Malaria transmission model

$$
\begin{aligned}
\frac{dS}{dt} &= \frac{P}{L} - \left(\lambda + \frac{1}{L}\right) S + \frac{1}{d_{\text{imm}}} R \\
\frac{dI_1}{dt} &= \lambda S - \left(\frac{\eta_0 p_1}{d_{\text{treat}}} + \frac{1-\eta_0 p_1}{d_{\text{in}}} + \frac{1}{L}\right) I_1 \\
\frac{dI_2}{dt} &= \lambda R - \left(\frac{\eta_0 p_2}{d_{\text{treat}}} + \frac{1-\eta_0 p_2}{d_{\text{in}}} + \frac{1}{L}\right) I_2 \\
\frac{dR}{dt} &= \left(\frac{\eta_0 p_1}{d_{\text{treat}}} + \frac{1-\eta_0 p_1}{d_{\text{in}}}\right) I_1 + \left(\frac{n_0 p_2}{d_{\text{treat}}} + \frac{1-\eta_0 p_2}{d_{\text{in}}}\right) I_2 - \left(\lambda + \frac{1}{L} + \frac{1}{d_{\text{imm}}}\right) R \\
\frac{dW}{dt} &= \lambda S \eta_0 p_1 + \lambda R \eta_0 p_2
\end{aligned}
$$

where

$$\lambda = r_0 \left(\frac{1}{L} + \frac{\eta_0 p_1}{d_{\text{treat}}} + \frac{1-\eta_0 p_1}{d_{\text{in}}}\right) \frac{I_1 + I_2}{P}$$

at steady state, and

$$\lambda(t) = \left(A \cos\left(2\pi(t-\phi)\right) + r_0\right) \left(\frac{1}{L} + \frac{\eta_0 p_1}{d_{\text{treat}}} + \frac{1-\eta_0 p_1}{d_{\text{in}}}\right) \frac{I_1 + I_2}{P}$$

otherwise. Parameter values used for simulation of the ODE are

parameter	definition	value
P	population size	10^6
L	average life expectancy	50 years
d_{imm}	average duration of immunity in the absence of re-exposure	1 year
d_{in}	average duration of untreated infections	0.5 year
d_{treat}	average duraction of treated sensitive infection	2 weeks
p_1	average proportion of infected individuals without preexisting immunity that have clinical malaria	0.87
p_2	average proportion of infected individuals that have clinical malaria who have experienced infection previously within a year	0.08
η_0	proportion of individuals with clinical infection that receive treatment	0.11
r_0	basic reproductive number	1.23
ϕ		0.25
A		0.67

Observations are given by

$$\log(y(t)) \sim \text{Normal}(\log(W(t) - W(t-1)), \sigma^2)$$

where $t = 1, \dots, T$ and $W(0) = 0$, and the unknowns parameters are $\theta = (d_{\text{in}}, \phi, \eta_0, \sigma)$ with prior

$$\begin{aligned} d_{\text{in}} &\sim \text{Half-Normal}(2^2) \\ \phi, \eta_0 &\sim \text{Logit-Normal}(0, 1) \\ \sigma &\sim \text{Log-Normal}(10, 4^2) \end{aligned}$$

A log tranformation is used for d_{in} and σ, and a logit transformation for ϕ and η_0.

Model code

```julia
using Pkg
Pkg.activate(".")

pkgs = string.([
    :Random,
    :Distributions,
    :StatsBase,
    :LinearAlgebra,
    :PDMats,
    :Roots,
    :LogExpFunctions,
    :Printf,
    :DifferentialEquations,
    :CairoMakie,
    :PlotlyJS,
    :SpecialFunctions
])

Pkg.add(pkgs)

using Random, Distributions, StatsBase
using LinearAlgebra, PDMats, Roots
using LogExpFunctions
using Printf
using DifferentialEquations
using CairoMakie, PlotlyJS
using SpecialFunctions
```

