Coding Stochastic Weather Generators: Challenges and Perspectives

David Métivier

StochasticWeatherGenerators.jl

INRAE
science for people, life & earth

Mistea
Mathématiques, Informatique et STatistique pour l'Environnement et l'Agronomie

The SWG Workflow

1. Need: Identify the application (impact study, risk assessment, etc.)
2. Existing model: Survey literature, test if available models work

The SWG Workflow

1. Need: Identify the application (impact study, risk assessment, etc.)
2. Existing model: Survey literature, test if available models work
3. New model: Propose modifications or new approaches
4. Data: Access and prepare weather station/gridded data

The SWG Workflow

1. Need: Identify the application (impact study, risk assessment, etc.)
2. Existing model: Survey literature, test if available models work
3. New model: Propose modifications or new approaches
4. Data: Access and prepare weather station/gridded data
5. Code: Fitting log-likelihood and simulation

The SWG Workflow

1. Need: Identify the application (impact study, risk assessment, etc.)
2. Existing model: Survey literature, test if available models work
3. New model: Propose modifications or new approaches
4. Data: Access and prepare weather station/gridded data
5. Code: Fitting log-likelihood and simulation
6. Validation: Check model performance against observations against other models?
7. Publication: Write paper, share code/package?

Major challenges

1. **Model fitting**
   - How to make our model as we dream of and not transform it to make it compatible with “the” existing package e.g. I want a seasonal model
   - How to optimize complex likelihood functions?

Major challenges

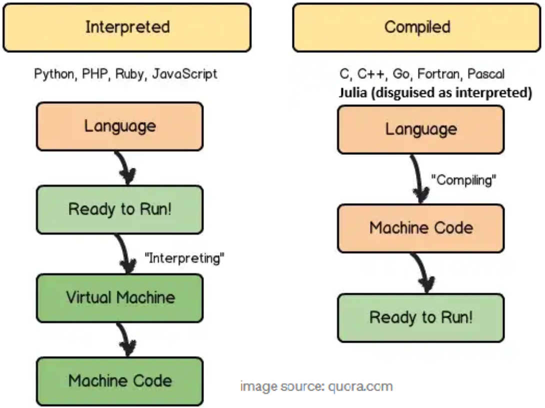
1. **Model fitting**
   - How to make our model as we dream of and not transform it to make it compatible with “the” existing package e.g. I want a seasonal model
   - How to optimize complex likelihood functions?

2. **Fast scenario generation**
   - One of the biggest selling points of SWGs is speed
   - Need a lot of long time series to explore uncertainties
   - How to write fast simulation code without spending months in C/C++?

Major challenges

1. **Model fitting**
   - How to make our model as we dream of and not transform it to make it compatible with “the” existing package e.g. I want a seasonal model
   - How to optimize complex likelihood functions?
2. **Fast scenario generation**
   - One of the biggest selling points of SWGs is speed
   - Need a lot of long time series to explore uncertainties
   - How to write fast simulation code without spending months in C/C++?
3. **Code accessibility & reuse**
   - Reproducibility is great but different from (re)usability
   - To be user-friendly it has to be somewhat fast
   - Packaging research code is a lot of work (interface, testing, docs, maintenance)

The Two-Language Problem: Compiled vs. Interpreted

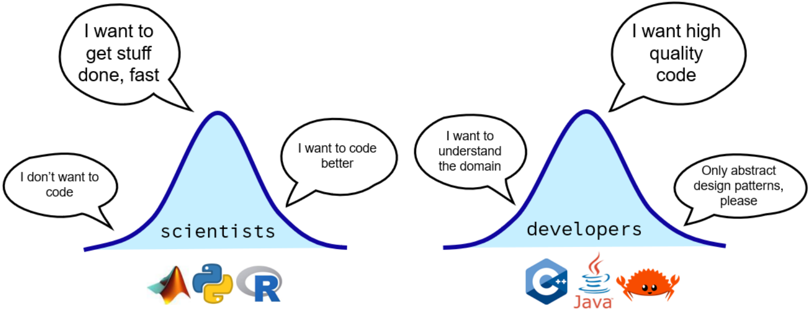


Who knows `C++`? Who wants to learn it?

