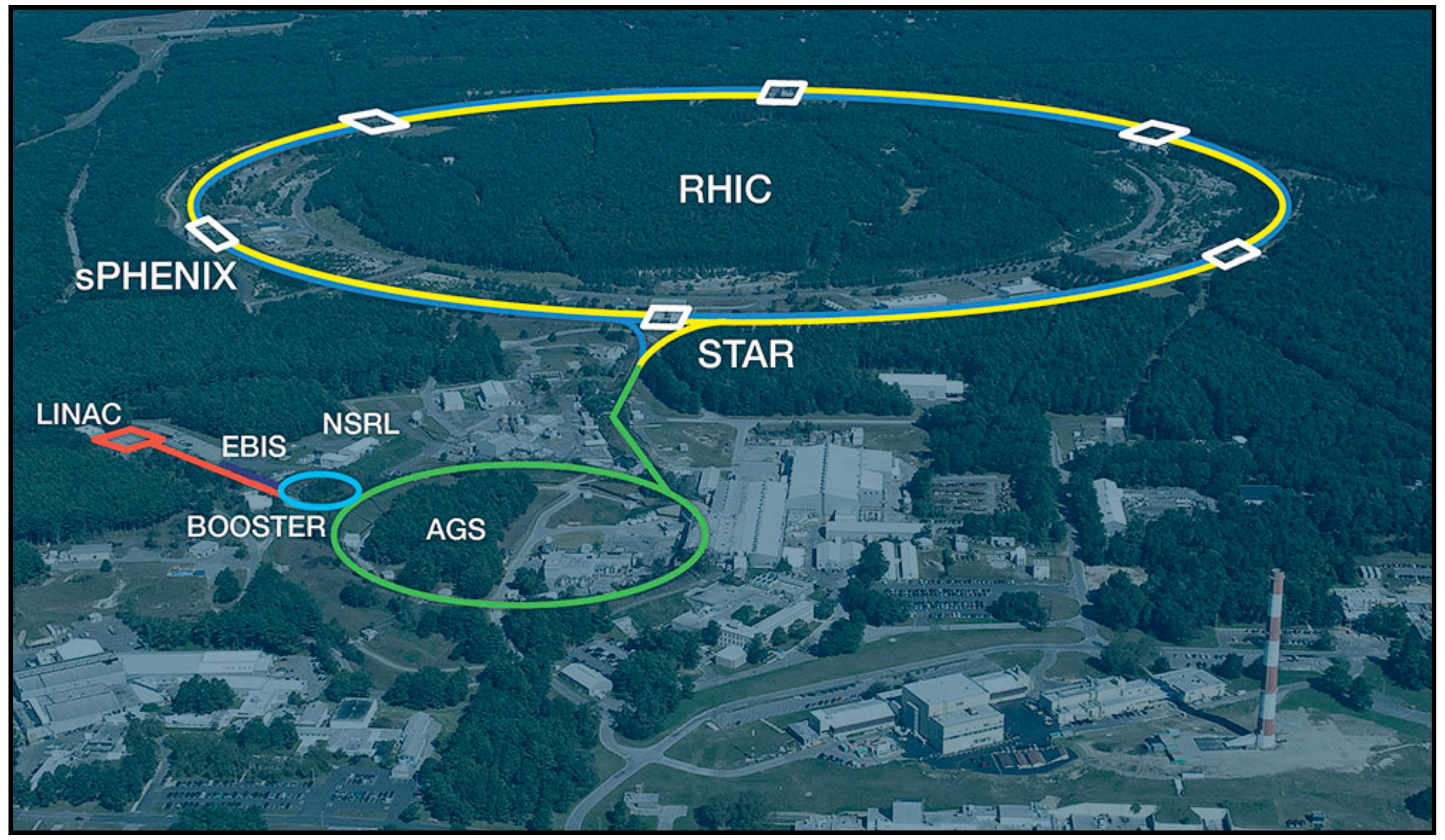
# Overview

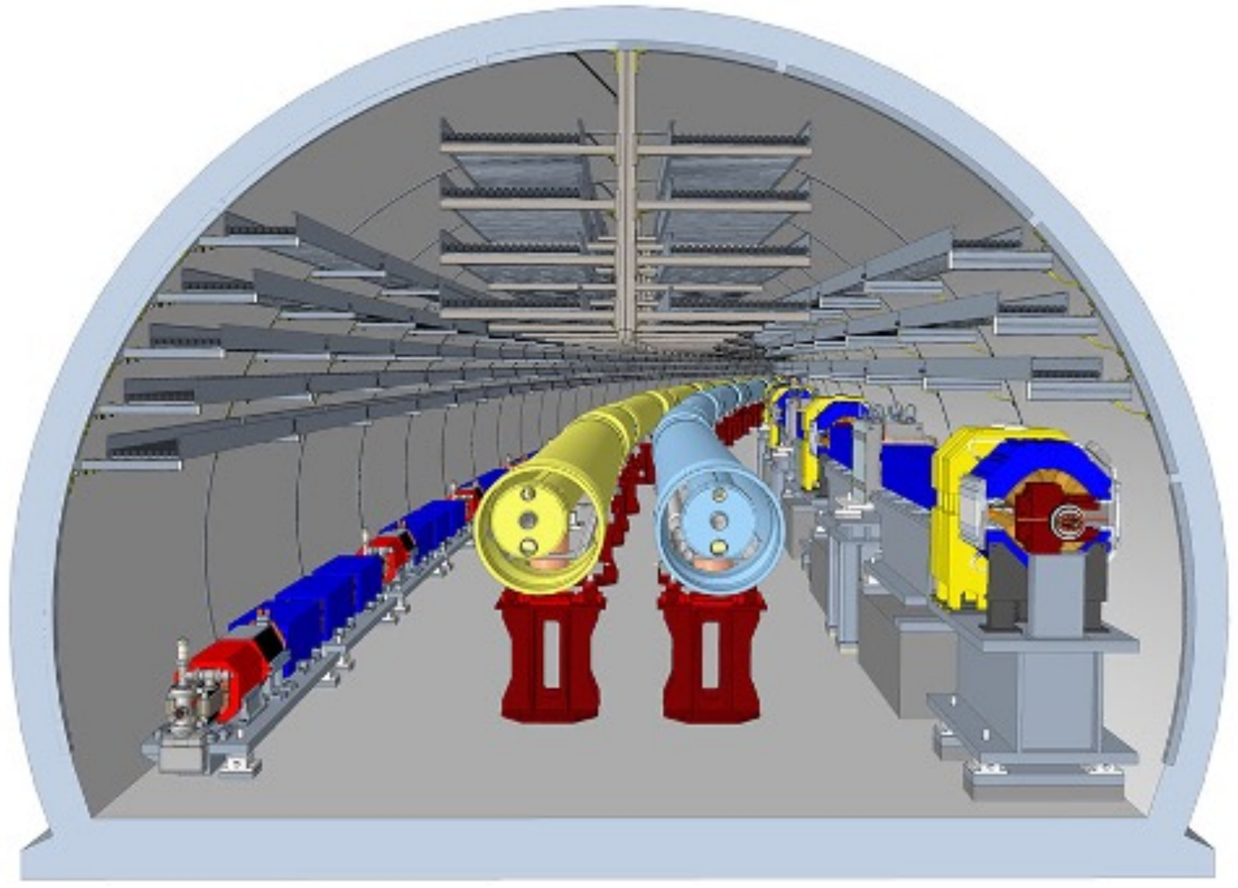
- Last year I spoke about uncertainty quantification (UQ) for accelerator control
    - Estimating digital twin parameters with error bars
    - Chris Kelly’s talk summarizes the outcome of this work
- Looking forward, I would like to discuss in more detail:
    - Computational methods for UQ
    - Stochastic control theory
    - Optimal experimental design
    - Differentiable programming

# Uncertainty in accelerator control

- <u>Objective:</u> Steer the beam (or control other beam properties)
- <u>Problem:</u> Imperfect knowledge of the relationship between system inputs (currents) and outputs (beam position)
    - Magnet misalignments
    - Transfer function between current and magnetization
    - Current set points not identical to realized currents in system
- Imperfect modeling can lead to *incorrect* control policy, but we never have *perfect* knowledge







# Review of uncertainty quantification

- We consider *probabilistic parameter estimation*
  - e.g., estimate Bmad parameters from beam position / ORM data, with uncertainties
- Parameter estimation is often done to improve predictions
  - Although sometimes we care about the parameter values themselves (e.g. theory fits)
  - Here we will focus on constraining/improving digital twin predictions for control
- This is often formulated as a (nonlinear) *regression* problem:
  - *Observations = Model(parameters) + Error*
  - $y_i = m_i(c; \theta) + \epsilon$
- Example:
  - $y_i$ is a BPM measurement, $m_i(c; \theta)$ is Bmad's prediction (for known control currents $c$ and unknown parameters $\theta$), and $\epsilon \sim N(0, \sigma^2)$ is a random measurement error variable

# Bayesian parameter estimation

- In *point estimation* such as least squares fitting, goal is to find single best parameter vector
  - $\hat{\theta} = \text{arg min}_\theta \sum_i (y_i - m_i(c;\theta))^2$
- *Bayesian inference* seeks a probability distribution of parameters, conditional on the data:
  - $p(\theta\,|\,y)$
- Bayes's theorem gives this *posterior distribution* in terms of a *likelihood* and *prior:*
  - $p(\theta\,|\,y) \propto p(y\,|\,\theta)\,p(\theta)$
- For the regression probability model and *iid* normal errors this becomes:

$$p(\theta\,|\,y) \propto p(y\,|\,\theta)\,p(\theta) = \frac{1}{\left(\prod_i \sqrt{2\pi\sigma_i^2}\right)} \exp\left[-\frac{1}{2}\frac{\sum_{i=1}^{N}(y_i - m_i(c;\theta))^2}{\sigma_i^2}\right] \times \prod_{k=1}^{K} p(\theta_k)$$

# Probabilistic programming

- The equations get complicated and messy (and will be mores for more complex models)
- Can we implement this in a more “declarative” style closer to the model we’re using:
  - $y_i = m_i(c; \theta) + \epsilon$
- *Probabilistic programming* defines a statistical model and sample it with Monte Carlo:
  - We use Turing.jl in Julia; Python has PyMC, PyStan, Pyomo, …
- Model definition:

```
@model function bmad_regression_model(Δbpm, c₊, c₋)
    θ ~ product_distribution(LogNormal.(logμ_θ, σ_θ))
    Δbpm ~ product_distribution(Normal.(bmad_pos(c₊,θ) - bmad_pos(c₋,θ), σ))
end
```

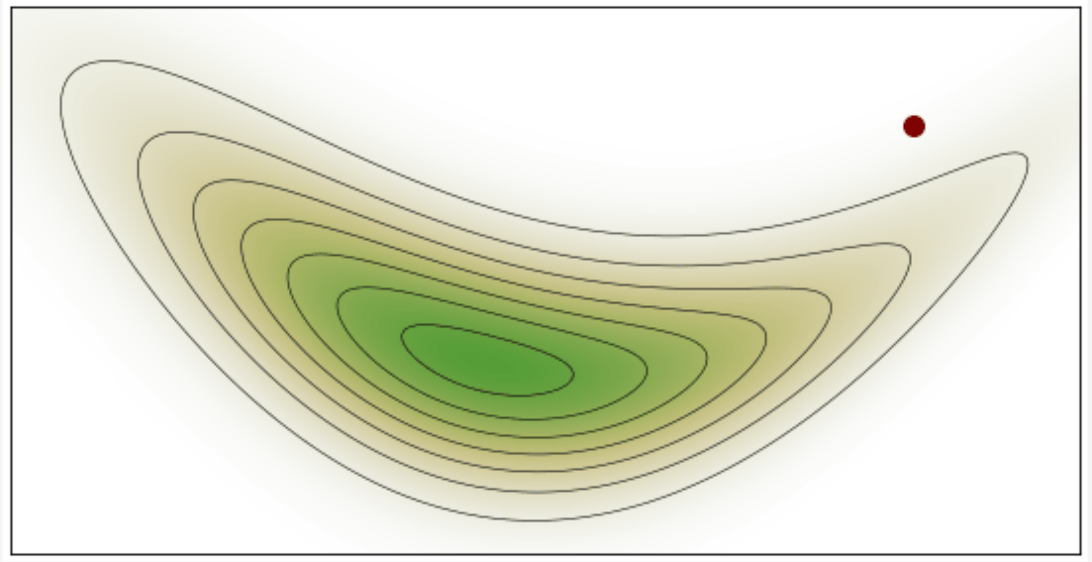
- Sampling:

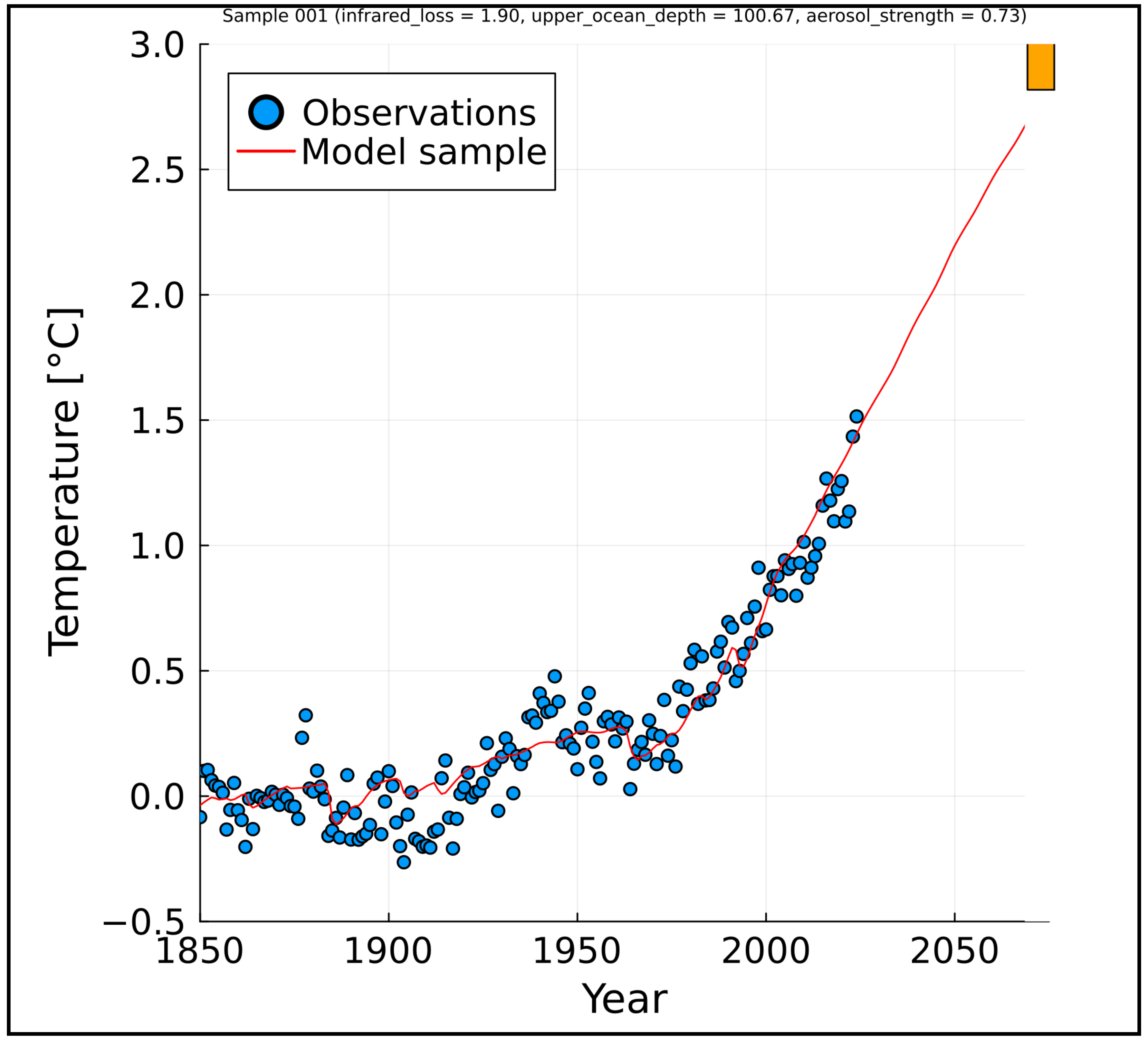
```
chain = sample(bmad_regression_model(Δbpm,c₊,c₋), NUTS(), num_iterations)
```

# Hybrid (or Hamiltonian) Monte Carlo sampling

- Hybrid Monte Carlo: idea from lattice field theory
- Let $U(\theta) = -\log \pi(\theta \mid y)$ be “potential energy”
- “Kinetic energy” $K(p) = \frac{1}{2} p^T M^{-1} p$ (fictitious momentum)
- Propose samples by integrating Hamiltonian dynamics of particle in potential (configuration space = parameter space), $\dot{\theta} = \partial \mathscr{H} / \partial p, \dot{p} = -\partial \mathscr{H} / \partial \theta$
- Metropolis test to accept/reject trajectories
- Requires a *differentiable model* (or surrogate)

```julia
@model function EBM(obs)
    p ~ arraydist([LogNormal(log(1.2),log(2)/2),LogNormal(log(100),log(2)/2),
                   LogNormal(log(1),log(2)/2)])
    σ ~ LogNormal(log(0.1),log(2)/2)
    m = predict(p)
    obs ~ MvNormal(m[obs_times], σ*I)
    return m
end
chain = sample(EBM(temp_obs), NUTS(0.65), 250)
```





# Imperfect models: From parameter to function estimation

- We assumed all uncertainties are in *parameters*, such as transfer function coefficients
- What if the uncertainties about the form of the functions themselves?
- Examples:
  - Transfer function shape beyond low-order polynomial
  - Unknown functional dependence of parameters on other variables (such as the environment, hysteresis history, …)
  - Overall form of a lattice element’s Lie map
- Can we learn “missing physics” in the digital twin as ML function approximations?
  - May need to preserve “structure" (monotonicity, convexity, symplecticity, …)
  - Learn operators acting on *distributions* of particles (avoid Monte Carlo simulation)?
  - High dimensional inverse problem: computationally challenging and may need more data
- **This amounts to adding data-driven ML corrections to the digital twin** (“hybrid model”)

# Stochastic optimization for control inputs

- **Control** $c$: inputs that the operator can specify
- **Parameters** $\theta$: unknown system characteristics (random variable from distribution $\pi(\theta)$)
- **Model** $m(c;\theta)$: the modeled system response to inputs (e.g., beam position)
- **Objective**: a metric of system performance (e.g., a loss function) to optimize
  - $\mathscr{L}(m(c;\theta\,|\,y)) = \sum_i (\bar{z}_i - m_i(c;\theta))^2$ (deviation of beam position from target at BPMs)
- Stochastic control is *robust to uncertainties* in quantities we can't estimate perfectly
- Find control that optimizes *expected* objective (average over Monte Carlo samples $\{\theta_j\}$):

$$c^\star = \arg\min_c \mathbb{E}_{\theta|y}[\mathscr{L}(m(c;\theta))]$$
$$\approx \frac{1}{J}\sum_{j=1}^{J}\sum_{i=1}^{N}(\bar{z}_i - m_i(c;\theta_j))^2$$

# Risk-averse control

- Expected loss minimization: find control that minimize *expected* loss

$$c^\star = \text{arg min}_c\, \mathbb{E}_{\theta|y}[\mathscr{L}(m(c;\theta))]$$

- This finds the control policy that does best on average
- But some rare scenarios could be very bad; we want to be robust to "long-tailed risk"
- **Conditional value-at-risk (CVaR)**: idea from financial risk management
  - Instead of the objective to minimize being "average loss" …
  - … minimize "average loss in the worst (1-α)% of outcomes" (CVaR)
  - e.g., if α=0.95 (95th percentile), select scenarios leading to the 5% worst losses, and minimize the average loss over just these "tail risk" scenarios
  - *Value-at-risk (VaR):* loss at α quantile, $VaR = \ell$ s.t. $Pr[\mathscr{L} \leq \ell] = \alpha$
  - *Conditional value-at-risk (CVaR):* average loss above the 95% percentile, $CVaR = \mathbb{E}_{\theta|y}[\mathscr{L} \,|\, \mathscr{L} > VaR]$

# Robust control

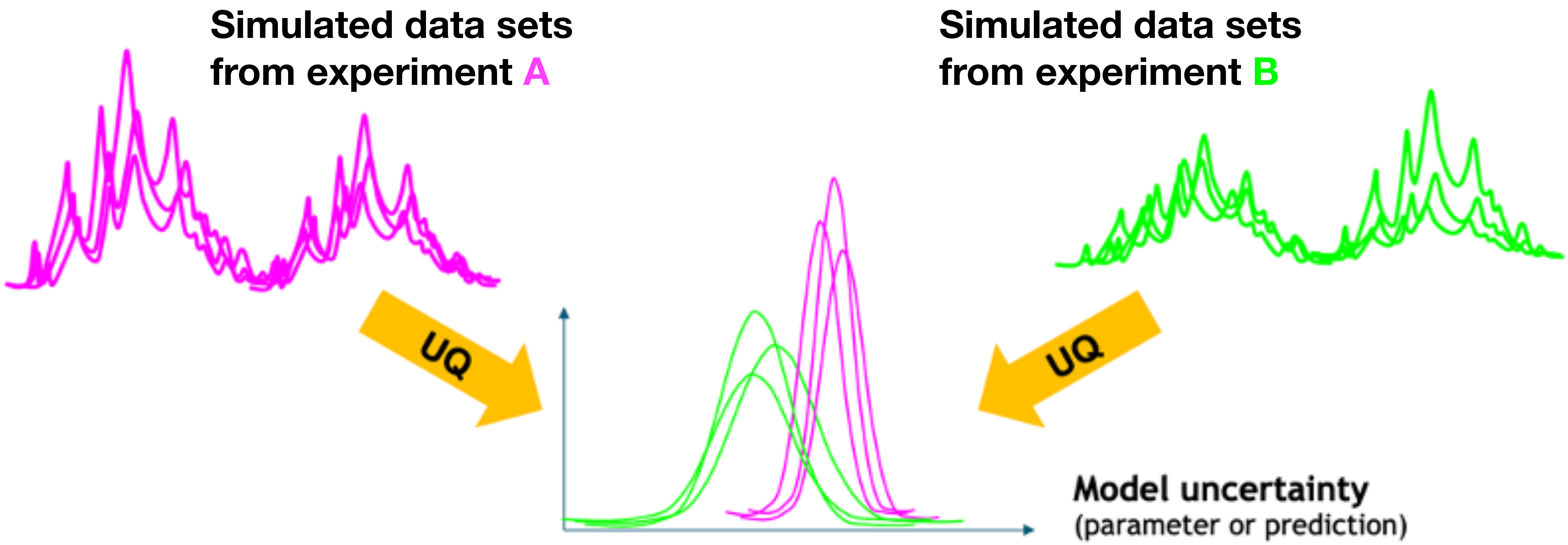
- Other topics in robust control theory
  - **Barrier certificates:** “Safety indicator” to monitor system approaching unsafe states
  - **Reachability:** Prove system can’t reach unsafe state from different controls
  - **Distributionally robust control:** Without making probability assumptions, find robust control policies over a worst-case “ambiguity set” of possible scenarios
- Harder under uncertainty, model misspecification, black-box or nonlinear models
  - State-of-art in control theory research
  - Might be able to get bounds/certification from truncated Taylor map expansion?
  - But all bets are off without any kind of bound on digital twin model error

# A workflow for accelerator control

- There are many approaches to control (BO, RL, model-based, model-free etc.)
- My own preferred approach:
  - **Write down regression model** with DT error term:
    - *Observations = Model(parameters) + <u>DT error</u> + Measurement error*
    - **Improve DT** (e.g., with <u>learnable internal correction terms</u>) so DT error becomes simpler
  - *(Optional: build surrogate of DT if DT is not fast enough)*
  - **Calibrate regression model**: learn parameters & DT error term
  - **Optimize control inputs using** DT via differentiable optimization (e.g., gradient descent)
    - May need state estimation along with DT parameter estimation
  - **Deploy** on real machine
- Streaming update loop (as data comes in: update regression, re-optimize with DT)
- "Greedy" algorithm; can use RL if sequences of decisions matter

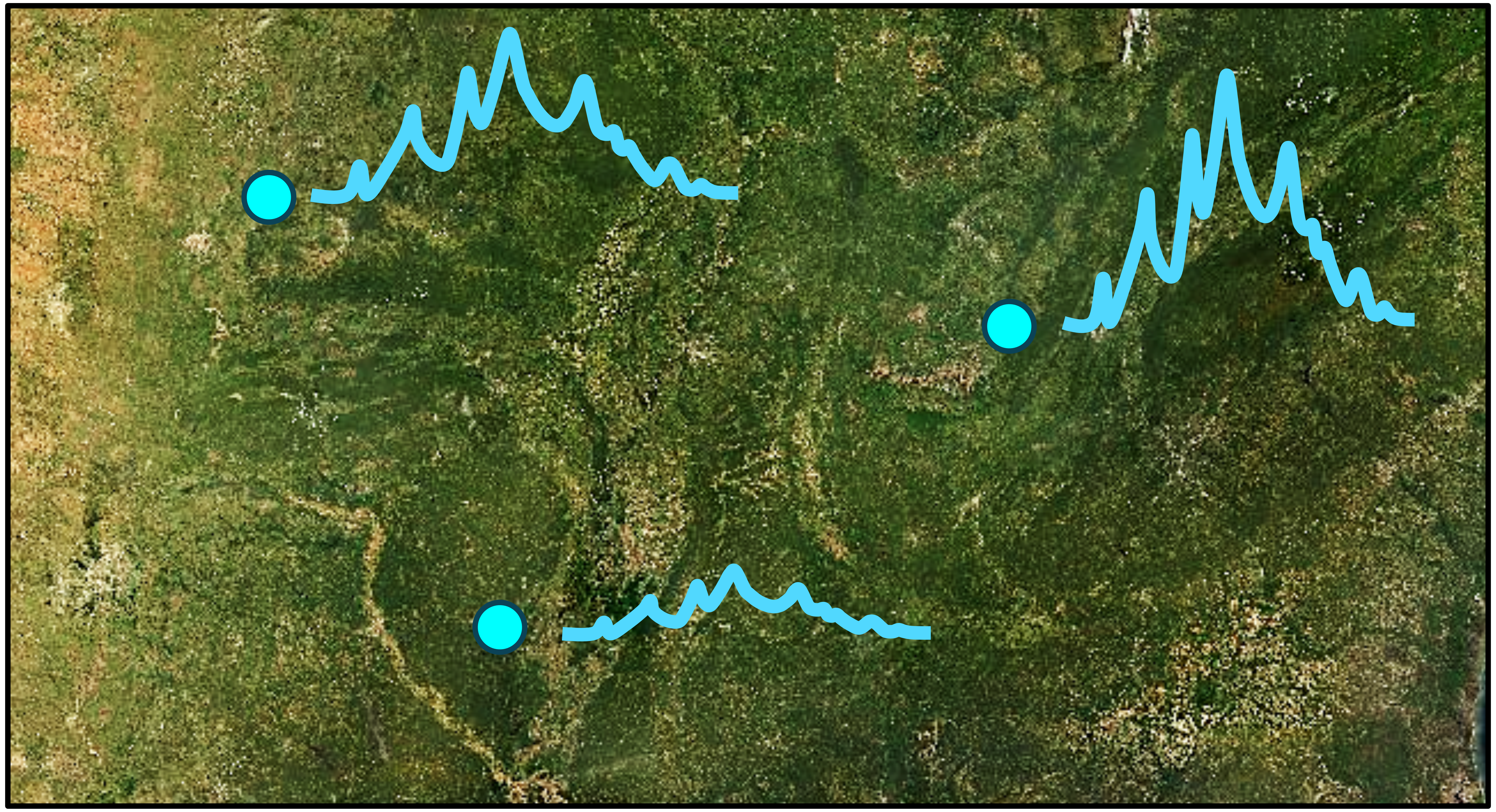
# Optimal experimental design (OED)

- Beyond UQ: what should we do to *reduce* uncertainty?
- Which machine-probing experiments help us control the beam better?
  - Limited downtime between science runs - which magnets to perturb & how much?
- Choose experiments whose data would reduce uncertainties the most?
  - Or rather, most reduce the objective to the stochastic optimal control problem