ODE

```
function mt_drift0!(dx, x, θ, t)
  P, L, dimm, din, dtreat, p1, p2, A, r0, φ, η0 = θ

  a1 = η0*p1/dtreat + (1-η0*p1)/din
  a2 = η0*p2/dtreat + (1-η0*p2)/din
  λ = r0*(1/L + a1)*(x[2]+x[3])/P

  dx[1] = P/L - (λ + 1/L)*x[1] + x[4]/dimm
  dx[2] = λ*x[1] - (a1 + 1/L)*x[2]
  dx[3] = λ*x[4] - (a2 + 1/L)*x[3]
  dx[4] = a1*x[2] + a2*x[3] - (λ + 1/dimm + 1/L)*x[4]
end


function mt_drift!(dx, x, θ, t)
  P, L, dimm, din, dtreat, p1, p2, A, r0, φ, η0 = θ

  a1 = η0*p1/dtreat + (1-η0*p1)/din
  a2 = η0*p2/dtreat + (1-η0*p2)/din
  λ = (A*cos(2*pi*(t-φ)) + r0)*(1/L + a1)*(x[2]+x[3])/P

  dx[1] = P/L - (λ + 1/L)*x[1] + x[4]/dimm
  dx[2] = λ*x[1] - (a1 + 1/L)*x[2]
  dx[3] = λ*x[4] - (a2 + 1/L)*x[3]
  dx[4] = a1*x[2] + a2*x[3] - (λ + 1/dimm + 1/L)*x[4]
  dx[5] = λ * x[1] * η0 * p1 + λ * x[4] * η0 * p2
end
```

Prior

```
prior_dists = [Normal(0, 2), Normal(0, 1), Normal(0, 1), Normal(10, 4)]

function prior_logpdf(tθ; which_parameters = "all")
  f = 0.5 * log(2/pi) - log(prior_dists[1].σ) - exp(2*tθ[1])/(2*prior_dists[1].σ^2) +
↪  tθ[1] # log-pdf of log-transformed half-normal dist
  if which_parameters == "mean"
    f += logpdf(product_distribution(prior_dists[2:3]), tθ[2:3])
  else which_parameters == "all"
    f += logpdf(product_distribution(prior_dists[2:4]), tθ[2:4])
  end
  return f
```

```julia
end

function prior_rnd(N; which_parameters = "all")
  if which_parameters == "mean"
    draws = rand(product_distribution(prior_dists[1:3]), N)
  else
    draws = rand(product_distribution(prior_dists), N)
  end
  draws[1, :] = log.(abs.(draws[1, :]))
  return [draws[:, j] for j = 1:N]
end

function transform(θ; which_parameters = "all")
  if which_parameters == "mean"
    return [log(θ[1]), logit(θ[2]), logit(θ[3])]
  elseif which_parameters == "noise"
    return log(θ[1])
  else
    return [log(θ[1]), logit(θ[2]), logit(θ[3]), log(θ[4])]
  end
end

function transform_back(tθ; which_parameters = "all")
  if which_parameters == "mean"
    return [exp(tθ[1]), logistic(tθ[2]), logistic(tθ[3])]
  elseif which_parameters == "noise"
    return exp(tθ[1])
  else
    return [exp(tθ[1]), logistic(tθ[2]), logistic(tθ[3]), exp(tθ[4])]
  end
end
```

