Investigating parameter identifiability and "sloppiness" in a dynamical model of reading

Abstract from our 2024 Virtual MathPsych conference talk:

> Theories of many cognitive processes can be expressed as dynamical process models. In order to test the hypotheses that the models implement, we must calibrate them to experimental data by fitting free parameters. In this work, we study a version of the SWIFT model of eye-movement control during reading (Engbert et al., 2005, Psych. Rev.; Engbert & Rabe, 2023, under review) to illustrate two related issues that can arise in models with multiple free parameters: parameter identifiability and sloppiness. The parameters of a model are identifiable for a given data set when it is possible to find a finite confidence interval for the parameter (Raue et al., 2009, Bioninformatics). When a parameter is non-identifiable, parameter fitting can be difficult and misleading, even if the fitted model's output looks reasonable. Sloppiness arises when there are large differences in how sensitive the model's output is to changes in different parameters (Brown & Sethna, 2003, Phys. Rev. E). Sloppiness can also lead to difficulty in model calibration and make interpreting model output challenging, as an analysis of sloppiness often reveals that there are combinations of parameters that vary systematically together with no change in the model's predictions. To our knowledge, parameter identifiability and sloppiness have received little attention in cognitive science, even though the structure of many models is susceptible to these problems. In this talk, we will discuss methods for identifying and addressing parameter non-identifiability and model sloppiness, which can lead to simpler models and more informative fits to experimental data.

Selection deleted

See the talk for further references and suggestions for additional tools one can use.

Overview

Here, I use simulated data to investigate the identifiability and sloppiness of models. I implement versions of the simplified SWIFT model from Engbert & Rabe (2024, *J.Math.Psych.*) using the probabilistic programming tool Turing.jl to set up the likelihood functions.

```
begin
    using Distributions, StatsBase, LinearAlgebra
    using Plots, StatsPlots
    using Turing, Optim, ForwardDiff, DynamicPPL
    using ExponentialUtilities
    using OrderedCollections, DataFrames
    using PlutoUI
end
```

Table of Contents


```
TableOfContents()
```

Selection deleted

Simple model of eye-movement control

The overall processing dynamics are closely based on Engbert & Rabe (2024, *J.Math.Psych.*). The main exception for the sloppy model has to do with how word frequency, activation spreading, and activation growth interact. In the sloppy model:

1. The maximum activation for each word is set to 1.0 and
2. Word frequency affects activation spreading: $\lambda(1 - \exp(-\beta * wordfreq))$, which in turn interacts with the `r` parameter contolling how fast activation grows when a word is within the processing span.

The following functions are all common to both the sloppy and correct model versions.

```
update_proc_rates! (generic function with 3 methods)
```

```
function update_proc_rates!(λ, currword, ν=0.3, σ=inv(1 + 2*ν + ν^2))
    nwords = length(λ)
    λ .= zero(eltype(λ))
    (currword-1 >= 1) && (λ[currword-1] = ν * σ)
    λ[currword] = σ
    (currword+1 <= nwords) && (λ[currword+1] = ν * σ)
    (currword+2 <= nwords) && (λ[currword+2] = ν^2 * σ)
    nothing
end
```

update_activations! (generic function with 2 methods)

```
function update_activations!(activations, r, λ, fixdur, maxact=1.0)
    activations .+= r .* λ .* fixdur .* 0.001
    idx = activations .>= maxact
    activations[idx] .= maxact[idx]
    nothing
end
```

update_saliencies! (generic function with 3 methods)

```
function update_saliencies!(activations, saliencies, maxact=1.0, minsal=0.001)
    saliencies .= (maxact .* sinpi.(activations ./ maxact)) .+ minsal
    nothing
end
```

update_probabilities! (generic function with 2 methods)

```
function update_probabilities!(probs, sal, γ=1.0)
    salexp = sal.^γ
    probs .= salexp ./ sum(salexp)
    nothing
end
```

Simulating word frequencies

In order to test the models, we need to generate data. Since the word frequency parameter β plays a large role in our analyses, we need to have a realistic distribution of word frequencies. For that, I generate word frequencies according to a Zipfian distribution.

Selection deleted

zetaprobs (generic function with 1 method)

```
function zetaprobs(s, xs)
    ptemp = xs.^(-s) ./ sum(x^(-s) for x in 1:maximum(xs))#zeta(s, 1)
    return ptemp ./ sum(ptemp)
end
```

rzipf =

[63, 13, 2, 20, 40, 3, 28, 11, 78, 36, 1, 1, 17, 1, 2, 7, 2, 58, 78, 2, more ,1, 8, 17,

```
# Creating a 100-word corpus with a million samples
rzipf = rand(Categorical(zetaprobs(1.1, 1:100)), 1_000_000)
```

cts =

Dict(5 ⇒ 39815, 56 ⇒ 2776, 35 ⇒ 4640, 55 ⇒ 2995, 60 ⇒ 2654, 30 ⇒ 5510, 32 ⇒ 5212,

```
cts = countmap(rzipf)
```

mx = 233042

```
mx = maximum(values(cts))
```

nlogfreqs =

Dict(5 ⇒ 0.170849, 56 ⇒ 0.011912, 35 ⇒ 0.0199106, 55 ⇒ 0.0128518, 60 ⇒ 0.0113885, 3

```
nlogfreqs = Dict(k => cts[k] / mx for k in keys(cts))
```

```
[0.515873, 0.916474, 1.0, 1.0]
```

```
begin
    atemp = fill(0.5, 4)
    update_activations!(atemp, p0.r * 10, ones(4) .* (1 .- exp.(-1 .* p0.β .*
    range(0.01, 0.99, 4))), 100, ones(4))
    atemp
end
```

Actually simulating an experimental data set

For demonstration purposes, I am not simulating different parameters for each participant or item. I am just simulating a number of sentences with a single parameter.

