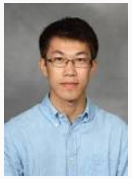
Structured Numerical Linear Algebra for Gaussian Process Modeling

David Bindel

20 May 2021

Collaborators

Ian Delbridge
(Klaviyo)



Kun Dong
(Facebook)

David Eriksson
(FB Research)

Jake Gardner
(U Penn)

Eric Lee
(SIGOPT)

Hannes Nickisch
(Phillips Research)

Geoff Pleiss
(Columbia)

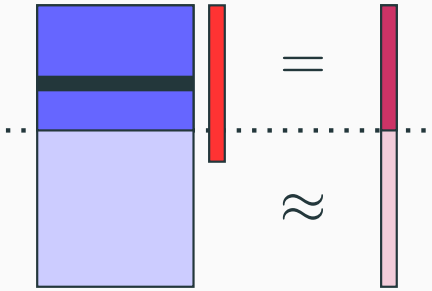
Kilian Weinberger
(Cornell)

Andrew Wilson
(NYU)

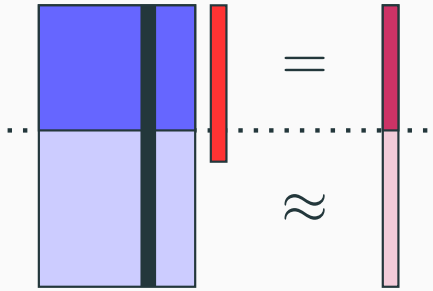
Kernel and GP basics

Kernel-Based Regression: Four Stories

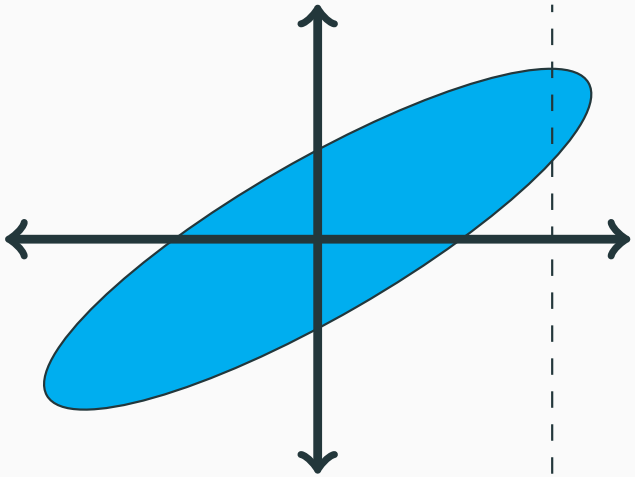
Feature map



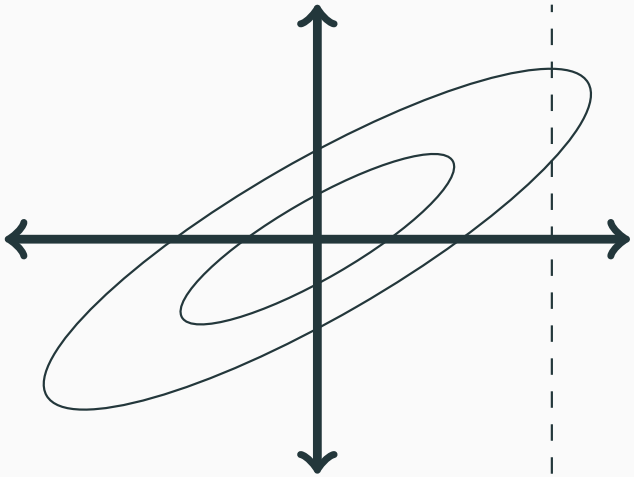
Data-dependent basis



Energy minimization



Gaussian process



The Big Picture

Gaussian processes (GPs) are

- Key building block for ML and spatio-temporal statistics
- Tightly connected to integral equations, kernel regression
- Straightforward to reason about (just linear algebra)
- But hard to scale to big data (because of dense LA)

Goal today: Make these methods scale!

Basic ingredient: Gaussian Processes (GPs)

Our favorite continuous distributions over

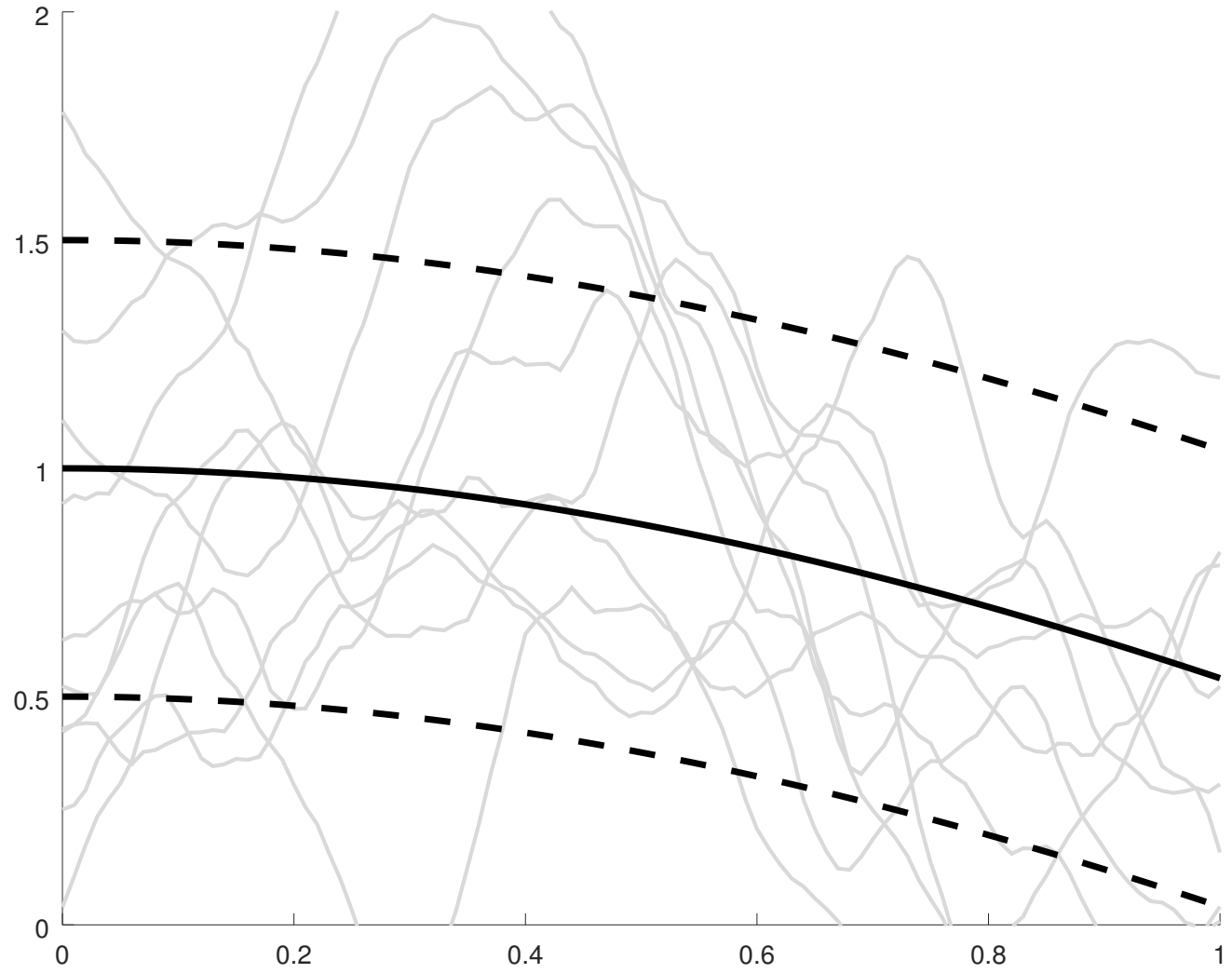
$$\begin{array}{lll} \mathbb{R}: & \text{Normal}(\mu, \sigma^2), & \mu, \sigma^2 \in \mathbb{R} \\ \mathbb{R}^n: & \text{Normal}(\mu, C), & \mu \in \mathbb{R}^n, C \in \mathbb{R}^{n \times n} \\ \mathbb{R}^d \to \mathbb{R}: & \text{GP}(\mu, k), & \mu : \mathbb{R}^d \to \mathbb{R},\ k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R} \end{array}$$