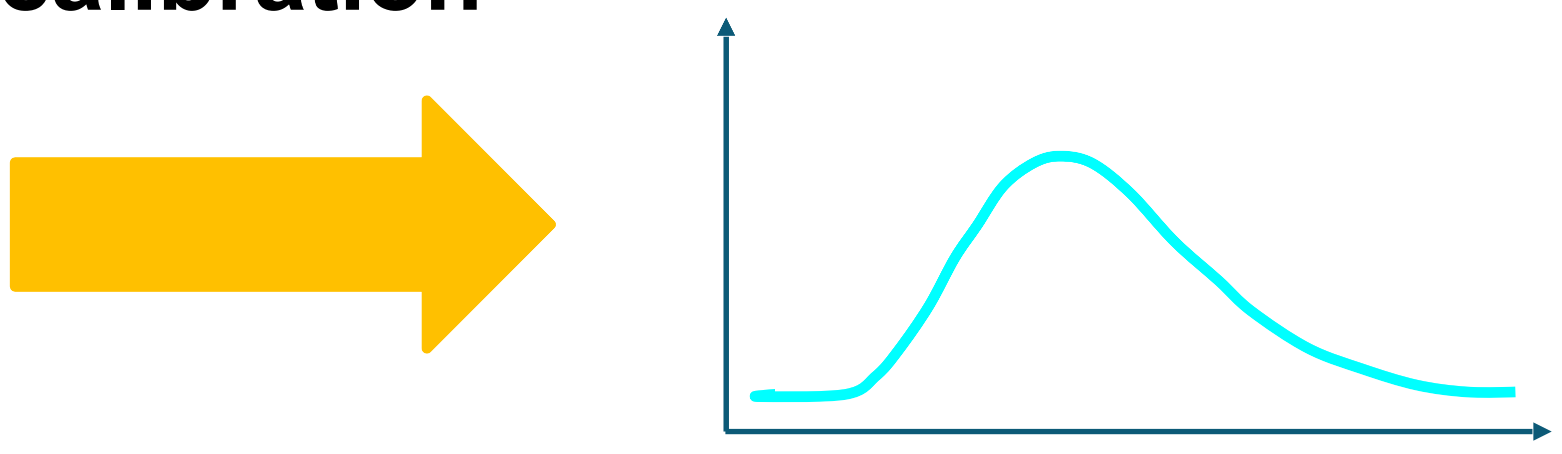


# OED for sensor placement example

- Our experimental design problem might be how to perturb magnet currents and measure beam positions to learn about model parameters (transfer functions, magnet misalignments, …)



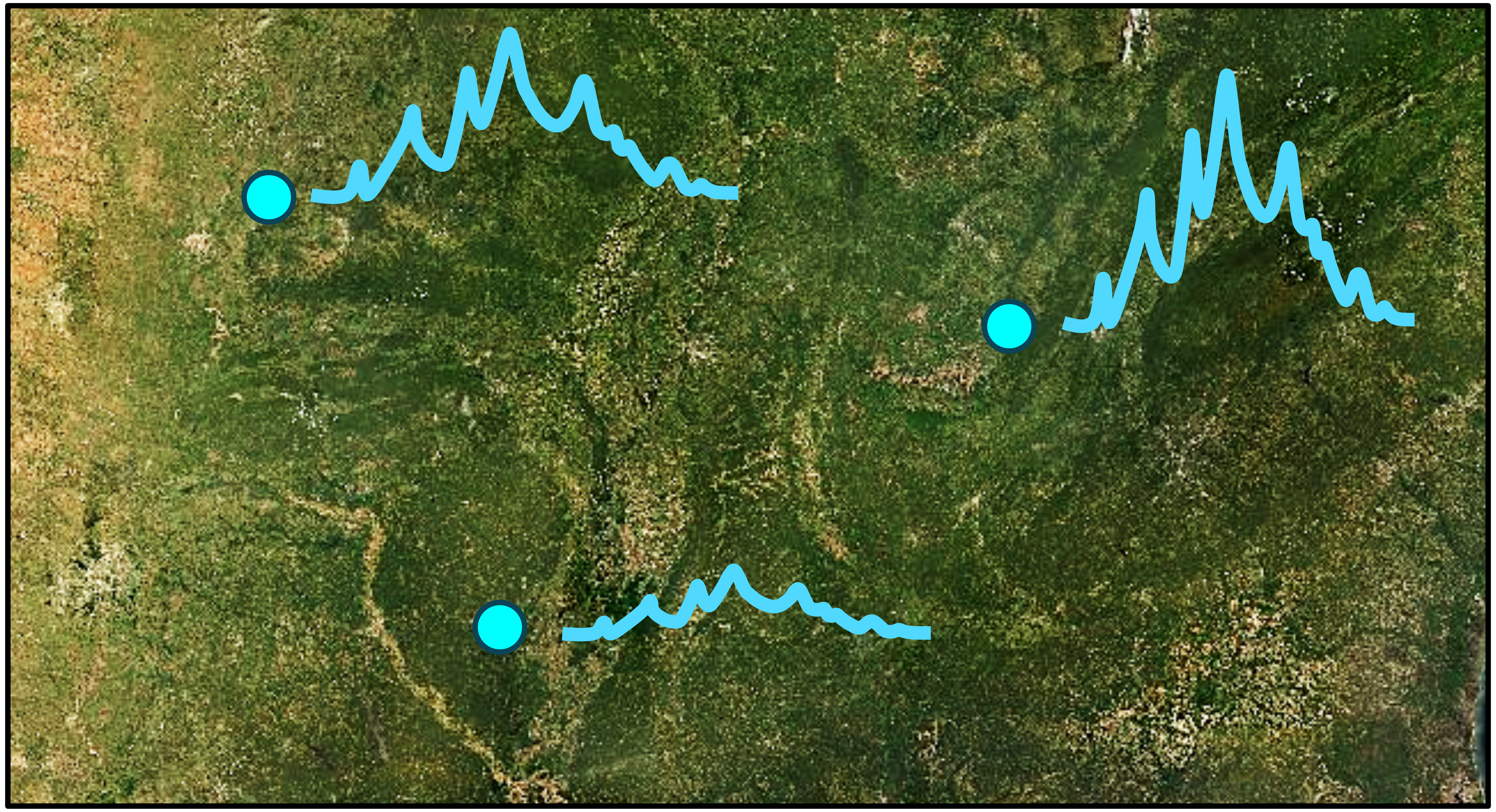
**Data-model calibration**



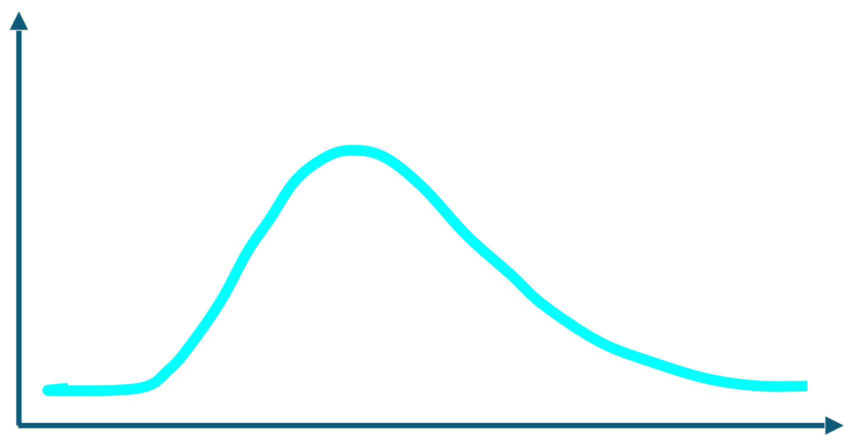
**Model uncertainty**
(parameter or prediction)

# OED for sensor placement example

- Here is an example of how this works for an environmental sensor placement problem
  - Sensor network: start with UQ (infer terrestrial vegetation model parameters)



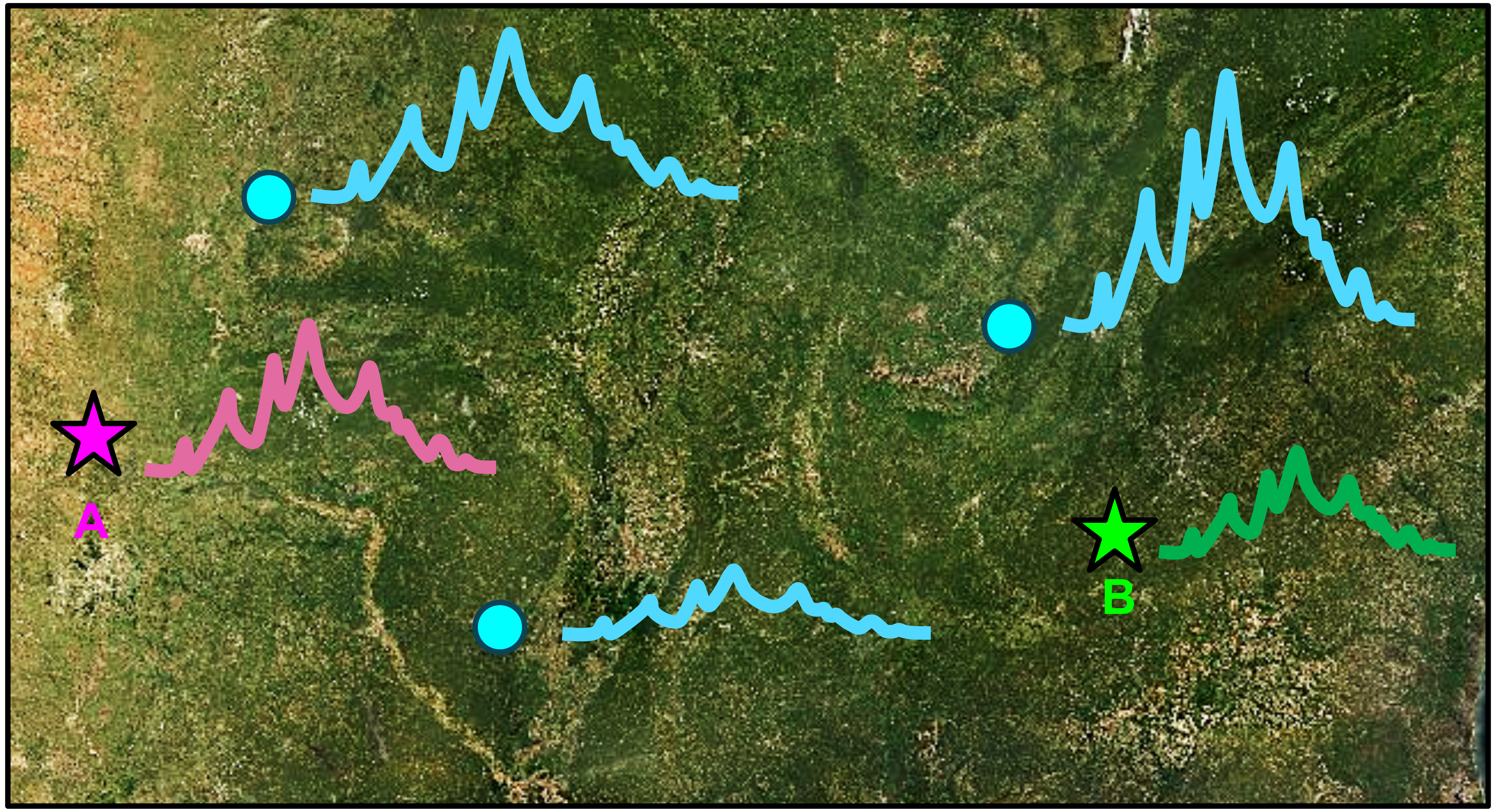
**Data-model calibration**



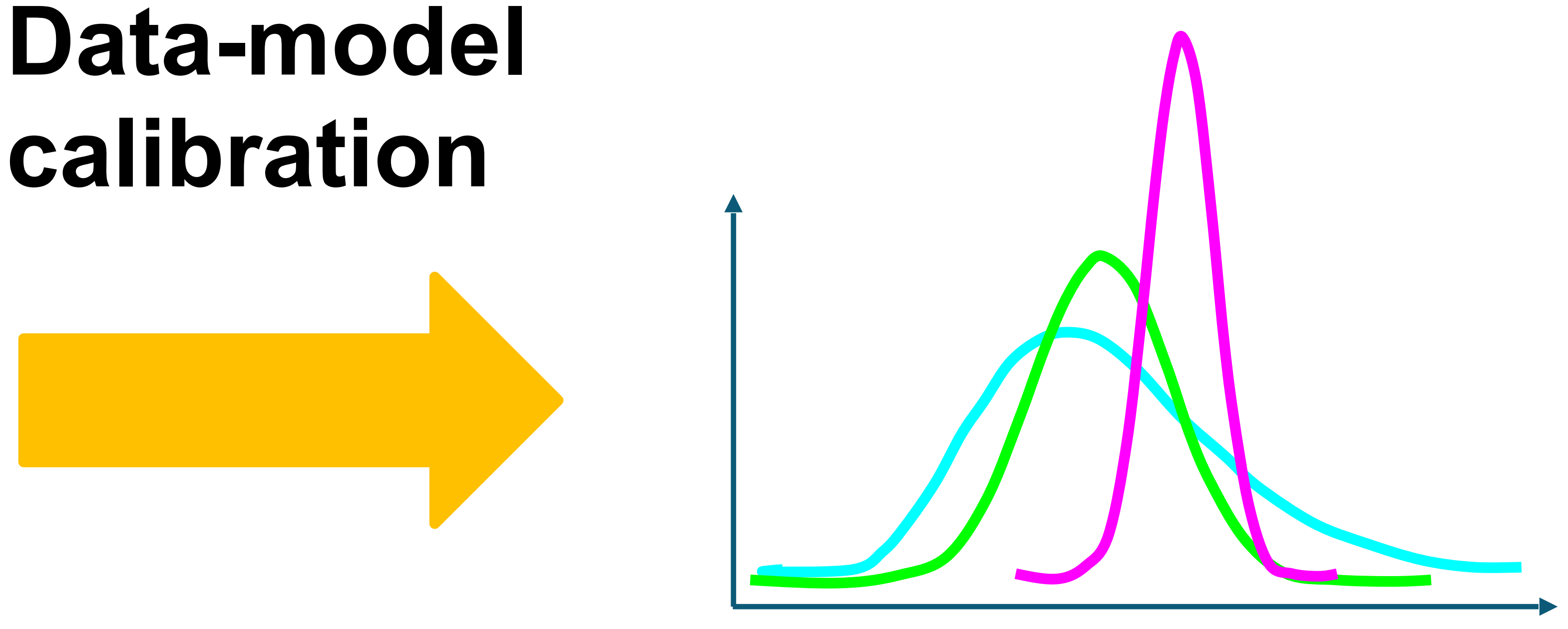
**Model uncertainty**
(parameter or prediction)

# OED for sensor placement example

- Of two proposed new site locations (A and B), which should we choose?
  - Select the location whose data, *if measured*, reduces model uncertainty the most



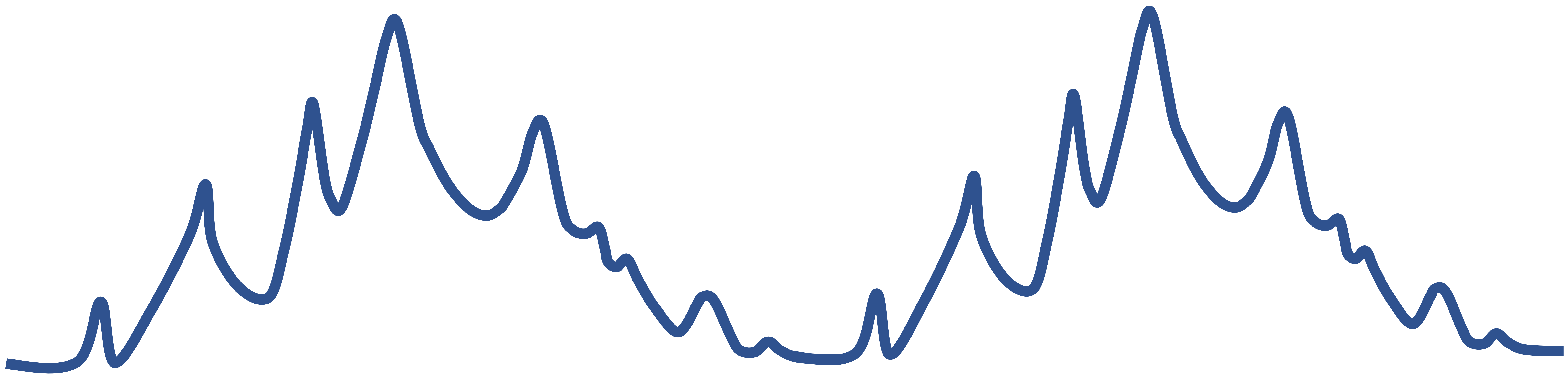

**Data-model calibration**



**Model uncertainty**
(parameter or prediction)

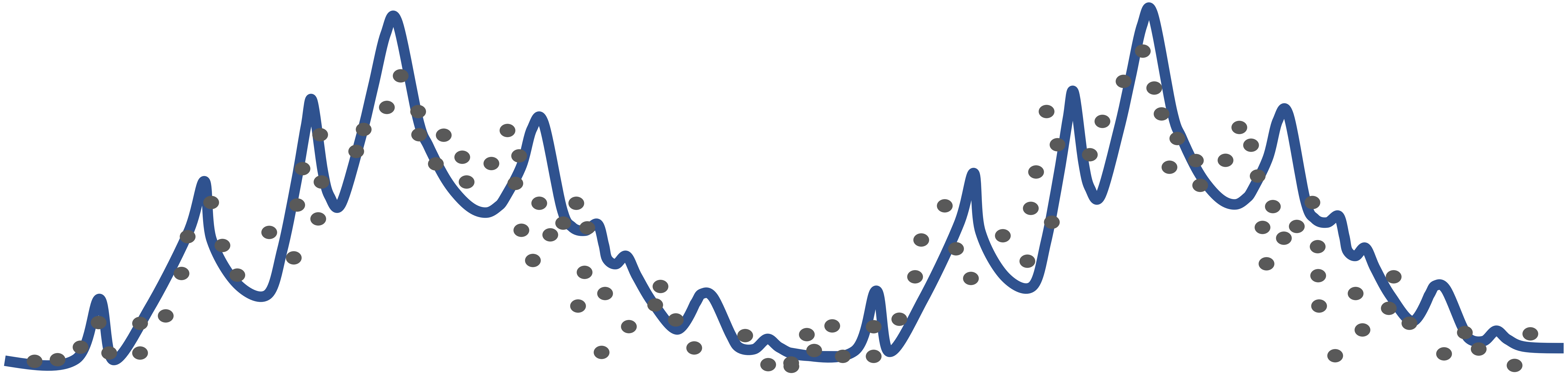
# Simulating experiments from digital twins

- Problem: we haven’t measured any data at these sites
  - Simulate the data
  - *Step 1: run the model*



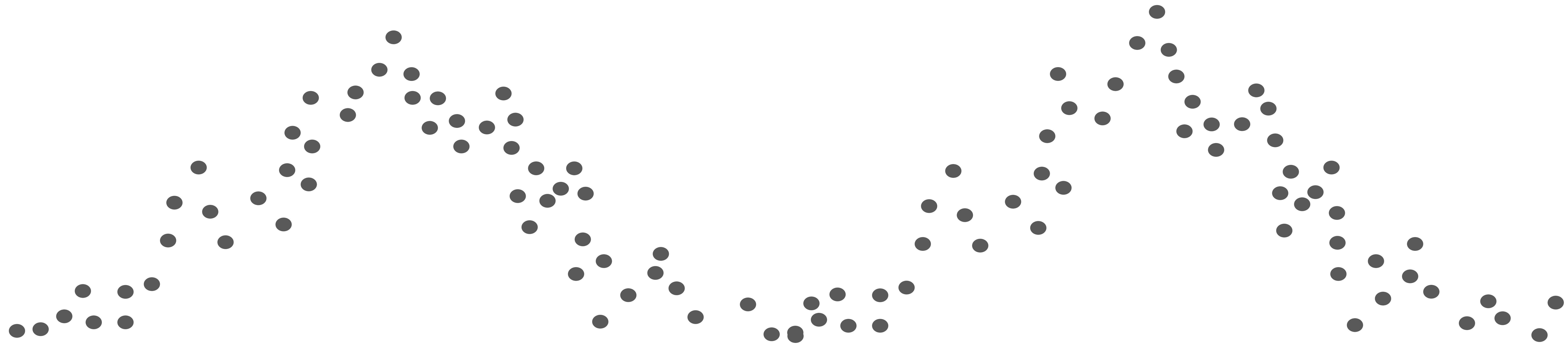
# Simulating experiments from digital twins

- Problem: we haven’t measured any data at these sites
  - Simulate the data
  - Step 1: run the model
  - *Step 2: simulate observations (add unmodeled variability, model bias, instrument error)*



# Simulating experiments from digital twins

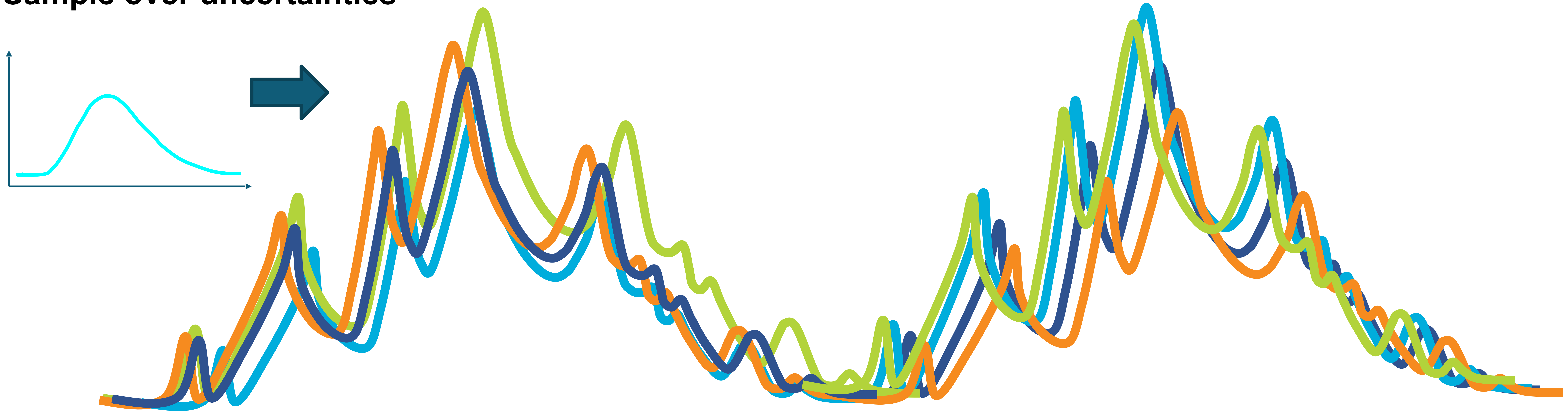
- Problem: we haven't measured any data at these sites
  - Simulate the data
  - Step 1: run the model
  - Step 2: simulate observations (add unmodeled variability, model bias, instrument error)



# Simulating experiments from digital twins

- Another problem: we are *uncertain* what data will be measured at a site
  - (we don't know the model parameters, plus there is random measurement error)
  - ⇒ Simulate *ensembles* of data for each site (sampling parameter & measurement uncertainty)

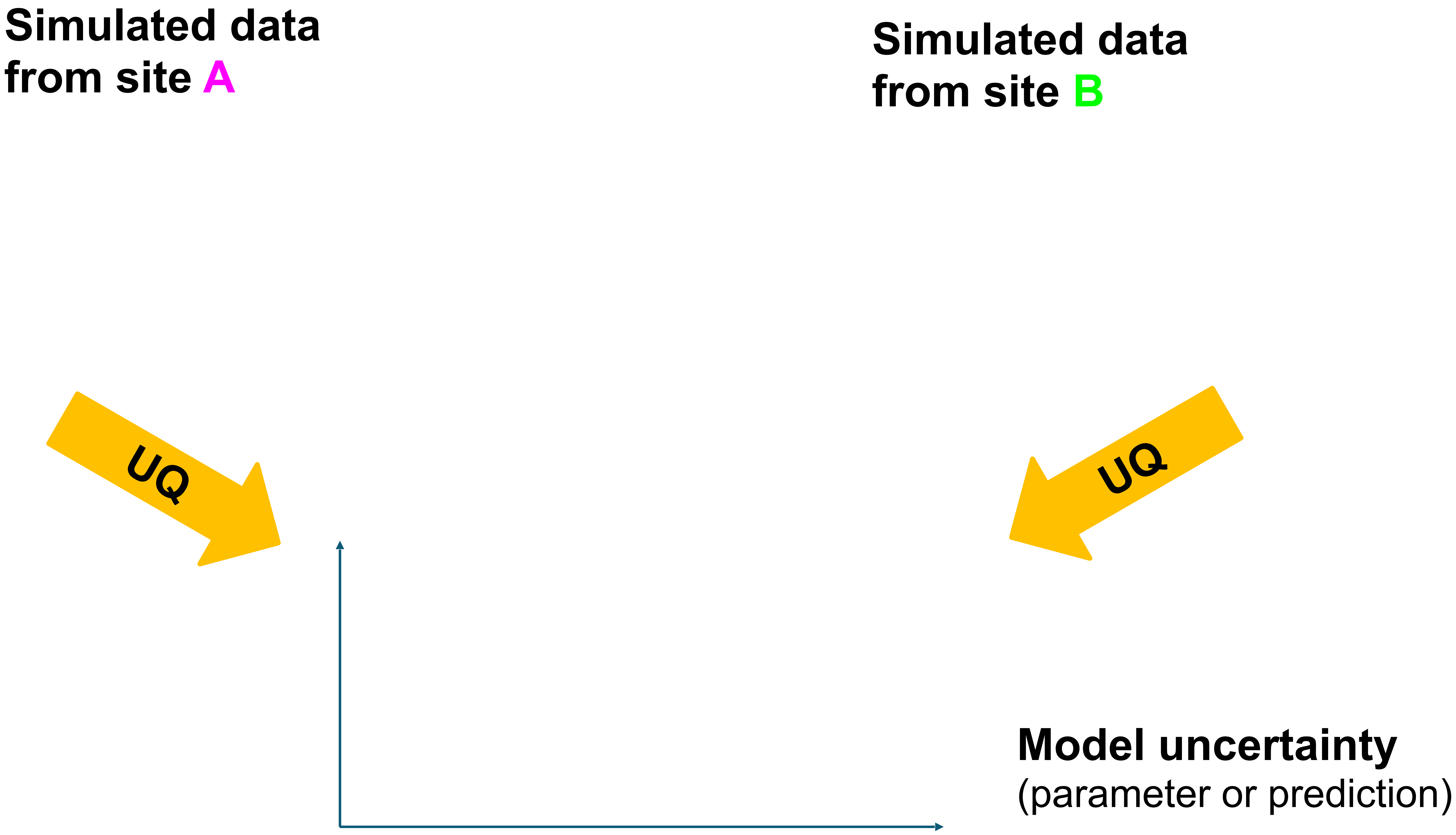
**Sample over uncertainties**



** For visual clarity, this only shows step 1 (model uncertainty), not step 2 (observation uncertainty)*

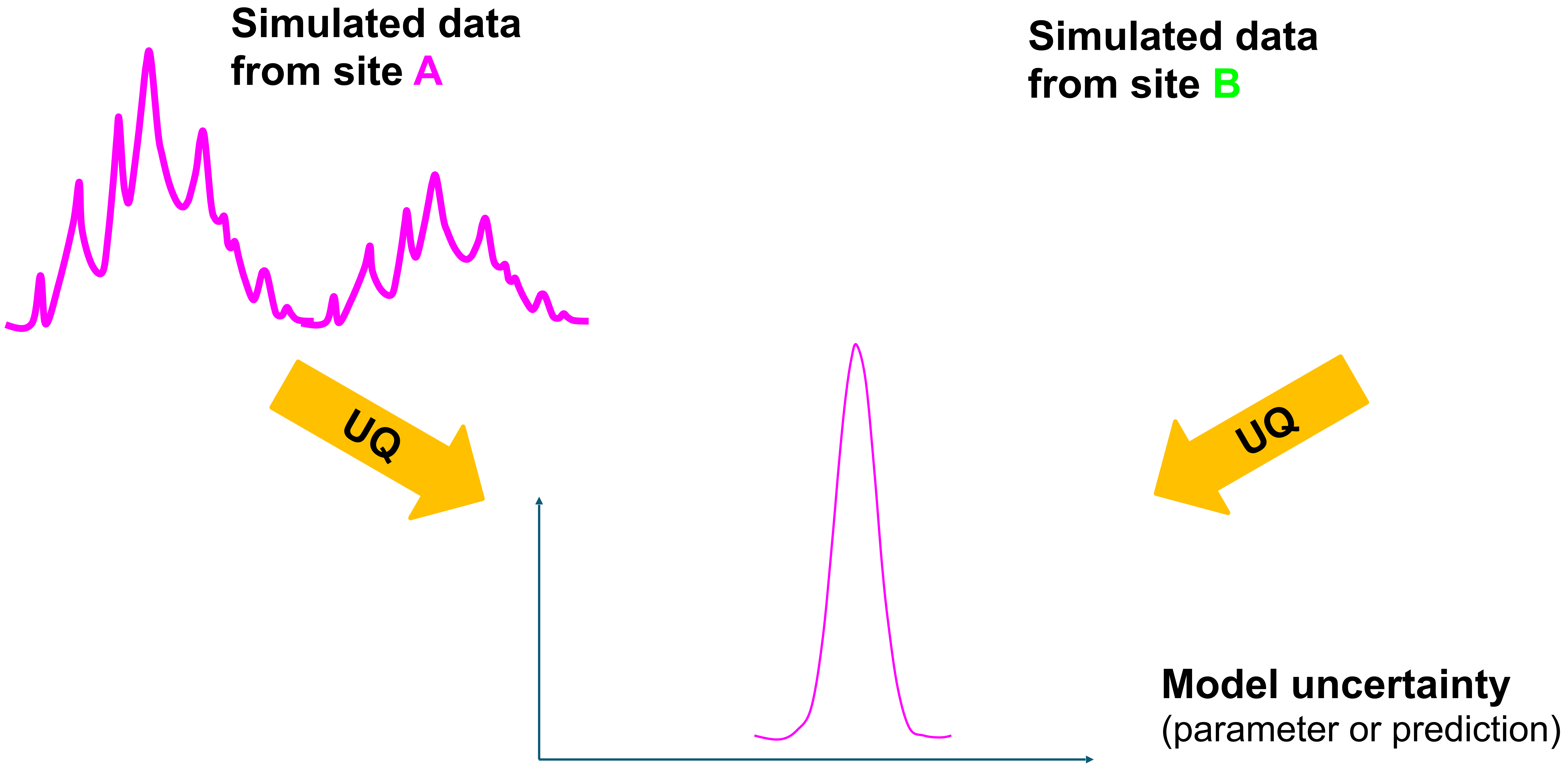
# Simulating experiments from digital twins