Selection deleted

simsent

```
simsent(nwords, ps, worddict, sentnr = missing)
```

Simulates a single sentence with `nwords` using parameters `ps` and a dictionary containing word indices and their normalized log frequencies. `sentnr` is an optional sentence ID to save with the generated data.

```
"""
    simsent(nwords, ps, worddict, sentnr = missing)

Simulates a single sentence with `nwords` using parameters `ps` and a dictionary containing word indices and their normalized log frequencies. `sentnr` is an optional sentence ID to save with the generated data.
"""
function simsent(nwords, ps, worddict, sentnr = missing)
    shapeparam = 9.0
    rate0 = shapeparam / ps.μ
    σ = inv(1 + 2*ps.ν + ps.ν^2)
    minsal = 10^-3
    wordids = rand(keys(worddict), nwords)
    #maxact = 1 .- ps.β .* [nlogfreqs[k] for k in wordids]
    maxact = ones(nwords)
    betas = ps.β .* [nlogfreqs[k] for k in wordids]
    activations = zeros( nwords)
    saliencies = zeros(nwords)
    probabilities = zeros(nwords)
    λ = zeros(nwords)
    currword = 1
    # Preallocating a place to save scanpath
    scanpath = Array{NamedTuple}(undef, 0)
    sizehint!(scanpath, nwords)

    while any(activations .< maxact) && currword ≠ nwords
    #while any(activations .< 1) && currword < nwords
        # Generate new fixation duration
        currscl = inv(rate0)
        fixdur = currscl / 2 * rand(Chisq(2 * shapeparam))

        # Updating
        update_proc_rates!(λ, currword, ps.ν, σ)
        update_activations!(activations, ps.r * 10, λ .* (1 .- exp.(-1 .* betas)), fixdur, maxact)
        update_saliencies!(activations, saliencies, maxact, minsal)
        update_probabilities!(probabilities, saliencies, ps.γ)

        # Saving
        push!(scanpath, (sentnr = sentnr, sentlength = nwords, fixatedword = currword, wordid = wordids[currword], fixationduration = fixdur, wordids = wordids))

        # Make a saccade
        currword = rand(Categorical(probabilities))
    end
```

Selection deleted

```
    # Last fixation
    currscl = inv(rate0)
    fixdur = currscl / 2 * rand(Chisq(2 * shapeparam))

    push!(scanpath, (sentnr = sentnr, sentlength = nwords, fixatedword = currword,
    wordid = wordids[currword], fixationduration = fixdur, wordids = wordids))

    return scanpath
end
```

```
p0 =   (μ = 0.2, β = 1.6, ν = 0.25, r = 1.0, γ = 1.0, minsal = 0.001)
```

```
# True parameter settings for generating data
p0 = (μ = 0.200, β = 1.6, ν = 0.25, r = 1.0, γ = 1.0, minsal=0.001)
```

```
[(sentnr = 1, sentlength = 10, fixatedword = 1, wordid = 90, fixationduration = 0.159033
```

```
simsent(10, p0, nlogfreqs, 1)
```

simexp

Simulate an experiment.

```
"""
Simulate an experiment.
"""
function simexp(nwordrange, nsents, worddict, ps)
    # Preallocating space to save data
    expdata = Array{NamedTuple}(undef, 0)
    sizehint!(expdata, maximum(nwordrange) * nsents)
    for i in 1:nsents
        nwords = rand(nwordrange)
        push!(expdata, simsent(nwords, ps, worddict, i)...)
    end
    return DataFrame(expdata)
end
```

	sentnr	sentlength	fixatedword	wordid	fixationduration	wordids
1	1	6	1	50	0.26628	[50, 58, 16, 55, 74, 9
2	1	6	3	16	0.141109	[50, 58, 16, 55, 74, 9
3	1	6	6	93	0.241074	[50, 58, 16, 55, 74, 9
4	2	11	1	85	0.158381	[85, 33, 54, 16, 55, 5
5	2	11	4	16	0.264585	[85, 33, 54, 16, 55, 5
6	2	11	5	55	0.279725	[85, 33, 54, 16, 55, 5
7	2	11	10	64	0.19744	[85, 33, 54, 16, 55, 5
8	2	11	3	54	0.270002	[85, 33, 54, 16, 55, 5
9	2	11	8	69	0.163831	[85, 33, 54, 16, 55, 5
10	2	11	1	85	0.233848	[85, 33, 54, 16, 55, 5
more						
1266	100	12	12	51	0.071249	[58, 35, 78, 18, 80, 5

```
begin
    sentlens =  6:12 #20 #4:30 # 10:20 # [2]
    nsents = 100 # 114 # 200
    data = simexp(sentlens, nsents, nlogfreqs, p0)
end
```

Setting up the sloppy model

```
basemodel (generic function with 4 methods)
```

```
@model function basemodel(nlogfreqs, data, fixatedwords, fixationdurations, ::Type{T} = Float64) where {T}
    # Priors
    μ ~ LogNormal(log(0.200), 1)
    ν ~ Beta(1.25, 1.25)
    β ~ Normal(0, 0.05)
    r ~ LogNormal(log(1), 0.5)
    γ = one(T)
    minsal = 10^-3 * one(T)

    shapeparam = 9.0
    rate0 = shapeparam / μ
    σ = inv(1 + 2*ν + ν^2)

    Threads.@threads for pi in data
        nwords = pi[1, :sentlength]
        maxact = ones(T, nwords)
        nfix = nrow(pi)
        betas = β .* [nlogfreqs[k] for k in pi[1, :wordids]]

        activations = zeros(T, nwords)
        saliencies = zeros(T, nwords)
        probabilities = zeros(T, nwords)
        λ = zeros(T, nwords)
        currword = pi[1, :fixatedword]

        for row in eachrow(@view pi[1:end-1, :])
            rowidx = parentindices(row)[1]

            update_proc_rates!(λ, currword, ν, σ)
            # Introducing an interaction between ν and β
            update_activations!(activations, r * 10, λ .* (1 .- exp.(-1 .* betas)), row.fixationduration, maxact)
            update_saliencies!(activations, saliencies, maxact, minsal)
            update_probabilities!(probabilities, saliencies, γ)

            # Spatial LL
            if !isprobvec(probabilities)
                Turing.@addlogprob! -Inf
            else
                fixatedwords[rowidx+1] ~ Categorical(probabilities)
            end

            # Temporal LL
            currword = fixatedwords[rowidx+1]
            currscl = inv(rate0)
            if currscl <= eps(typeof(currscl))
                Turing.@addlogprob! -Inf
            else
                fixationdurations[currword] ~ currscl / 2 * Chisq(2 * shapeparam)
            end
        end
```

```
    end
end
```

```
gdata = groupby(data, :sentnr);
```

```
mod = basemodel(nlogfreqs, gdata, data.fixatedword, data.fixationduration);
```

```
(μ = 0.2, β = 1.6, ν = 0.25, r = 1.0, γ = 1.0, minsal = 0.001)
```

```
p0
```