More technically, define GPs by looking at finite sets of points:

$$\begin{aligned} \forall X = (x_1, \ldots, x_n),\ x_i \in \mathbb{R}^d, & \\ \text{have } f_X \sim N(\mu_X, K_{XX}), & \text{ where} \\ f_X \in \mathbb{R}^n, \quad & (f_X)_i \equiv f(x_i) \\ \mu_X \in \mathbb{R}^n, \quad & (\mu_X)_i \equiv \mu(x_i) \\ K_{XX} \in \mathbb{R}^{n \times n}, \quad & (K_{XX})_{ij} \equiv k(x_i, x_j) \end{aligned}$$

When X is unambiguous, we will sometimes just write K.

Basic ingredient: Gaussian Processes (GPs)



Basic ingredient: Kernel functions

Call the *kernel* (or *covariance*) function k. Required (today):

- **Pos def**: K_{XX} is always positive definite

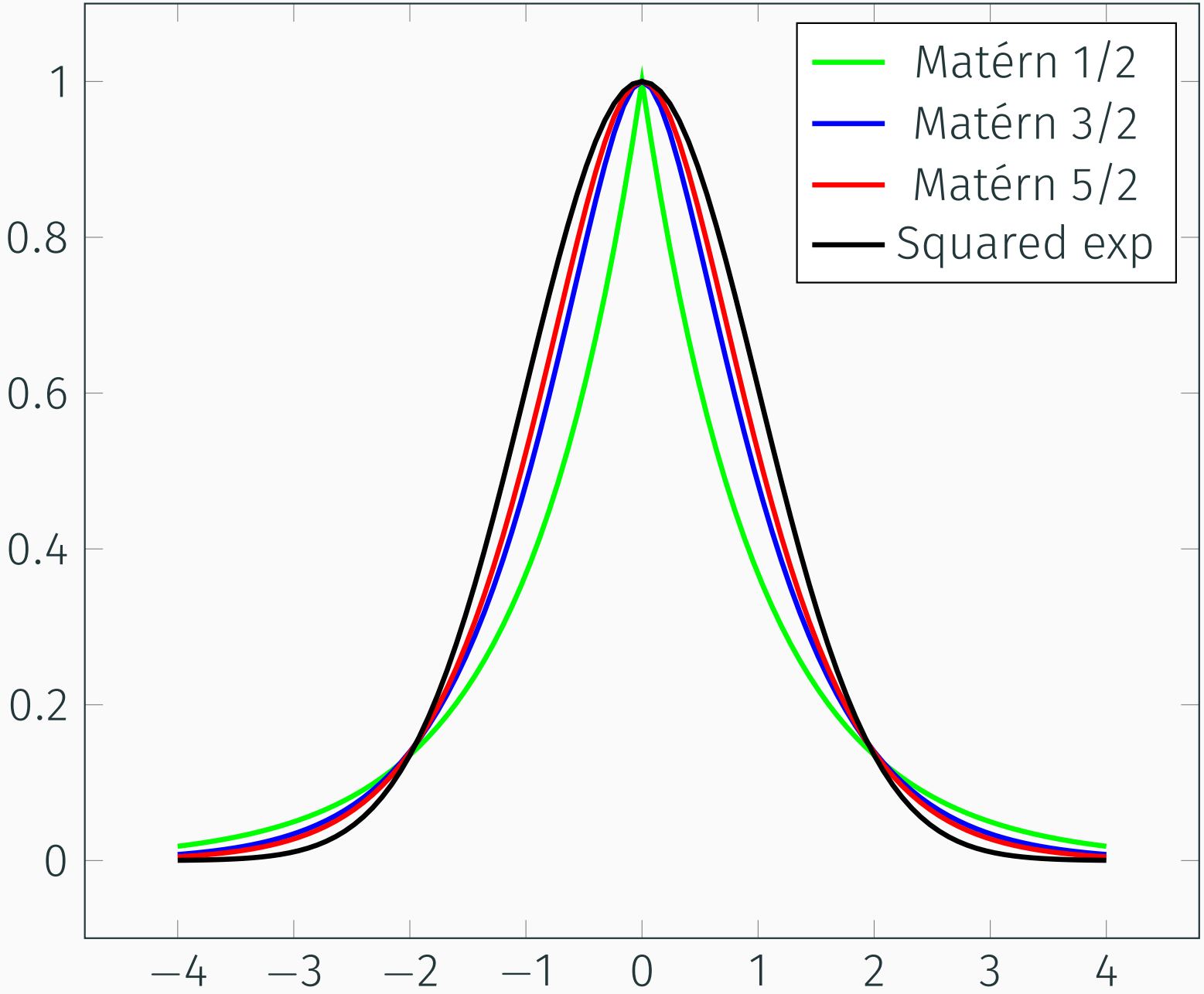
Often desirable:

- **Stationary**: $k(x, y)$ depends only on $x - y$
- **Isotropic**: $k(x, y)$ depends only on x and $\|x - y\|$

Often want both (sloppy notation: $k = k(r)$).

Common examples (e.g. Matérn, SE) also depend on *hyper-parameters* θ — suppressed in notation unless needed.