- **Compiled languages** are fast but hard to write
- **Interpreted languages** are easy to write but slow
- Interpreted languages need compiled languages under the hood

The Two-Language Problem: Scientists vs. Developers

From "My Target Audience" by Matthijs Cox



Common workflow:

1. Prototype in R/Python
2. Rewrite some parts in C++/Fortran or Cython, Rcpp, Pytorch etc.

The Two-Language Problem: Scientists vs. Developers

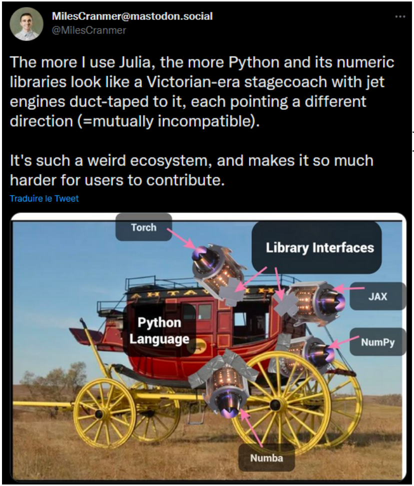
From 'My Target Audience' by Matthijs Cox

Common workflow:

1. Prototype in R/Python
2. Rewrite some parts in C++/Fortran or Cython, Rcpp, Pytorch etc.

or use Julia "feels like Python/R but fast like C" + a lot more

Composability in Python



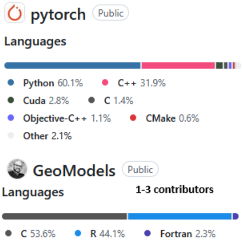
Python HMM packages

- `hmmlearn` (NumPy), `pomegranate` (PyTorch), `dynamax` (JAX)
- Each locked to its framework
- Built-in distributions only e.g.
  `jax.random.multivariate_normal`
  `torch.randn`
  `numpy.random.multivariate_normal`

$\Rightarrow$ Each framework $\rightarrow$ isolated ecosystem = mutually incompatible

Two languages packages examples

glue code

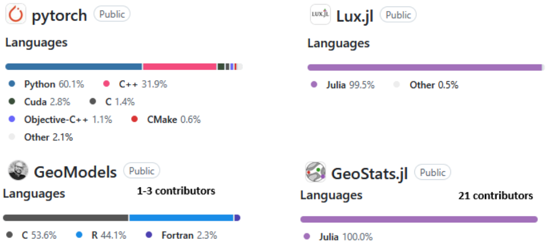


Costs of two languages:

- Few people know both languages well enough to contribute
- Installation/Development/Maintenance nightmare with dependencies

Two languages packages examples

glue code



Costs of two languages:

- Few people know both languages well enough to contribute
- Installation/Development/Maintenance nightmare with dependencies

In Julia most packages just work together!

- Each package = one domain (distributions, optim, diff equations, etc.)
- Improving one package can benefit all others and users
- **Easier to contribute!**

Simulating SWG

Dynamic languages (R/Python/Matlab) require *vectorization*

- “Life is too short to spend writing for loops” (MATLAB manual)
- “Learning to use vectorized operations is a key skill in R” (R introduction blog post)

Simulating SWG

Dynamic languages (R/Python/Matlab) require *vectorization*

- "Life is too short to spend writing for loops" (MATLAB manual)
- "Learning to use vectorized operations is a key skill in R" (R introduction blog post)

But... Not everything can be vectorized!
Especially for SWG with temporal dependency (e.g., Markov chains, complex dependencies)

Benchmark: HMM Simulation

4 hidden states

12-dimensional MvNormal

$N = 10^4 \sim 27$ year

See the associated notebook.

```julia
function rand_HMM(Q, dist, N)
    Z = zeros(Int, N)
    Y = zeros(N)
    Z[1] = 1  # Z₁ = 1
    Y[1] = rand(dist[Z[1]])
    for t in 2:N
        Z[t] = rand(Categorical(Q[Z[t-1], :]))
        Y[t] = rand(dist[Z[t]])
    end
    return Z, Y
end
```

Benchmark: HMM Simulation

4 hidden states
12-dimensional MvNormal
$N = 10^4 \sim 27$ year
See the associated notebook.

```julia
function rand_HMM(Q, dist, N)
    Z = zeros(Int, N)
    Y = zeros(N)
    Z[1] = 1  # Z₁ = 1
    Y[1] = rand(dist[Z[1]])
    for t in 2:N
        Z[t] = rand(Categorical(Q[Z[t-1], :]))
        Y[t] = rand(dist[Z[t]])
    end
    return Z, Y
end
```

Language	Relative Speed
Julia (baseline)	1×
C	$\sim$ 45×
R	$\sim$ 700×

Key takeaways:

- R loops unavoidably slow (~700× slower)
- C complex & error-prone: bad C can be slower than Julia!
- Julia: prototype $\simeq$ production code

Fitting SWG

Goal: Fit a Gaussian or HMM-based SWG, EGPD distributions, etc.

