近似される. すなわち,

$$Y_n = \sqrt{n}\,(\bar{X} - \mu) = \frac{(X_1 - \mu) + \cdots + (X_n - \mu)}{\sqrt{n}}$$

が従う分布は, nが大きいとき, 期待値 0 と標準偏差 σ を持つ正規分布で近似され,

$$Z_n = \frac{\sqrt{n}\,(\bar{X} - \mu)}{\sigma} = \frac{(X_1 - \mu) + \cdots + (X_n - \mu)}{\sqrt{n}\,\sigma}$$

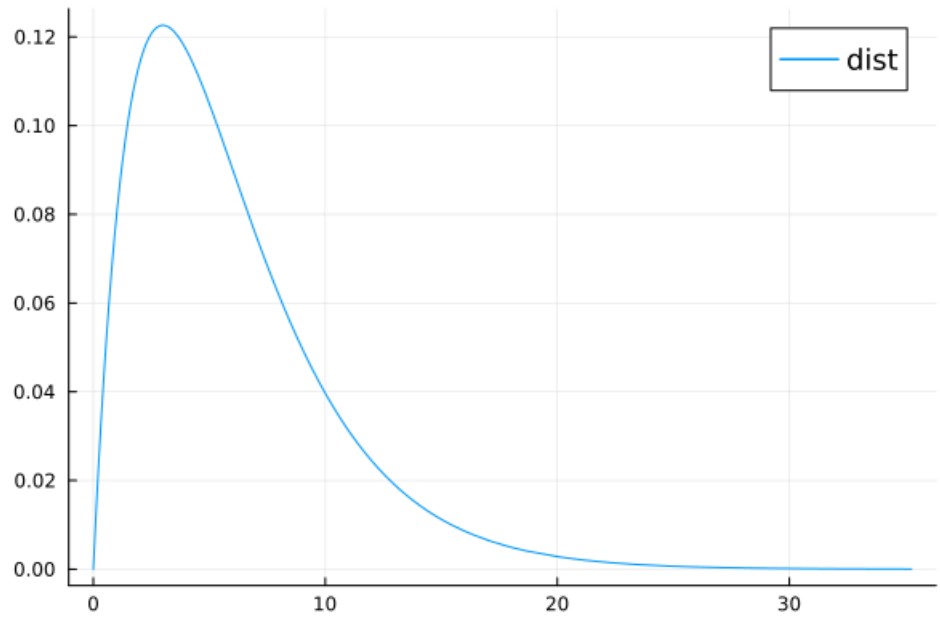
が従う分布は, nが大きいとき, 標準正規分布で近似される.

In [18]:

```
dist = Gamma(2, 3)
@show mu, sigma = mean(dist), std(dist)
plot(dist; label="dist")
```

```
(mu, sigma) = (mean(dist), std(dist)) = (6.0, 4.242640687119285)
```

Out[18]:



以下で使う stephist の説明を見てみよう。

In [19]:

```
?stephist
```

```
search: stephist stephist! scatterhist testhist scatterhist! testhist! barhist
```

Out[19]:

```
stephist(x)
stephist!(x)
```

ヒストグラムステッププロットを作成します（ビンのカウントはバーの代わりに水平線で表されます）。 histogram を参照してください。

stephist は縦線が表示されないヒストグラムを表示してくれる。

In [20]:
```
?histogram
```

```
search: histogram histogram! histogram2d ea_histogram histogram2d! ea_histogram!
```