Matérn and SE kernels



Observations on kernel matrices

Kernel is *chosen by modeler*

- Matérn / SE for regularity and simplicity
- Many more exotic options (deep kernel learning, spectral mixture kernels, etc)
- Rarely have the intuition to pick the "right" kernel
- Common choices are *universal* — can recover anything
    - ... with less data for "good" choice (inductive bias)

Properties of kernel matrices:

- Positive definite by design, but not well conditioned!
- Weyl: $k(r) \in C^\nu \implies |\lambda_n| = o(n^{-\nu-1/2})$
- SE case: eigenvalues decay (super)exponentially
- Adding $\sigma^2 I$ "wipes out" small eigenvalues

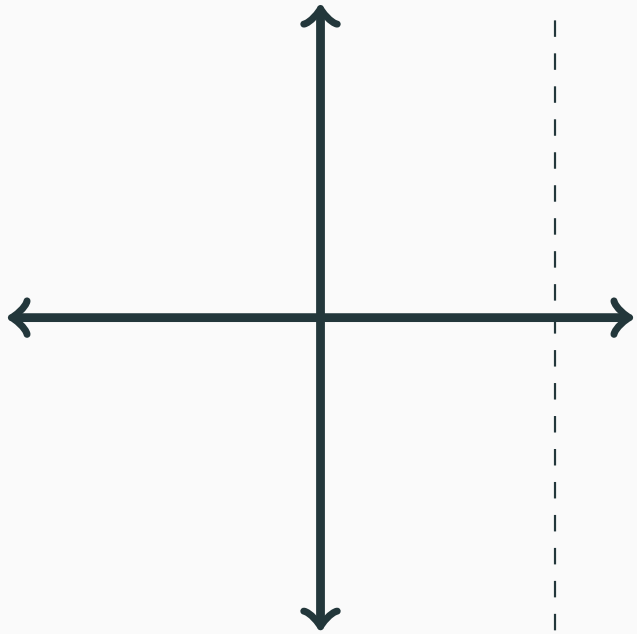
Fun with kernel matrices

Want solves, Schur complements, determinants, etc with kernel matrices. Usually, dense *but*:

- Small data (a couple 1000) – use standard direct methods
- Iterative methods are great for bigger problems
- Smooth kernels give nearly low-rank matrices
- Low-dimensional problems give rank-structured matrices
- Modelers can choose kernels for convenience

Learning parameters and hyperparameters

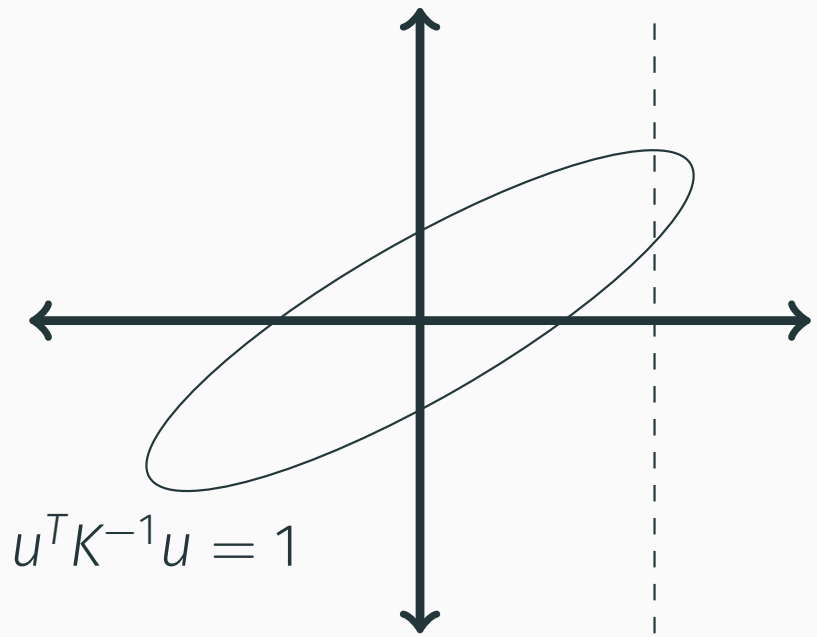
Simple and impossible



Let $u = (u_1, u_2)$. Given u_1, what is u_2?

We need an assumption!

Being Bayesian



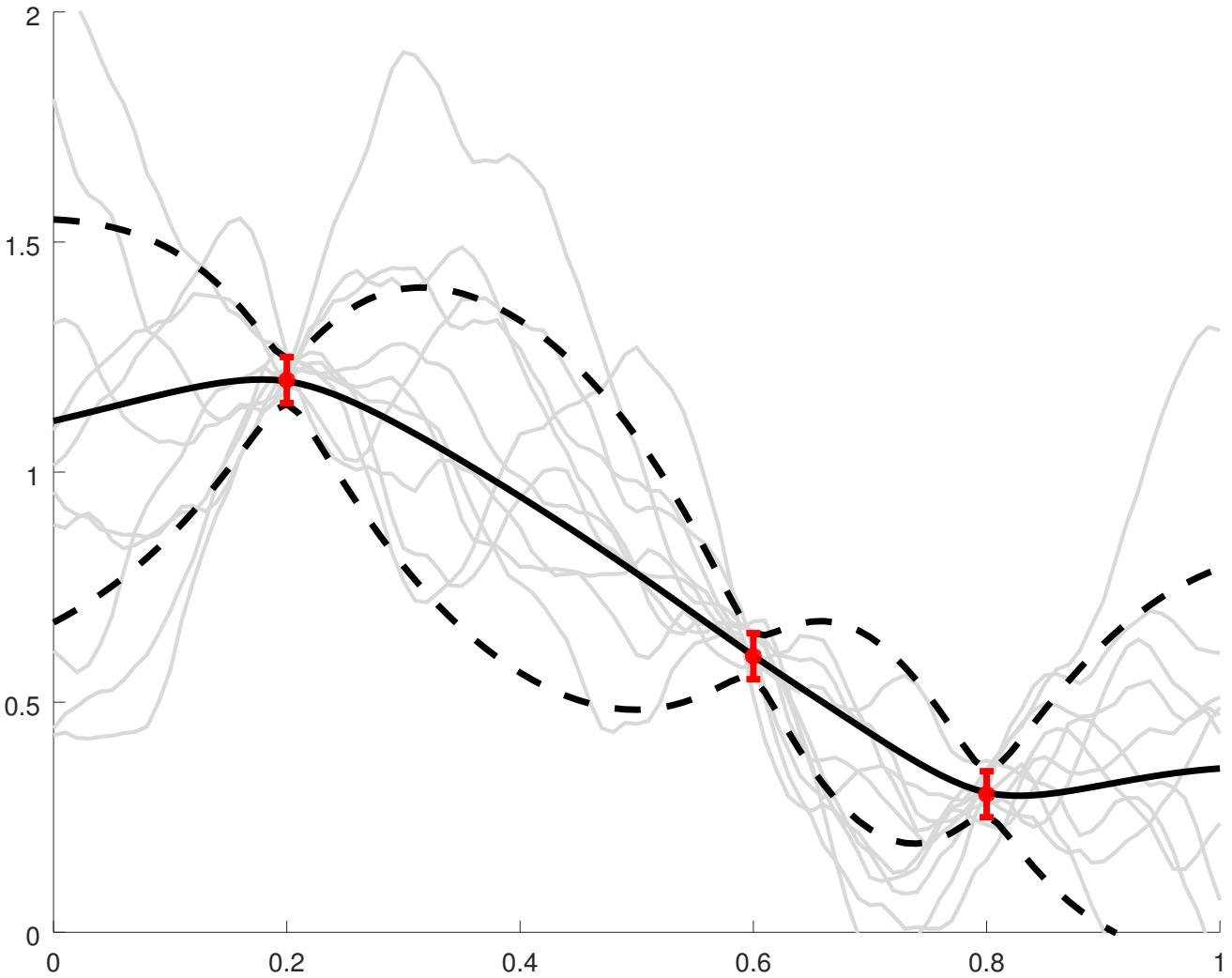
Let $U = (U_1, U_2) \sim N(0, K)$. Given $U_1 = u_1$, what is U_2?