- Each simulated data set gives a simulated reduction in uncertainty
  - We pick the site that has the greatest *average* uncertainty reduction



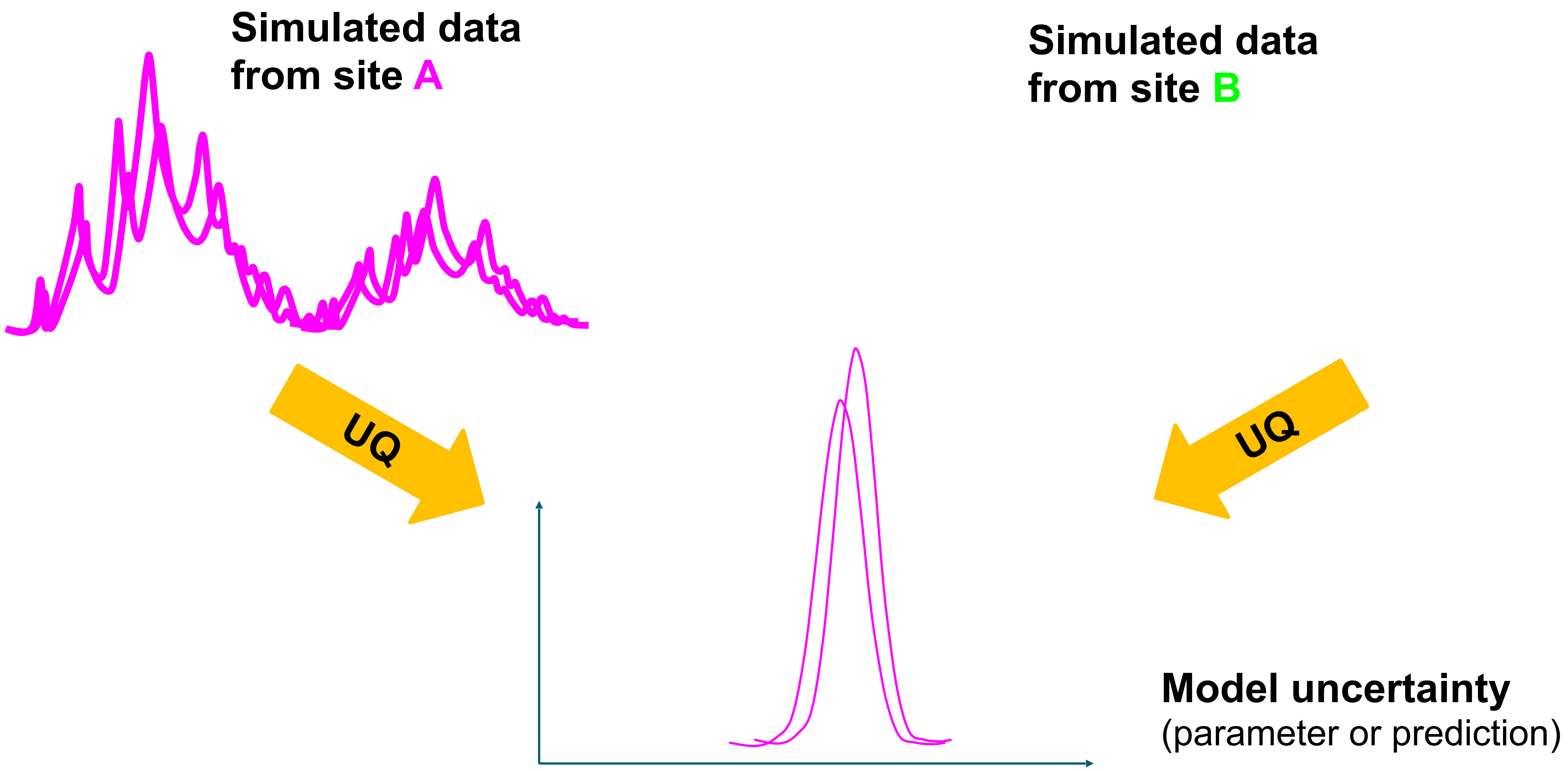
# Simulating experiments from digital twins

- Each simulated data set gives a simulated reduction in uncertainty
  - We pick the site that has the greatest *average* uncertainty reduction



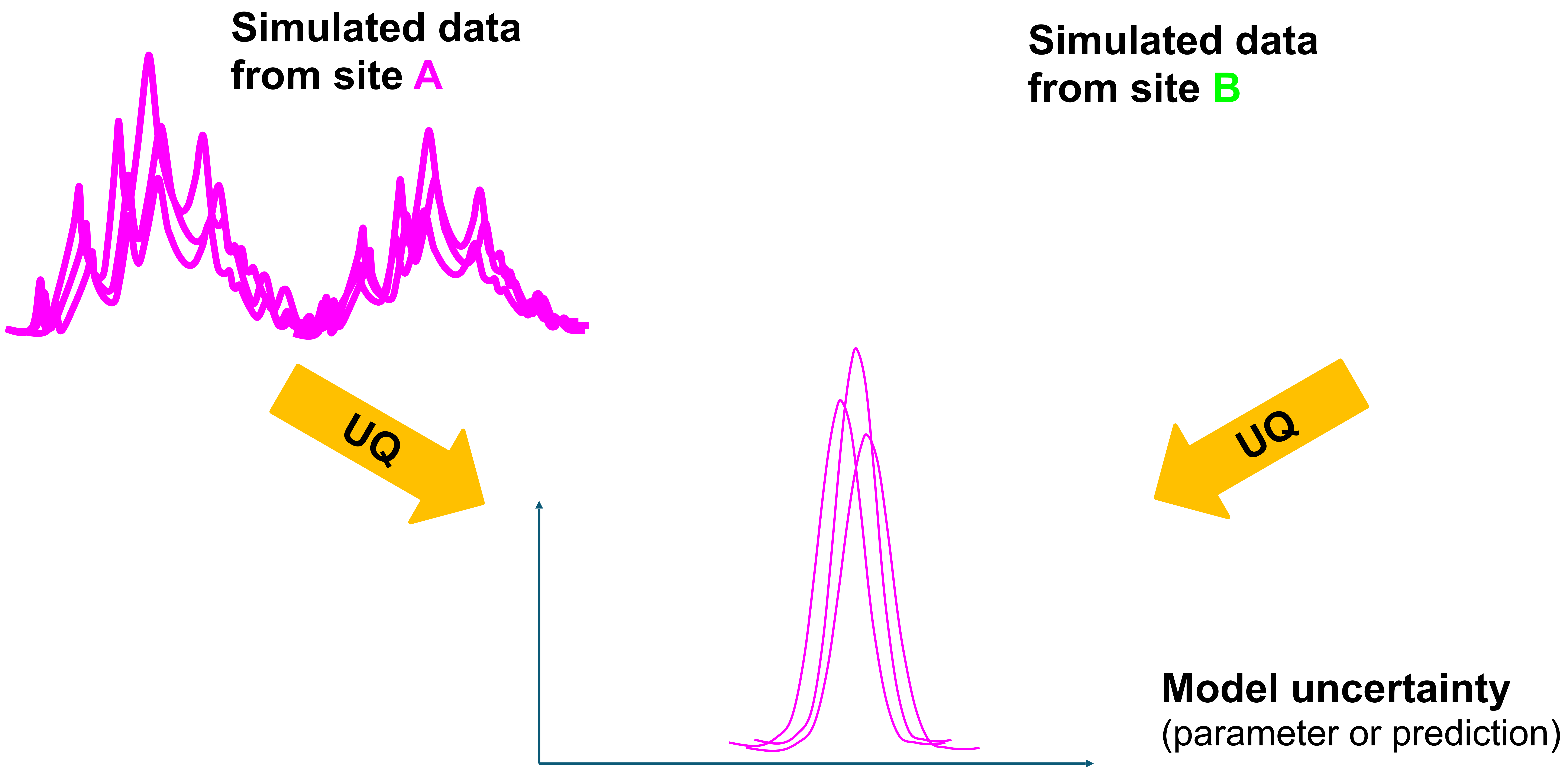
# Simulating experiments from digital twins

- Each simulated data set gives a simulated reduction in uncertainty
  - We pick the site that has the greatest *average* uncertainty reduction



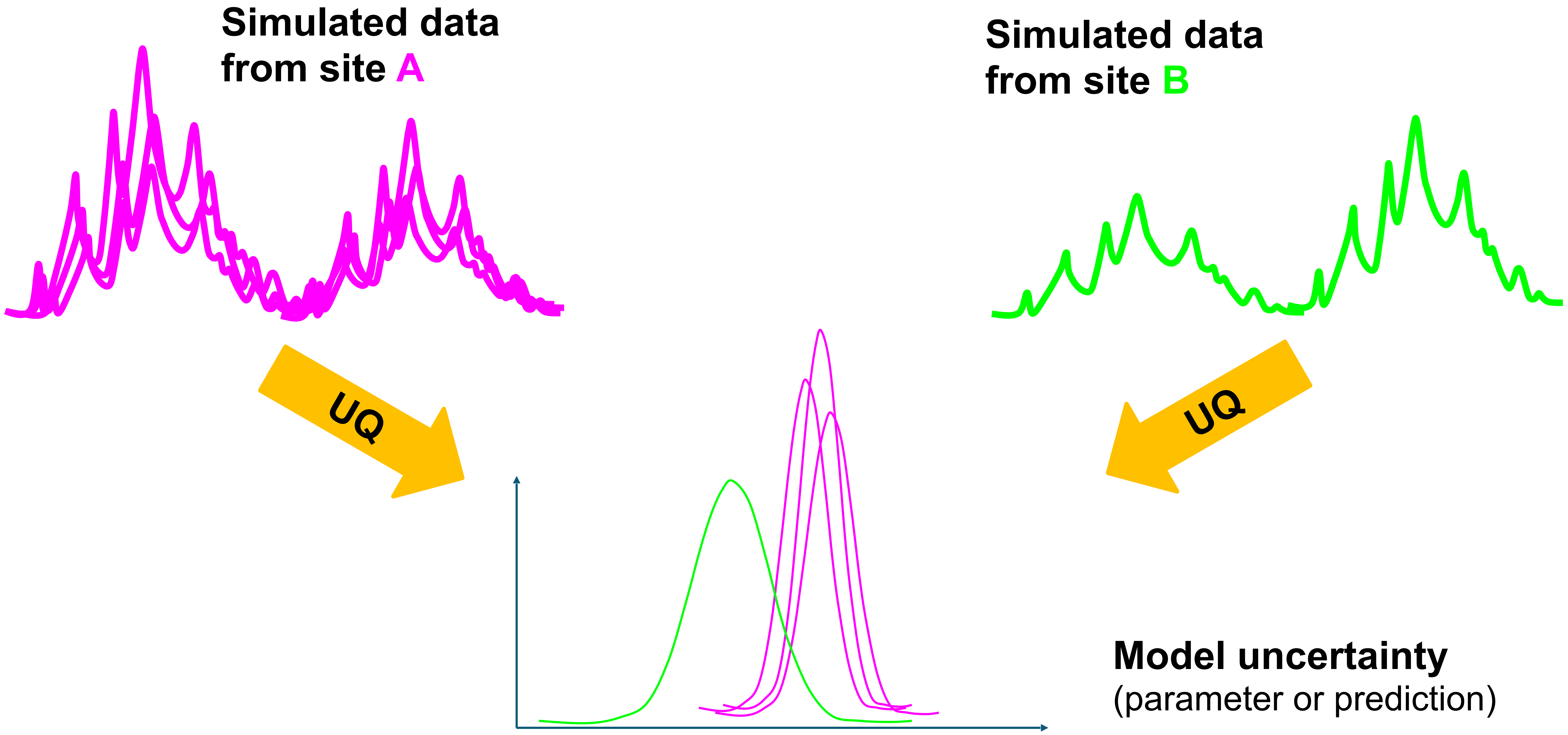
# Simulating experiments from digital twins

- Each simulated data set gives a simulated reduction in uncertainty
  - We pick the site that has the greatest *average* uncertainty reduction



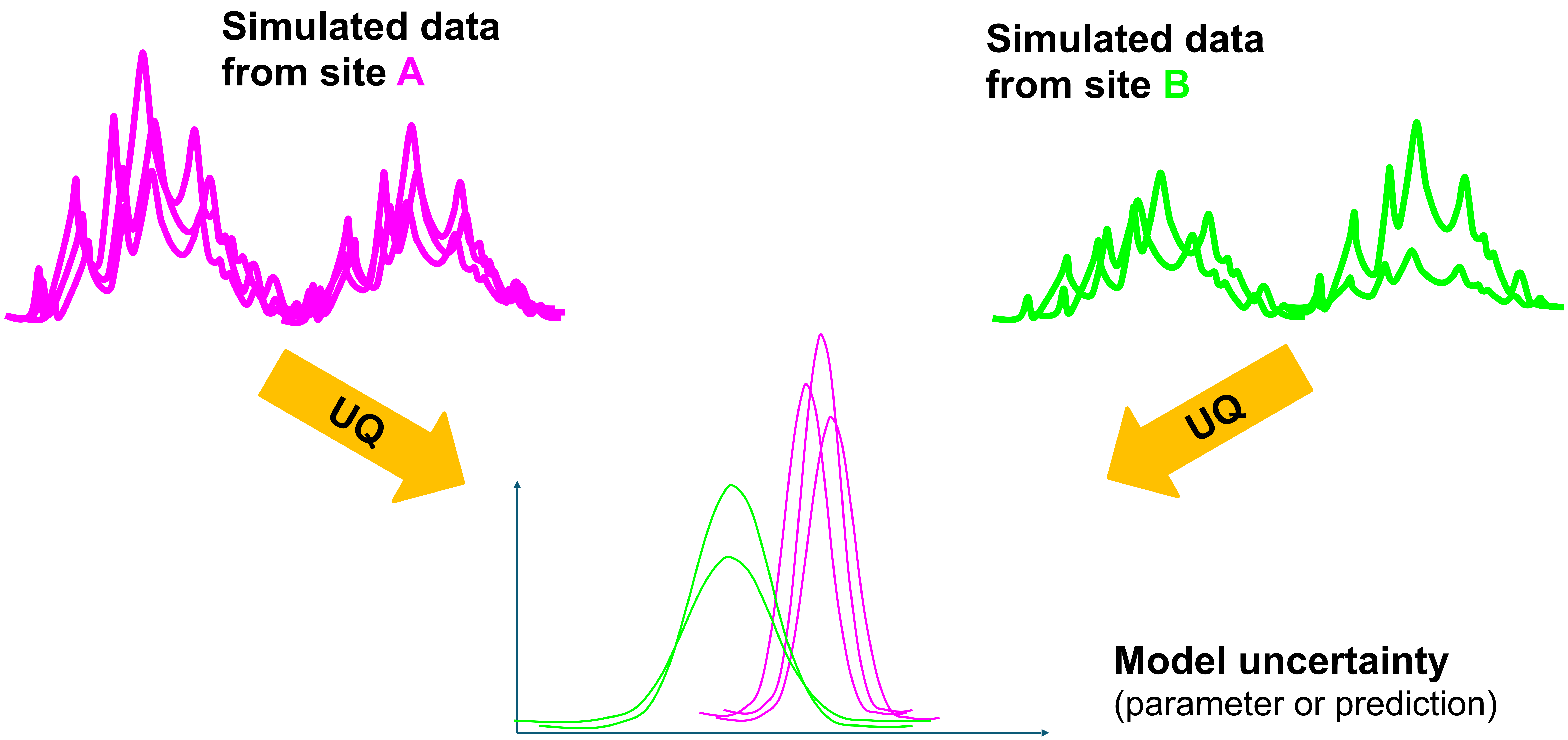
# Simulating experiments from digital twins

- Each simulated data set gives a simulated reduction in uncertainty
  - We pick the site that has the greatest *average* uncertainty reduction



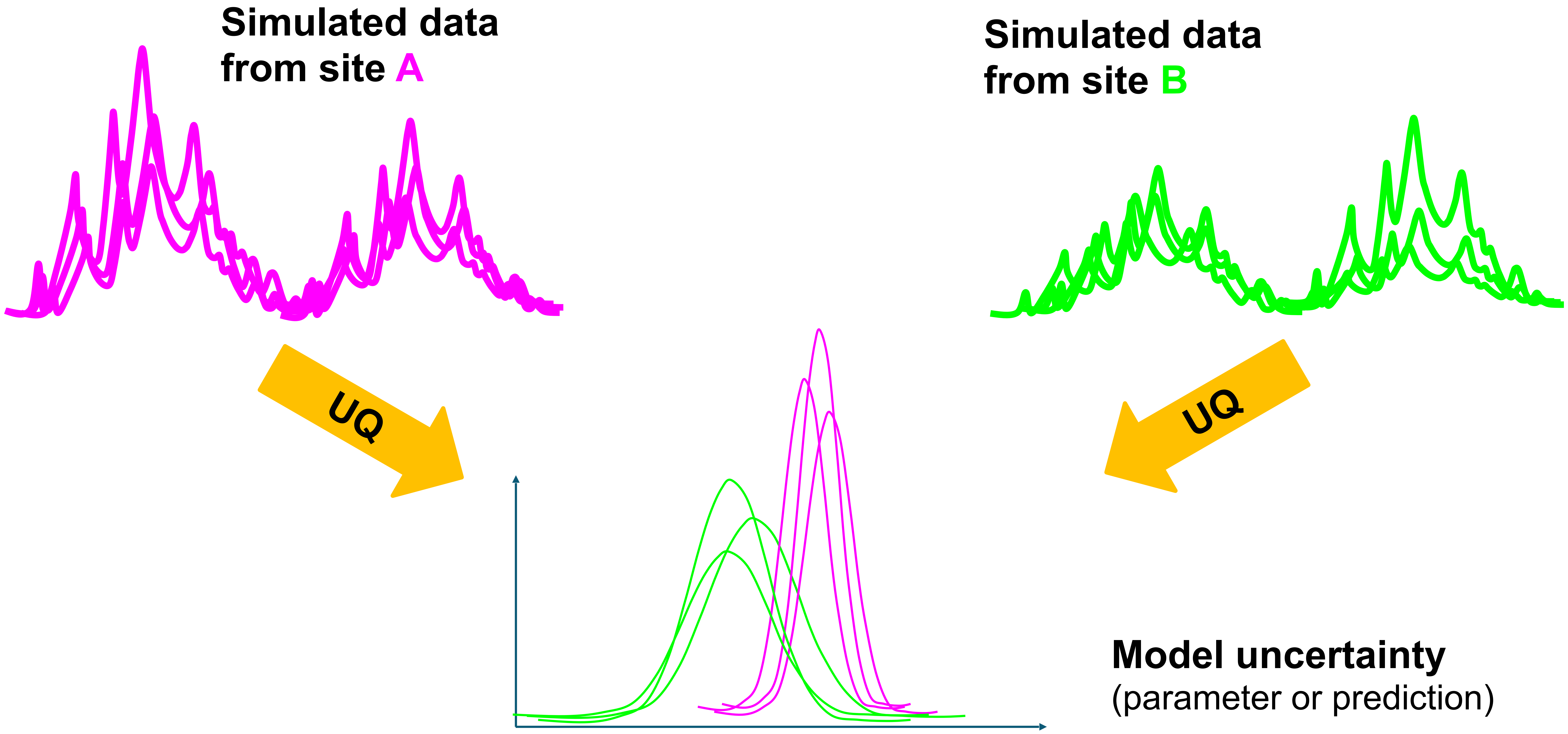
# Simulating experiments from digital twins

- Each simulated data set gives a simulated reduction in uncertainty
  - We pick the site that has the greatest *average* uncertainty reduction



# Simulating experiments from digital twins

- Each simulated data set gives a simulated reduction in uncertainty
  - We pick the site that has the greatest *average* uncertainty reduction



# Optimal experimental design: Mathematics

- Uncertainty about parameter distribution $p(\theta)$ given by *entropy* $H[\theta] = \mathbb{E}_\theta[\log p(\theta)]$
- What experiment $d$ would most reduce the entropy (maximize *information gain*)
  - Possible experimental outcomes are random, with probability distribution $p(y\,|\,\theta, d)$
  - Observing an outcome $y$ gives a new distribution $p(\theta\,|\,y)$ with entropy $H[\theta\,|\,y]$.
  - We want to maximize information gain (entropy reduction) $H[\theta] - H[\theta\,|\,y]$
- The problem is, we don't know which outcome $y$ we will measure
- Choose $d$ to maximize *expected* information gain (EIG), averaged over possible outcomes
  - $EIG = \mathbb{E}_{y|\theta,d}\Big[H[\theta] - H[\theta\,|\,y]\Big]$
- Other formulations maximize *predictive* (instead of parameter) information gain
- *Decision-theoretic* OED optimizes the solution to an inner control problem

# The importance of derivatives

- We have a model (digital twin) that makes predictions of outputs: $m_i(c; \theta)$
- Many modern algorithms require or benefit from access to *derivatives* of outputs
  - Gradients ($\nabla m$ such as $\nabla_\theta m(c; \theta)$ or $\nabla_c m(c; \theta)$), Hessians ($\nabla \nabla m$), etc.
  - Hybrid Monte Carlo, variational inference, gradient descent, function approximation, control, reinforcement learning, other sampling- and optimization- based algorithms
  - Other algorithms try to approximate those derivatives if you don't have them (e.g., black-box optimization methods)
- This is often the main way to scale to high-dimensional problems
- A star example is *backpropagation* in machine learning, which enables *training by stochastic gradient descent* (optimization of neural net parameters)
- This capability of SciBmad would be very useful for everything I've discussed

# Differentiable programming

- How do you get the gradient of a model output with respect to an input?
- We know how to symbolically differentiate analytic formulas
- We know how to numerically differentiate code functions (e.g. finite differences)
- Third way: *automatic differentiation* (AD) or *differentiable programming* (∂P)
    - Trace each function being executed, evaluate the derivatives of each primitive operation (e.g. `+`, `sin`, `exp`), and compose them via the chain rule
- AD applied to the loss function of a neural network is called *backpropagation*
    - Computes gradient of loss with respect to NN weights
    - Used in gradient descent step to minimize loss
    - PyTorch is “just” a giant AD library with efficient derivative code for many NNs

# Differentiable programming and adjoint models: Fitting digital twins to data with backpropagation

- Neural networks are trained by gradient descent + backprop
  - Calculate the network's error, and adjust its parameters to decrease the error
- We can fit a digital twin like Bmad the same way ("differentiable programming")
  - We just need automatic differentiation to calculate the gradient of our DT
  - The gradient of a loss w.r.t a forward model solution is the *adjoint model*
  - The adjoint of a ODE is another ODE integrated backward in time
    - Chain rule / backprop in continuous time, not discrete layers!
    - But we can also autodiff a discrete time-stepper code

**Adjoint model**

$$\frac{\partial \mathcal{L}}{\partial u(t)} \equiv \lambda(t) = \lambda_T - \int_T^t \left(\partial_u f\big(u(s);\theta\big)\right)^T \lambda(s)\, ds$$

$$\frac{\partial \mathcal{L}}{\partial \theta} = \int_0^T \frac{\partial \mathcal{L}}{\partial u(t)} \frac{\partial u(t)}{\partial \theta}\, dt = -\int_T^0 \lambda(s)^T \partial_\theta f\big(u(s);\theta\big)\, ds$$

**Forward model** $\dfrac{\mathrm{d}u}{\mathrm{d}t} = f(u(t);\, \theta)$

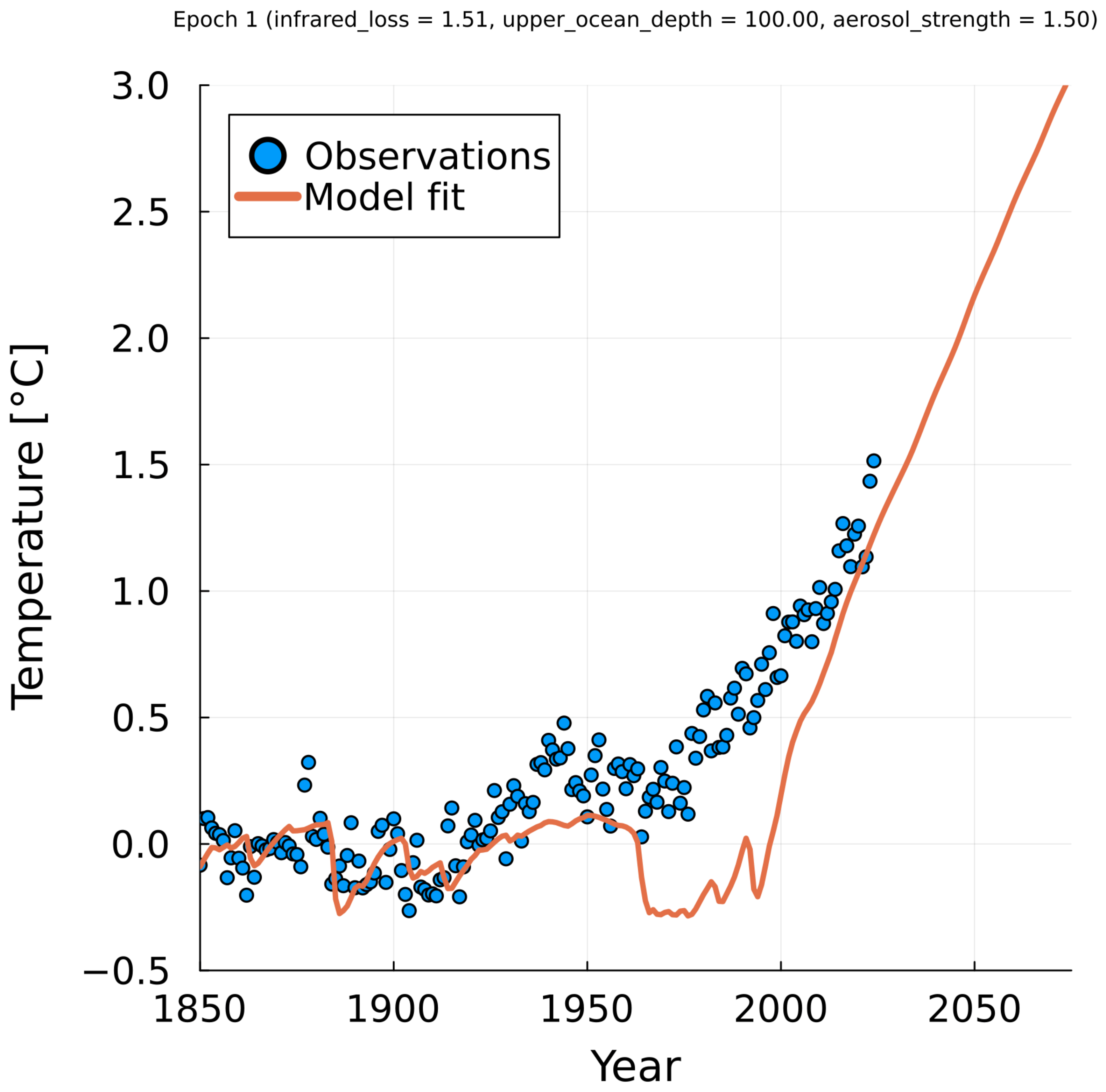
# Differentiable programming and adjoint models: Fitting digital twins to data with backpropagation