Likelihood

```julia
function likelihood_mean(init_cond0, tθ, T, dt)
  # P, L, dimm, din, dtreat, p1, p2, A, r0, φ, η0
  θ = transform_back(tθ; which_parameters = "mean")
  params = [29203486, 66.67, 0.93, θ[1], 3/52, 0.87, 0.08, 0.67, 1.23, θ[2], θ[3]]

  prob0 = SteadyStateProblem(mt_drift0!, init_cond0, params)
  sol0 = solve(prob0, DynamicSS(Tsit5());verbose=false)
```

```
  init_cond = [sol0.u; 0.0]

  prob1 = ODEProblem(mt_drift!, init_cond, (0.0, T), params)
  sol = solve(prob1, Tsit5(); saveat=dt, save_idxs = [5], verbose=false)

  x_diff = max.(diff(reduce(vcat, sol.u)), 0.0) # convert to count per month
  return log.(x_diff)
end

function likelihood_logpdf(y, init_cond0, tθ, T, dt)
  x = likelihood_mean(init_cond0, tθ[1:3], T, dt)
  if length(x) < length(y)
    return -Inf
  else
    σ = transform_back(tθ[4]; which_parameters = "noise")
    return sum(logpdf.(Normal.(x, σ), y))
  end
end

function likelihood_logpdf(y, x, σ)
  if length(x) < length(y)
    return -Inf
  else
    return sum(logpdf.(Normal.(x, σ), y))
  end
end

function likelihood_rnd(init_cond0, tθ, T, dt; σ = NaN)
  x = likelihood_mean(init_cond0, tθ[1:3], T, dt)
  if isnan(σ)
    σ = transform_back(tθ[4]; which_parameters = "noise")
  end
  return rand.(Normal.(x, σ))
end
```

Sequential Monte Carlo (SMC) sampler

```
function estimate_ess(g_new::Float64, g::Float64, loglikelihood::Vector{Float64})
    logW = (g_new - g)*loglikelihood
```

```julia
    logW[isnan.(logW)] .= [-Inf]
    logNW = logW .- LogExpFunctions.logsumexp(logW)
    return exp(-LogExpFunctions.logsumexp(2*logNW))
end

function MCMC_mutation!(tθ_particles, θ_loglike, cts, tθ_cov, g, T, dt, init_cond0, y, N)
    # current posterior
    logposterior_curr = prior_logpdf.(tθ_particles) .+ g.*θ_loglike

    # proposal
    prop_tθ = rand.(MvNormal.(tθ_particles, Ref(tθ_cov)))
    prop_ll = likelihood_logpdf.(Ref(y), Ref(init_cond0), prop_tθ, T, dt)
    logposterior_prop = prior_logpdf.(prop_tθ) .+ g.*prop_ll

    # accept/reject
    inds = findall(exp.(logposterior_prop .- logposterior_curr) .> rand(N))
    tθ_particles[inds] .= prop_tθ[inds]
    θ_loglike[inds] .= prop_ll[inds]
    cts[inds] .+= 1
end

function SMC(N, target_ess)
  # initialise
  tθ_particles = prior_rnd(N)
  θ_loglike = likelihood_logpdf.(Ref(y), Ref(init_cond0), tθ_particles, T, dt)
  θ_logweights = zeros(N) .- log(N)

  g = 0.0
  g_hist = [0.0]
  R_hist = []

  S = 20
  while g < 1.0
      # reweight
      if estimate_ess(1.0, g, θ_loglike) >= target_ess
        g = 1.0
      else
        g = find_zero(newg -> estimate_ess(newg, g, θ_loglike) - target_ess, (g + eps(Float64), 1.0))
      end
      ess = estimate_ess(g, g_hist[end], θ_loglike)
```

```
    θ_logW = (g - g_hist[end]).*θ_loglike
    push!(g_hist, g)
    θ_logweights .= θ_logW .- LogExpFunctions.logsumexp(θ_logW)

    display(@sprintf("next g is %f", g))
    display(@sprintf("ESS is %f", ess))

    # resample
    inds = StatsBase.sample(1:N, Weights(exp.(θ_logweights)), N) # multinomial
↪   resampling
    tθ_particles .= tθ_particles[inds]
    θ_loglike .= θ_loglike[inds]
    θ_logweights .= zeros(N) .- log(N)

    # MCMC mutation
    tθ_cov = cov(tθ_particles)
    cts = zeros(N)
    @inbounds for _ in 1:S
        MCMC_mutation!(tθ_particles, θ_loglike, cts, tθ_cov, g, T, dt, init_cond0, y,
        ↪   N)
    end
    p = mean(cts./S)
    R = min(ceil(S/p), 5000)
    display(@sprintf("Number of MCMC repeats: %d", R))
    @inbounds for _ in 1:(R-S)
        MCMC_mutation!(tθ_particles, θ_loglike, cts, tθ_cov, g, T, dt, init_cond0, y,
        ↪   N)
    end
    push!(R_hist, R)
  end
  return tθ_particles, R_hist, g_hist
end
```