Posterior distribution: $(U_2 | U_1 = u_1) \sim N(w, S)$ where

$$w = K_{21} K_{11}^{-1} u_1$$

$$S = K_{22} - K_{21} K_{11}^{-1} K_{12}$$

Being Bayesian



Being Bayesian

Now consider prior of $f \sim \mathrm{GP}(\mu, k)$, noisy measurements

$$f_X \sim y + \epsilon, \quad \epsilon \sim N(0, W), \qquad \text{typically } W = \sigma^2 I$$

Posterior is $f \sim \mathrm{GP}(\mu', k')$ with

$$\mu'(x) = \mu(x) + K_{xX} c \qquad \tilde{K} = K_{XX} + W$$
$$k'(x, x') = K_{xx'} - K_{xX} \tilde{K}^{-1} K_{Xx'} \qquad c = \tilde{K}^{-1}(y - \mu_X)$$

The expensive bit: solves with $\tilde{K}$.

NB: Other (linear) measurements also allowed –
e.g. derivatives.

Kernel hyper-parameters

How to estimate *hyper-parameters* θ?

- Bayesian approach? Expensive...
- Usually just do maximum likelihood estimation (MLE)

Likelihood function is same as for a multivariate normal:

$$\ell(\theta|y) = \frac{1}{\sqrt{\det(2\pi\tilde{K})}} \exp\left(-\frac{1}{2}(y-\mu_X)^T\tilde{K}^{-1}(y-\mu_X)\right).$$

Of course, we usually work with *log-likelihood* and derivatives.

Kernel hyper-parameters

How to estimate *hyper-parameters* θ?

- Bayesian approach? Expensive...
- Usually just do maximum likelihood estimation (MLE)

Log-likelihood function for kernel hypers θ

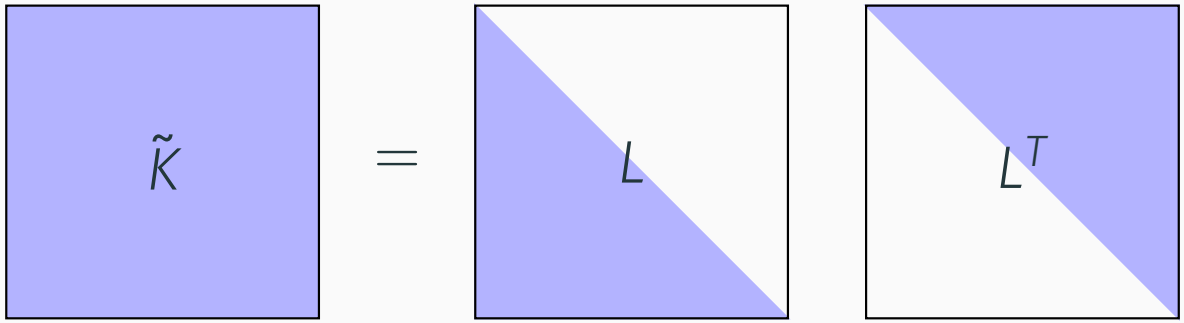
$$\mathcal{L}(\theta|y) = \mathcal{L}_y + \mathcal{L}_{|K|} - \frac{n}{2}\log(2\pi)$$

where (again with $c = \tilde{K}^{-1}(y - \mu_X)$)

$$\mathcal{L}_y = -\frac{1}{2}(y - \mu_X)^T c, \qquad \frac{\partial \mathcal{L}_y}{\partial \theta_i} = \frac{1}{2} c^T \left(\frac{\partial \tilde{K}}{\partial \theta_i}\right) c$$

$$\mathcal{L}_{|K|} = -\frac{1}{2}\log\det \tilde{K}, \qquad \frac{\partial \mathcal{L}_{|K|}}{\partial \theta_i} = -\frac{1}{2}\,\mathrm{tr}\left(\tilde{K}^{-1}\frac{\partial \tilde{K}}{\partial \theta_i}\right)$$

Learning parameters and hypers: small n



In an optimization loop:

```
1  L = chol(Ktilde(theta));
2  c = L'\(L\(y-mu));
3  liky  = -0.5*((y-mu)'*c);
4  likK  = -sum(log(diag(L)));
5  for i = 1:len(theta)
6     dliky = 0.5*(c'*dK(theta,i)*c);
7     dlikK = -0.5*trace(L'\(L\dk(theta,i)));
8  end
```

Scalability bottlenecks

Consider n data points

- Straightforward regression: factor $\tilde{K}$ at $O(n^3)$ cost
- Kernel hyper MLE requires multiple $O(n^3)$ ops
  - To compute $\log\det \tilde{K}$ is $O(n^3)$ per step
  - To compute $\mathrm{tr}\left(\tilde{K}^{-1}\frac{\partial \tilde{K}}{\partial \theta_i}\right)$ is $O(n^3)$ per hyper per step

Two possible work-arounds

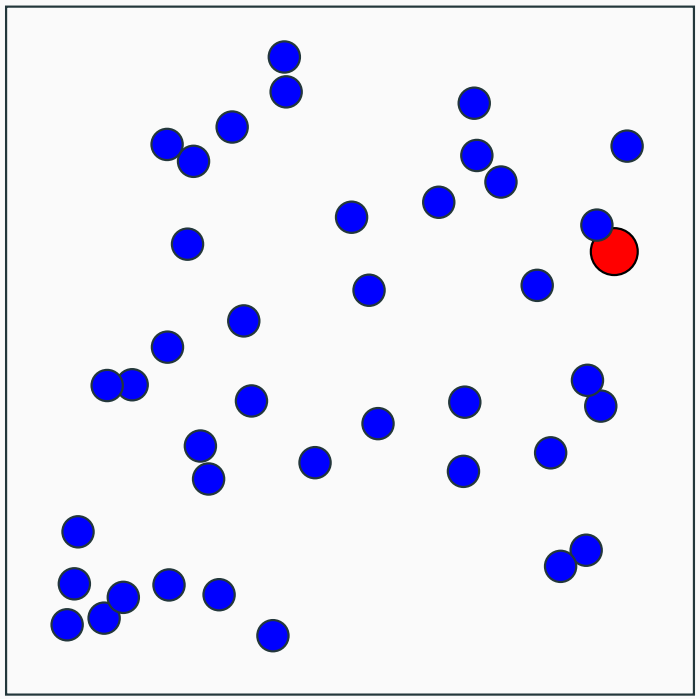
- Data-sparse factorization methods
- Methods that avoid factorization (e.g. iterative solvers)
  - Q: how to handle determinants and traces?