Finding the MLE

The MLE will be important for a number of techniques below, so I fit it first.

```
mle0 =
ModeResult with maximized lp of -721.78
[0.21210957911517067, 0.3105451950105806, 2.6894479576224315, 0.5475136418966452]
```

```
mle0 = optimize(mod, MLE(), [p0.μ, p0.ν, p0.β, p0.r]; autodiff=:forward)
```

```
paramnames =  (:μ, :ν, :β, :r)
```

```
paramnames = DynamicPPL.syms(DynamicPPL.VarInfo(mod))
```

Bias and coefficient of variation

A first check we can do is to look at the bias in the MLE estimates. We see that there is quite a bit of bias, especially for β and r. Bias should be as close to zero as possible (Cole, 2020, *Parameter redundancy and identifiability*), but β is off by ~400% and r is off by 45%.

Selection deleted

The coefficient of variability (SE(θ) / $\bar{\theta}$) should also be less than one, indicating low variability in the parameter estimate relative to the magnitude of the mean parameter estimate. Here, ν, β, and r are all much higher than μ, although they are less than one.

```
relbias =  [(:μ, 0.0605479), (:ν, 0.0378407), (:β, 4.35779), (:r, 0.452486)]
```

```
# Some of these are quite biased...
relbias = [(p, abs(mle0.values.array[i] - p0[p]) / abs(p0[i])) for (i, p) in
enumerate(paramnames)]
```

```
[(:μ, 0.00976179), (:ν, 0.352069), (:β, 0.652869), (:r, 0.530849)]
```

```
# CVs ≳ 1, too much variability for this sample size
# Note: using the absolute value might be weird
try
    cv = [(p, abs(diag(vcov(mle0))[i])^0.5 / abs(mle0.values.array[i])) for (i, p)
    in enumerate(paramnames)]
catch
    @warn "Bad matrix"
end
```

Observed information matrix and related techniques

The observed information matrix is the negative of the Hessian matrix of the likelihood evaluated at the MLE. The Hessian matrix is the matrix of partial second derivatives of the likelihood function and provides information about the curvature of the likelihood surface.

The observed information matrix tells us how informative small changes in a parameter are with regard to the model's output. If an entry in the observed information matrix is large, then a small change in that parameter leads to a big change in the likelihood of the data.

Often it is most informative to look at an eigenvalue decomposition of the observed information matrix.

```
infomat = 4×4 Named Matrix{Float64}
          A ╲ B │          μ          ν          β          r
          ──────┼──────────────────────────────────────────────
          μ     │ 2.33249e5       -0.0       -0.0       -0.0
          ν     │      -0.0    156.555   -15.0002   -107.488
          β     │      -0.0   -15.0002    3.89593     24.138
          r     │      -0.0   -107.488     24.138    163.533
```

```
1 infomat = informationmatrix(mle0)
```

Selection deleted

```
false
```

```
1 # Positive definiteness as a proxy for a well-behaved matrix
2 isposdef(infomat)
```

Eigendecomposition analysis

```
1 md"""
2 ### Eigendecomposition analysis
3 """
```

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
4-element Vector{Float64}:
      0.31675983200866
     53.18254846834681
    270.4850431378334
 233249.3786217384
vectors:
4×4 Matrix{Float64}:
  0.0         0.0         0.0       1.0
 -0.0103646  -0.723987   -0.689736  0.0
  0.988177   -0.112934    0.103693  0.0
 -0.152967   -0.680506    0.716598  0.0
```

```
1 evals, evecs = eigen(infomat)
```

```
[-0.49927, 1.72577, 2.43214, 5.36782]
```

```
1 log10.(evals)
```

```
5.8670903914874675
```

```
# The ratio of the larges to the smallest eigenvalues provides info about sloppiness
log10(maximum(abs.(evals)) / minimum(abs.(evals)))
```

```
736360.3432374633
```

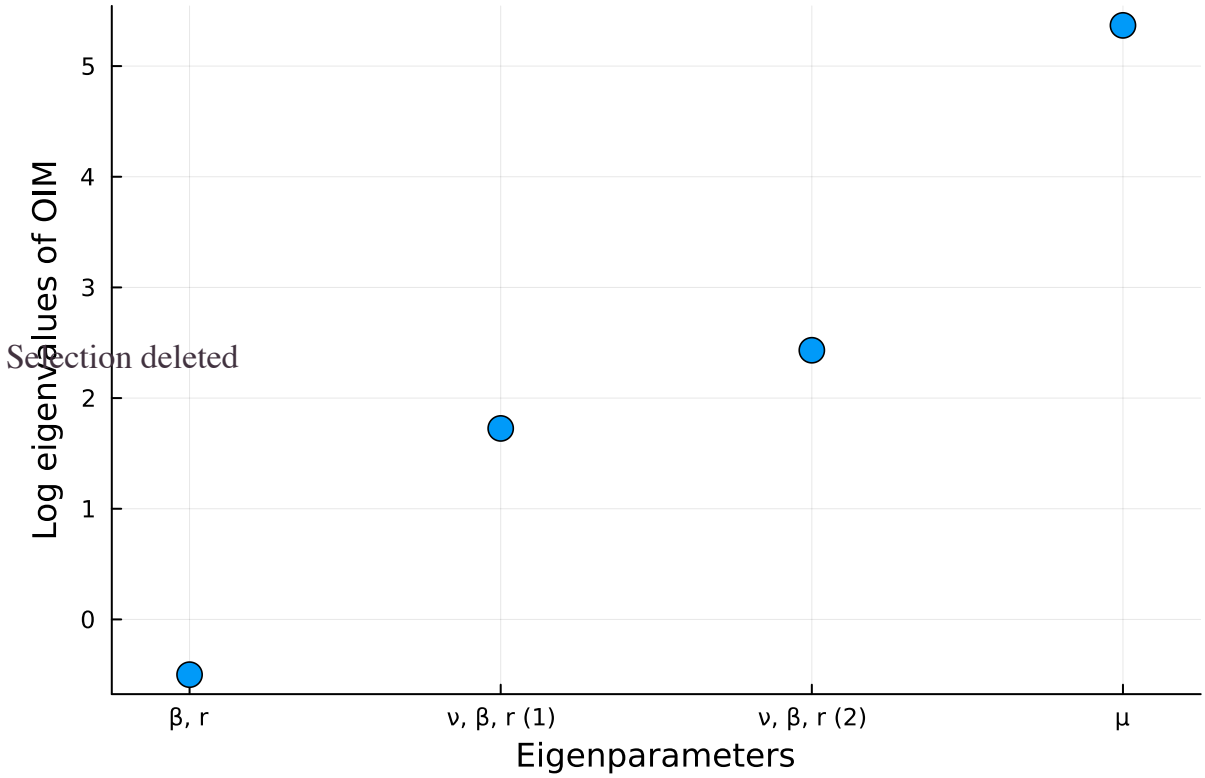
```
cond(infomat)
```