Out[20]:

```
histogram(x)
histogram!(x)
```

ヒストグラムをプロットします。

引数

- x : ビンに分ける値の AbstractVector
- bins::Union{Integer, Symbol, Tuple{Integer, Integer}, AbstractVector} : デフォルトは :auto (Freedman-Diaconis ルール)。ヒストグラムタイプの場合、目指すビンの概数を定義するか、使用する自動ビン分けアルゴリズムを指定します (:sturges, :sqrt, :rice, :scott または :fd)。細かい制御が必要な場合は、ブレーク値のベクトルを渡します。例: range(minimum(x), stop = maximum(x), length = 25) 。エイリアス: (:bin, :nb, :nbin, :nbins)。
- weights : x の値に対する重みのベクトル、重み付きビンカウント用
- normalize::Union{Bool, Symbol} : ヒストグラムの正規化モード。可能な値は: false/:none (正規化なし、デフォルト)、true/:pdf (離散PDFに正規化、ビンの合計面積が1)、 :probability (ビンの高さの合計が1) および :density (各ビンの面積がカウントに等しい - 不均一なビンサイズに便利)。 エイリアス: (:norm, :normalized, :normalizes, :normed)。
- bar_position::Symbol : :overlay (デフォルト) または :stack から選択します。(警告: 部分的にのみ実装されている可能性があります)。エイリアス: (:bar_positions, :barpositions)。
- bar_width::Real : データ座標におけるバーの幅。 nothing の場合、 x (または orientation = :h の場合は y) に基づいて選択します。エイリアス: (:bar_widths, :barwidths)。
- bar_edges::Bool : バーをエッジ (true) に揃えるか、センター (デフォルト) に揃えるか？。
- permute::Tuple{Symbol, Symbol} : タプルで与えられた軸のデータと軸のプロパティを入れ替えます。例: (:x, :y)。エイリアス: (:permutes,)。

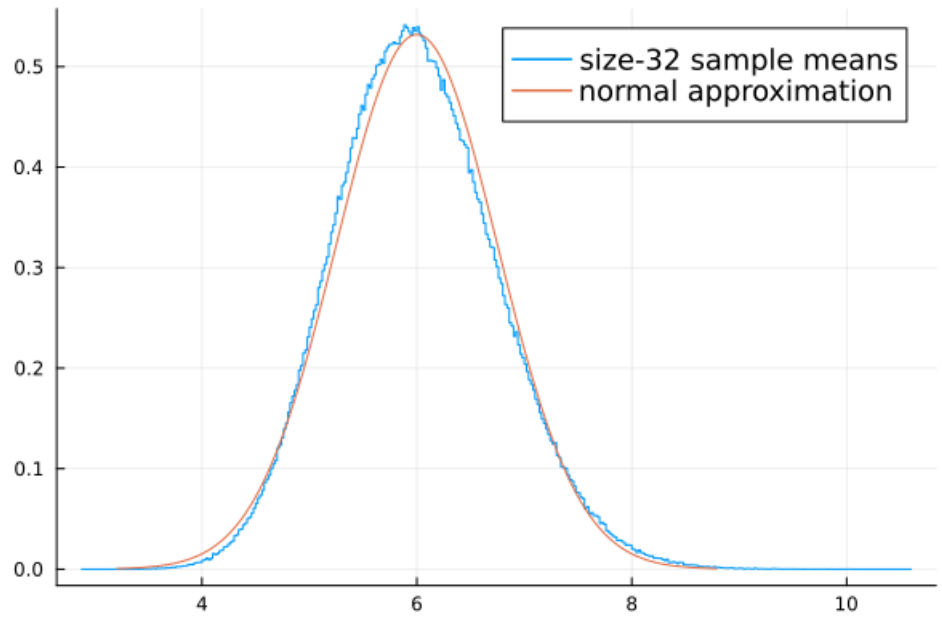
例

```
julia> histogram([1,2,1,1,4,3,8],bins=0:8)
julia> histogram([1,2,1,1,4,3,8],bins=0:8,weights=weights([4,7,3,9,12,2,6]))
```

stephist や histogram では normalize=true もしくは norm=true と設定して使う習慣しておくと、統計学の学習時に便利である。そのようにしておくと、確率密度関数のスケールでヒストグラムをプロットしてくれる。