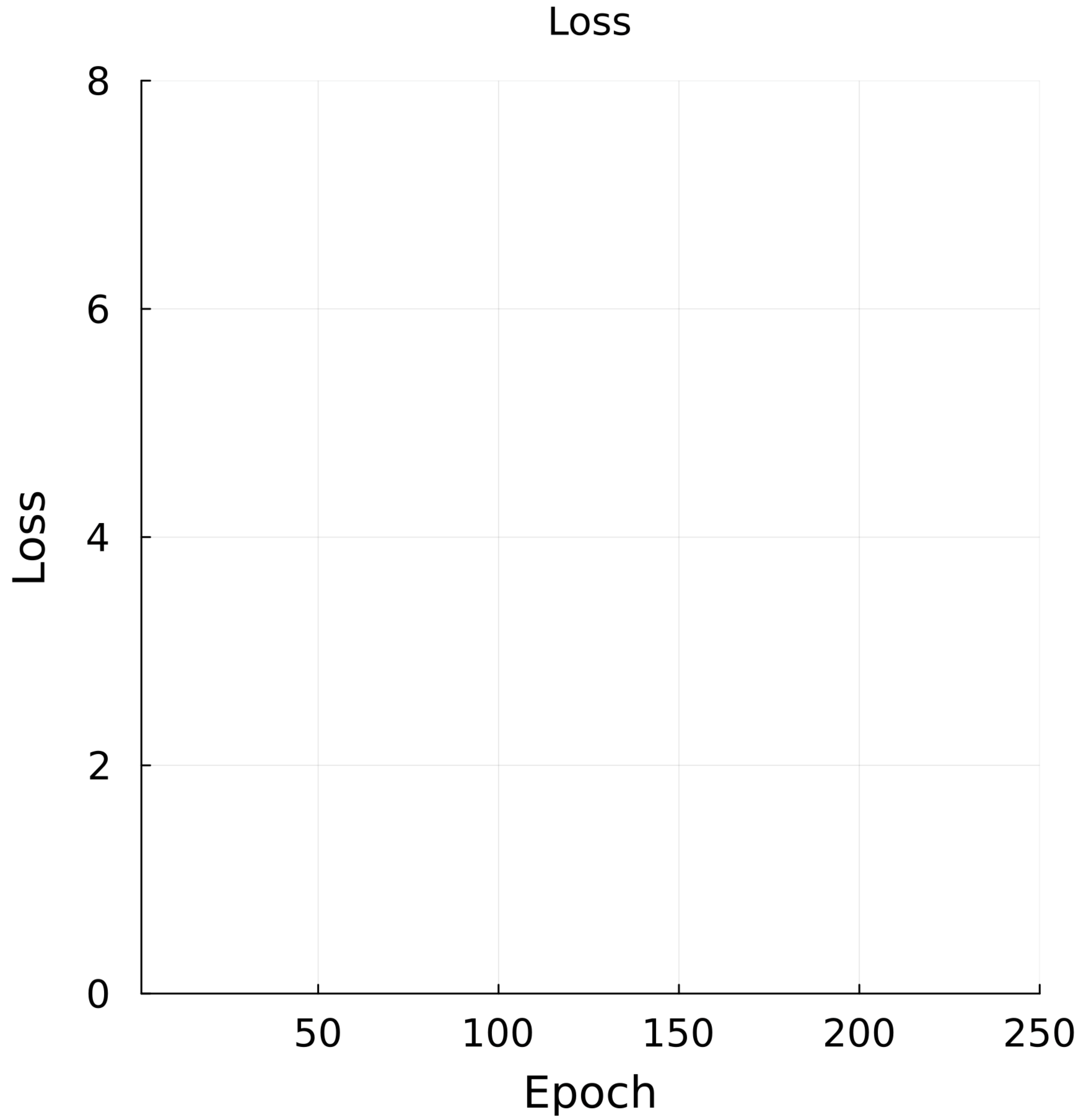
- Neural networks are trained by gradient descent + backprop
  - Calculate the network’s error, and adjust its parameters to decrease the error
- We can fit a model to data the same way (“differentiable programming”)

Gradient descent
(first-order optimization)

```
loss(p) = sum((obs - model(p)).^2)

N = 250
η = 1e-2   (large step size)

for i = 1:N
    p = p - η * gradient(loss, p)
end
```

julia



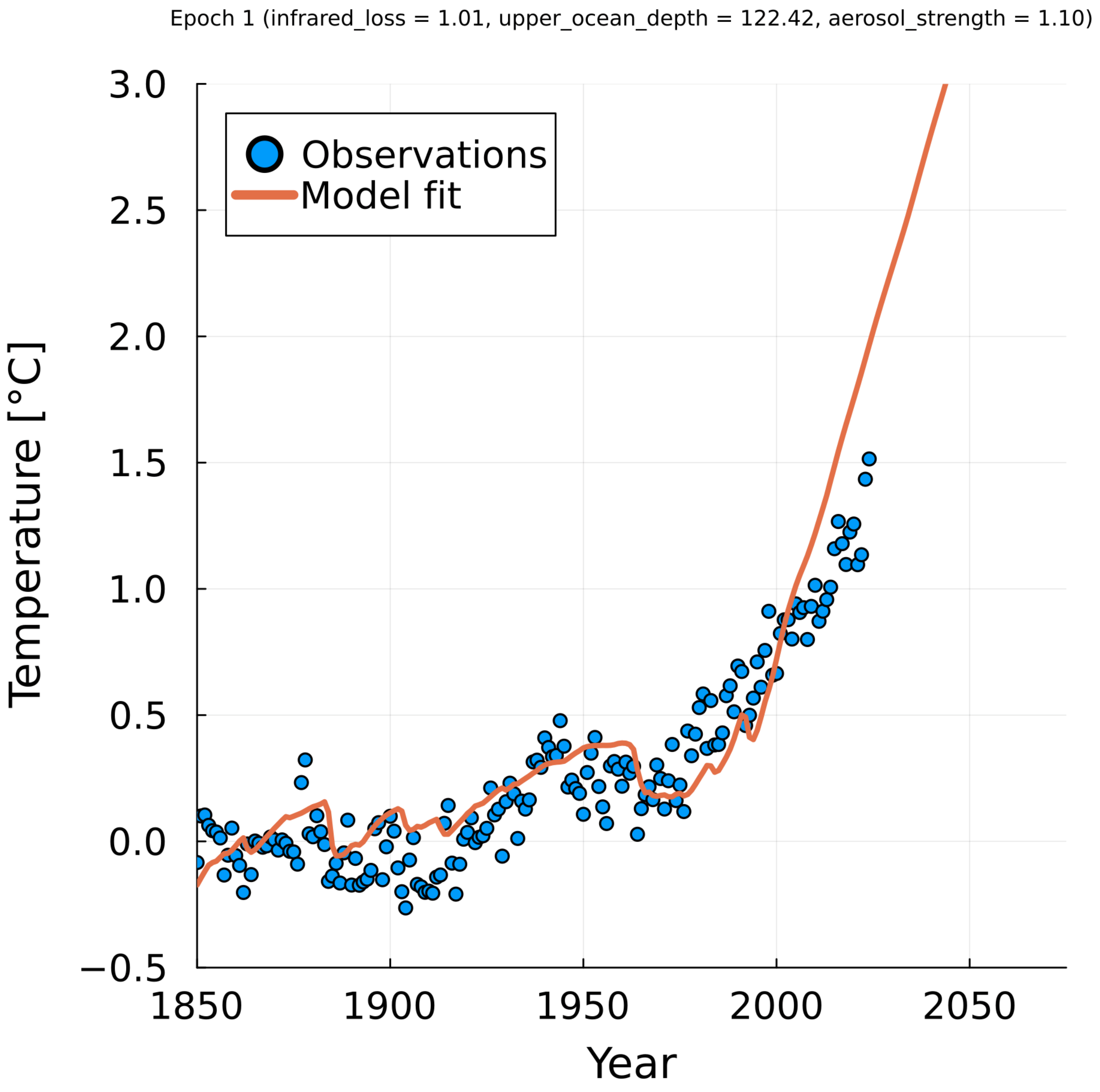



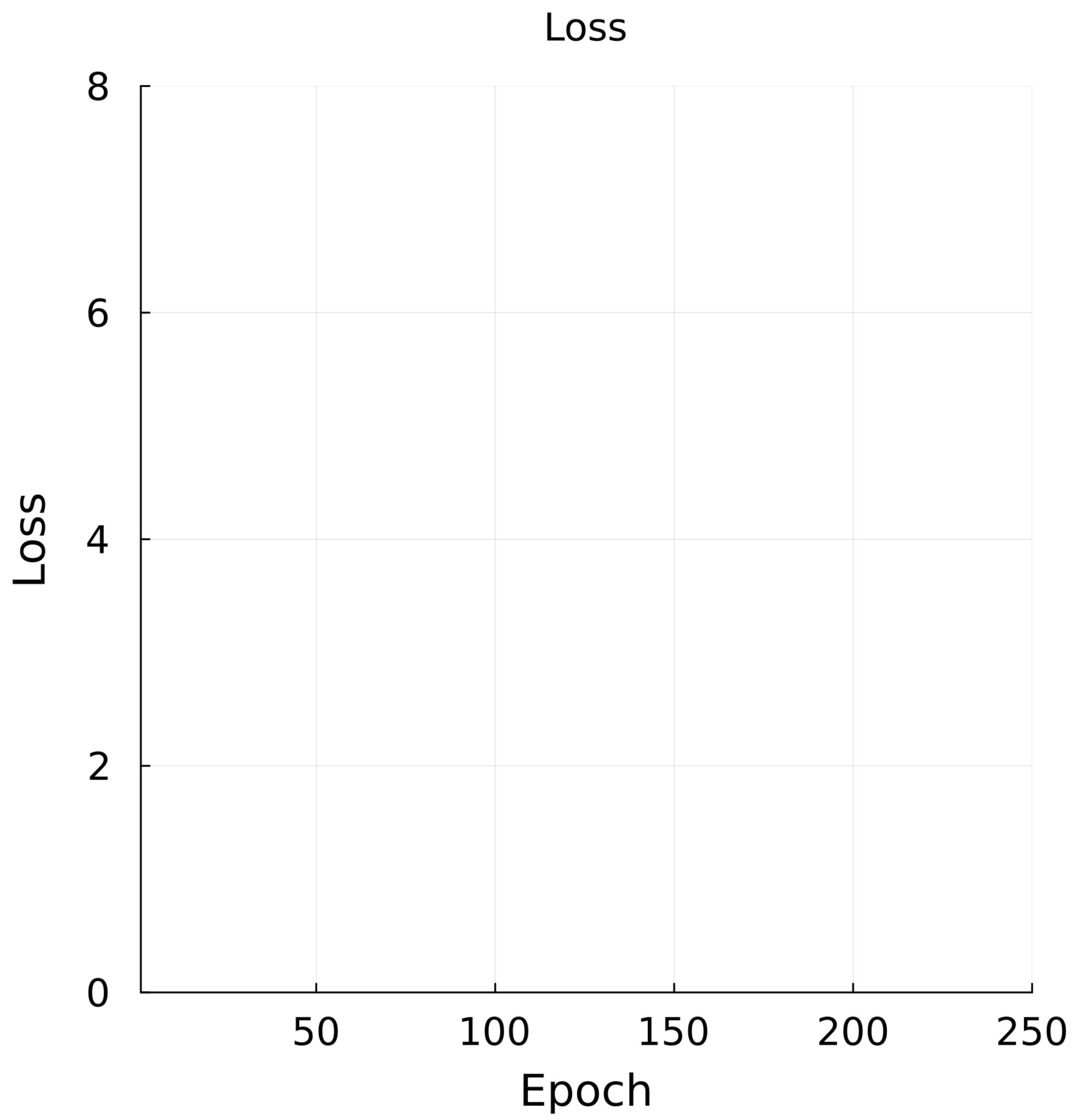
# Differentiable programming and adjoint models: Fitting digital twins to data with backpropagation

- Neural networks are trained by gradient descent + backprop
  - Calculate the network's error, and adjust its parameters to decrease the error
- We can fit a model to data the same way ("differentiable programming")

Newton's method
(second-order optimization)

```
loss(p) = sum((obs - model(p)).^2)

N = 250

for i = 1:N
    p = p - hessian(loss, p) \ gradient(loss, p)
end
```



# Backpropagation: Calculating the gradient

- We calculate gradients using the matrix chain rule
  - $\partial f(g(x))/\partial x = (\partial g/\partial x)^T(\partial f/\partial g)$
  - (being <u>sloppy</u> with gradient/Jacobian notation)
- We want gradient of loss w.r.t. the *parameters* of each layer
- … start with gradient w.r.t. the ("pre-activation") *output* of each layer
- First, the gradient of the loss w.r.t. the last layer:
  - $\partial \mathscr{L}/\partial z_5 \equiv \partial \mathscr{L}/\partial a_5 = 2(Y - a_5)$
- Then the gradient of the loss w.r.t. the next-to-last layer's pre-activation:

$$\partial \mathscr{L}/\partial z_4 = (\partial a_4/\partial z_4)^T(\partial z_5/\partial a_4)^T(\partial \mathscr{L}/\partial z_5)$$

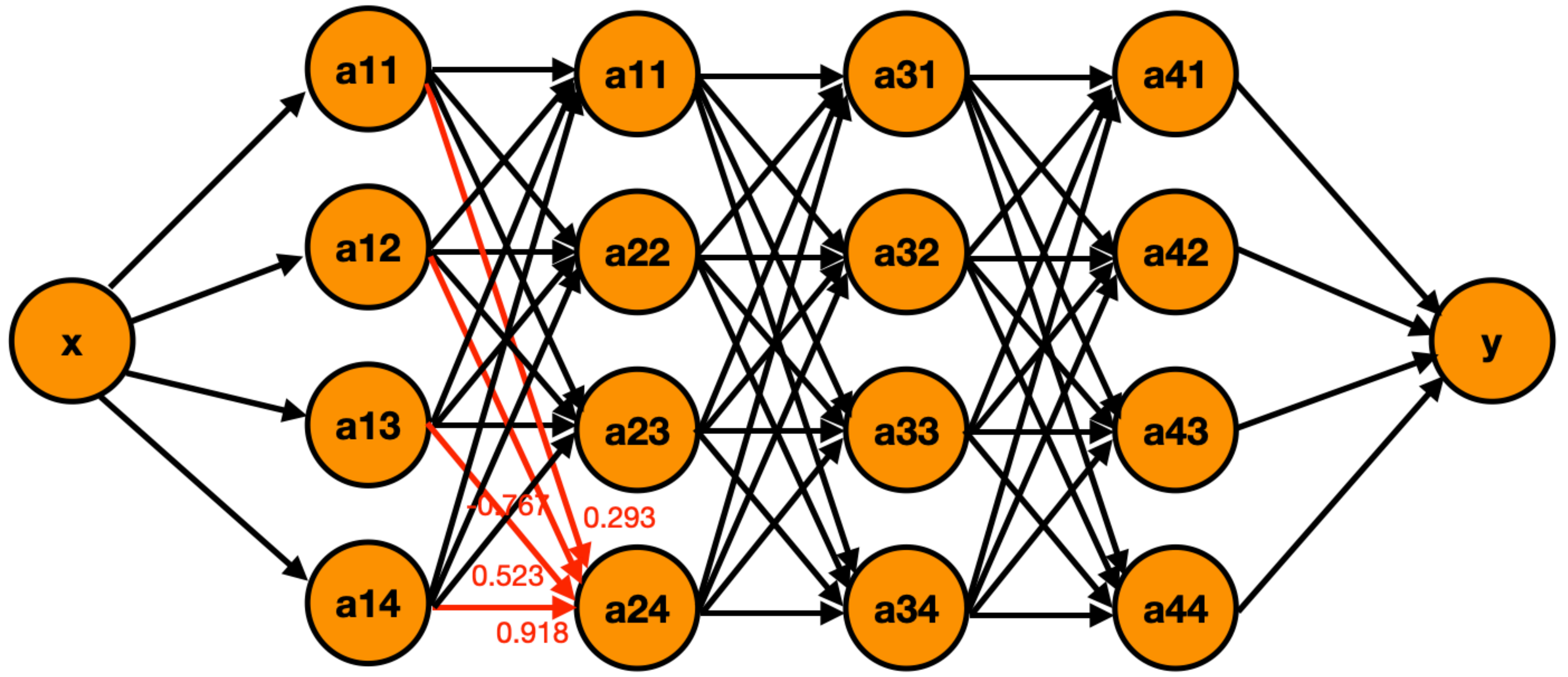
- $$= \left(\frac{\partial \sigma(z_4)}{\partial z_4}\right)\left(\frac{\partial (W_5 a_4 + b_5)}{\partial a_4}\right)\left(\frac{\partial \mathscr{L}}{\partial z_5}\right)$$

$$= \sigma'(z_4) W_5^T (\partial \mathscr{L}/\partial z_5)$$

$$\begin{aligned} z_1 &= W_1 x + b_1, & a_1 &= \sigma(z_1) \\ z_2 &= W_2 a_1 + b_2, & a_2 &= \sigma(z_2) \\ z_3 &= W_3 a_2 + b_3, & a_3 &= \sigma(z_3) \\ z_4 &= W_4 a_3 + b_4, & a_4 &= \sigma(z_4) \\ z_5 &= W_5 a_4 + b_5, & a_5 &= z_5 \end{aligned}$$

$$\mathscr{L}[\theta] = (Y - a_5)^T(Y - a_5)$$

**Deep neural network**



# Backpropagation: Calculating the gradient

- Neural network is composition of affine layers + pointwise nonlinearity (activation)
- Recursively derive gradient of loss with respect to each layer's activation from last layer to first (reverse-mode AD = backpropagation)

**Forward pass**

$$
\begin{aligned}
z_1 &= W_1 x + b_1, & a_1 &= \sigma(z_1) \\
z_2 &= W_2 a_1 + b_2, & a_2 &= \sigma(z_2) \\
z_3 &= W_3 a_2 + b_3, & a_3 &= \sigma(z_3) \\
z_4 &= W_4 a_3 + b_4, & a_4 &= \sigma(z_4) \\
z_5 &= W_5 a_4 + b_5, & a_5 &= z_5
\end{aligned}
$$

$$\mathscr{L} = (Y - a_5)^T (Y - a_5)$$

**Backward pass**

$$
\begin{aligned}
\nabla_{z_5} \mathscr{L} &= 2(z_5 - Y) \\
\nabla_{z_4} \mathscr{L} &= (W_5^T \nabla_{z_5} \mathscr{L}) \circ \sigma'(z_4) \\
\nabla_{z_3} \mathscr{L} &= (W_4^T \nabla_{z_4} \mathscr{L}) \circ \sigma'(z_3) \\
\nabla_{z_2} \mathscr{L} &= (W_3^T \nabla_{z_3} \mathscr{L}) \circ \sigma'(z_2) \\
\nabla_{z_1} \mathscr{L} &= (W_2^T \nabla_{z_2} \mathscr{L}) \circ \sigma'(z_1)
\end{aligned}
$$

$$
\begin{aligned}
\nabla_{W_5} \mathscr{L} &= (\nabla_{z_5} \mathscr{L})\, a_4^T, & \nabla_{b_5} \mathscr{L} &= \nabla_{z_5} \mathscr{L} \\
\nabla_{W_4} \mathscr{L} &= (\nabla_{z_4} \mathscr{L})\, a_3^T, & \nabla_{b_4} \mathscr{L} &= \nabla_{z_4} \mathscr{L} \\
\nabla_{W_3} \mathscr{L} &= (\nabla_{z_3} \mathscr{L})\, a_2^T, & \nabla_{b_3} \mathscr{L} &= \nabla_{z_3} \mathscr{L} \\
\nabla_{W_2} \mathscr{L} &= (\nabla_{z_2} \mathscr{L})\, a_1^T, & \nabla_{b_2} \mathscr{L} &= \nabla_{z_2} \mathscr{L} \\
\nabla_{W_1} \mathscr{L} &= (\nabla_{z_1} \mathscr{L})\, x^T, & \nabla_{b_1} \mathscr{L} &= \nabla_{z_1} \mathscr{L}
\end{aligned}
$$

# Backpropagation: Calculating the gradient

- Or in code:

**Forward pass**

```
a1 = σ.(W1*x + b1)
a2 = σ.(W2*a1 + b2)
a3 = σ.(W3*a2 + b3)
a4 = σ.(W4*a3 + b4)
a5 =     W5*a4 + b5
```

**Backward pass**

```
∇z5 = 2*(a5 - Y)
∇z4 = (W5' * ∇z5) .* σ′.(z4)
∇z3 = (W4' * ∇z4) .* σ′.(z3)
∇z2 = (W3' * ∇z3) .* σ′.(z2)
∇z1 = (W2' * ∇z2) .* σ′.(z1)

∇W5 = ∇z5 * a4' ;  ∇b5 = ∇z5
∇W4 = ∇z4 * a3' ;  ∇b4 = ∇z4
∇W3 = ∇z3 * a2' ;  ∇b3 = ∇z3
∇W2 = ∇z2 * a1' ;  ∇b2 = ∇z2
∇W1 = ∇z1 * x'  ;  ∇b1 = ∇z1
```

# Conclusions

- So far we have focused on “learning the machine”: UQ for Bmad parameters encoding mismatch between the digital twin and the real machine
- We could do more of this — larger part of the accelerator, more parameters, etc.
  - UQ becomes even more important with many parameters (non-identifiability)
- We could also consider other things:
  - Learning non-parametric uncertainties
  - Robust/risk-aware control — is it different from deterministic control?
  - Optimal experimental design
- All these options greatly benefit from a differentiable model

# RESERVE SLIDES

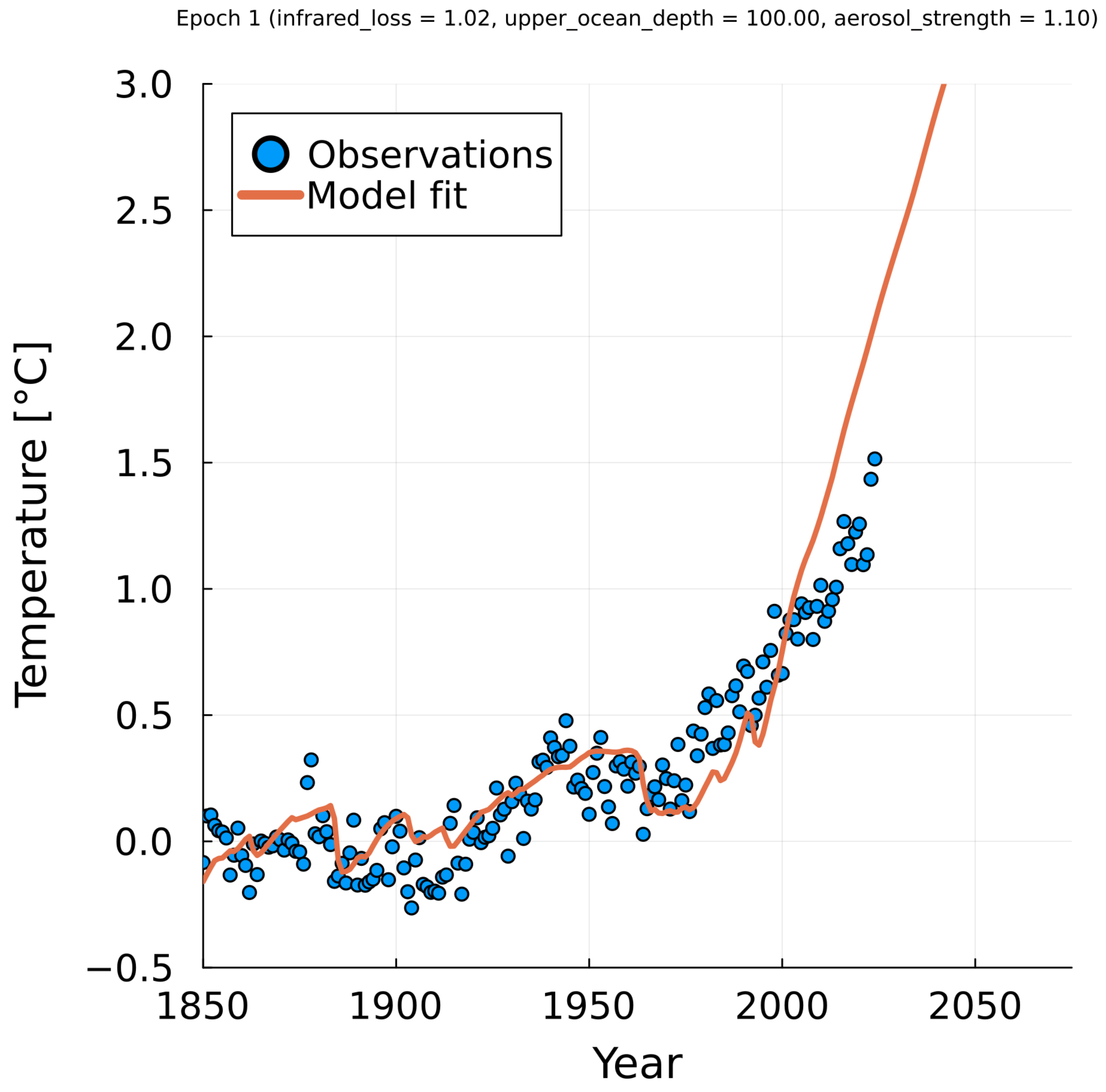
# Differentiable programming and adjoint models: Fitting digital twins to data with backpropagation

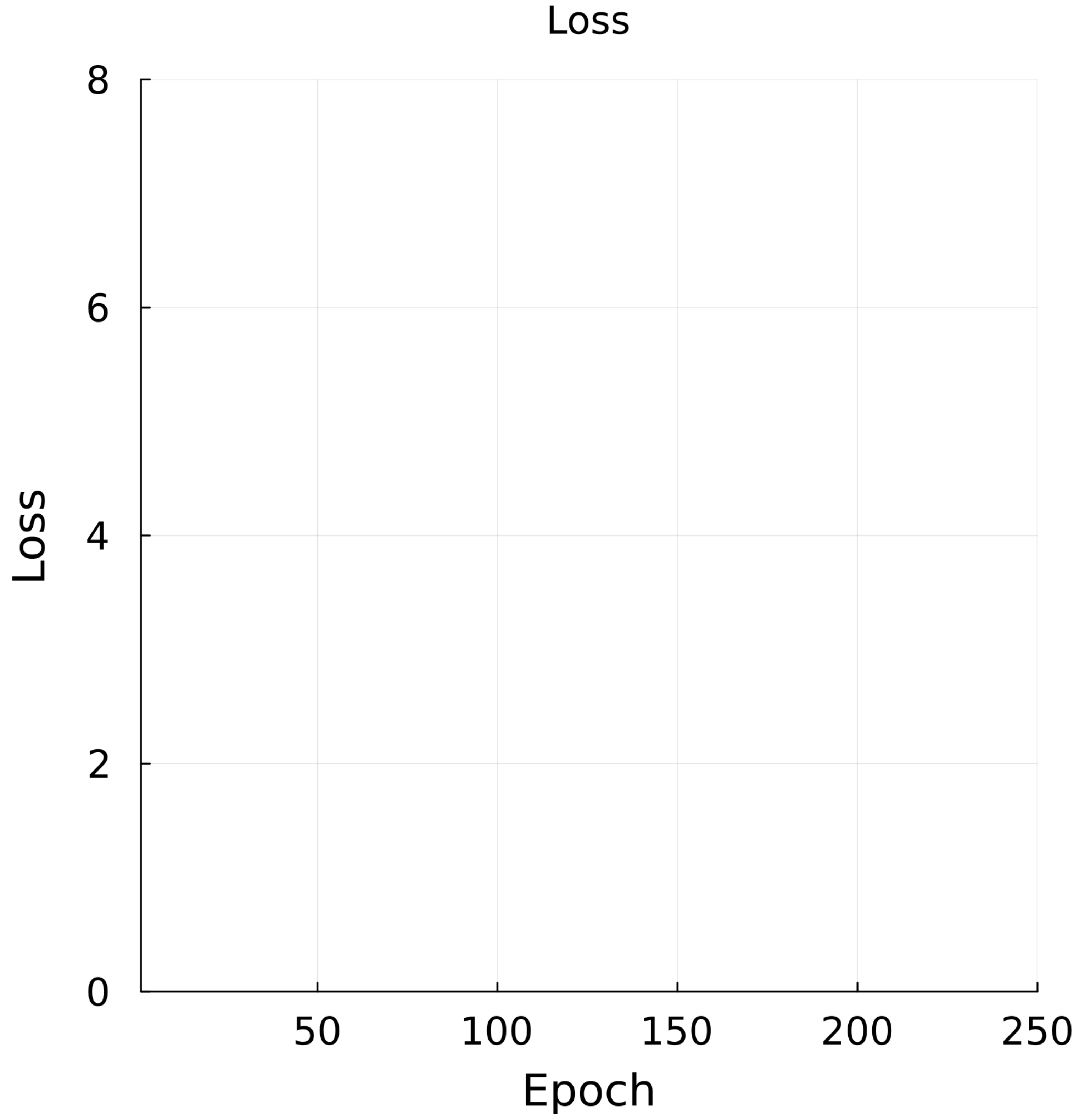
- Neural networks are trained by gradient descent + backprop
  - Calculate the network's error, and adjust its parameters to decrease the error
- We can fit a model to data the same way ("differentiable programming")