```
[1.35803e-6, 0.000228007, 0.00115964, 1.0]
```

```
# Suggests there are two unidentifiable parameters, r and β
# b/c their eigenvalues are below the cutoff of 0.001, Cole 2020
evals ./ maximum(evals)
```

```
(2, 4)
```

```
rank(infomat, rtol=0.001), rank(infomat)
```




```
begin
    eigvalplot = scatter(1:4, log10.(evals), xlims=(0.75, 4.25), xticks=(1:4, ["β, r", "ν, β, r (1)", "ν, β, r (2)", "μ"]), ylabel="Log eigenvalues of OIM", markersize=7, legend=false, xlabel="Eigenparameters")
end
```

These eigendecomposition analyses provide a number of interesting points.

- First, the log eigenvalues span more than three orders of magnitude and they are more or less evenly spaced from smalles to largest, so the common criteria for sloppiness are met.

  This means that some parameters will be easy to estimate while others become gradually hard to estimate. For the parameters associated with the smallest eigenvalues, there will be broad swaths of parameter space that all produce roughly the same model behavior (as indexed by the likelihood function).

- In addition, using a loose tolerance to calculate the rank of the observed information matrix, we have evidence that the parameters `r` and `β` are nearly redundant (Cole, 2020): There is a very similar model with those parameters eliminated or combined that produces approximately the same behavior.

From these analyses, we should expect there to be parameter (combinations) for which large changes only result in small changes in model predictions.

```
#savefig(eigvalplot, "eigvalplot.pdf")
```

The expected Fisher information matrix

The expected Fisher information matrix (FIM) is closely related to the observed information matrix. The Fisher information matrix is evaluated at the *true* parameter values instead of the MLE (like the observed information matrix), so the FIM provides information about how easy/hard it is to recover the true parameter values instead of how hard it was to estimate the MLE *for this particular data set*. Here, we're estimating the expected Fisher information with a numerical approximation. The observed information (above) also tells us about how easy/hard it is to estimate, but it's based *on a single data set* and is therefore less general. (See, e.g., Pawitan, 2001 *In all likelihood* for a discussion of the differences.)

The discussion in Pawitan (2001, Ch. 8.3, pp. 216-217) can be summarized as: increasing FIM means the parameter(s) is/are easier to estimate, and we can get away with smaller sample. On the other hand, a lower FIM means harder estimation and a need for more data.

Selection deleted

The results of the analyses almost always point in the same direction.

efim

```
efim(gdata, mle, returnscore=false)
```

Calculate an estimate of the Fisher information matrix, evaluated at `mle` and averaged over each data point. Optionally returns the average of the score function (i.e., the average gradient of the log-likelihood).

```
"""
    efim(gdata, mle, returnscore=false)

Calculate an estimate of the Fisher information matrix, evaluated at `mle` and averaged over each data point. Optionally returns the average of the score function (i.e., the average gradient of the log-likelihood).
"""
function efim(gdata, mle, returnscore=false)
    I = zeros(size(mle, 1), size(mle, 1))
    score = zeros(size(mle, 1), length(gdata))
    for i in 1:length(gdata)
        mod = basemodel(nlogfreqs, gdata[i:i], data.fixatedword, data.fixationduration)
        ∇l = ForwardDiff.gradient(x -> loglikelihood(mod, (μ = x[1], ν = x[2], β = x[3], r = x[4])), mle)
        I .+= ∇l * ∇l'
        score .+= ∇l
    end
    if returnscore
        return I ./ length(gdata), score
    else
        return I ./ length(gdata)
    end
end
```

```
(
  1: 4×4 Matrix{Float64}:
      13131.0      28.9977    -7.82127    -15.7218
         28.9977    0.937233  -0.167057    -0.32007
         -7.82127  -0.167057   0.0691093    0.127382
        -15.7218   -0.32007    0.127382     0.244859
  2: 4×100 Matrix{Float64}:
      3176.95     3176.95     3176.95     …  3176.95     3176.95     3176.95
        26.0128     26.0128     26.0128        26.0128     26.0128     26.0128
        -6.86654    -6.86654    -6.86654       -6.86654    -6.86654    -6.86654
       -14.2402    -14.2402    -14.2402       -14.2402    -14.2402    -14.2402
)
```

```
# Pawitan 2001, Ch. 8.3, p. 216--217:
# Fisher information evaluated at param values that generated the data
fim, score = efim(gdata, [p0.μ, p0.ν, p0.β, p0.r], true)#, p0.γ], true)#, p0.γ, p0.minsal])#, true)
#fim = efim(gdata, [p0.μ, p0.β, p0.r, p0.γ])#, p0.γ, p0.minsal])#, true)
```

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
4-element Vector{Float64}:
     0.0022195191167488776
     0.14553476414081848
     1.0159209308547876
 13131.038753229173
vectors:
4×4 Matrix{Float64}:
 -3.07509e-5  0.000328073   0.00256077   0.999997
 -0.00118372  0.404949     -0.914336     0.00220852
  0.883936    0.427974      0.188399    -0.000595673
 -0.467606    0.807994      0.358454    -0.00119738
```

```
1 fevals, fevecs = eigen(fim)
```