Ensemble Kalman inversion (EKI) sampler

```
function cross_cov(x::Array, y::Array)
    μx = mean(x)
    μy = mean(y)
```

```
    N = size(x, 1)
    Nx = length(x[1])
    Ny = length(y[1])

    C = zeros(Nx, Ny)
    for n = 1:N
        C = C + (x[n] - μx)*(y[n] - μy)'
    end
    C = C/(N-1)
    return C
end

function EKI(N, target_ess, σ; initial_resample = true)
  tθ_particles = prior_rnd(N; which_parameters = "mean")
  θ_loglike_mean = likelihood_mean.(Ref(init_cond0), tθ_particles, T, dt)
  θ_loglike = likelihood_logpdf.(Ref(y), θ_loglike_mean, σ)
  θ_logweights = zeros(N) .- log(N)
  Γ = σ^2*diagm(ones(length(y)))

  g = 0.0
  g_hist = [0.0]

  # This step isn't a standard part of EKI, but it gets rid of the worst of the prior draws.
  if initial_resample
    # initial reweight and resample
    g = find_zero(newg -> estimate_ess(newg, g, θ_loglike) - target_ess, (eps(Float64), 1.0))
    θ_logW = (g - g_hist[end]).*θ_loglike
    θ_logweights .= θ_logW .- LogExpFunctions.logsumexp(θ_logW)
    push!(g_hist, g)

    inds = StatsBase.sample(1:N, Weights(exp.(θ_logweights)), N) # multinomial resampling
    tθ_particles .= tθ_particles[inds]
    θ_loglike_mean .= θ_loglike_mean[inds]
    θ_loglike .= θ_loglike[inds]
    θ_logweights .= zeros(N) .- log(N)
  end

  while g < 1
      # determine stepsize
```

```
    if estimate_ess(1.0, g, θ_loglike) >= target_ess
      g = 1.0
    else
      g = find_zero(newg -> estimate_ess(newg, g, θ_loglike) - target_ess, (g +
↪  eps(Float64), 1.0))
    end
    Δg = g - g_hist[end]
    push!(g_hist, g)
    display(@sprintf("next g is %f", g))

    # Kalman gain matrix (H)
    Cgg = cov(θ_loglike_mean)
    Cθg = cross_cov(tθ_particles, θ_loglike_mean)
    H = Cθg*pinv(Cgg + (1/Δg)*Γ)

    # update particles
    kernel_means = tθ_particles .+ Ref(H).*(Ref(y) .- θ_loglike_mean)
    kernel_cov = Hermitian((1/Δg)*(H*Γ*H'))
    tθ_particles = rand.(MvNormal.(kernel_means, Ref(kernel_cov)))

    # update log-likelihood
    θ_loglike_mean = likelihood_mean.(Ref(init_cond0), tθ_particles, T, dt)
    θ_loglike = likelihood_logpdf.(Ref(y), θ_loglike_mean, σ)
  end
  return tθ_particles, g_hist
end
```