Gradient descent
(first-order optimization)

```
loss(p) = sum((obs - model(p)).^2)

N = 250
η = 2e-3   (small step size)

for i = 1:N
    p = p - η * gradient(loss, p)
end
```

julia



# Backpropagation: Calculating the gradient

- Write out our NN by hand:

$$
\begin{aligned}
z_1 &= W_1 x + b_1, & a_1 &= \sigma(z_1)\\
z_2 &= W_2 a_1 + b_2, & a_2 &= \sigma(z_2)\\
z_3 &= W_3 a_2 + b_3, & a_3 &= \sigma(z_3)\\
z_4 &= W_4 a_3 + b_4, & a_4 &= \sigma(z_4)\\
z_5 &= W_5 a_4 + b_5, & a_5 &= z_5
\end{aligned}
$$

  - with the loss $\mathscr{L}[\theta] = (Y - a_5)^T(Y - a_5)$
- We want the gradient of the loss with respect to parameters, $\nabla_{W_l}\mathscr{L}$ and $\nabla_{b_l}\mathscr{L}$
- We calculate the gradient using the matrix chain rule (being <u>sloppy</u> with notation):
  - $\partial f(g(x))/\partial x = (\partial g/\partial x)^T(\partial f/\partial g)$

# Backpropagation: Calculating the gradient

- We calculate gradients using the matrix chain rule
  - $\partial f(g(x))/\partial x = (\partial g/\partial x)^T (\partial f/\partial g)$
  - (being <u>sloppy</u> with gradient/Jacobian notation)
- We want gradient of loss w.r.t. the *parameters* of each layer
- … but start with the gradient w.r.t. the ("pre-activation") *output* of each layer
- First, the gradient of the loss w.r.t. the last layer:
  - $\partial \mathscr{L}/\partial z_5 \equiv \partial \mathscr{L}/\partial a_5 = 2(Y - a_5)$
- Then the gradient of the loss w.r.t. the next-to-last layer's pre-activation:
-
$$
\begin{aligned}
\partial \mathscr{L}/\partial z_4 &= (\partial a_4/\partial z_4)^T (\partial z_5/\partial a_4)^T (\partial \mathscr{L}/\partial z_5) \\
&= \left(\frac{\partial \sigma(z_4)}{\partial z_4}\right)\left(\frac{\partial (W_5 a_4 + b_5)}{\partial a_4}\right)\left(\frac{\partial \mathscr{L}}{\partial z_5}\right) \\
&= \sigma'(z_4) W_5^T (\partial \mathscr{L}/\partial z_5)
\end{aligned}
$$

$$
\begin{aligned}
z_1 &= W_1 x + b_1, & a_1 &= \sigma(z_1) \\
z_2 &= W_2 a_1 + b_2, & a_2 &= \sigma(z_2) \\
z_3 &= W_3 a_2 + b_3, & a_3 &= \sigma(z_3) \\
z_4 &= W_4 a_3 + b_4, & a_4 &= \sigma(z_4) \\
z_5 &= W_5 a_4 + b_5, & a_5 &= z_5
\end{aligned}
$$

$$\mathscr{L}[\theta] = (Y - a_5)^T (Y - a_5)$$

# Backpropagation: Calculating the gradient

- We can keep working backwards recursively to get gradients for each layer

$$\partial \mathscr{L} / \partial z_4 = (\partial a_4 / \partial z_4)^T (\partial z_5 / \partial a_4)^T (\partial \mathscr{L} / \partial z_5)$$

- $$= \left( \frac{\partial \sigma(z_4)}{\partial z_4} \right) \left( \frac{\partial (W_5 a_4 + b_5)}{\partial a_4} \right) \left( \frac{\partial \mathscr{L}}{\partial z_5} \right)$$

$$= \sigma'(z_4) W_5^T (\partial \mathscr{L} / \partial z_5)$$

$$\partial \mathscr{L} / \partial z_3 = (\partial a_3 / \partial z_3)^T (\partial z_4 / \partial a_3)^T (\partial \mathscr{L} / \partial z_4)$$

- $$= \left( \frac{\partial \sigma(z_3)}{\partial z_3} \right) \left( \frac{\partial (W_4 a_3 + b_4)}{\partial a_3} \right) \left( \frac{\partial \mathscr{L}}{\partial z_4} \right)$$

$$= \sigma'(z_3) W_4^T (\partial \mathscr{L} / \partial z_4)$$

$$z_1 = W_1 x + b_1, \qquad a_1 = \sigma(z_1)$$
$$z_2 = W_2 a_1 + b_2, \qquad a_2 = \sigma(z_2)$$
$$z_3 = W_3 a_2 + b_3, \qquad a_3 = \sigma(z_3)$$
$$z_4 = W_4 a_3 + b_4, \qquad a_4 = \sigma(z_4)$$
$$z_5 = W_5 a_4 + b_5, \qquad a_5 = z_5$$
$$\mathscr{L}[\theta] = (Y - a_5)^T (Y - a_5)$$

# Backpropagation: Calculating the gradient

- More compactly, we get:

**Forward pass**

$$
\begin{aligned}
z_1 &= W_1 x + b_1\,, & a_1 &= \sigma(z_1) \\
z_2 &= W_2 a_1 + b_2\,, & a_2 &= \sigma(z_2) \\
z_3 &= W_3 a_2 + b_3\,, & a_3 &= \sigma(z_3) \\
z_4 &= W_4 a_3 + b_4\,, & a_4 &= \sigma(z_4) \\
z_5 &= W_5 a_4 + b_5\,, & a_5 &= z_5
\end{aligned}
$$

$$\mathscr{L} = (Y - a_5)^T (Y - a_5)$$

**Backward pass**

$$
\begin{aligned}
\nabla_{z_5} \mathscr{L} &= 2(z_5 - Y) \\
\nabla_{z_4} \mathscr{L} &= (W_5^T \nabla_{z_5} \mathscr{L}) \circ \sigma'(z_4) \\
\nabla_{z_3} \mathscr{L} &= (W_4^T \nabla_{z_4} \mathscr{L}) \circ \sigma'(z_3) \\
\nabla_{z_2} \mathscr{L} &= (W_3^T \nabla_{z_3} \mathscr{L}) \circ \sigma'(z_2) \\
\nabla_{z_1} \mathscr{L} &= (W_2^T \nabla_{z_2} \mathscr{L}) \circ \sigma'(z_1)
\end{aligned}
$$

# Backpropagation: Calculating the gradient

- Now we're ready to compute the gradients w.r.t. *parameters*

$$\frac{\partial \mathscr{L}}{\partial W_5} = \left(\frac{\partial z_5}{\partial W_5}\right)^T \frac{\partial \mathscr{L}}{\partial z_5}$$

$$= \left(\frac{\partial (W_5 a_4 + b_5)}{\partial W_5}\right)^T \frac{\partial \mathscr{L}}{\partial z_5}$$

- $$= \frac{\partial \mathscr{L}}{\partial z_5} a_4^T$$

$$\frac{\partial \mathscr{L}}{\partial b_5} = \left(\frac{\partial z_5}{\partial b_5}\right)^T \frac{\partial \mathscr{L}}{\partial z_5}$$

$$= \left(\frac{\partial (W_5 a_4 + b_5)}{\partial b_5}\right)^T \frac{\partial \mathscr{L}}{\partial z_5}$$

- $$= \frac{\partial \mathscr{L}}{\partial z_5}$$

$$\begin{aligned}
z_1 &= W_1 x + b_1, & a_1 &= \sigma(z_1) \\
z_2 &= W_2 a_1 + b_2, & a_2 &= \sigma(z_2) \\
z_3 &= W_3 a_2 + b_3, & a_3 &= \sigma(z_3) \\
z_4 &= W_4 a_3 + b_4, & a_4 &= \sigma(z_4) \\
z_5 &= W_5 a_4 + b_5, & a_5 &= z_5
\end{aligned}$$

$$\mathscr{L}[\theta] = (Y - a_5)^T (Y - a_5)$$

# Backpropagation: Calculating the gradient

- More compactly, we get:

**Forward pass**

$$
\begin{aligned}
z_1 &= W_1 x + b_1\,, & a_1 &= \sigma(z_1) \\
z_2 &= W_2 a_1 + b_2\,, & a_2 &= \sigma(z_2) \\
z_3 &= W_3 a_2 + b_3\,, & a_3 &= \sigma(z_3) \\
z_4 &= W_4 a_3 + b_4\,, & a_4 &= \sigma(z_4) \\
z_5 &= W_5 a_4 + b_5\,, & a_5 &= z_5
\end{aligned}
$$

$$\mathscr{L} = (Y - a_5)^T (Y - a_5)$$

**Backward pass**

$$
\begin{aligned}
\nabla_{z_5}\mathscr{L} &= 2(z_5 - Y) \\
\nabla_{z_4}\mathscr{L} &= (W_5^T \nabla_{z_5}\mathscr{L}) \circ \sigma'(z_4) \\
\nabla_{z_3}\mathscr{L} &= (W_4^T \nabla_{z_4}\mathscr{L}) \circ \sigma'(z_3) \\
\nabla_{z_2}\mathscr{L} &= (W_3^T \nabla_{z_3}\mathscr{L}) \circ \sigma'(z_2) \\
\nabla_{z_1}\mathscr{L} &= (W_2^T \nabla_{z_2}\mathscr{L}) \circ \sigma'(z_1)
\end{aligned}
$$

$$
\begin{aligned}
\nabla_{W_5}\mathscr{L} &= (\nabla_{z_5}\mathscr{L})\, a_4^T, & \nabla_{b_5}\mathscr{L} &= \nabla_{z_5}\mathscr{L} \\
\nabla_{W_4}\mathscr{L} &= (\nabla_{z_4}\mathscr{L})\, a_3^T, & \nabla_{b_4}\mathscr{L} &= \nabla_{z_4}\mathscr{L} \\
\nabla_{W_3}\mathscr{L} &= (\nabla_{z_3}\mathscr{L})\, a_2^T, & \nabla_{b_3}\mathscr{L} &= \nabla_{z_3}\mathscr{L} \\
\nabla_{W_2}\mathscr{L} &= (\nabla_{z_2}\mathscr{L})\, a_1^T, & \nabla_{b_2}\mathscr{L} &= \nabla_{z_2}\mathscr{L} \\
\nabla_{W_1}\mathscr{L} &= (\nabla_{z_1}\mathscr{L})\, x^T, & \nabla_{b_1}\mathscr{L} &= \nabla_{z_1}\mathscr{L}
\end{aligned}
$$

# Backpropagation: Calculating the gradient

- Or in code:

**Forward pass**

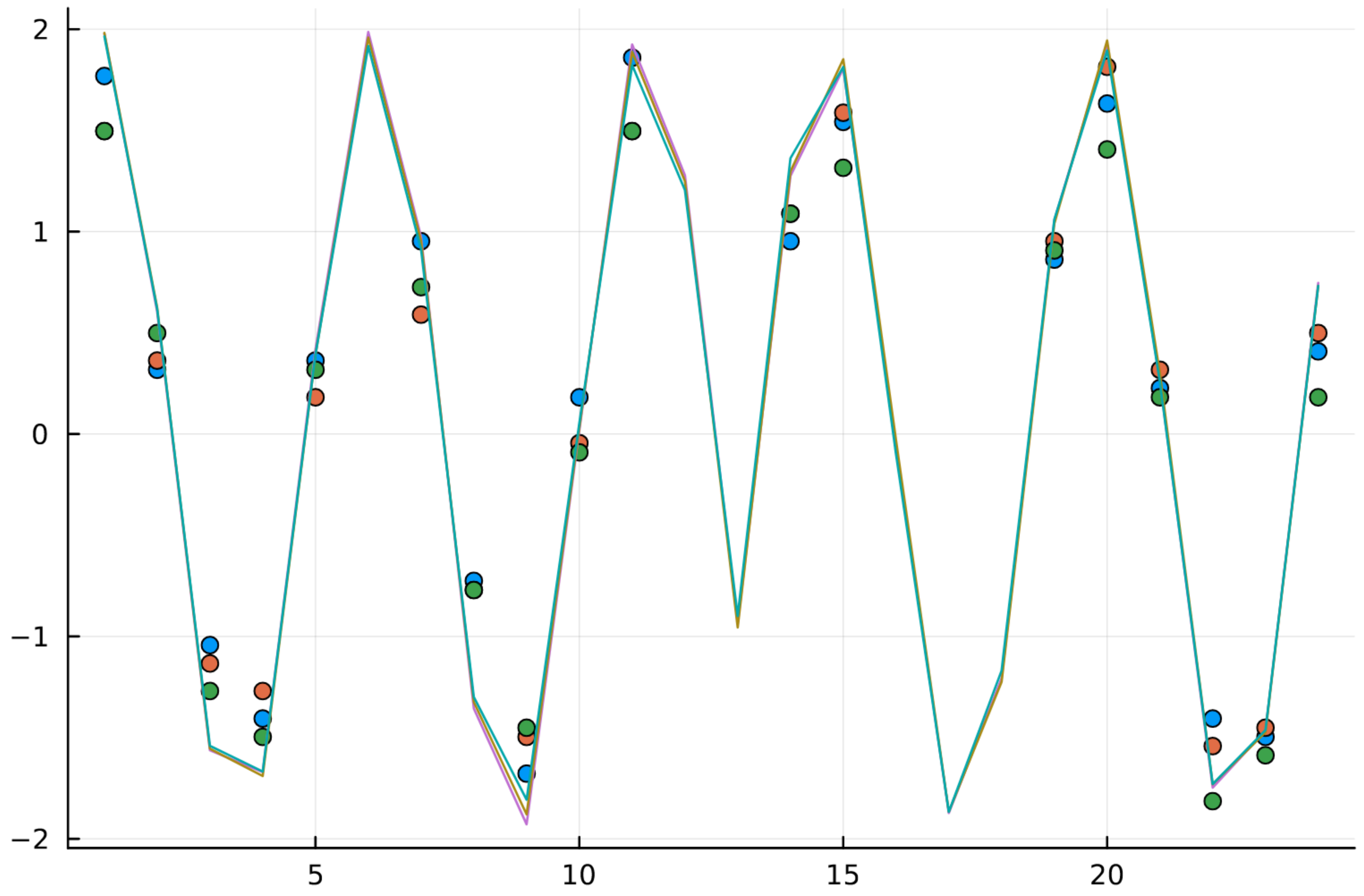
```
a1 = σ.(W1*x + b1)
a2 = σ.(W2*a1 + b2)
a3 = σ.(W3*a2 + b3)
a4 = σ.(W4*a3 + b4)
a5 =    W5*a4 + b5
```

**Backward pass**

```
∇z5 = 2*(a5 - Y)
∇z4 = (W5' * ∇z5) .* σ′.(z4)
∇z3 = (W4' * ∇z4) .* σ′.(z3)
∇z2 = (W3' * ∇z3) .* σ′.(z2)
∇z1 = (W2' * ∇z2) .* σ′.(z1)

∇W5 = ∇z5 * a4' ;  ∇b5 = ∇z5
∇W4 = ∇z4 * a3' ;  ∇b4 = ∇z4
∇W3 = ∇z3 * a2' ;  ∇b3 = ∇z3
∇W2 = ∇z2 * a1' ;  ∇b2 = ∇z2
∇W1 = ∇z1 * x'  ;  ∇b1 = ∇z1
```

# Accelerator control

- For this talk: use Bmad model to predict beam position in response to operator inputs
    - Can control other quantities (polarization, emittance, luminosity, “figure of merit”, …)
- Actual beam position measured (with error) at 24 BPMs
- Bmad can be used in an optimizer to find inputs that better <u>control the beam</u>
    - If Bmad is an accurate “twin” of the real machine
    - Model accuracy depends on assumed, but unknown characteristics of the machine



# Parameter estimation (tuning)

- **Controls** $c$: known inputs that the operator specifies (currents, …)
- **Parameters** $\theta$: fixed but unknown system properties (misalignments, current biases, …)
- **Model** $m(c;\theta)$: response of the system to its controls, assuming parameters are known
  - e.g., predicted beam position due to currents, if we knew all machine characteristics
  - Here we use Bmad as a “digital twin”
- **Measurements** $y(c)$: observed system response to the control

- Estimate parameters by fitting model to measurements, e.g. by least squares:

$$\hat{\theta} = \text{arg min}_\theta \sum_i (y_i - m_i(c;\theta))^2$$

# Parameter estimation (inference)

- In **parameter fitting**, the goal is to find the best-fitting set of parameters

$$\hat{\theta}$$

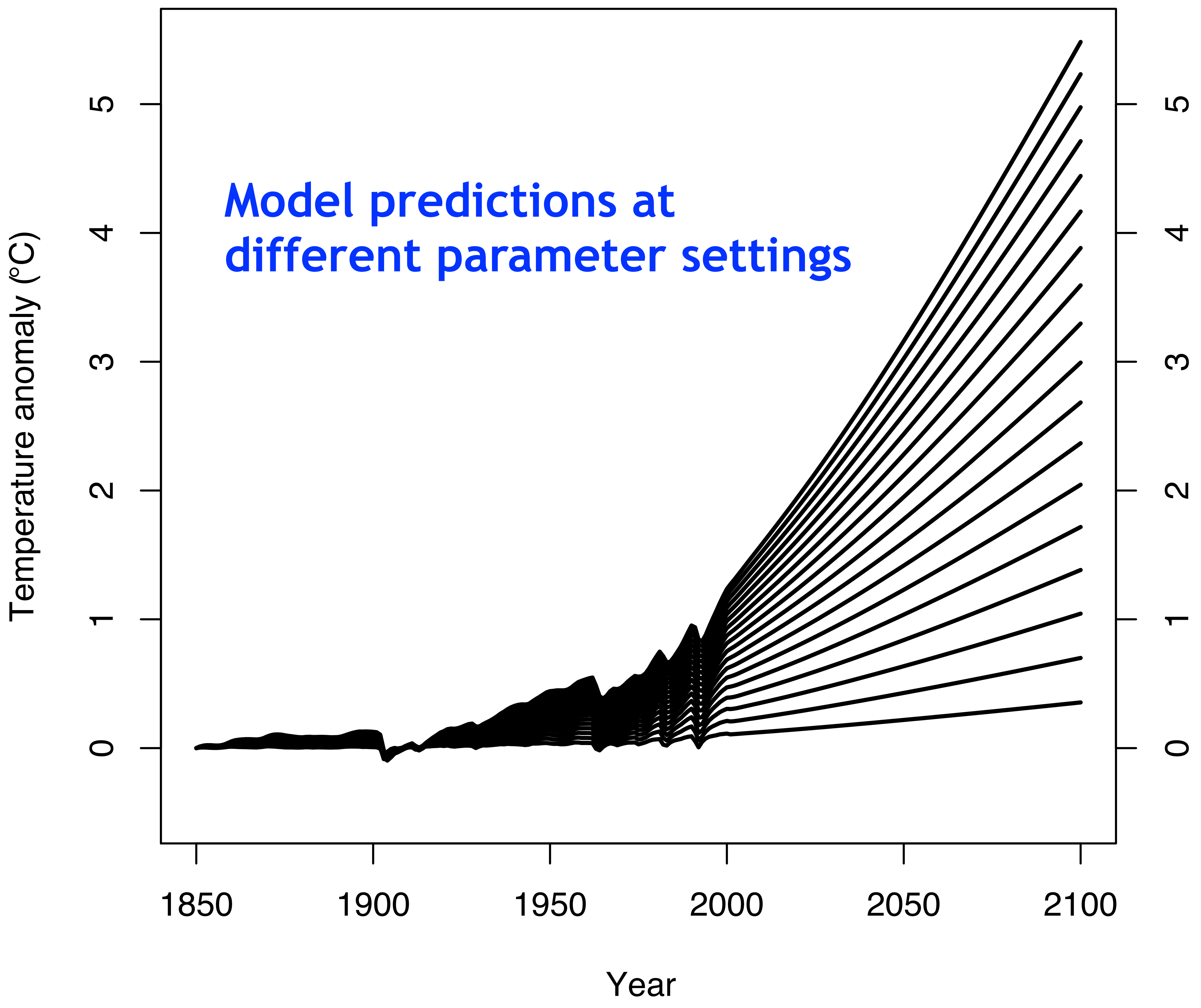
- In Bayesian **uncertainty quantification** (UQ), the goal is to estimate a probability distribution over the unknown parameters, not just a single point estimate (best fit).
  - Posterior distribution (probability of unknown parameters, conditional on measurements):

$$p(\theta | y)$$

- When do you want to go to the trouble of UQ?
  - May be *many* "best fits", with different implications for predicted behavior
  - (in pure science) To put error bars on predictions (e.g,. compare theory and experiment)
  - (in control) Nonlinear response / non-Gaussian errors mean that *best fit parameters* don't correspond to *controller with best average performance*
  - (in control) We might want to know the expected reliability of a control policy

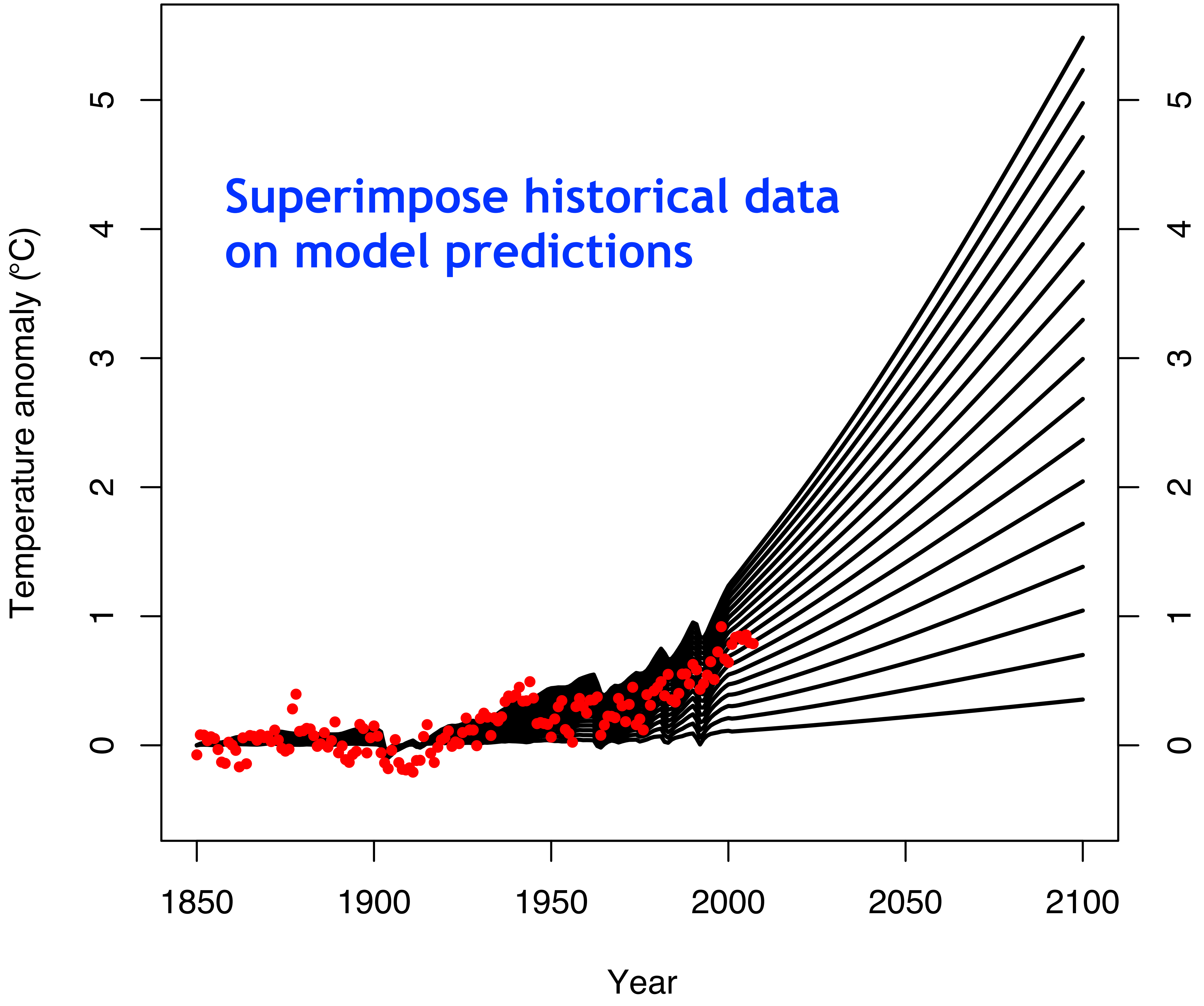
# Probabilistically fitting a model to data

- Example of a 3-parameter model from climate science
- Could tune these parameters to data
- But rather than a point estimate, we can assign each parameter value a probability weight
  - Weight given by “goodness of fit”
- It is (probabilistic, nonlinear) **regression**



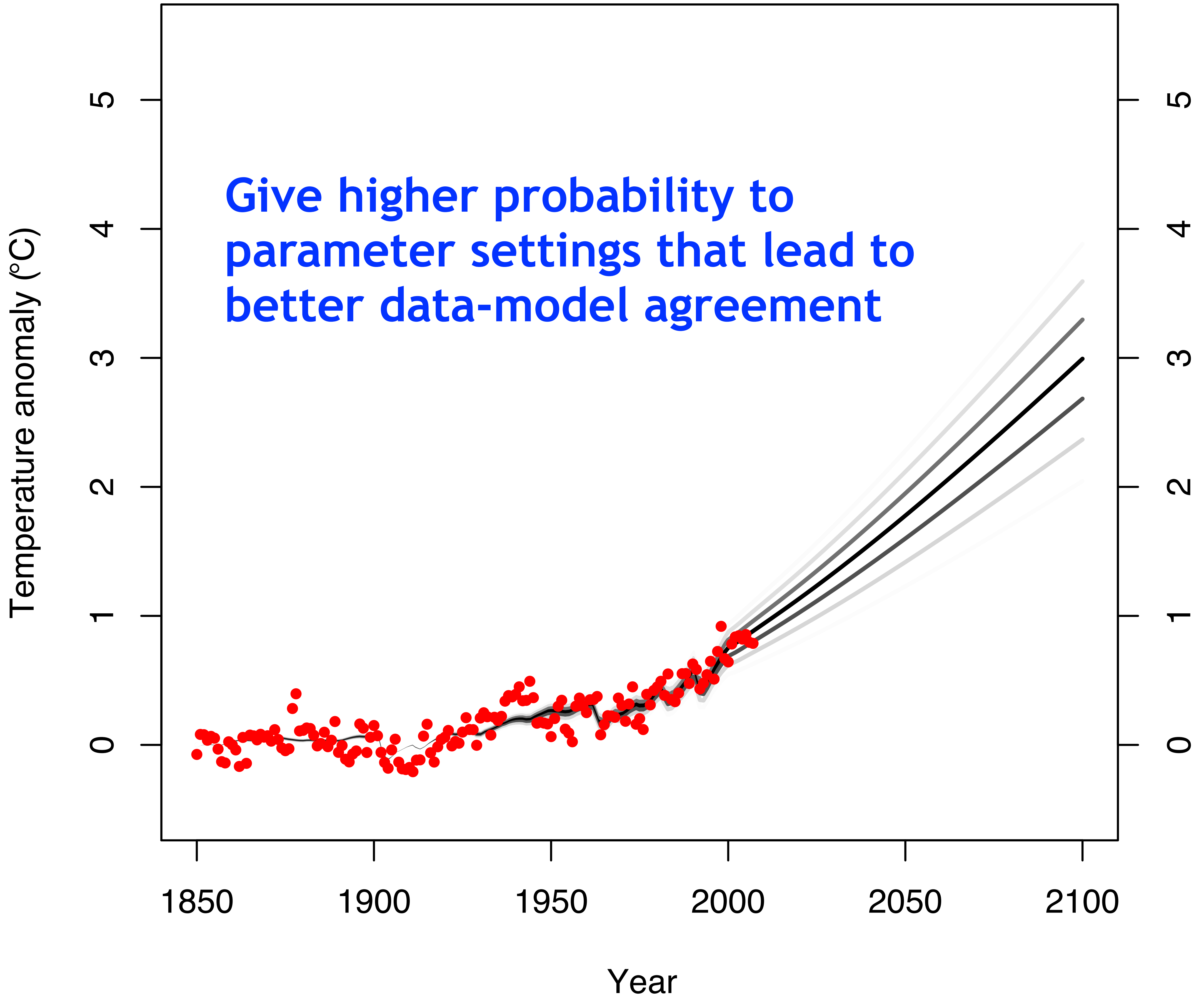
# Probabilistically fitting a model to data

- Example of a 3-parameter model from climate science
- Could tune these parameters to data
- But rather than a point estimate, we can assign each parameter value a probability weight
  - Weight given by “goodness of fit”
- It is (probabilistic, nonlinear) **regression**