Selection deleted

```
1 scatter(ones(length(paramnames)), log10.(fevals), xlims=(0.5, 1.5), xticks=nothing,
  ylabel="Order of magnitude of eigenvalues", legend=false, marker=:rect,
  markersize=5)
```

```
((:μ, :ν, :β, :r), (μ = 0.2, β = 1.6, ν = 0.25, r = 1.0, γ = 1.0, minsal = 0.001))
```

```
1 paramnames, p0
```

Profile likelihoods

Profile likelihoods approximate a multidimensional likelihood function that would be otherwise hard to visualize. For each parameter in the model, we pick a range of values to evaluate. We then fix the parameter at that value and maximize the restricted likelihood over the remaining parameters. Once we've done this for the whole range of values, we can plot the profile likelihood for the parameter. This is repeated for each parameter in the model.

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

```
function profll(mod, var, vals, inits; mll=nothing)
    mles = zeros(size(vals, 1))
    prevmle = zeros(length(inits))
    for (i, v) in enumerate(vals)
        if i == 1
            omle = optimize(fix(mod, (; var => v)), MLE(), inits; autodiff=:forward)
            mles[i] = omle.lp
            prevmle = omle.values.array
        else # Use the previous MLE as the starting point.
            omle = optimize(fix(mod, (; var => v)), MLE(), prevmle;
            autodiff=:forward)
            mles[i] = omle.lp
            prevmle = omle.values.array
        end
    end
    return mles
end
```

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

```
function allprofs(mod, vals, inits; mll=nothing)
    @assert size(vals, 1) == length(inits) "Number of ranges must match number of
    variables"
    vars = DynamicPPL.syms(DynamicPPL.VarInfo(mod))
    mles = OrderedDict(v => Float64[] for v in vars)
    for (i, v) in enumerate(vars)
        mles[v] = profll(mod, v, vals[i], inits[eachindex(inits) .∉ [i]]; mll)
    end
    return mles
end
```

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

```
function plotprofs(lls, vals, mles = nothing, trues = nothing, args...; cutoff =
nothing, ml = nothing, kwargs...)
    p = plot(layout=(ceil(Int, length(lls) // 2), 2), margin=1*Plots.mm, args...;
    kwargs...)
    for (i, v) in enumerate(keys(lls))
        if !isnothing(ml)
            plot!(p[i], vals[i], lls[v] .- ml[i], xlabel=String(v), label=nothing)
        else
            plot!(p[i], vals[i], lls[v], xlabel=String(v), label=nothing)
        end
        if cutoff != nothing
            hline!(p[i], [cutoff], label="Cutoff")
        end
        if mles != nothing
            vline!(p[i], [mles[i]], label="MLE")
        end
        if trues != nothing
            vline!(p[i], [trues[v]], label="True value")
        end
    end
    return p
end
```

nvals = 50

```
nvals = 50
```

OrderedDict(:μ ⇒ [-1767.06, -1705.46, -1646.63, -1590.49, -1536.91, -1485.79, -1437.06,

```
begin
    vals = [range(0.14, 0.22, nvals), # μ
            range(0.01, 0.99, nvals),
            range(0.01, 4, nvals), # β
            range(0.01, 2, nvals)]
    lls = allprofs(mod, vals, mle0.values.array; mll=mle0.lp)
end
```

OrderedDict(:μ ⇒ [-1767.06, -1705.46, -1646.63, -1590.49, -1536.91, -1485.79, -1437.06,

```
lls
```

cutoff = -2.0794415416798357

```
# When considering all params at once:
#cutoff = -0.5 * quantile(Chisq(length(paramnames)), 0.95) # mle0.lp

# Likelihood interval, Royall 1997
cutoff = -log(8)
#cutoff = log(95/100)
```

(-4.74386, -3.46574, -2.07944, -4.56435)

```
-0.5 * quantile(Chisq(length(paramnames)), 0.95), -log(32), -log(8), -log(32 *
(length(paramnames) - 1))
```

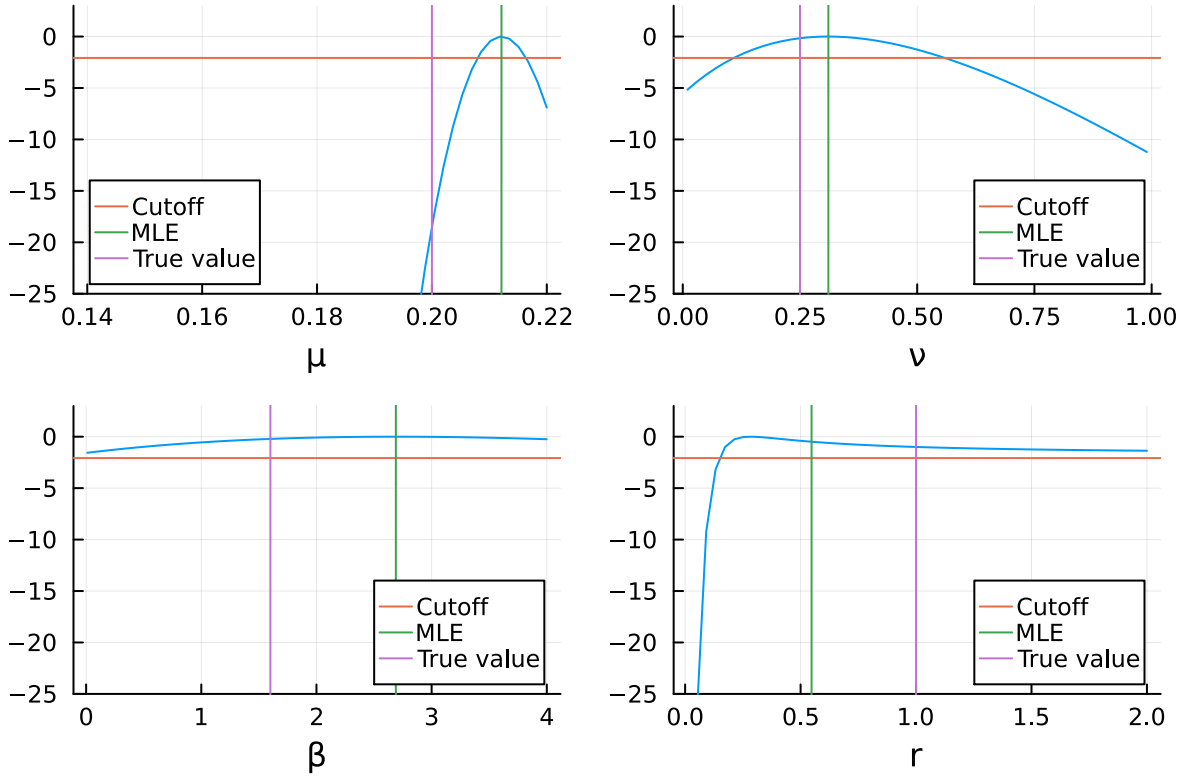
prof_mle = [-721.784, -721.775, -721.775, -727.366]

```
prof_mle = [maximum(x) for x in values(lls)]
```