Fitting SWG

Goal: Fit a Gaussian or HMM-based SWG, EGPD distributions, etc.

- Write down likelihood function
- Instinct: Reduce to a known problem $\Rightarrow$ use an existing package

Fitting SWG

Goal: Fit a Gaussian or HMM-based SWG, EGPD distributions, etc.

- Write down likelihood function
- Instinct: Reduce to a known problem $\Rightarrow$ use an existing package
- \+ Packages are presumably well-tested and optimized
- \+ Save time on implementation

Fitting SWG

Goal: Fit a Gaussian or HMM-based SWG, EGPD distributions, etc.

- Write down likelihood function
- Instinct: Reduce to a known problem ⇒ use an existing package

+ Packages are presumably well-tested and optimized
+ Save time on implementation

But...

- Research models are most of the time somewhat new
- Need flexibility e.g. seasonality, could it be simpler to code yourself?

What is under the hood of the package you use?

When you call `optim`?

Fitting SWG

Goal: Fit a Gaussian or HMM-based SWG, EGPD distributions, etc.

- Write down likelihood function
- Instinct: Reduce to a known problem ⇒ use an existing package

+ Packages are presumably well-tested and optimized
+ Save time on implementation

But...

- Research models are most of the time somewhat new
- Need flexibility e.g. seasonality, could it be simpler to code yourself?

What is under the hood of the package you use?

When you call `optim`?

```
optim(par, fn, gr = NULL, …,
      method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN",
                 "Brent"),
      lower = -Inf, upper = Inf,
      control = list(), hessian = FALSE)
```

Optimization methods

$$\max_{\theta \in \mathbb{R}^d} L(\theta)$$

- Gradient free $\rightarrow$ just evaluate $L(\theta)$ e.g. Nelder-Mead
- Gradient based $\rightarrow$ uses $\nabla L(\theta)$ e.g. (L-)BFGS, Newton's method

Optimization methods

$$\max_{\theta \in \mathbb{R}^d} L(\theta)$$

- Gradient free $\rightarrow$ just evaluate $L(\theta)$ e.g. Nelder-Mead
- Gradient based $\rightarrow$ uses $\nabla L(\theta)$ e.g. (L-)BFGS, Newton's method

1. Analytic/Symbolic: Derive formulas by hand or using computer (Mathematica, SymPy)
   - Labor-intensive, error-prone
   - Expressions explode in size, doesn't scale

Optimization methods

$$\max_{\theta \in \mathbb{R}^d} L(\theta)$$

- Gradient free $\rightarrow$ just evaluate $L(\theta)$ e.g. Nelder-Mead
- Gradient based $\rightarrow$ uses $\nabla L(\theta)$ e.g. (L-)BFGS, Newton's method

1. Analytic/Symbolic: Derive formulas by hand or using computer
2. Approximate: Finite differences $(f(x+h) - f(x))/h$
   - Truncation error + floating point error
   - Slow: $O(n)$ function calls for n parameters

Optimization methods

$$\max_{\theta \in \mathbb{R}^d} L(\theta)$$

- Gradient free $\rightarrow$ just evaluate $L(\theta)$ e.g. Nelder-Mead
- Gradient based $\rightarrow$ uses $\nabla L(\theta)$ e.g. (L-)BFGS, Newton's method

1. Analytic/Symbolic: Derive formulas by hand or using computer
2. Approximate: Finite differences $(f(x+h) - f(x))/h$
3. Automatic: Exact gradients, fast, scales to complex code
    - Evaluate derivatives of functions specified by computer programs
    - Every computation is a *sequence of elementary operations* $(+, -, \times, \exp, \log, \sin, \ldots)$
    - Apply *chain rule repeatedly* to these operations
    - Get exact derivatives to machine precision!

Optimization methods

$$\max_{\theta\in\mathbb{R}^d} L(\theta)$$

- Gradient free $\to$ just evaluate $L(\theta)$ e.g. Nelder-Mead
- Gradient based $\to$ uses $\nabla L(\theta)$ e.g. (L-)BFGS, Newton's method

1. Analytic/Symbolic: Derive formulas by hand or using computer
2. Approximate: Finite differences $(f(x+h)-f(x))/h$
3. Automatic: Exact gradients, fast, scales to complex code

`optim`: Nelder-Mead (gradient free) or BFGS with approx. gradients.
$\to$ Does NOT know how to AD
Julia is automatically differentiable almost everywhere natively

Example: Fitting a Spatial Model

Gaussian Random Field with **Matérn covariance function:**

$$\rho_{\text{Matérn}}(h;\nu,\rho) = \sigma^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{h}{\rho}\right)^{\nu} K_\nu\left(\frac{h}{\rho}\right)$$

where K_ν is the modified Bessel function, ν is smoothness, ρ is range
Julia AD for Matérn covariance: *Geoga et al. (2023)*

Results for 15 locations for
i) estimating (ρ, ν) with $\sigma = 1$ and **ii)** estimating (ρ, ν, σ) :
See the associated notebook.