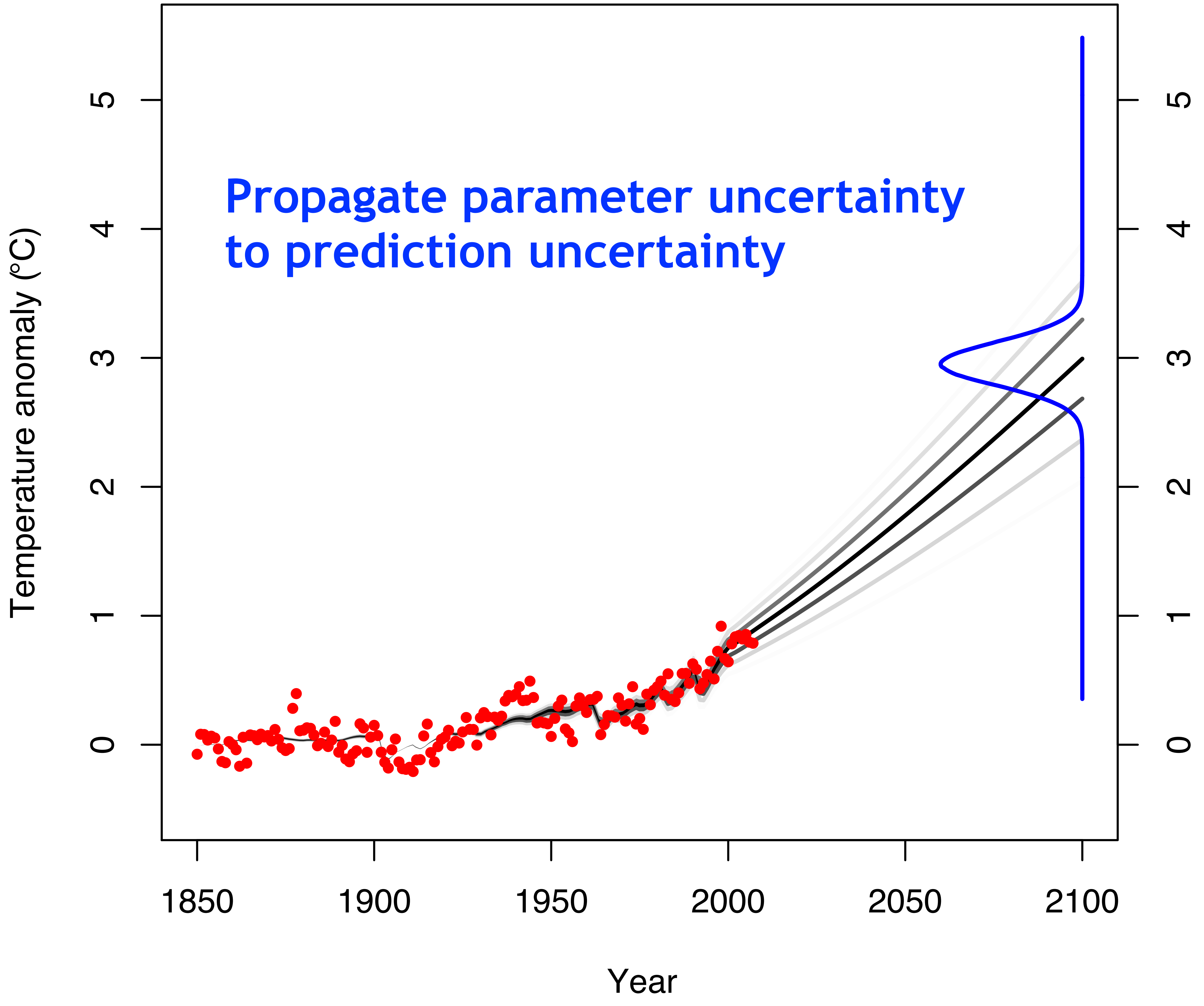
# Probabilistically fitting a model to data

- Example of a 3-parameter model from climate science
- Could tune these parameters to data
- But rather than a point estimate, we can assign each parameter value a probability weight
    - Weight given by “goodness of fit”
- It is (probabilistic, nonlinear) **regression**



# Probabilistically fitting a model to data

- Example of a 3-parameter model from climate science
- Could tune these parameters to data
- But rather than a point estimate, we can assign each parameter value a probability weight
  - Weight given by “goodness of fit”
- It is (probabilistic, nonlinear) **regression**



# Posterior distribution: p(parameters|data)

# Bayesian inference (probabilistic parameter estimation)

- Goal: infer parameter probability density functions (PDFs) from data
  - *Conditional* inference: infer parameter uncertainties from known data

<u>Bayes theorem:</u> $p(\text{parameters}|\text{data}) = p(\text{data}|\text{parameters})\, p(\text{parameters}) \,/\, p(\text{data})$

posterior ∝ likelihood × prior

To infer <u>posterior</u> PDF, need to know <u>likelihood function</u> (data-generating distribution) and <u>prior</u> distribution (beliefs about parameters before seeing the data).

Bayesian uncertainty quantifies “ignorance” about the true parameter values.

# Prior distribution: $p$(parameters)

- What you believe about the parameters before you've seen the data
  - Use outside information (physical predictions, other data sources)
  - Priors <u>must</u> be independent of conditioning data (no double-counting)
  - Can use posterior inferred from other data as prior (sequential Bayesian update)

- Elicit booster prior uncertainties from operators
  - trim current errors $\approx\pm 10^{-3}$ (1-$\sigma$)
  - magnet misalignments informed from previous surveys
  - transfer function coefficient ranges harder to elicit (not directly measured)

# Likelihood function: *p*(data|parameters)

Assume data is distributed randomly (additively) around an accelerator model (e.g. Bmad):

**Measurements(*BPM location i*) = Model(*control*; *parameters*) + Noise**

$$y_i = m(c;\theta) + \varepsilon$$

Assume noise process is noise process ($\varepsilon$) is normal (independent and identically distributed, or *iid*), zero mean: $\varepsilon \sim N(0, \sigma^2)$

$$y_i \sim N(\mu = m_i(c;\theta), \sigma^2)$$

*(Likelihood: one observation)*

$$p(y_i\,|\,\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2}\frac{(y_i - m_i(c;\theta))^2}{\sigma^2}\right]$$

*(Likelihood: all observations)*

$$p(y\,|\,\theta) = \Pi_i\, p(y_i\,|\,\theta) = \frac{1}{\left(\Pi_i \sqrt{2\pi\sigma_i^2}\right)} \exp\left[-\frac{1}{2}\frac{\sum_i (y_i - m_i(c;\theta))^2}{\sigma^2}\right]$$

# Likelihood function: $p$(data|parameters)

Note: for an *iid* normal likelihood model, the *maximum likelihood estimate* (MLE) for $\theta$ is the same as a *least squares* or *minimum* $\chi^2$ fit.

*(Likelihood: all observations)*

$$p(y\,|\,\theta) = \Pi_i\, p(y_i\,|\,\theta) = \frac{1}{\left(\Pi_i \sqrt{2\pi\sigma_i^2}\right)} \exp\left[-\frac{1}{2}\frac{\sum_i (y_i - m_i(c;\theta))^2}{\sigma^2}\right] \propto \exp(-\chi^2/2)$$

Assume noise process is noise process ($\varepsilon$) is normal (independent and identically distributed, or *iid*), zero mean: $\varepsilon \sim N(0, \sigma^2)$

$$y_i \sim N(\mu = m_i(c;\theta), \sigma^2)$$

# Posterior distribution: *p*(parameters|data)

The posterior is proportional to the product of the likelihood and prior (which we will assume is independent for each parameter).

$$p(\theta\,|\,y) \propto p(y\,|\,\theta)\,p(\theta) = \frac{1}{\left(\prod_i \sqrt{2\pi\sigma_i^2}\right)} \exp\left[-\frac{1}{2}\frac{\sum_{i=1}^{N}(y_i - m_i(c;\theta))^2}{\sigma_i^2}\right] \times \prod_{k=1}^{K} p(\theta_k)$$

The log posterior is like a “regularized” least squares fit. If the priors are assumed normal around some typical mean, $\theta_k \sim N(\bar{\theta}_k, \nu_k^2)$, then the “maximum a posteriori” (MAP) estimate arises from minimizing a least squares term with an additional “penalty” term on the parameters.

$$-\log p(\theta\,|\,y) \propto \sum_{i=1}^{N} \frac{(y_i - m_i(c;\theta))^2}{\sigma^2} + \sum_{k=1}^{K} \frac{(\theta_k - \bar{\theta}_k)^2}{\nu^2} + const$$

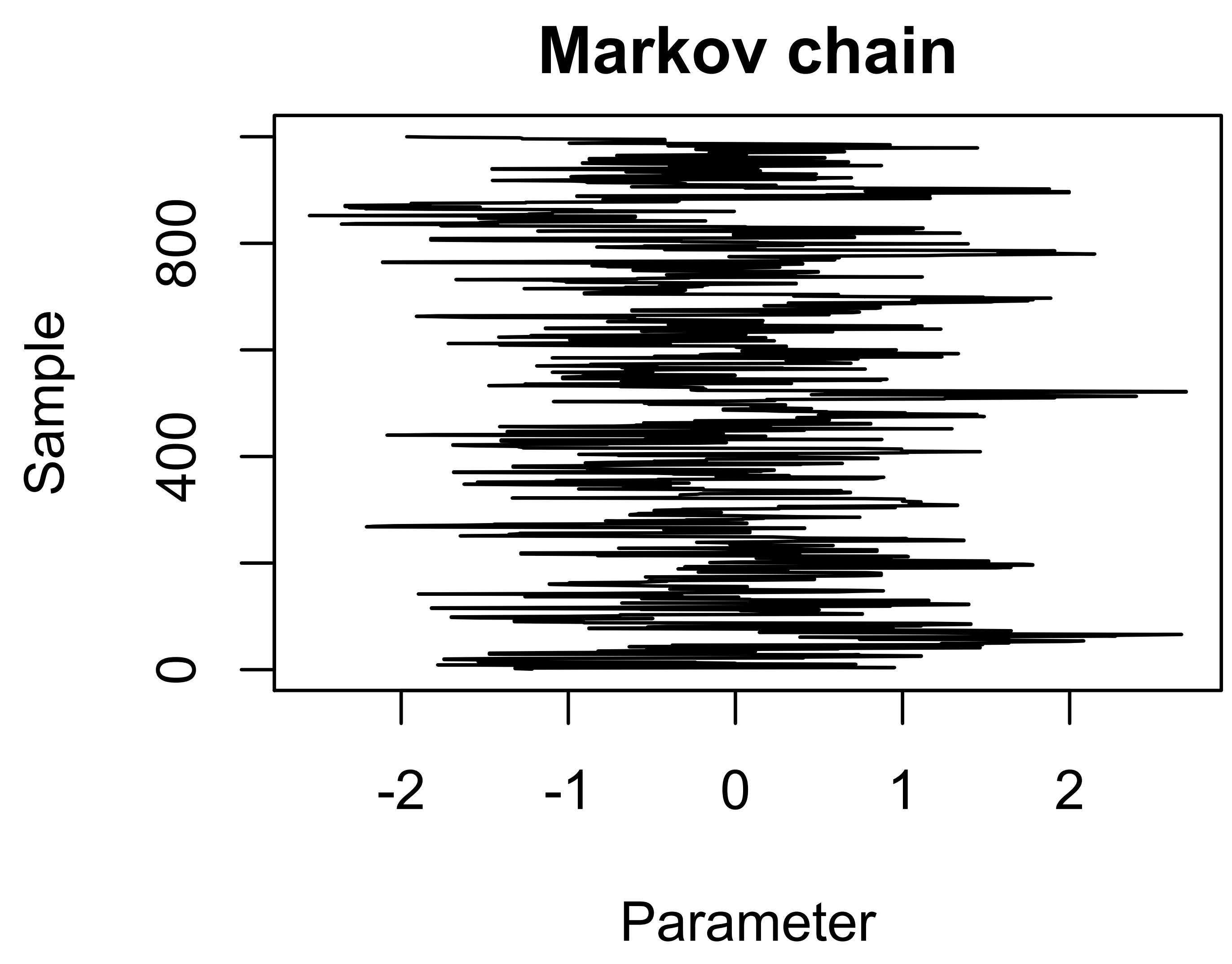
# Posterior distribution: *p*(parameters|data)

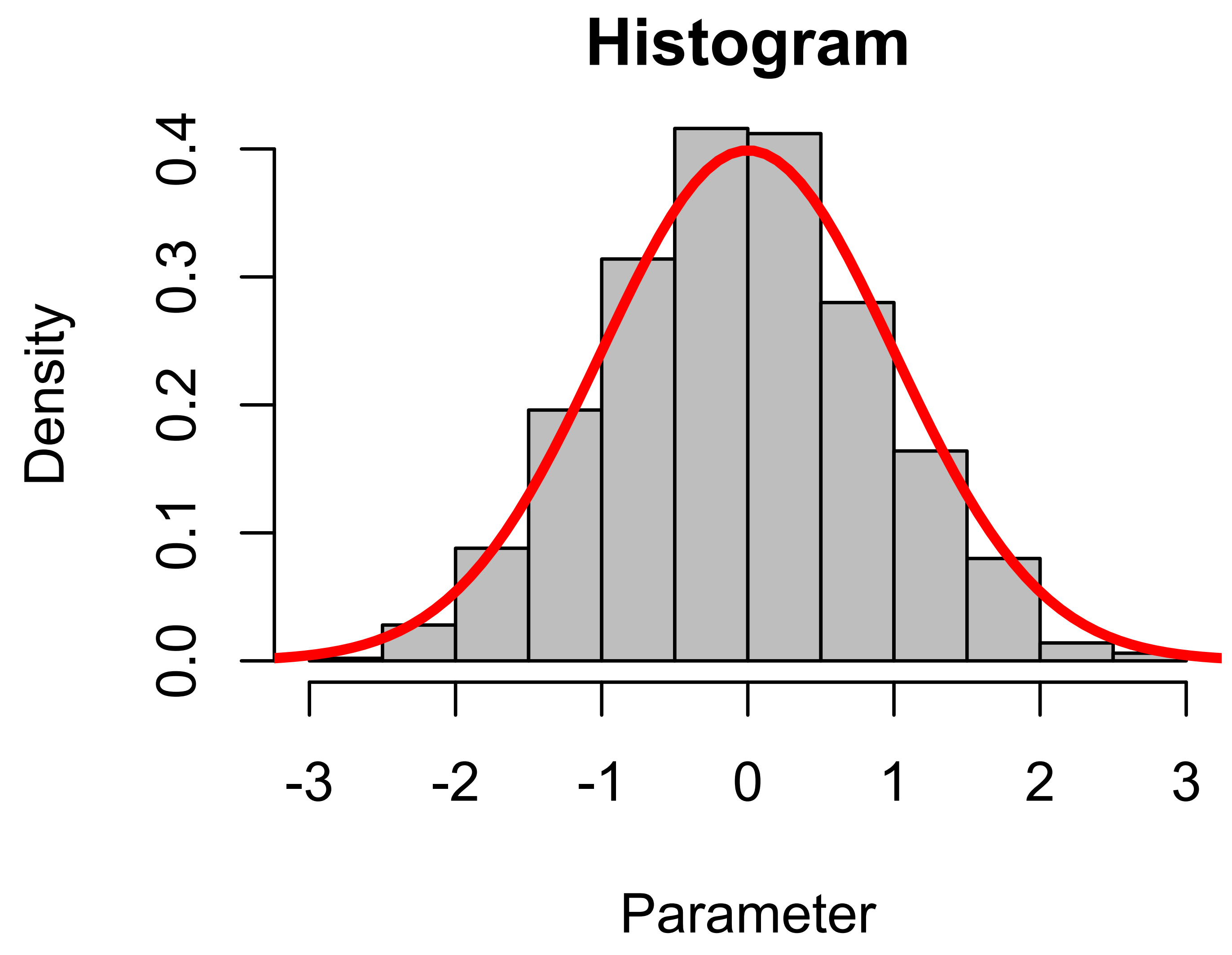
- *However*: These relationships are just to connect to some familiar concepts.
- In UQ, we usually are not interested in point estimates.
  - (and if we do make a point estimate, it's usually the posterior mean, not MAP)
- Our real goal is *uncertianty*, which means the full posterior distribution
  - Its mean, variance, and all higher moments

$$p(\theta \,|\, y) \propto p(y \,|\, \theta)\, p(\theta) = \frac{1}{\left(\prod_i \sqrt{2\pi\sigma_i^2}\right)} \exp\left[ -\frac{1}{2} \frac{\sum_{i=1}^{N} (y_i - m_i(c;\theta))^2}{\sigma_i^2} \right] \times \Pi_{k=1}^{K}\, p(\theta_k)$$

# Markov chain Monte Carlo (MCMC) sampling

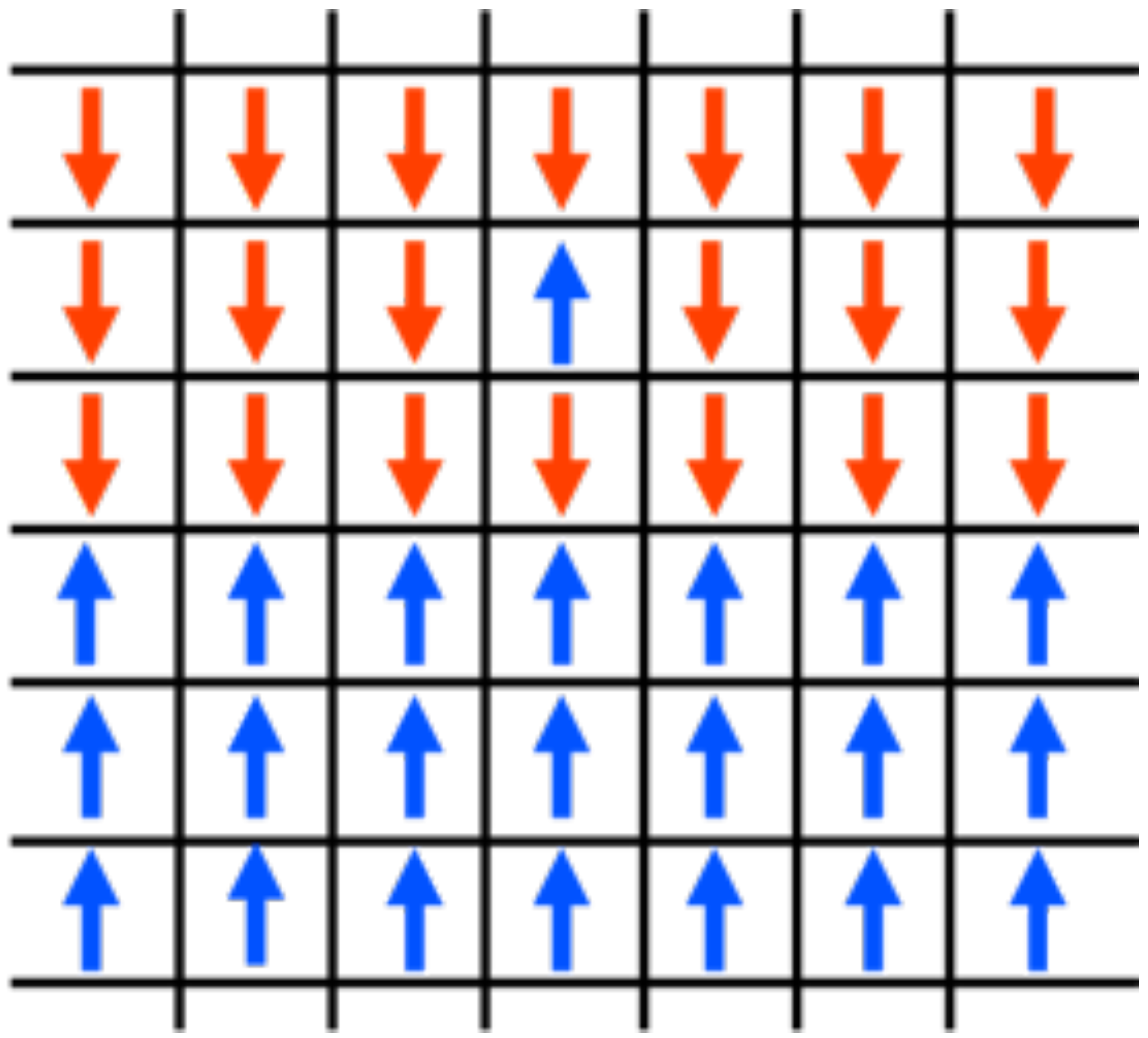
- We want to calculate the posterior distribution. In high dimensions, Monte Carlo sampling works best.
  - sampling converges like $1/\sqrt{N}$, where *N* is # of samples
- How to sample from an arbitrary distribution?
- Approach: importance-biased random walk
  - spend more time sampling high-probability regions
  - (note: samples from a random walk are not independent)

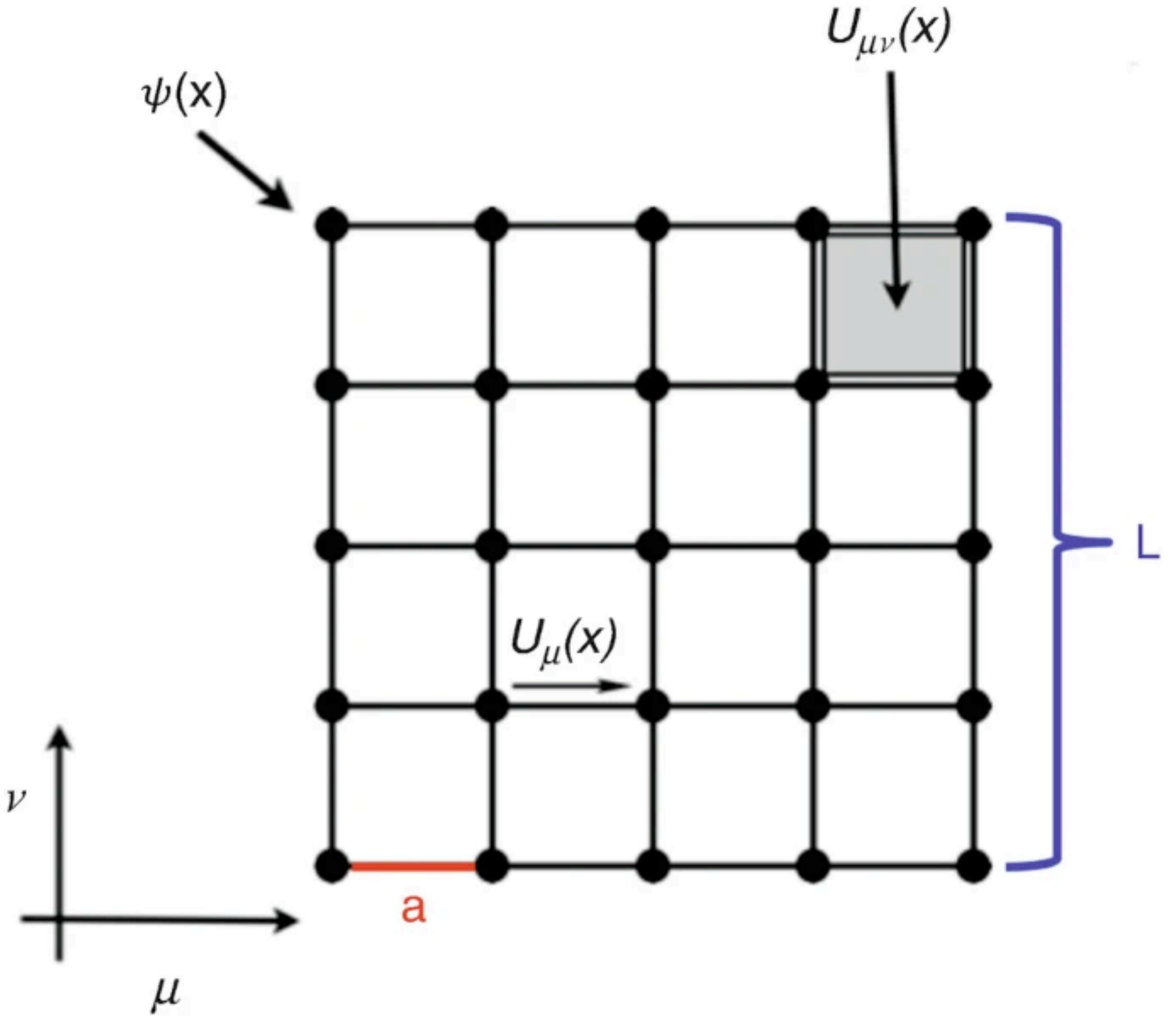


# Physics note: MCMC

- Sampling from a probability distribution $p(x)$ is directly analogous to statistical mechanics
  - Sample Boltzmann distribution $p(x) \propto e^{-\beta E(x)}$
  - -log $p(x)$ is analogous to *potential energy*
- Or lattice gauge theory
  - $p(x) \propto e^{-S[x]}$
  - -log $p(x)$ is analogous to the *action*
- Advanced Bayesian inference uses *hybrid Monte Carlo* (HMC), just like lattice QCD
  - Requires calculating gradient of $p(x)$
  - Which for us means the gradient of the model output (e.g., Bmad beam position) w.r.t. the parameters
  - <u>*Differentiable Bmad* would be very helpful</u>





# Metropolis MCMC algorithm


LANL

- Let the target distribution $\pi(\theta)$ be the posterior, $p(\theta|y)$
- Construct a random walk as follows:

1. Start at point $\theta$
2. Propose moving to a new point $\theta'$ randomly, according to some easy to sample symmetric distribution $t(\theta'|\theta)$ (e.g., a Gaussian perturbation)
3. If this moves us to a <u>higher</u> probability point, $\pi(\theta') > \pi(\theta)$, <u>accept</u> the move to $\theta'$
4. If this moves us to a <u>lower</u> probability point, <u>accept randomly with probability</u> $\pi(\theta')/\pi(\theta)$; else reject and stay at the same point $\theta$
5. Either way, record the point you end up at to construct the Markov chain
6. Repeat

# Code for Bayesian regression

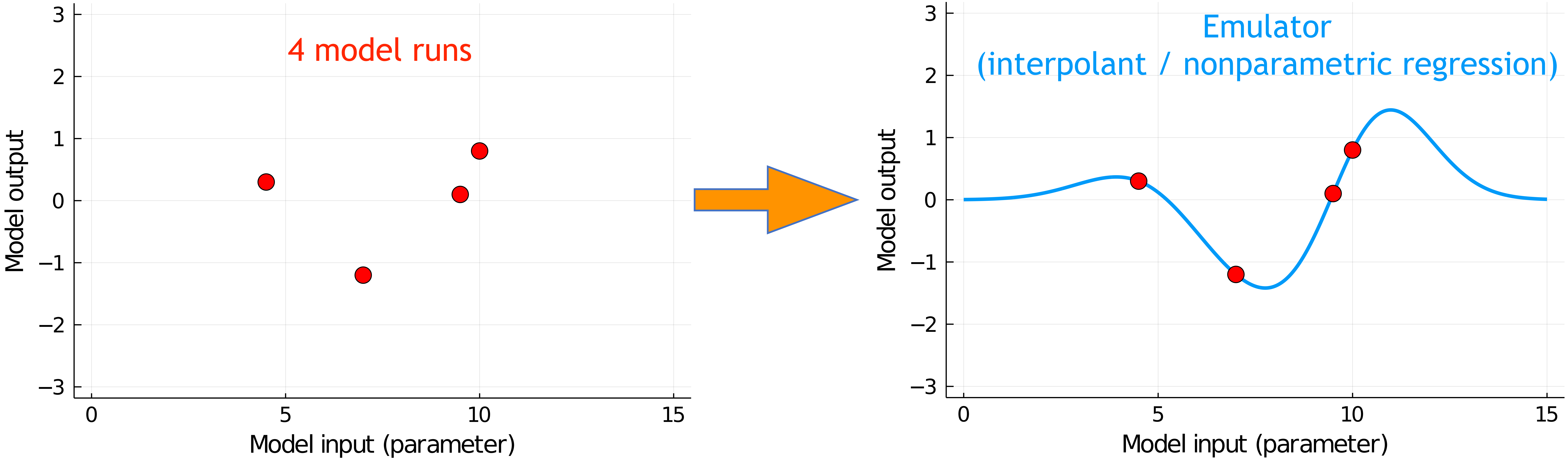
```julia
function metropolis(lpdf, num_iter, x₀, step)
    D = length(x₀)
    chain = zeros(num_iter, D)
    chain[1,:] = x₀
    x, lp = x₀, lpdf(x₀)
    num_accept = 0

    for i = 2:num_iter
        x′ = x + step .* randn(D) # proposal
        lp′ = lpdf(x′)

        if log(rand()) < lp′ - lp # Metropolis
            x, lp = x′, lp′
            num_accept = num_accept + 1
        end

        chain[i,:] = x
    end

    return (chain, num_accept/num_iter)
end
```

```julia
function model(p)
    λ,d,α,T₀ = p
    Δt = 31557600. # year [s]
    C = 4184000 * d # heat capacity/area [J/K/m^2]
    F = forcing_non_aerosol + α*forcing_aerosol
    T = zero(F)
    for i in 1:length(F)-1
        T[i+1] = T[i] + (F[i] - λ*T[i])/C * Δt
    end
    return T .+ T₀
end
```

```julia
function log_posterior(p)
    λ,d,α,T₀ = p
    log_post = -Inf

    if λ > 0 && d > 0 && α > 0 # parameters in range
        F2xCO₂ = 4.0 # forcing for doubled CO₂ [W/m²]
        lpri_λ = logpdf(LogNormal(log(3), log(2)/2), F2xCO₂/λ)
          + log(F2xCO₂/λ^2) # ECS prior + Jacobian (ECS = F2xCO₂/λ)
        lpri_d = logpdf(Normal(100, 25), d)
        lpri_α = logpdf(LogNormal(log(1), log(1.5)/2), α)
        lpri_T₀ = 0
        log_pri = lpri_λ + lpri_d + lpri_α + lpri_T₀ # prior

        σ = 0.1 # observational noise standard deviation [K]
        r = temp_obs - model(p)[midx] # data-model residual
        log_lik = sum(logpdf.(Normal(0,σ), r)) # likelihood
        log_post = log_lik + log_pri # posterior
    end

    return log_post
end
```