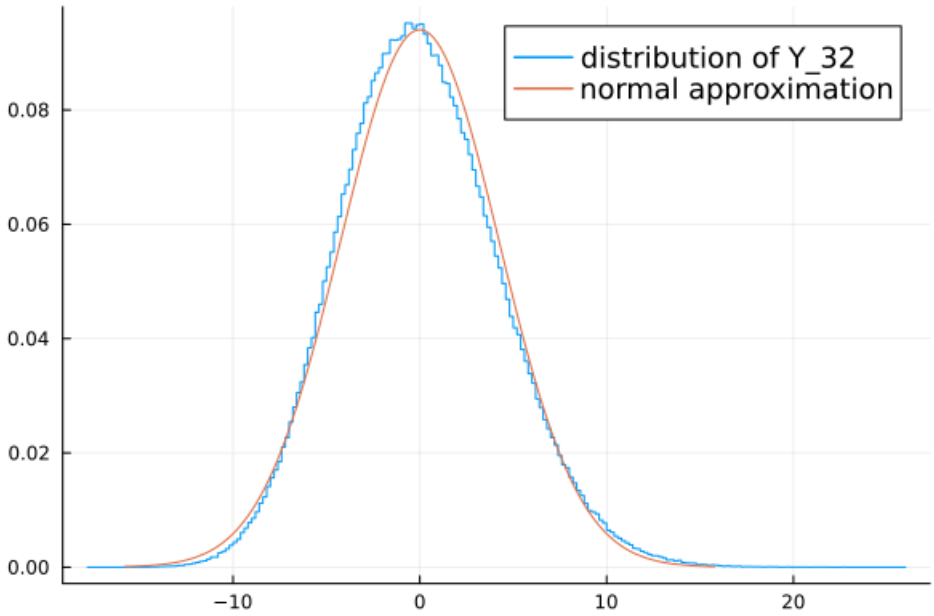
In [21]:
```
n = 2^5 # sample size
niters = 10^6 # number of iterations
Xbars = [mean(rand(dist, n)) for _ in 1:niters] # niters個の標本平均を計算

stephist(Xbars; norm=true, label="size-$n sample means")
plot!(Normal(mu, sigma/sqrt(n)); label="normal approximation")
```

Out[21]:



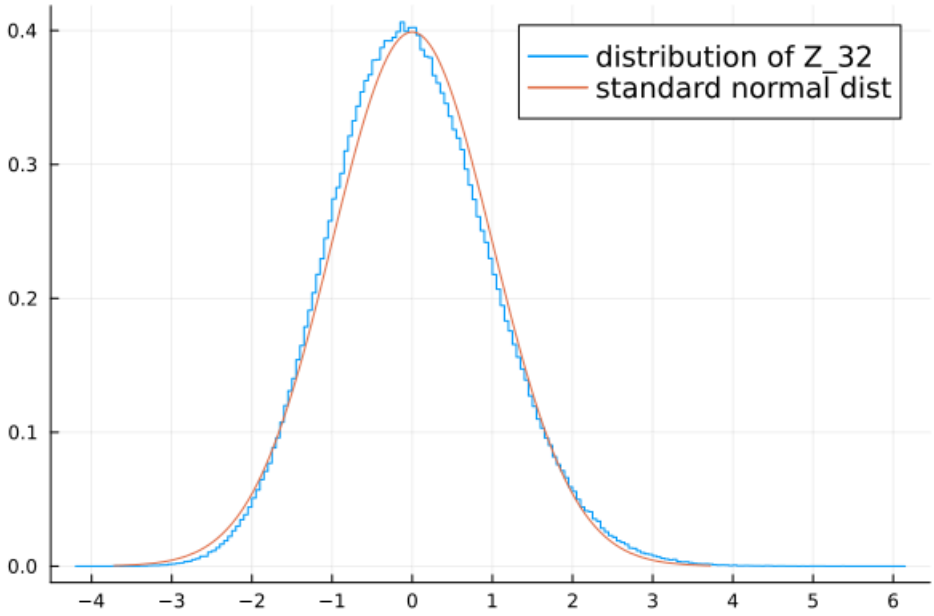
In [22]:
```
n = 2^5 # sample size
Yns = [sqrt(n) * (Xbar - mu) for Xbar in Xbars] # Z_nを繰り返し計算

stephist(Yns; norm=true, label="distribution of Y_$n")
plot!(Normal(0, sigma); label="normal approximation")
```

Out[22]:



In [23]:
```
n = 2^5 # sample size
Zns = [sqrt(n) * (Xbar - mu) / sigma for Xbar in Xbars] # Z_nを繰り返し計算

stephist(Zns; norm=true, label="distribution of Z_$n")
plot!(Normal(); label="standard normal dist")
plot!(xtick=-10:10)
```

Out[23]:



以下は自由に使って下さい

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:

```
1
```