((μ = 0.2, β = 1.6, ν = 0.25, r = 1.0, γ = 1.0, minsal = 0.001), 100)

```
p0, nsents
```

```
sloppyprofs =
```




```
1 sloppyprofs = plotprofs(lls, vals, mle0.values.array, p0; cutoff=cutoff,
  ml=prof_mle, ylims=(-25, 3))# ml=mle0.lp)
2 #plotprofs(lls, vals, nothing, (l1 = λ₁, l2 = λ₂); cutoff=nothing, ml=hypomle.lp)
```

```
1 #savefig(sloppyprofs, "sloppy_profs.pdf")
```

Interpreting the profile likelihoods

The profile likelihoods show us a couple of interesting things:

- First, there is often significant bias in the MLEs: The estimated and true values are often not very close.
- Second, the `r` parameter, which controls the speed of activation increase, seems to be practically non-identifiable (Raue et al., 2009, *Bioinformatics*). That is, its confidence interval is not bounded; it flattens out above the cutoff towards positive infinity.

  While there does seem to be a maximum for `r`, there is still an infinite range of `r` values that produce the same model behavior.
- It will likely be hard to estimate `β`: Its profile is quite flat, which means there are many parameter settings that are almost identical in their output, a sign of sloppiness. This is reinforced by the eigenvalue associated with the combination `r-β` eigen parameter shown above, which was much smaller than the other eigenvalues.

Overall, the profile likelihoods paint much the same picture as the information matrix analyses: The `r` and `β` parameter will be hard or impossible to estimate, as there are parameter settings that produce (almost) identical model outputs. The model is both sloppy and practically non-identifiable.

The correct model

Following Engbert & Rabe (2024, J.Math.Psych.)

The difference between this model and the sloppy model is that the β (word frequency) and r (activation growth rate) parameters are better separated and have stronger and more independent effects on the model output.

```
correctmodel (generic function with 4 methods)
```

```
@model function correctmodel(nlogfreqs, data, fixatedwords, fixationdurations,
::Type{T} = Float64) where {T}
    # Priors
    μ ~ LogNormal(log(0.200), 1)
    ν ~ Beta(1.25, 1.25)
    β ~ Beta(1.25, 1.25)
    r ~ LogNormal(log(1), 0.5)
    γ = one(T)
    minsal = 10^-3 * one(T)

    shapeparam = 9.0
    rate0 = shapeparam / μ
    σ = inv(1 + 2*ν + ν^2)

    Threads.@threads for pi in data
        nwords = pi[1, :sentlength]
        nfix = nrow(pi)
        maxact = 1 .- β .* [nlogfreqs[k] for k in pi[1, :wordids]]

        activations = zeros(T, nwords)
        saliencies = zeros(T, nwords)
        probabilities = zeros(T, nwords)
        λ = zeros(T, nwords)
        currword = pi[1, :fixatedword]

        for row in eachrow(@view pi[1:end-1, :])
            rowidx = parentindices(row)[1]

            update_proc_rates!(λ, currword, ν, σ)
            update_activations!(activations, r * 10, λ, row.fixationduration,
            maxact)
            update_saliencies!(activations, saliencies, maxact, minsal)
            update_probabilities!(probabilities, saliencies, γ)

            # Spatial LL
            if !isprobvec(probabilities)
                Turing.@addlogprob! -Inf
            else
                fixatedwords[rowidx+1] ~ Categorical(probabilities)
            end

            # Temporal LL
            currword = fixatedwords[rowidx+1]
            currscl = inv(rate0)
            if currscl <= eps(typeof(currscl))
                Turing.@addlogprob! -Inf
            else
                fixationdurations[currword] ~ currscl / 2 * Chisq(2 * shapeparam)
            end
        end
    end
end
```