- Finite differences: at least 10x slower than AD and very bad convergence for **i)** and **ii)**
- Gradient-free (Nelder-Mead): fast and good for **i)**. Did not converge for **ii)**.
- AD: best convergence and reasonable speed for **i)** and **ii)**

Examples: Seasonal (constant per month) Fitting

Parameters are seasonal: **BUT** “the” package only deals with stationary data

1. Fit each month separately $\Rightarrow 12 \times$ YEARS fits $\rightarrow$ take the median/mean of parameters

   ⚠ high variance

Examples: Seasonal (constant per month) Fitting

Parameters are seasonal: **BUT** “the” package only deals with stationary data

1. Fit each month separately $\Rightarrow$ 12 $\times$ YEARS fits $\rightarrow$ take the median/mean of parameters
   ⚠ high variance
2. Concatenate data for each month across years $\Rightarrow$ 12 independent fits
   ⚠ boundary issues

Examples: Seasonal (constant per month) Fitting

Parameters are seasonal: **BUT** "the" package only deals with stationary data

1. Fit each month separately $\Rightarrow$ 12 $\times$ YEARS fits $\rightarrow$ take the median/mean of parameters
   ⚠ high variance
2. Concatenate data for each month across years $\Rightarrow$ 12 independent fits
   ⚠ boundary issues
3. The "correct" likelihood is not that complicated to write:
   $\mathcal{L}(\theta \mid Y^{(1:N)} = y^{(1:N)}) = \sum_{n=1}^{N} \log f_{\theta_{\text{month}(n)}}(Y^{(n)} = y^{(n)} \mid \cdots)$
   ⚠ Probably not in a package + one big fit

Examples: Seasonal (constant per month) Fitting

Parameters are seasonal: **BUT** “the” package only deals with stationary data

1. Fit each month separately $\Rightarrow$ 12 $\times$ YEARS fits $\rightarrow$ take the median/mean of parameters
   ⚠ high variance
2. Concatenate data for each month across years $\Rightarrow$ 12 independent fits
   ⚠ boundary issues
3. The “correct” likelihood is not that complicated to write:
   $\mathcal{L}(\theta \mid Y^{(1:N)} = y^{(1:N)}) = \sum_{n=1}^{N} \log f_{\theta_{\text{month}(n)}}(Y^{(n)} = y^{(n)} \mid \cdots)$
   ⚠ Probably not in a package + one big fit

Toy example 2D-AR(2) with $N = 36524$ (100 years)

$$Y^{(n+1)} = A_{\text{month}(n)} Y^{(n)} + \Sigma_{\text{month}(n)} \epsilon^{(n)}, \quad A, \Sigma \in \mathbb{R}^{2\times 2}, \quad \epsilon^{(n)} \sim \mathcal{N}(0, I_2)$$

	Median (1)	Concatenation (2)	Total likelihood (3)
A_1	7.21%	15.08%	3.96%
A_2	25.45%	17.95%	8.64%
Σ	4.38%	10.48%	1.14%

$\rightarrow$ It does make a difference!

Conclusions

Takeaways:

- How to get speed: glue code or Julia?
- Write your own likelihoods or use packages?
- Which optimization methods (AD, Finite Different, No gradient)?

Packaging research code:

- Packages should be easy to install (few dependencies)
- Packages are hard to make: testing, documentation, user support (**if any**), not always rewarded
- Should benefit everyone, promote code reuse and collaboration
- `StochasticWeatherGenerators.jl`

Goal Aggregate SWG models in one place for easy comparison

? How to leverage Julia ecosystem

TODO Real documentation

Conclusions

Takeaways:

- How to get speed: glue code or Julia?
- Write your own likelihoods or use packages?
- Which optimization methods (AD, Finite Different, No gradient)?

Packaging research code:

- Packages should be easy to install (few dependencies)
- Packages are hard to make: testing, documentation, user support (**if any**), not always rewarded
- Should benefit everyone, promote code reuse and collaboration
- `StochasticWeatherGenerators.jl`

Goal Aggregate SWG models in one place for easy comparison

? How to leverage Julia ecosystem

TODO Real documentation

Thank You!