# Model emulation

- We can only afford a limited number of Bmad simulations; hard to embed in Monte Carlo sampler where many evaluations are required
- Can we estimate “what the model would have predicted at a new parameter setting” from an ensemble of training simulation output, without actually running the model?
- “Response surface” emulation: *interpolation* to the rescue
  - Gaussian processes (as in Bayesian optimization), neural networks, other regression approaches

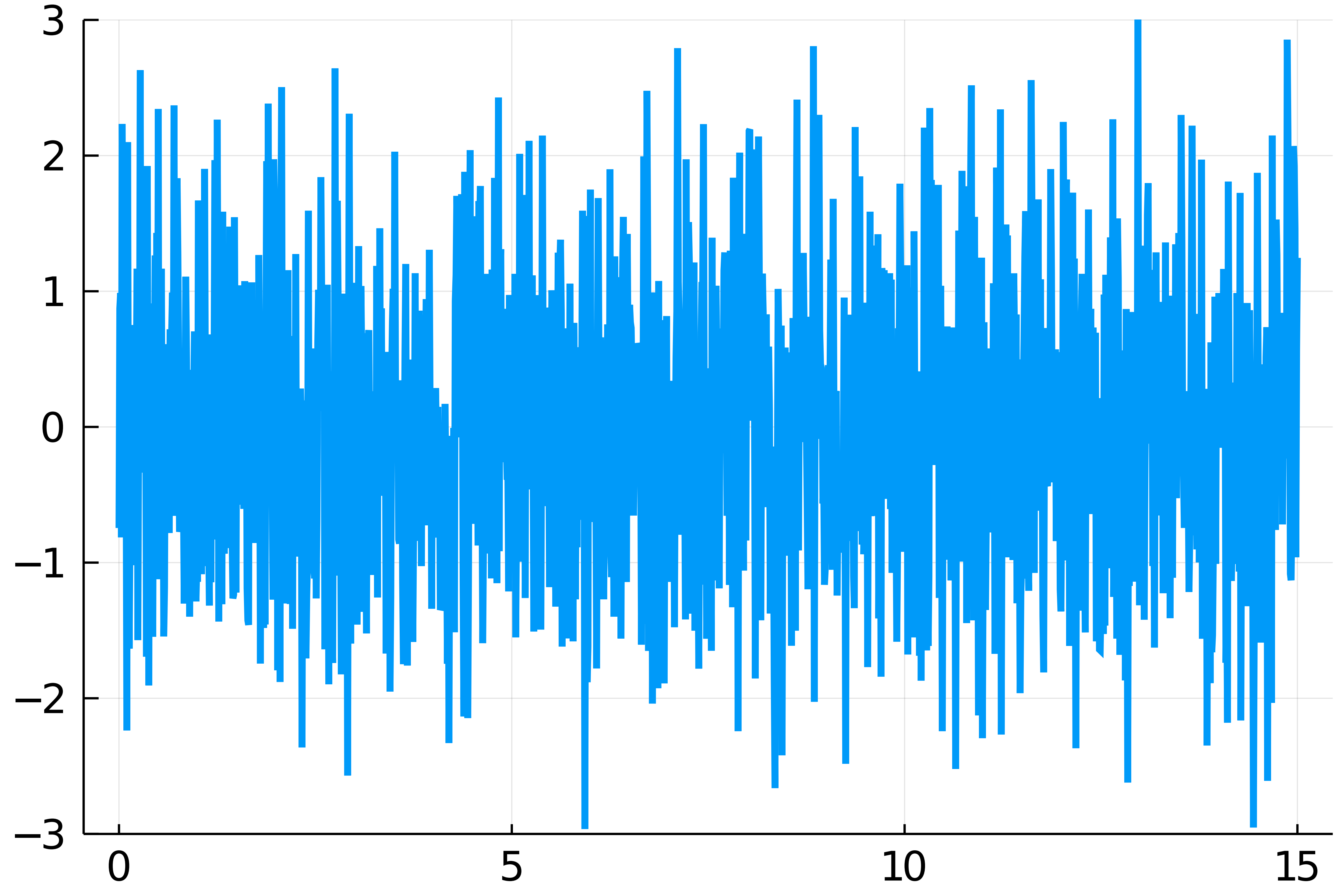


# Gaussian process regression as emulation

- A Gaussian processes is a probability distribution on a space of *functions*
- Can be used for *probabilistic* interpolation / regression

- Draw, say, 1000 Gaussian random samples and plot them over "space":

$Y_i \sim N(0,1)$



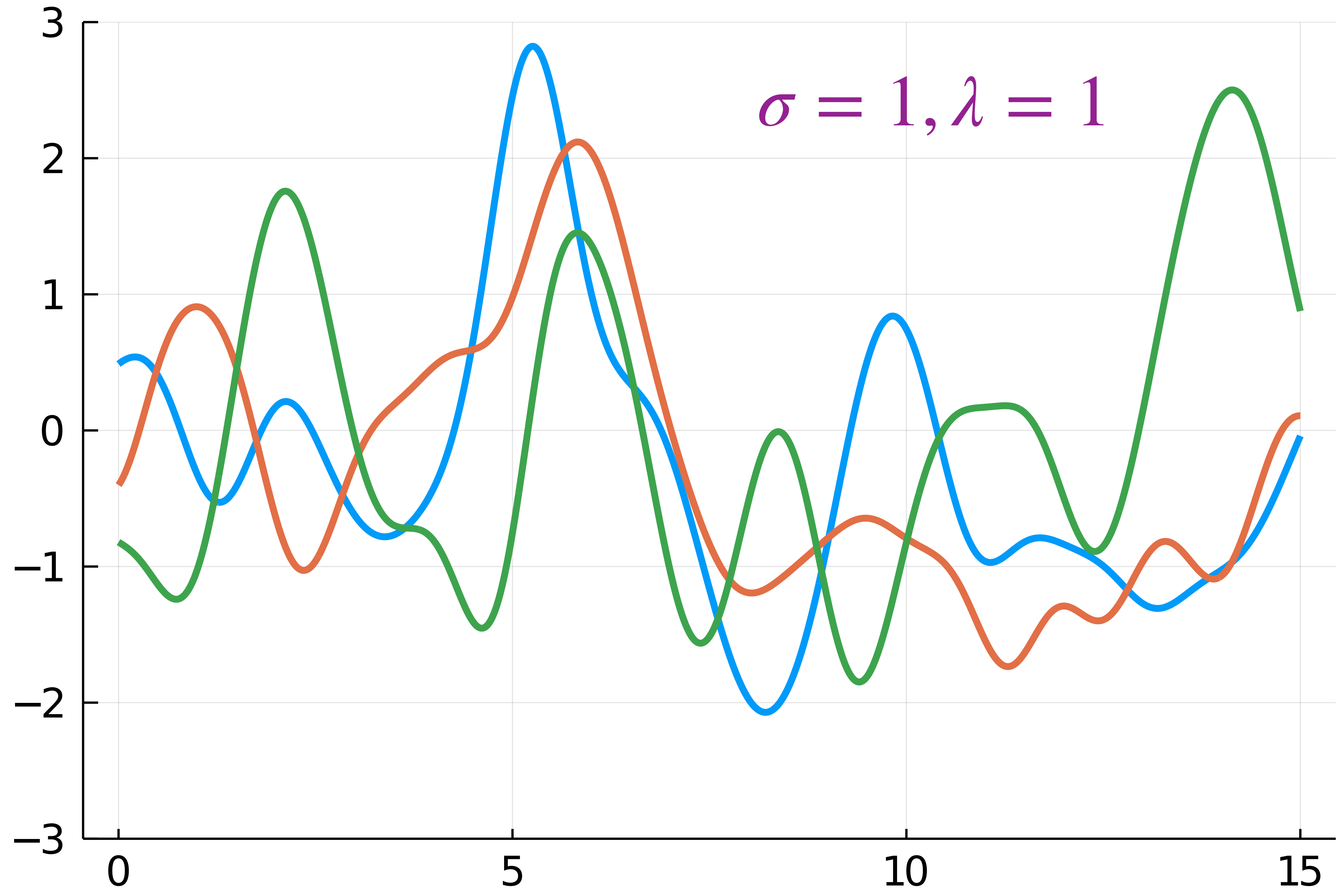
# Gaussian process regression as emulation

- A Gaussian processes is a probability distribution on a space of *functions*
- Can be used for *probabilistic* interpolation / regression

- Draw 1000 random variables, but *correlated with each other*; here are 3 draws:

$$Y \sim N(0,\Sigma), \qquad \Sigma_{ij} = \mathrm{Cov}(Y_i, Y_j)$$

$$\mathrm{Cov}(Y_i, Y_j) = \sigma^2 \exp\left[ -\left( \frac{X_i - X_j}{\lambda} \right)^2 \right]$$



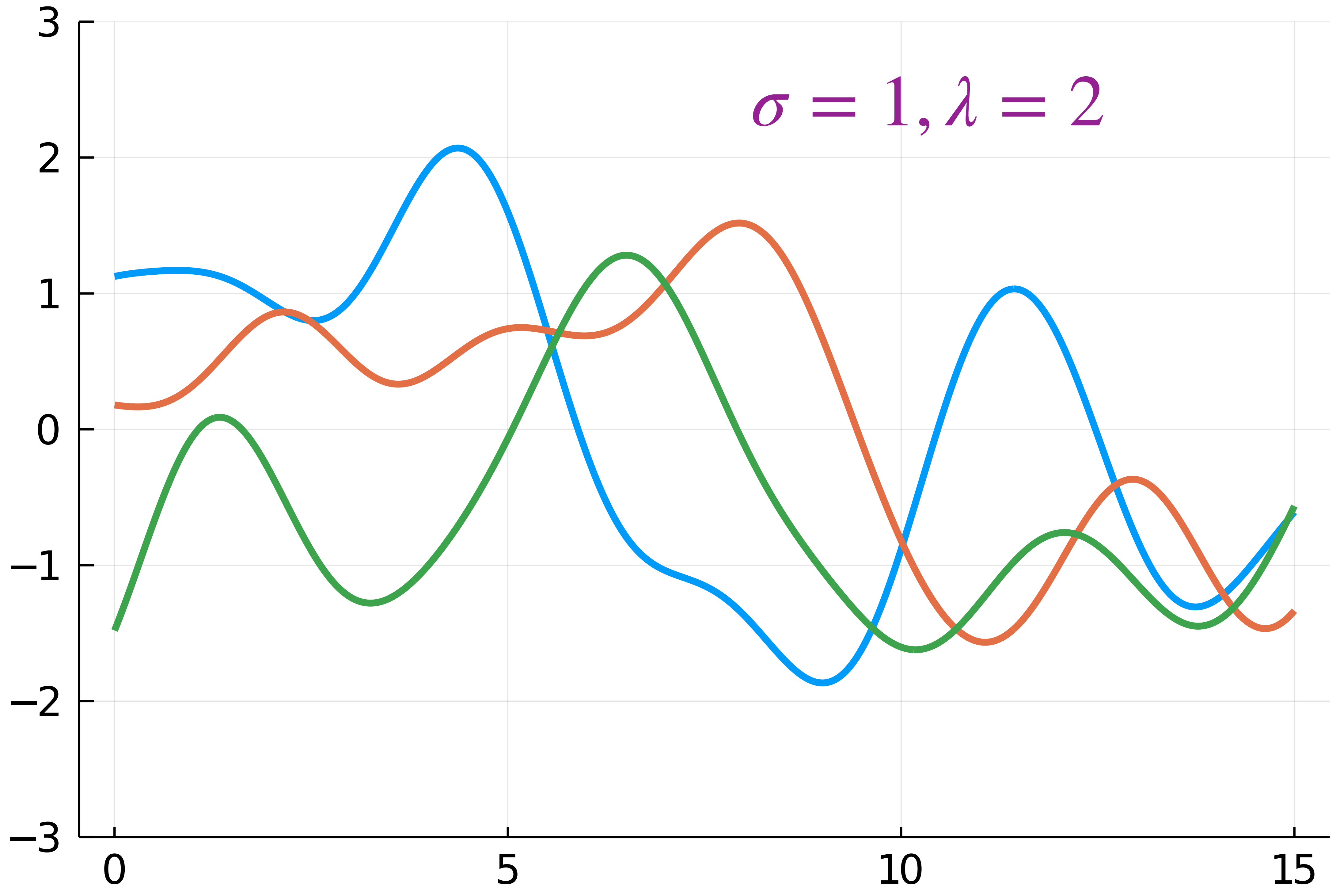
# Gaussian process regression as emulation

- A Gaussian processes is a probability distribution on a space of *functions*
- Can be used for *probabilistic* interpolation / regression

- Draw 1000 random variables, but *correlated with each other*; here are 3 draws:

$$Y \sim N(0, \Sigma), \qquad \Sigma_{ij} = \mathrm{Cov}(Y_i, Y_j)$$

$$\mathrm{Cov}(Y_i, Y_j) = \sigma^2 \exp\left[ -\left( \frac{X_i - X_j}{\lambda} \right)^2 \right]$$



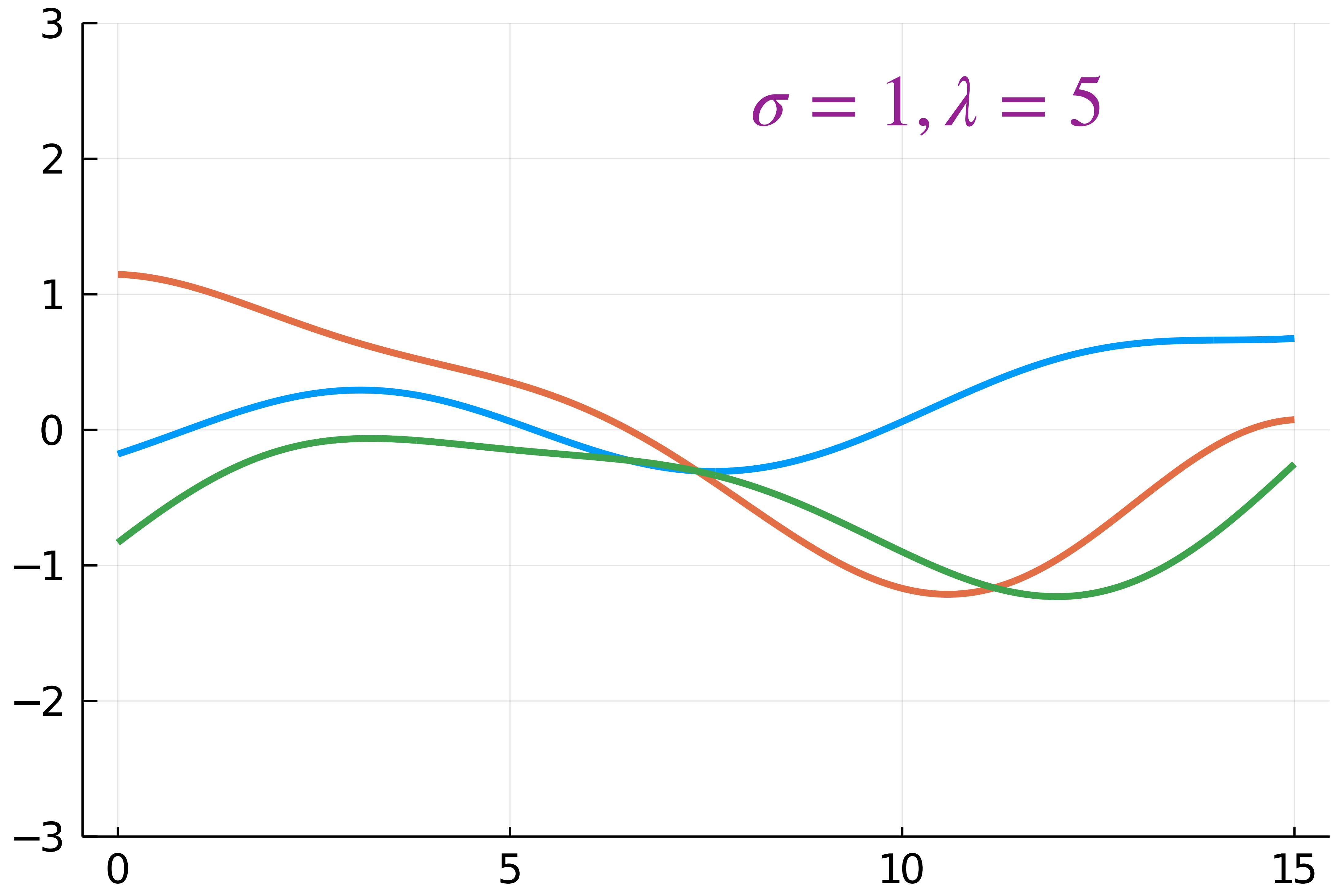
# Gaussian process regression as emulation

- A Gaussian processes is a probability distribution on a space of *functions*
- Can be used for *probabilistic* interpolation / regression

- Draw 1000 random variables, but *correlated with each other*; here are 3 draws:

$$Y \sim N(0,\Sigma), \qquad \Sigma_{ij} = \mathrm{Cov}(Y_i, Y_j)$$

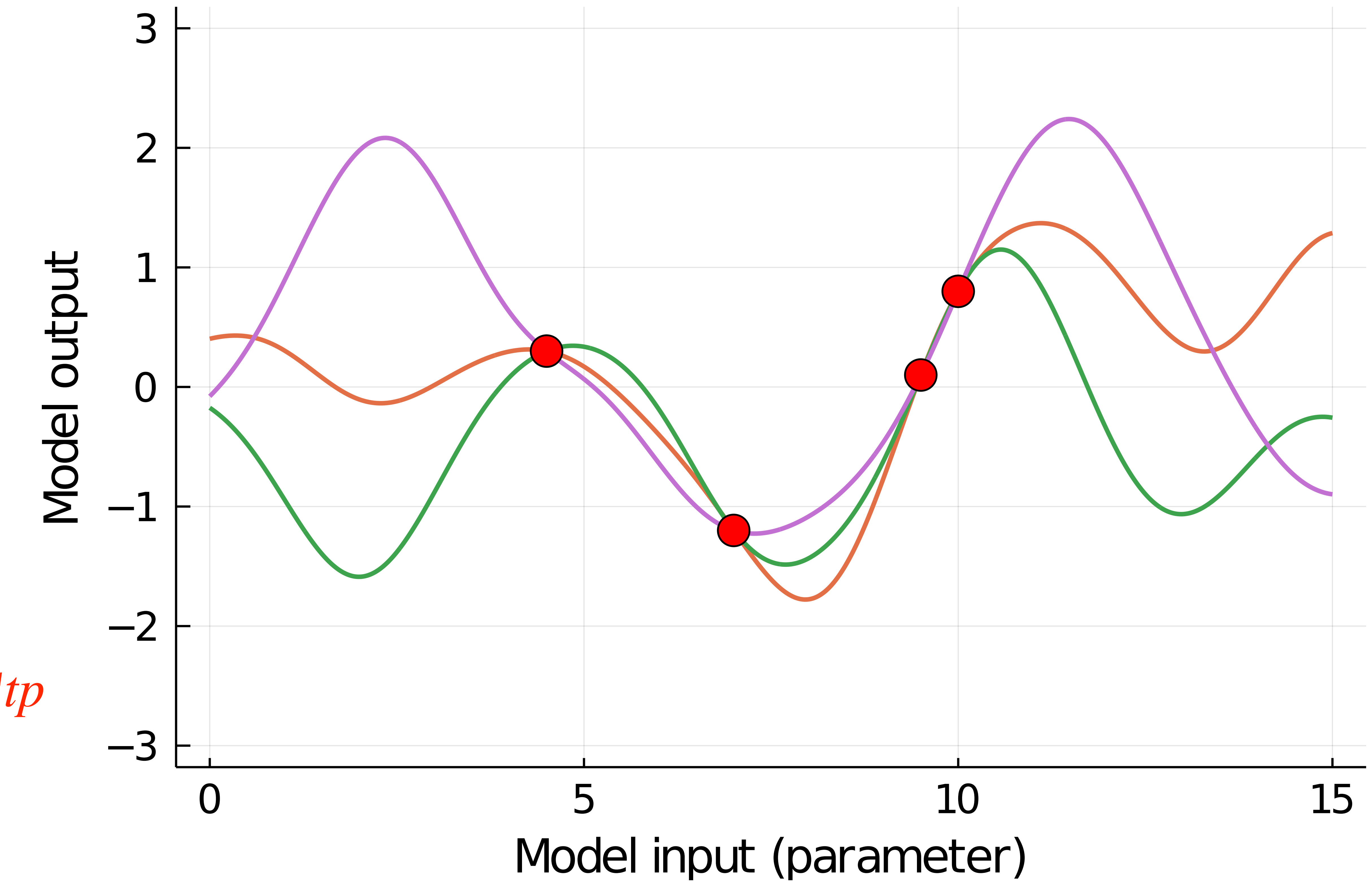
$$\mathrm{Cov}(Y_i, Y_j) = \sigma^2 \exp\left[-\left(\frac{X_i - X_j}{\lambda}\right)^2\right]$$

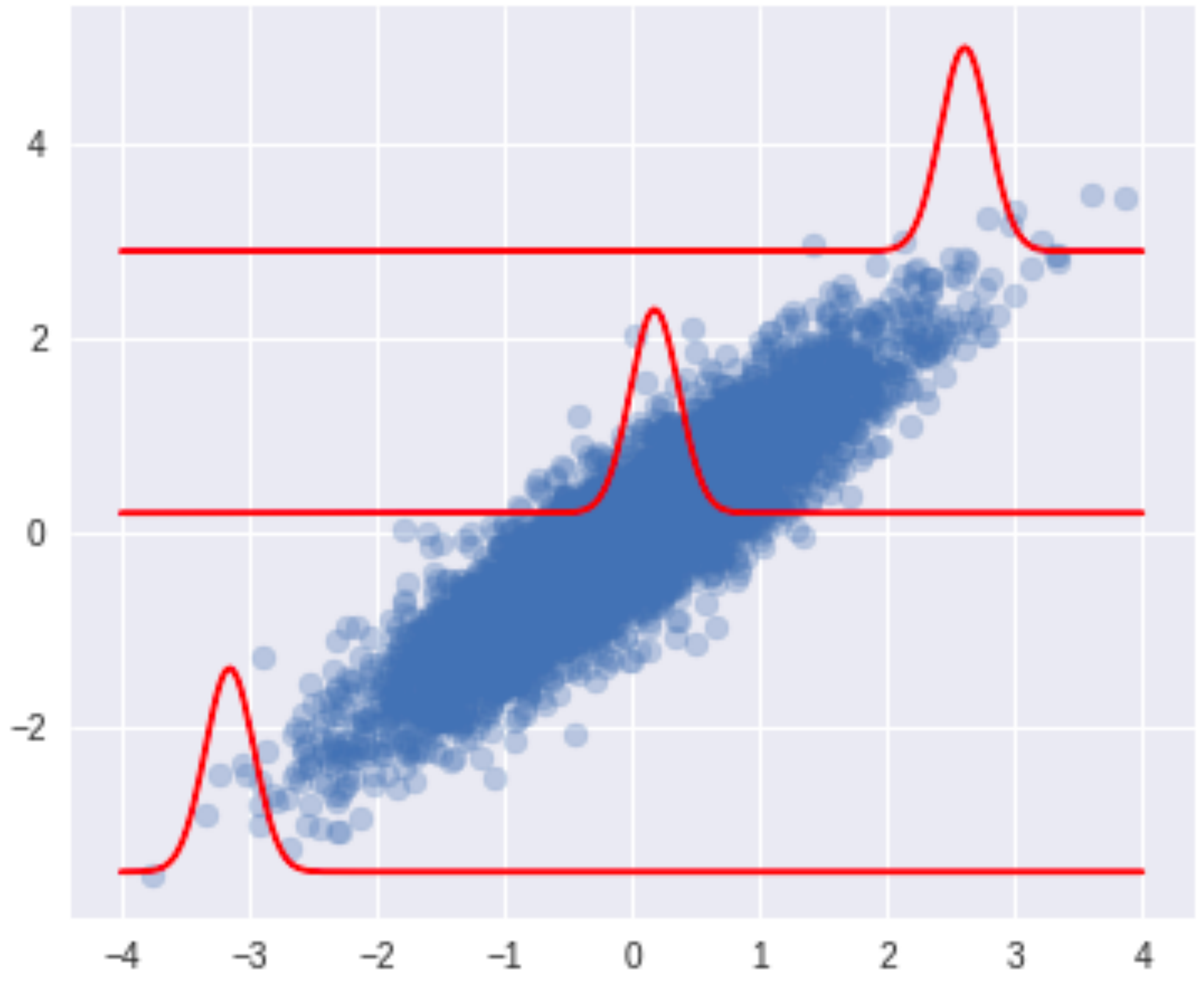


# Gaussian process regression as emulation

- We have seen that we can draw *random vectors* that have smooth behavior by imposing a correlation over space (nearer points are more correlated)
- A Gaussian process is the continuum limit of this idea to *random functions*
- We can be Bayesian, and *condition* on "observed" data to get a *posterior*:

$$Y \sim N(\mu^*, \Sigma^*)$$
$$\mu^* = \Sigma_{pt}\Sigma_{tt}^{-1}y_t$$
$$\Sigma^* = \Sigma_{pp} - \Sigma_{pt}\Sigma_{tt}^{-1}\Sigma_{tp}$$