simsent_correct

```
simsent_correct(nwords, ps, worddict, sentnr = missing)
```

Simulates a single sentence with `nwords` using parameters `ps` and a dictionary containing word indices and their normalized log frequencies. `sentnr` is an optional sentence ID to save with the generated data.

```
"""
    simsent_correct(nwords, ps, worddict, sentnr = missing)

Simulates a single sentence with `nwords` using parameters `ps` and a dictionary containing word indices and their normalized log frequencies. `sentnr` is an optional sentence ID to save with the generated data.
"""
function simsent_correct(nwords, ps, worddict, sentnr = missing)
    shapeparam = 9.0
    rate0 = shapeparam / ps.μ
    σ = inv(1 + 2*ps.ν + ps.ν^2)
    minsal = 10^-3
    wordids = rand(keys(worddict), nwords)
    maxact = 1 .- ps.β .* [nlogfreqs[k] for k in wordids]
    activations = zeros( nwords)
    saliencies = zeros(nwords)
    probabilities = zeros(nwords)
    λ = zeros(nwords)
    currword = 1
    # Preallocating a place to save scanpath
    scanpath = Array{NamedTuple}(undef, 0)
    sizehint!(scanpath, nwords)

    while any(activations .< maxact) && currword ≠ nwords
        # Generate new fixation duration
        currscl = inv(rate0)
        fixdur = currscl / 2 * rand(Chisq(2 * shapeparam))

        # Updating
        update_proc_rates!(λ, currword, ps.ν, σ)
        update_activations!(activations, ps.r * 10, λ, fixdur, maxact)
        update_saliencies!(activations, saliencies, maxact, minsal)
        update_probabilities!(probabilities, saliencies, ps.γ)

        # Saving
        push!(scanpath, (sentnr = sentnr, sentlength = nwords, fixatedword =
        currword, wordid = wordids[currword], fixationduration = fixdur, wordids =
        wordids))

        # Make a saccade
        currword = rand(Categorical(probabilities))
    end

    # Last fixation
    currscl = inv(rate0)
    fixdur = currscl / 2 * rand(Chisq(2 * shapeparam))
```

```
43
44      push!(scanpath, (sentnr = sentnr, sentlength = nwords, fixatedword = currword,
        wordid = wordids[currword], fixationduration = fixdur, wordids = wordids))
45
46      return scanpath
47  end
```

simexp_correct

Simulate an experiment.

```
"""
Simulate an experiment.
"""
function simexp_correct(nwordrange, nsents, worddict, ps)
    # Preallocating space to save data
    expdata = Array{NamedTuple}(undef, 0)
    sizehint!(expdata, maximum(nwordrange) * nsents)
    for i in 1:nsents
        nwords = rand(nwordrange)
        push!(expdata, simsent_correct(nwords, ps, worddict, i)...)
    end
    return DataFrame(expdata)
end
```

p0correct = (μ = 0.2, β = 0.6, ν = 0.25, r = 1.0, γ = 1.0, minsal = 0.001)

```
p0correct = (μ = 0.20, β = 0.6, ν = 0.25, r = 1.0, γ = 1.0, minsal = 0.001)
```

(μ = 0.2, β = 1.6, ν = 0.25, r = 1.0, γ = 1.0, minsal = 0.001)

```
p0
```

```
begin
    correctdata = simexp_correct(sentlens, nsents, nlogfreqs, p0correct);
    gdata_correct = groupby(correctdata, :sentnr);
    nothing
end
```

```
correctmod = correctmodel(nlogfreqs, gdata_correct, correctdata.fixatedword,
correctdata.fixationduration);
```

mlecorrect =
ModeResult with maximized lp of -10544.08
[0.194766929318489, 0.26691378211759853, 0.6015899725275644, 0.9791758946694226]

```
mlecorrect = optimize(correctmod, MLE(), [p0correct.μ, p0correct.ν, p0correct.β,
p0correct.r]; autodiff=:forward)
```

cinfomat = 4×4 Named Matrix{Float64}

A \ B	μ	ν	β	r
μ	2.4895e6	-0.0	-0.0	-0.0
ν	-0.0	6363.83	78.9172	110.372
β	-0.0	78.9172	224.492	492.484
r	-0.0	110.372	492.484	2163.73

```
cinfomat = informationmatrix(mlecorrect)
```

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
4-element Vector{Float64}:
  106.1726206776973
 2277.762037847354
 6368.116838238123
    2.489497805554399e6
vectors:
4×4 Matrix{Float64}:
  0.0         0.0         0.0        1.0
 -0.0081674  -0.0307426  -0.999494   0.0
  0.972596    0.232011   -0.0150838  0.0
 -0.232357    0.972227   -0.0280052  0.0
```

```
1 cevals, cevecs = eigen(cinfomat)
```

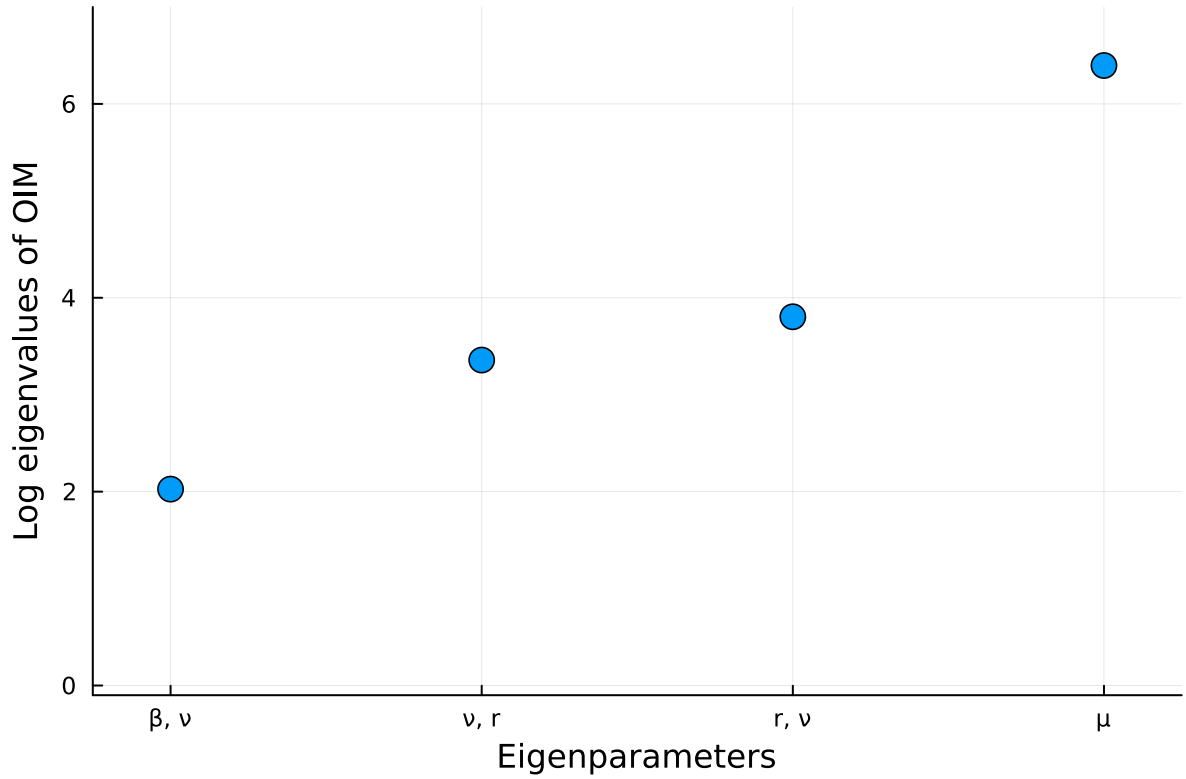
```
[3.157, 1.90502, 1.68141, 1.0]
```