Plots

```
function gen_plots(tθ_particles, y, init_cond0, T, dt, filename; σ = NaN)

  if isnan(σ)
    θ_particles = transform_back.(tθ_particles)
  else
    θ_particles = transform_back.(tθ_particles; which_parameters = "mean")
  end

  np = length(θ_particles[1])
  colors = Makie.wong_colors()
```

```
  fig = Figure(size = (850, 565))

  ######## marginal posterior densities
  θ_matrix = reduce(hcat, θ_particles)'
  param_names = ["din", "φ", "η0", "σ"]
  for j = 1:np
      ax = Axis(fig[1, j], title = param_names[j], xticklabelsize = 10.0)
      density!(ax, θ_matrix[:,j], color = (colors[1], 0.0), strokecolor = colors[1],
       ↪  strokewidth = 2)
  end

  ######## posterior predictive
  ax = Axis(fig[2, 1:np], title = "Posterior predictive", xticklabelsize = 10.0)

  # predict using the likelihood
  preds = likelihood_rnd.(Ref(init_cond0), tθ_particles, T, dt; σ = σ)
  preds_mat = reduce(hcat, preds)'
  preds_vec = [preds_mat[:,i] for i in 1:length(y)]

  # get percentiles and mean
  pred_lb = percentile.(preds_vec, 2.5)
  pred_mean = mean.(preds_vec)
  pred_ub = percentile.(preds_vec, 97.5)

  # plot
  t_ind = collect(0:dt:T)
  lines!(ax, t_ind, pred_mean, color = colors[1], linewidth = 2)
  band!(ax, t_ind, pred_lb, pred_ub, color = (colors[1], 0.5))
  scatter!(ax, t_ind, y, color = :black, label = "data")
  CairoMakie.save(filename*".png", fig)
end
```