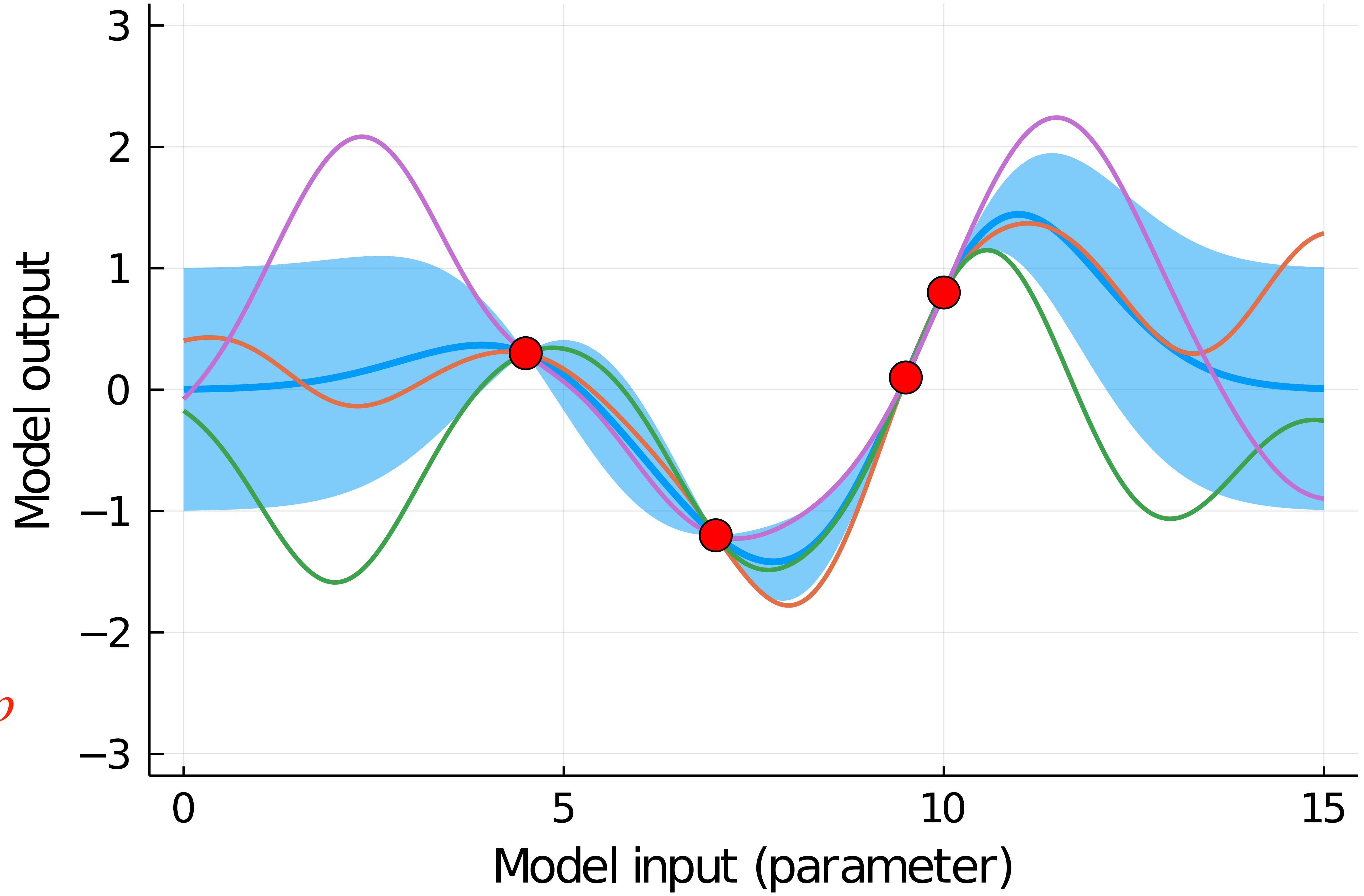



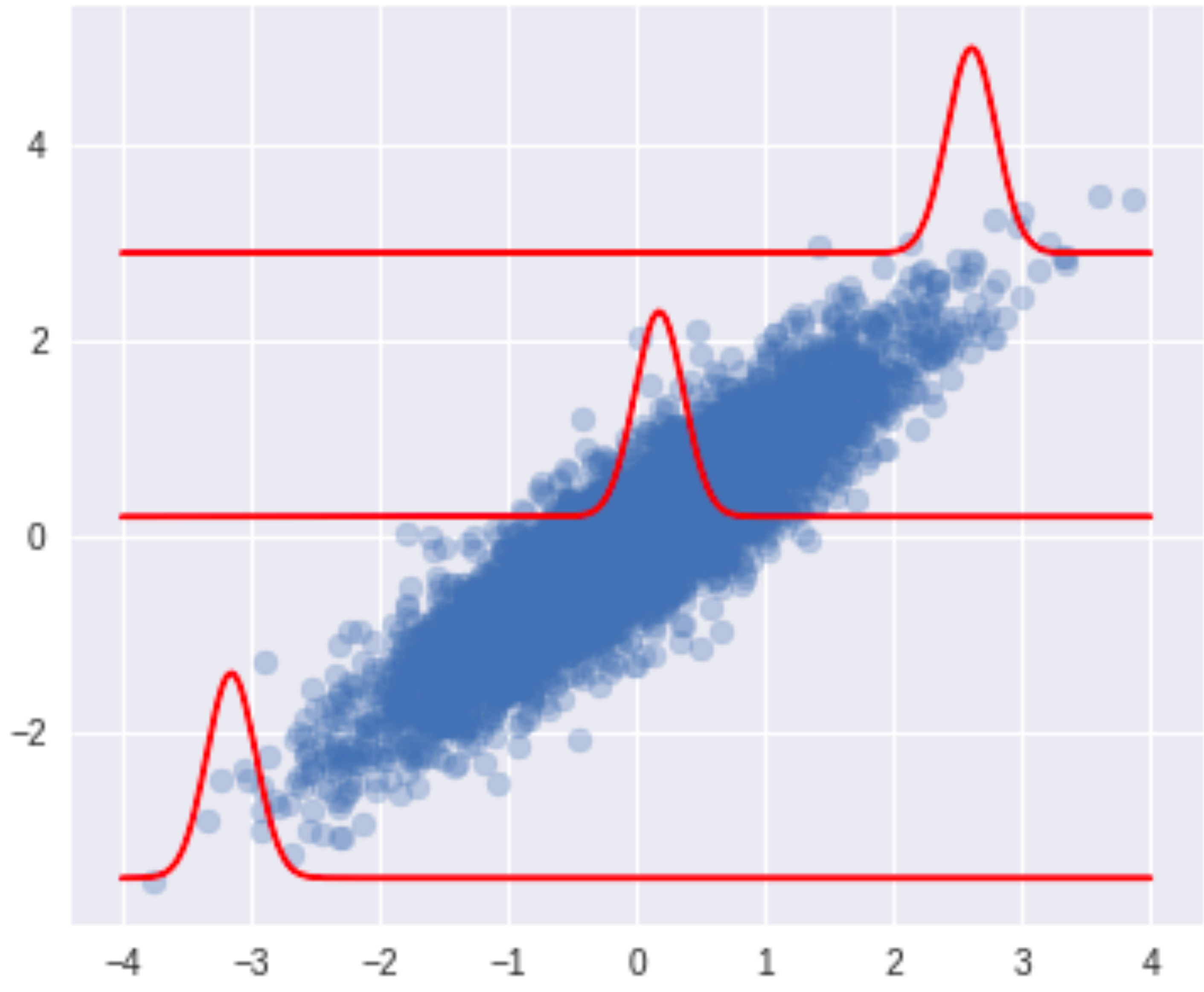

*CrossValidated (2019)*

# Gaussian process regression as emulation

- We have seen that we can draw *random vectors* that have smooth behavior by imposing a correlation over space (nearer points are more correlated)
- A Gaussian process is the continuum limit of this idea to *random functions*
- We can be Bayesian, and *condition* on "observed" data to get a *posterior*:

$$Y \sim N(\mu^*, \Sigma^*)$$
$$\mu^* = \Sigma_{pt}\Sigma_{tt}^{-1}y_t$$
$$\Sigma^* = \Sigma_{pp} - \Sigma_{pt}\Sigma_{tt}^{-1}\Sigma_{tp}$$




*CrossValidated (2019)*

# Errors in variables

- We have assumed that the controls (e.g., currents) are perfectly known, because we set them
- But what if the true control is unknown (currents fluctuate randomly, or there is a persistent but unknown bias between set point and realized current)?
    - The model has noisy inputs in addition to noisy outputs
- We can treat the "true" controls as *parameters* to infer ("latent variables")
    - Probability model for set current as random perturbation of true current: $\tilde{c}_d \sim N(c_d, \varsigma_d^2)$
    - Find joint posterior for parameters and true currents $p(\theta, c \,|\, y, \tilde{c})$

$$p(\theta, c \,|\, y, \tilde{c}) \propto p(y \,|\, \theta)\, p(c \,|\, \tilde{c})\, p(\theta)\, p(c)$$

$$\propto \exp\left[-\frac{1}{2}\frac{\sum_{i=1}^{N}(y_i - m_i(c;\theta))^2}{\sigma_i^2}\right] \times \prod_{k=1}^{K}\frac{(\theta_k - \bar{\theta}_k)^2}{\nu_i^2} \times \prod_{d=1}^{D}\frac{(\tilde{c}_d - c_d)^2}{\varsigma_d^2}$$

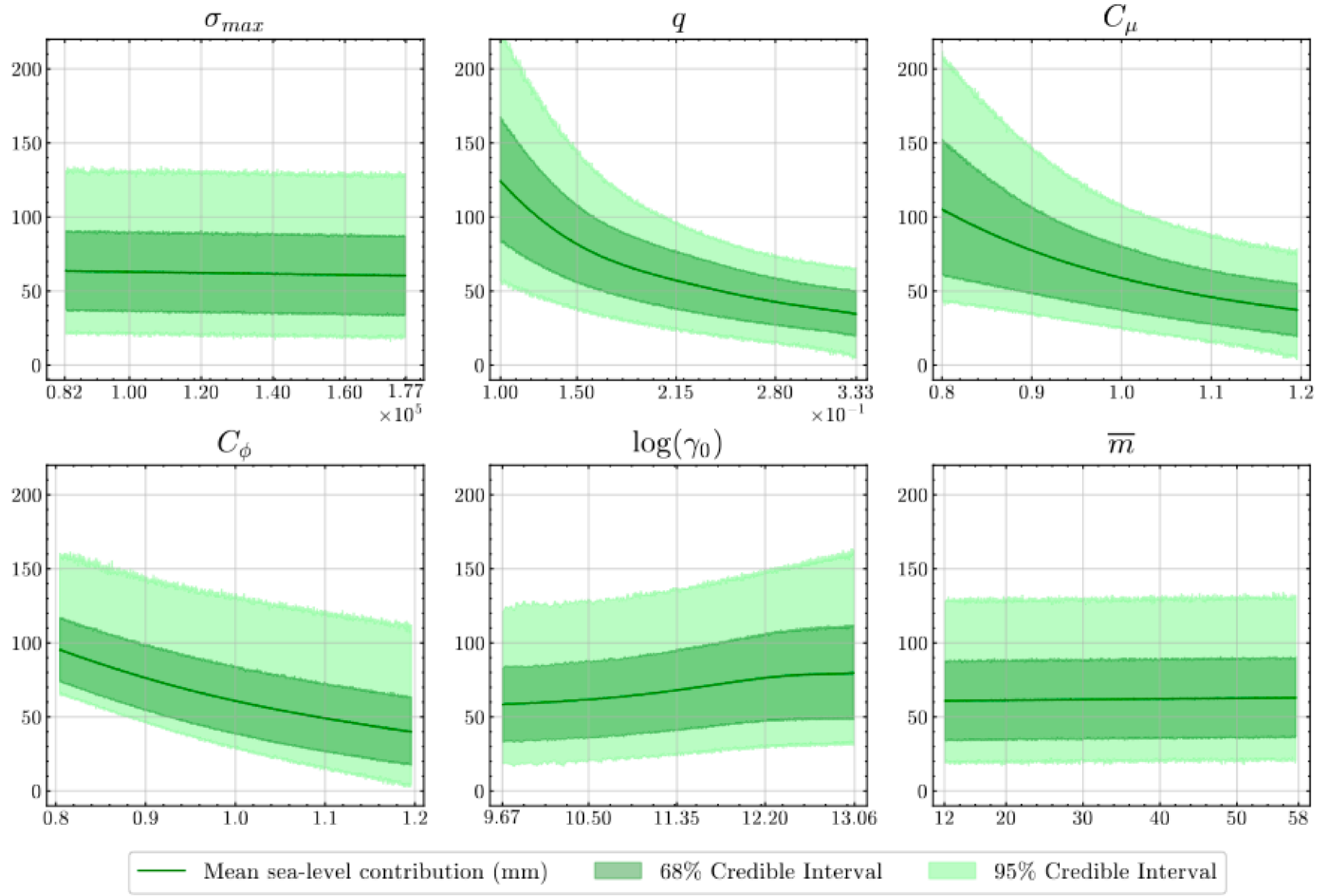
Obtain parameter posterior by integrating out ("marginalizing over") latent variables: $p(\theta \,|\, y, \tilde{c}) = \int p(\theta, c \,|\, y, \tilde{c})\, dc$

# Do any of these uncertainties matter?

- So far we've been proceeding under the assumption that we know which parameters are responsible for beam positioning, or Bmad model misfit
  - We just have to quantify their effects
- What if we don't know what matters?
  - Magnet misalignments, transfer function, trim currents
- Can we go through a list of suspects, and identify or quantify their importance?
  - In terms of influence on model prediction, or data-model misfit
- Characterizing the response of outputs to inputs is known as **sensitivity analysis**
- Traditional approach: "one-at-a-time" (OAT) parameter scan
  - Pick a parameter, change its value over a range (fixing all other parameters at nominal)
  - Doesn't pick up any interactions between parameters
  - Can be sample-inefficient (most of the time you aren't learning about most parameters)
  - Be aware of overconfidence: exploring parameters and stopping when one shows an effect

# Accounting for uncertainty in sensitivity analysis

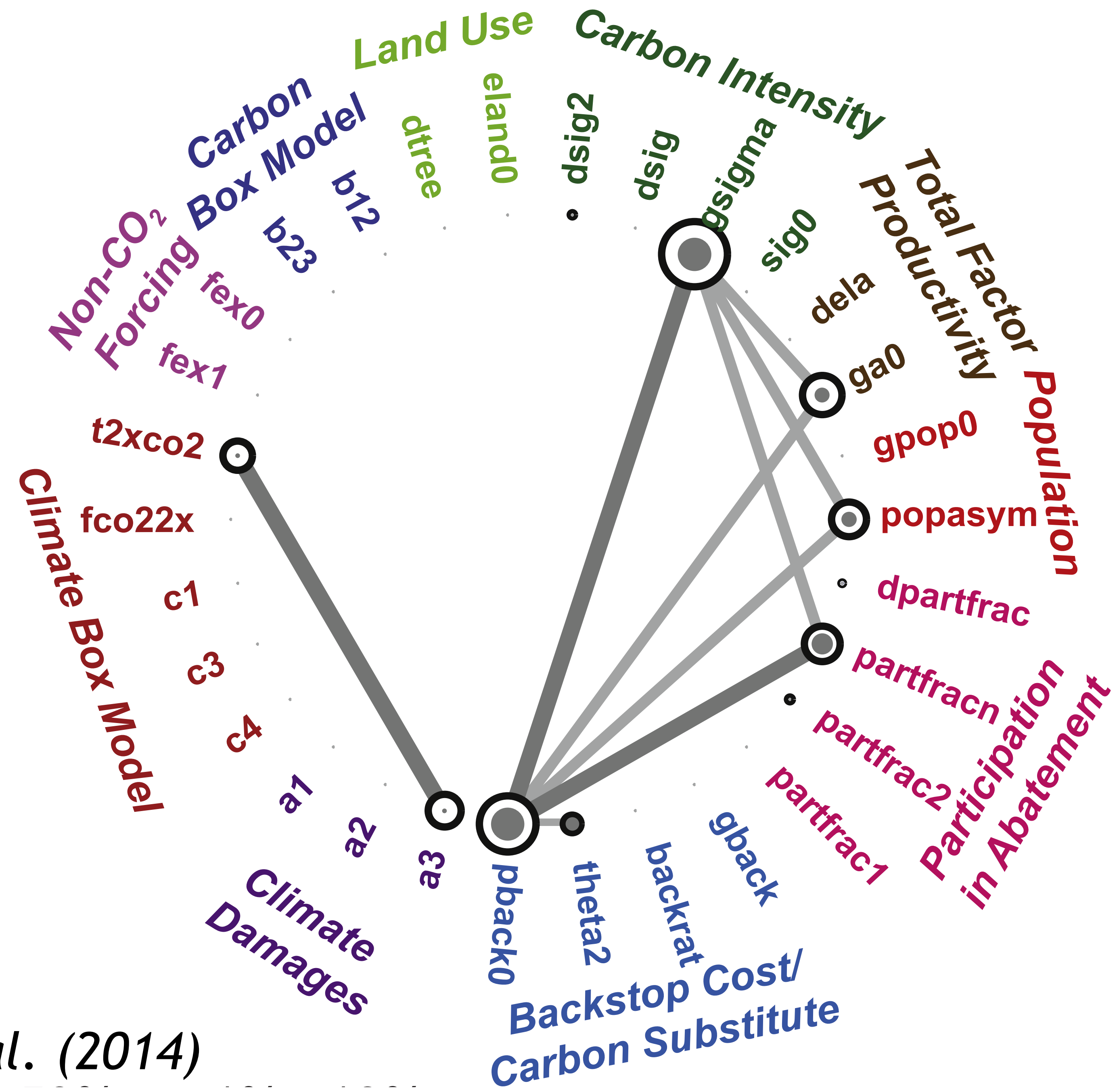
- OAT: change one parameter, holding all others fixed
- Alternative: change one parameter, *sampling randomly* over all other parameters (given a distribution)
  - Accounts for uncertainty in the response of one parameter, due to variability in other parameters




*Jantre et al. (2024)*

# Variance-based global sensitivity analysis (GSA)

- Sobol' decomposition: **Analysis-of-variance (ANOVA) to construct a model's "uncertainty budget"**
  - Requires user to specify a probability distribution over uncertain inputs
- **How much of the output uncertainty can be attributed to the uncertainty in a particular input?**
  - Or, how much could we reduce output uncertainty if we learned the true value of an input?
- **How much does an input contribute directly, and indirectly through correlations with other inputs?**
  - Quantifies importance of (2-way, 3-way, …) interactions between input variables
- **Contrast with "one-at-a-time" parameter scans**
  - Don't identify contributions to output uncertainty, or detect interactions
- **Specific advantages when GSA is coupled with an emulator:**
  - Fast, closed-form analytic solutions for sensitivity metrics
  - Change assumptions about input uncertainties without new simulations




*Butler et al. (2014)*

# Global sensitivity analysis, quantitatively

- How much would we reduce uncertainty in output $Y$, if we learned the value of the $i$<u>th</u> input, $X_i$?
  - Difficulty: we don't know the true value of $X_i$
- Uncertainty in output due to uncertainty in all inputs = $\mathbf{Var}(Y)$
- Uncertainty in output, after learning the true value $x$ of input $X_i$ = $\mathbf{Var}_{\sim i}(Y|X_i=x)$
- Expected output uncertainty after learning true input, averaged over input uncertainty = $\mathbf{E}_i(\mathbf{Var}_{\sim i}(Y|X_i))$
- Expected reduction in uncertainty after learning input $i$ = $\mathbf{Var}(Y) - \mathbf{E}_i(\mathbf{Var}_{\sim i}(Y|X_i))$
  - Also equal to $\mathbf{Var}_i(\mathbf{E}_{\sim i}(Y|X_i))$, via law of total variance
- Normalizing by the output variance gives the **first-order sensitivity index**, $S_i = \mathbf{Var}_i(\mathbf{E}_{\sim i}(Y|X_i)) / \mathbf{Var}(Y)$
- Nested expectations calculated by sampling, or (sometimes) analytically with an emulator of $Y(X)$

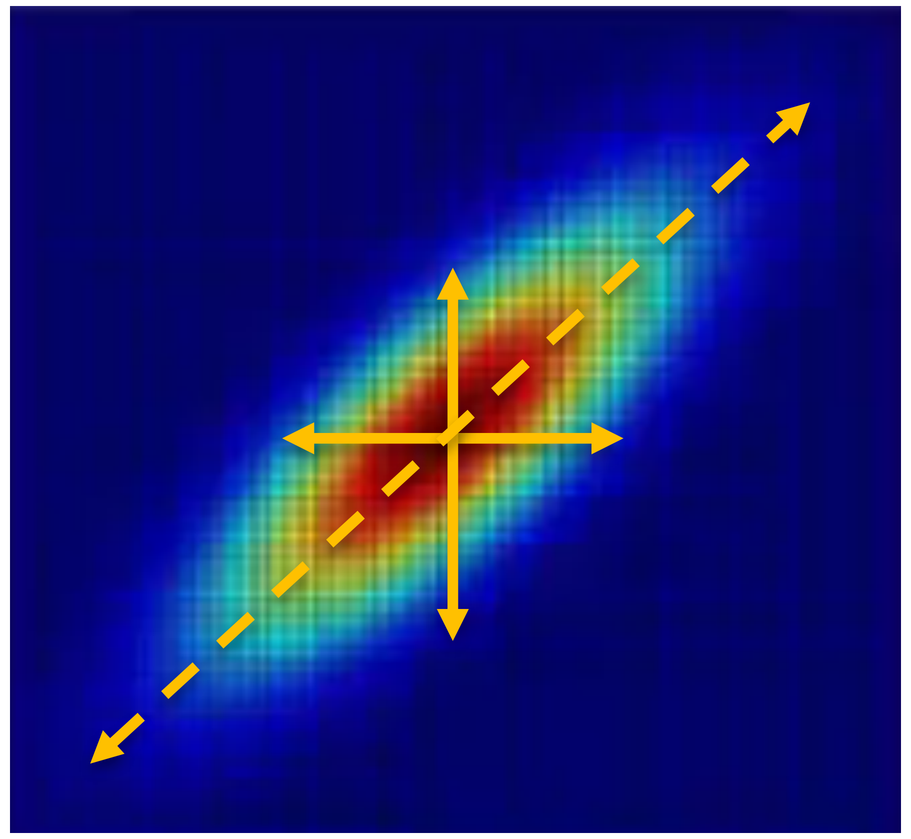
- We can define similar indices for *interactions* between pairs of variables, $S_{ij}$
- The sum of first-order and interaction sensitivities is the **total sensitivity index**, $T_i = \mathbf{E}_{\sim i}(\mathbf{Var}_i(Y|X_{\sim i})) / \mathbf{Var}(Y)$
- A large first-order sensitivity means it would be valuable to reduce uncertainty in that variable
- A small total sensitivity means that variable's uncertainty is negligible (it does not influence output uncertainty either directly, or indirectly through its interactions with other variables)

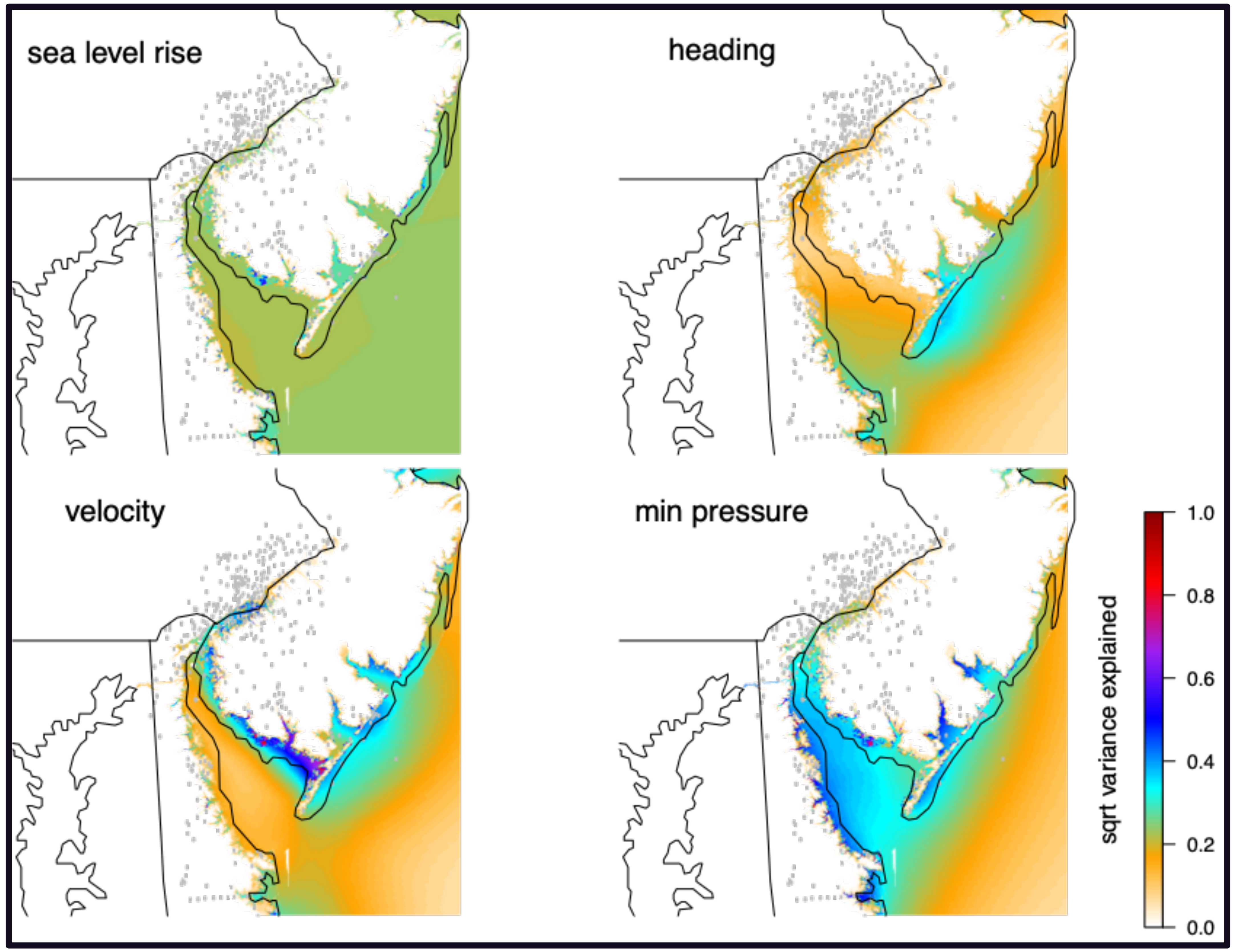
# Code for global sensitivity analysis

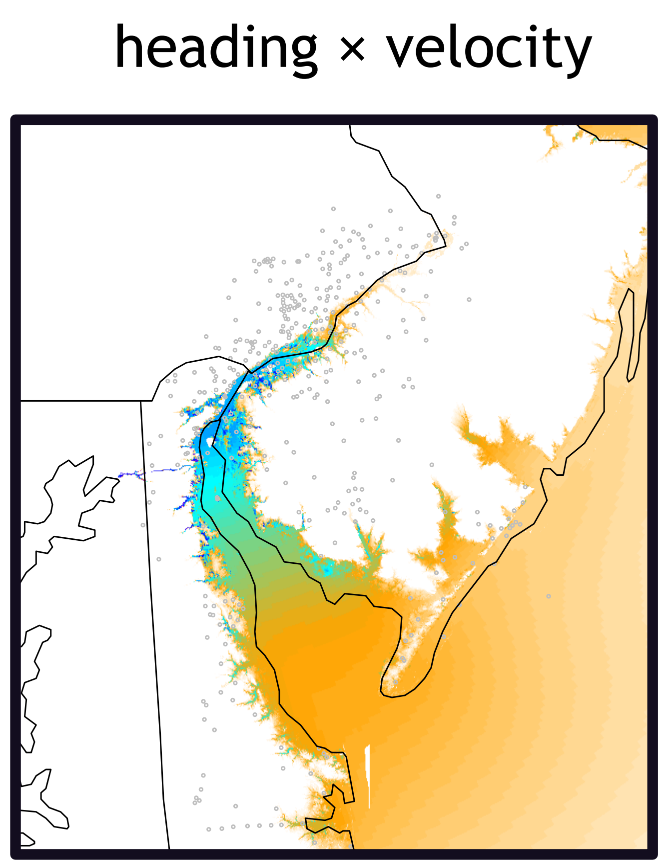
```julia
# conditional draw on xᵢ
randᵢ(d, i, xᵢ) = [j==i ? xᵢ : rand(d[j]) for j=1:length(d)]
# conditional draw on x₋ᵢ
rand!ᵢ(d, i, x!ᵢ) = [j==i ? rand(d[i]) : x!ᵢ[j] for j=1:length(d)]

# Sobol' first-order sensitivity index
S(m, d, i, N) = var(mean(m(randᵢ(d,i,xᵢ)) for k=1:N) for xᵢ in rand(d[i],N))
              / var(m(rand.(d)) for j=1:N^2)

# Sobol' total sensitivity index
T(m, d, i, N) = mean(var(m(rand!ᵢ(d,i,x!ᵢ)) for k=1:N) for x!ᵢ in (randᵢ(d,i,NaN) for j=1:N))
              / var(m(rand.(d)) for j=1:N^2)
```

# Global sensitivity analysis example

- Sensitivity of flooding to sea level rise and hurricane direction, speed, and intensity
- This does not mean these two inputs are correlated with each other (though they can be)
- Rather, nonlinear variations in the output may occur when two variables change together
- These effects would be invisible if the inputs were varied one-at-a-time






*Francom et al.*
*LA-UR-19-27244*

# Optimizing control inputs

- **Control** *c*: currents or other inputs that the operator can specify
- **Model** *m*(*c*): the modeled system response to inputs (e.g., beam position)
- **Objective**: a metric of system performance (e.g., a loss function) to optimize
  - $\mathcal{L}(m(c)) = \sum_i (\bar{z}_i - m_i(c))^2$ (deviation of beam position from target position at BPMs)
  - (e.g., $\bar{z}_i = 0$)
- Find control that optimizes objective:

$$c^\star = \text{arg min}_c\, \mathcal{L}(m(c))$$

- Solve using standard optimization algorithms (quasi-Newton, gradient descent, …)

# On optimal control methods

- There are many optimization methods floating around
  - Bayesian optimization, gradient descent, quasi-Newton methods, …
- There are many ways to formulate beam control as an optimization problem
  - Nonlinear loss minimization, expected utility maximization (with chance constraints), robust optimization/control, classical control theory, reinforcement learning
- Probably a digression to discuss pros/cons in this talk, but we should discuss in the project
- The methods discussed here are adapted for this setting:
  - There is a physical system model, which is much cheaper than real experiments
  - We can solve control policies offline using the physical model (digital twin)
  - The model is imperfect, but imperfections are learnable via data-model comparisons
  - There are many variables to control; maybe many uncertain system parameters
  - Decisions are one-off / non-sequential (if sequential, can extend to RL-like approaches)

# What next?

- We need to identify controls (and their ranges) that matter to the beam position
  - More expert elicitation, sensitivity analysis / parameter screening, …
- Perform UQ
  - Are results Gaussian? Correlated? May inform approximations we make in the future
- Stochastic optimization
  - Minimize expected loss via BFGS, gradient descent, BO, …
- Optimal experimental design
- How important are Bmad structural errors (biases, missing physics, …?)
  - Keep adding things to Bmad? Some other approach
- Sequential / realtime decision making?
  - Amortized myopic optimization (precompute policy: optimal solution conditional on state)
  - Reinforcement learning (accounting for future decisions in present actions)
    - RL with UQ: all state variables become *belief states* (infinite-dimensional distributions)