```
1  maximum(log10, cevals) ./ log10.(cevals)
```

```
(23447.6, 4)
```

```
1 cond(cinfomat), rank(cinfomat)
```

eigvalplot2 =




```
1 eigvalplot2 = scatter(1:4, log10.(cevals), xlims=(0.75, 4.25), xticks=(1:4, ["β,
  ν", "ν, r", "r, ν", "μ"]), ylabel="Log eigenvalues of OIM", markersize=7,
  legend=false, xlabel="Eigenparameters", ylims=(-0.1, 7))
```

```
1 #savefig(eigvalplot2, "eigvalplot2.pdf")
```

```
valscorrect =
  StepRangeLen{Float64, Base.TwicePrecision{Float64}, Base.TwicePrecision{Float64}, Int64
     1:  0.18:0.0008163265306122456:0.22000000000000003
     2:  0.01:0.02:0.99
     3:  0.01:0.02:0.99
     4:  0.75:0.01020408163265306:1.25
]
```

```
valscorrect = [range(p0.μ - 0.1*p0.μ, 0.1*p0.μ + p0.μ, nvals),
               range(0.01, 0.99, nvals), # ν
               range(0.01, 0.99, nvals), # β
               range(0.75, 1.25, nvals)] # r
```

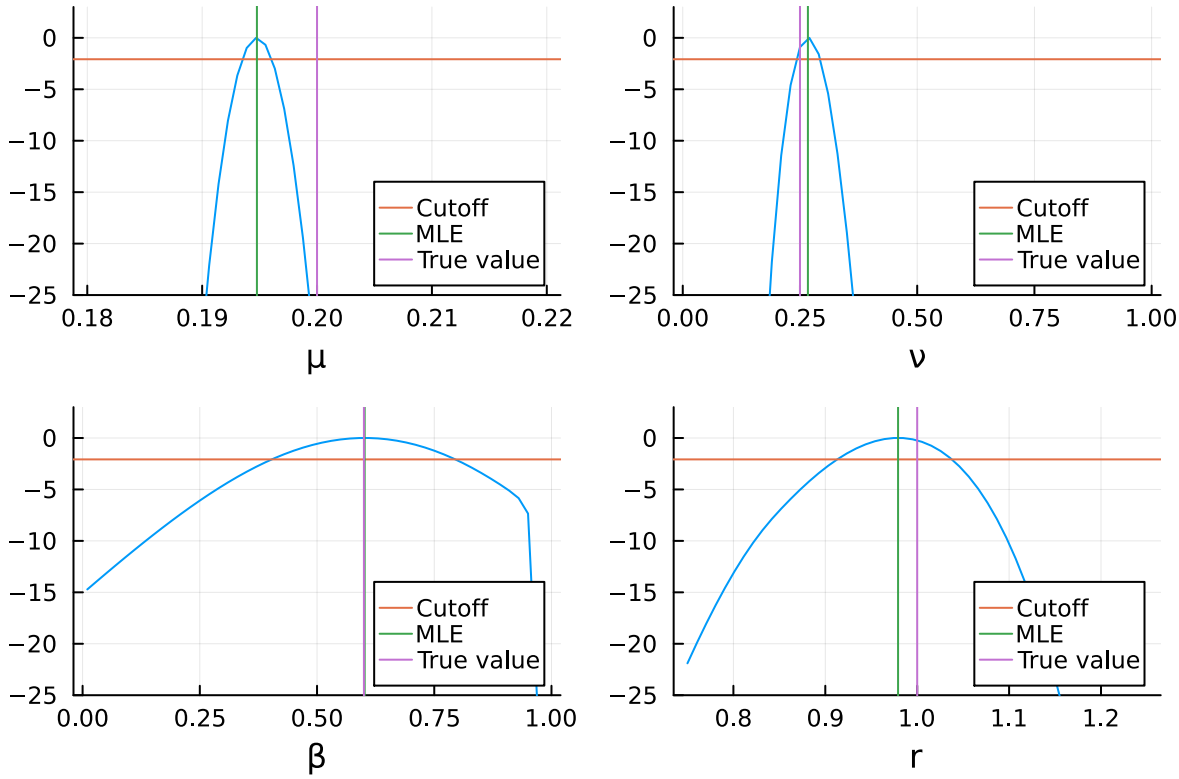
```
llscorrect =
  OrderedDict(:μ ⇒ [-10845.5, -10811.5, -10779.7, -10750.1, -10722.6, -10697.3, -10674.0,
```

```
llscorrect = allprofs(correctmod, valscorrect, mlecorrect.values.array;
mll=mlecorrect.lp)
```

```
prof_mle_correct =  [-10544.1, -10544.1, -10544.1, -10544.1]
```

```
prof_mle_correct = [maximum(x) for x in values(llscorrect)]
```

```
cprofplot =
```




```
cprofplot = plotprofs(llscorrect, valscorrect, mlecorrect.values.array, p0correct;
cutoff=cutoff, ml=prof_mle_correct, ylims=(-25, 3))
```

```
#savefig(cprofplot, "non-sloppy-profs.pdf")
```

Diagnostics for the correct model

As shown above, the diagnostics for sloppiness and identifiability look much better for the correct model:

- The eigenvalues of the observed information matrix are much less spread out and higher overall, indicating that each of the (eigen-) parameters has a stronger effect on the model's predictions.
- The profile likelihoods show clear peaks near the true parameter values. There is less bias, the confidence intervals are all bounded, and the peaks are more strongly curved, which all point to much improved identifiability.