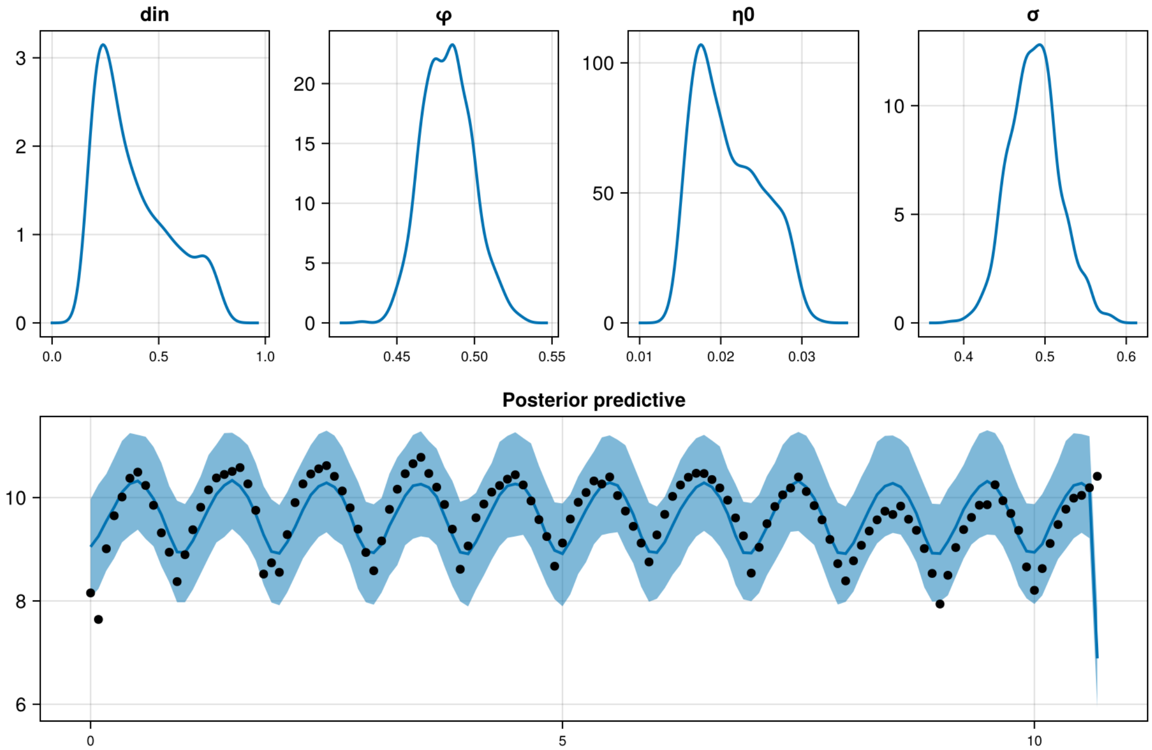
Afghanistan data

```
y = log.([3477.25092158, 2082.2602636, 8177.80134323, 15493.864566, 22287.2797051,
      32042.3673181, 36046.3061152, 27856.1329092, 18969.6005656, 11128.6168762,
      7643.53885775, 4332.17189315, 7291.31949705, 11819.4212998, 18264.6568702,
      25580.4676059, 32200.1716912, 34464.7275665, 36552.0375701, 39338.9890421,
      28710.0439327, 17210.018684, 5013.12932384, 6231.12659698, 5184.31550775,
```

```
      10757.9659648, 19990.9104681, 28700.7019139, 34797.5054285, 38454.7795789,
      40892.2890471, 33224.5114377, 25034.8432056, 18065.1921426, 11966.621219,
      7610.46306115, 5344.6447508, 9523.30454982, 17535.4744231, 25897.591274,
      34955.5622885, 42445.3365652, 48018.2295612, 35124.9810635, 26935.0603444,
      19268.0401959, 11950.4620512, 5502.19663687, 8635.56026865, 14905.8223501,
      19435.1865879, 24486.4414483, 27796.0410039, 31453.8201283, 34239.7616523,
      28139.1708327, 20646.3667121, 14373.5797606, 10365.8536585, 5833.20708983,
      9141.79669747, 14540.9786396, 20115.1340706, 24295.5612786, 30391.1023582,
      28647.9321315, 32826.0869565, 22894.7634197, 16970.156037, 12614.2503661,
      9128.66737363, 6338.68605767, 10692.3193456, 15918.2952078, 22537.746806,
      28111.902237, 32641.2664748, 35252.7394839, 35075.2411251, 31241.9835378,
      26537.3933242, 20961.9754583, 14863.1520477, 10506.9938898, 5104.27712973,
      8412.36176337, 13289.1481089, 18514.6189971, 23217.9467757, 26527.2938444,
      32797.3034389, 24781.8512347, 18855.981417, 14325.8597182, 9795.4855325,
      6134.92905115, 4390.74887643, 6479.57380195, 8742.86724234, 11529.3137403,
      14316.5176993, 16928.7481695, 15880.6746453, 18666.8686563, 14483.9165783,
      11695.7026713, 8210.37216583, 5073.4737161, 2806.39297076, 4894.96540928,
      8378.02353179, 11861.0816543, 14995.960208, 19001.9189012, 19174.6200071,
      28232.5910216, 20738.271979, 16208.1502803, 11678.2810685, 5753.42119881,
      3660.05150735, 5575.41786598, 9058.72847548, 13064.9396556, 17593.7989194,
      21774.4786144, 22992.9808615, 26650.0025249, 33268.4441751])

T = 10.67
dt = 1/12 # per month

I1 = 5
I2 = 10
S = 29203486 - I1 - I2
R = 0
init_cond0 = Float64.([S, I1, I2, R])

N = 400
target_ess = N*0.5

tθ_particles_smc, R_hist, g_hist = SMC(N, target_ess)
gen_plots(tθ_particles_smc, y, init_cond0, T, dt, "mt_smc_plots")
mean(transform_back.(tθ_particles_smc))

σ = 0.5 # fix based on SMC results
tθ_particles, g_hist = EKI(N, target_ess, σ, initial_resample = true)
gen_plots(tθ_particles, y, init_cond0, T, dt, "mt_eki_plots"; σ = σ)
```

SMC results

Top row shows marginal posterior densities of the parameters, and second row shows the posterior predictive (mean and 95% credible intervals) against the observations.



EKI results