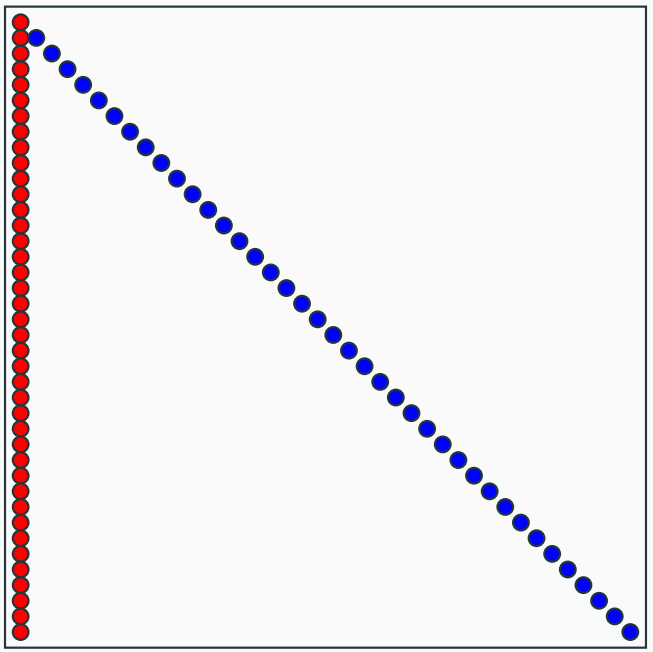
Scaling GPs: Factorization approach

Simplest data-sparse approach

For K (nearly) low rank:

Left-looking partial pivoted Cholesky



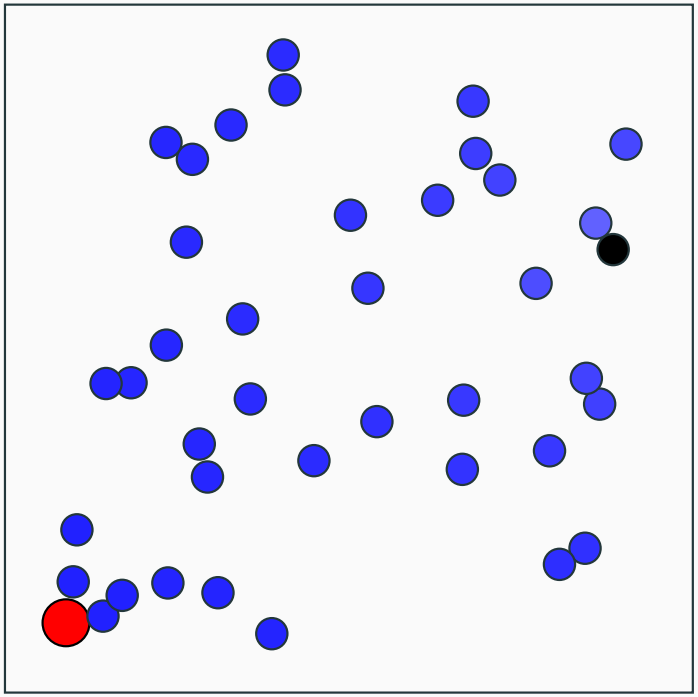


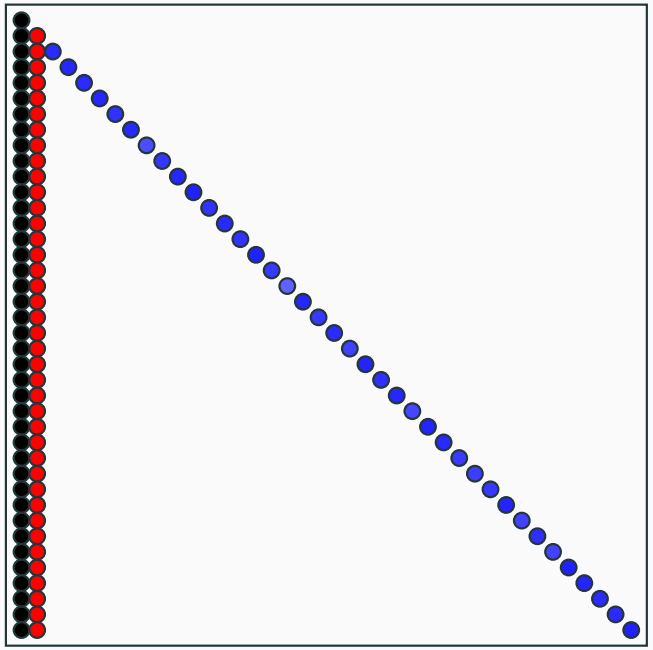
Diagonal element: 1.00e+00

Simplest data-sparse approach

For K (nearly) low rank:

Left-looking partial pivoted Cholesky



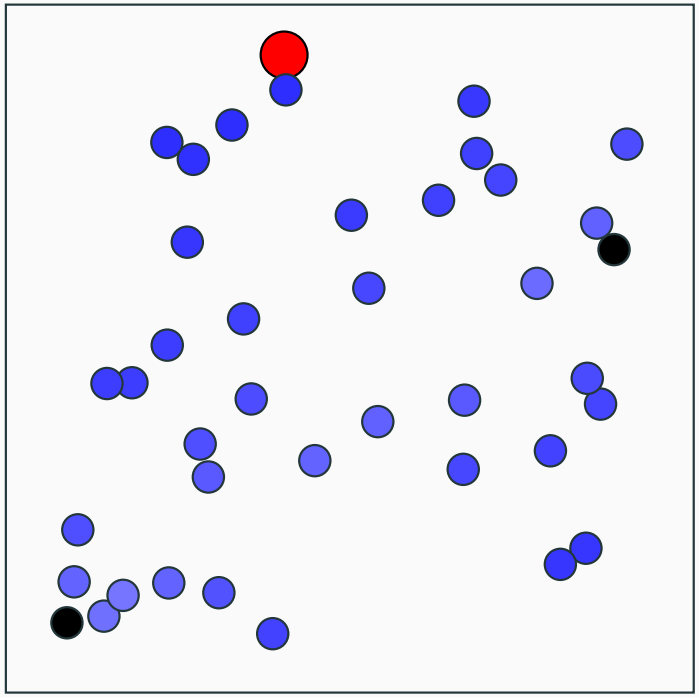


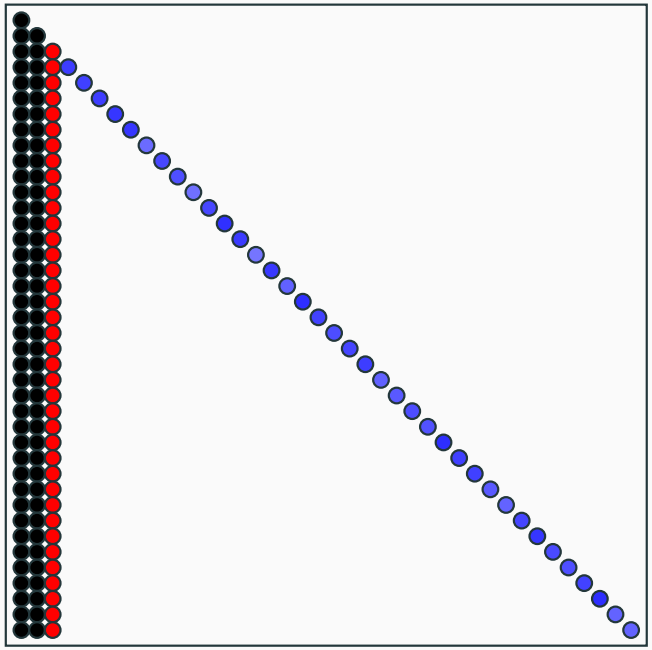
Diagonal element: 6.77e-02

Simplest data-sparse approach

For K (nearly) low rank:

Left-looking partial pivoted Cholesky



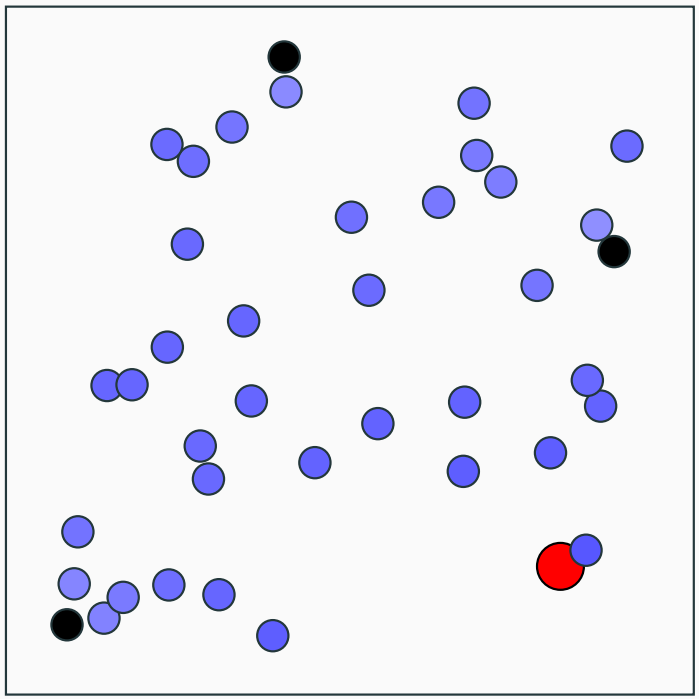


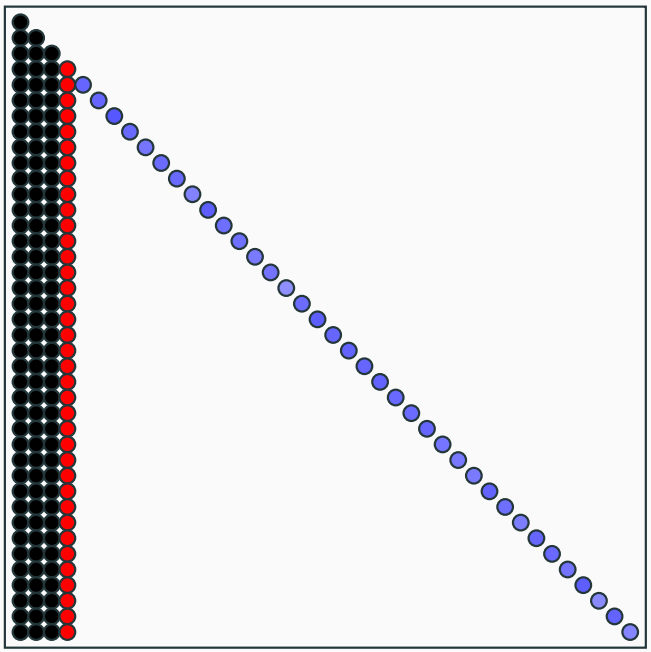
Diagonal element: 1.91e-02

Simplest data-sparse approach

For K (nearly) low rank:

Left-looking partial pivoted Cholesky



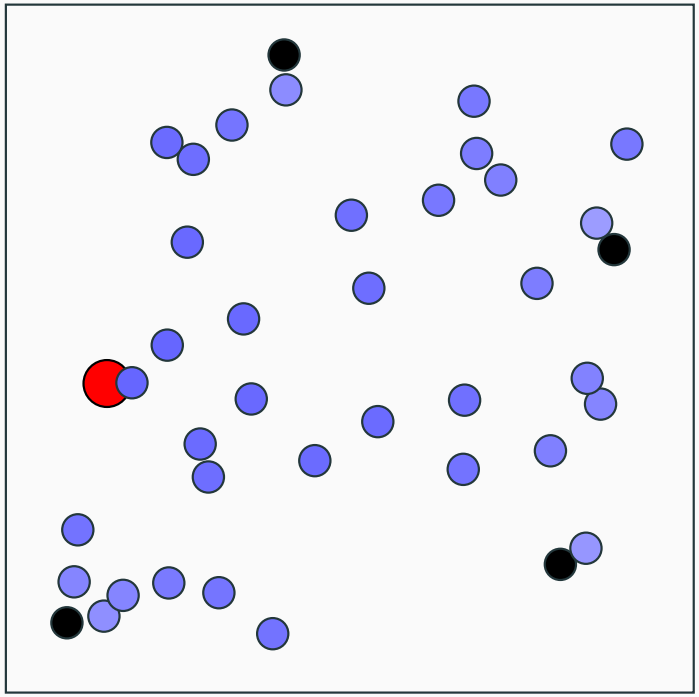


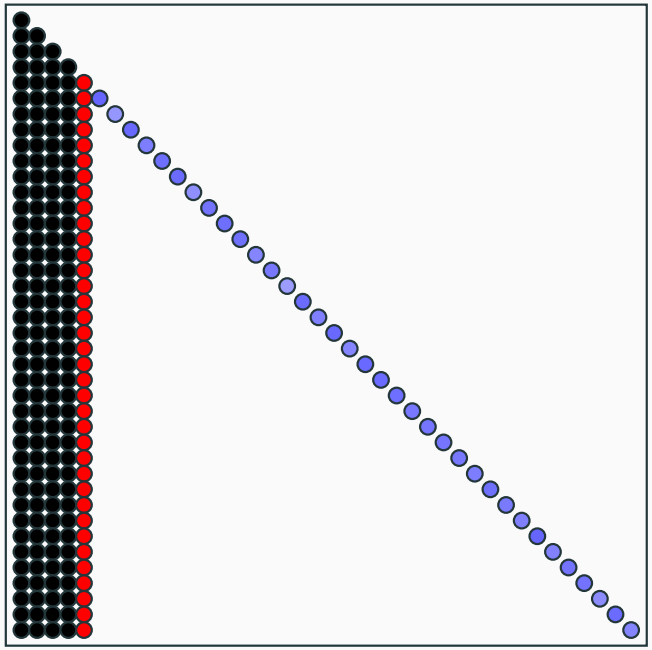
Diagonal element: 5.11e-04

Simplest data-sparse approach

For K (nearly) low rank:

Left-looking partial pivoted Cholesky



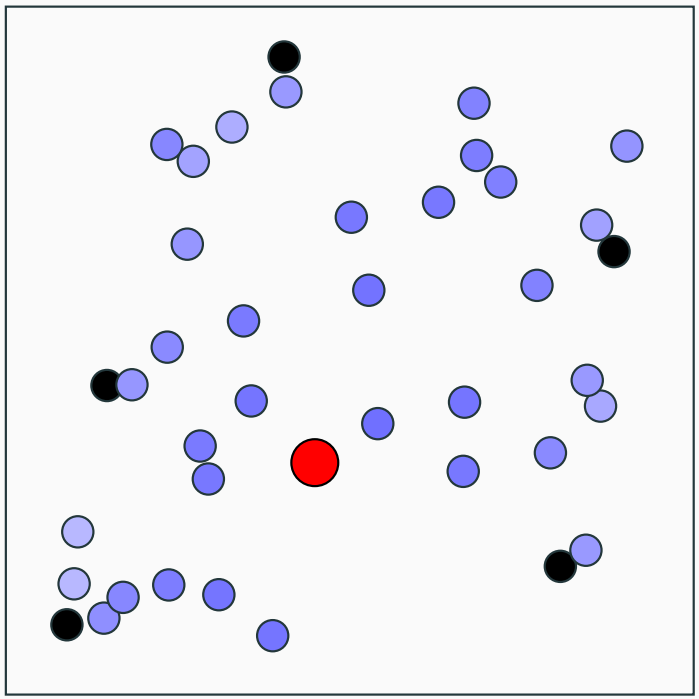


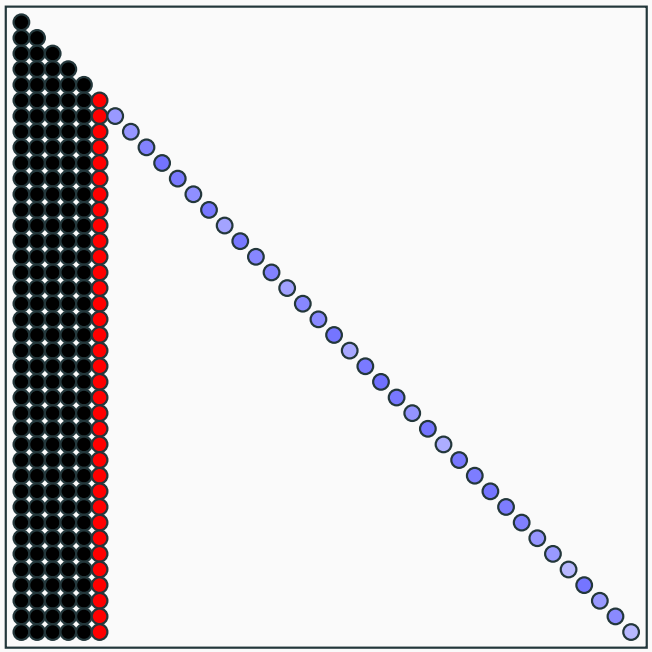
Diagonal element: 1.19e-04

Simplest data-sparse approach

For K (nearly) low rank:

Left-looking partial pivoted Cholesky



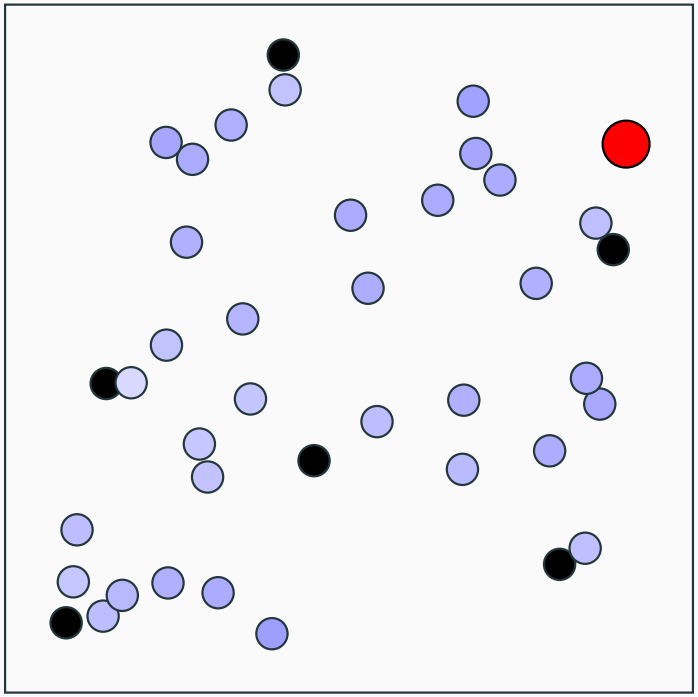


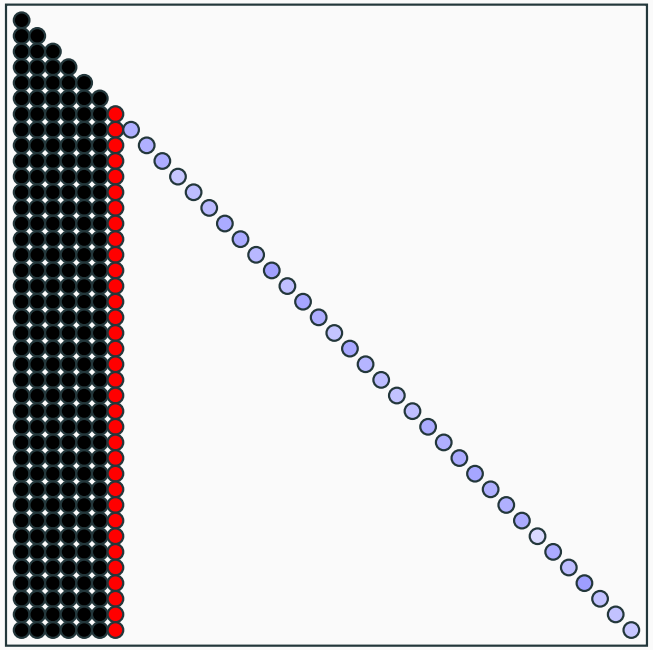
Diagonal element: 4.18e-05

Simplest data-sparse approach

For K (nearly) low rank:

Left-looking partial pivoted Cholesky



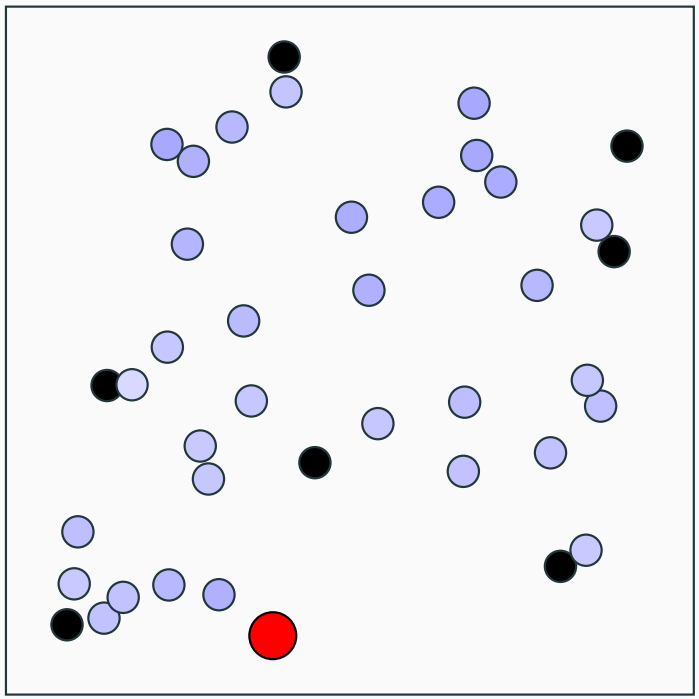


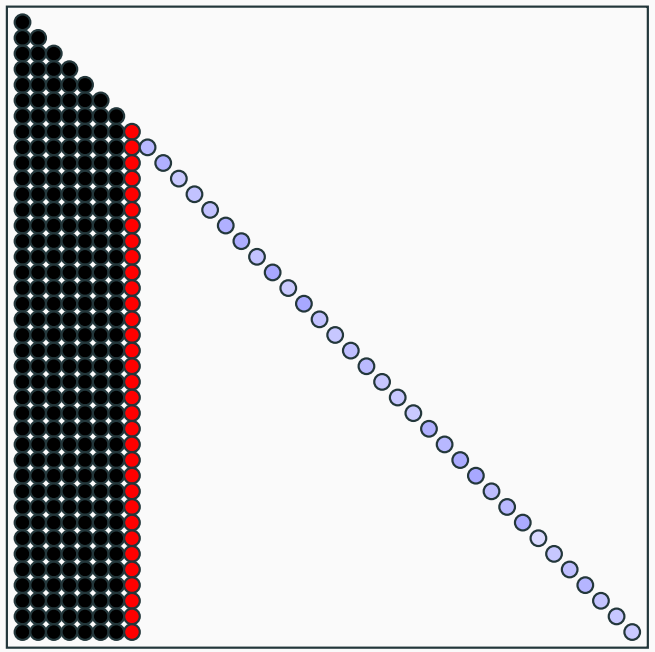
Diagonal element: 8.54e-07

Simplest data-sparse approach

For K (nearly) low rank:

Left-looking partial pivoted Cholesky



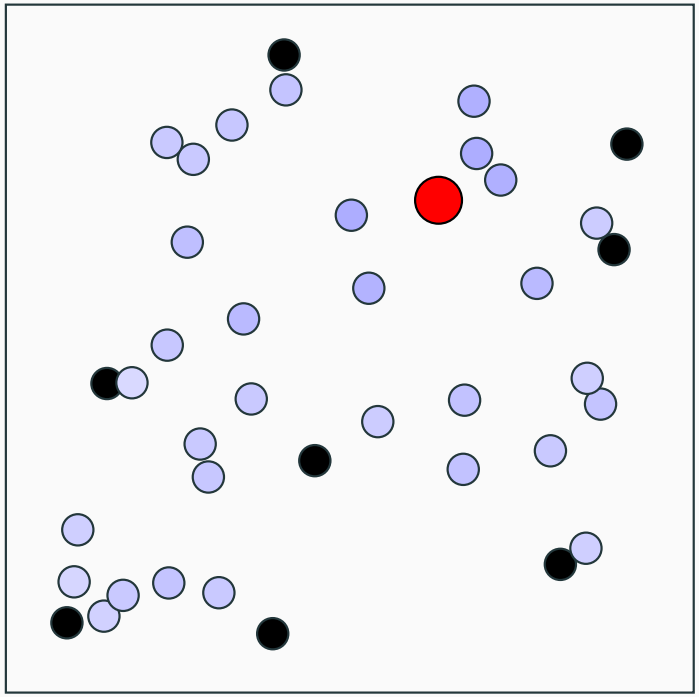


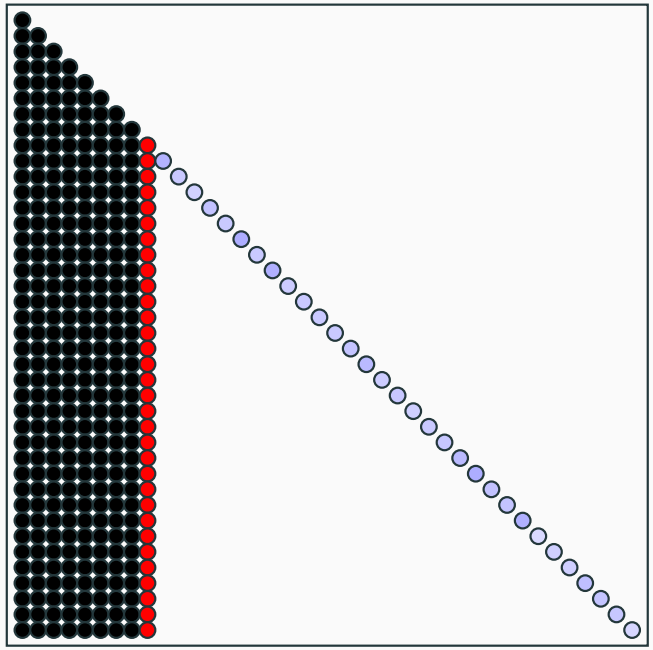
Diagonal element: 3.58e-07

Simplest data-sparse approach

For K (nearly) low rank:

Left-looking partial pivoted Cholesky





Diagonal element: 1.92e-07

Simplest data-sparse approach

Smooth kernel and long length scales: K nearly rank $m \ll n$

- $P(LL^T)P^T =$ partial pivoted Cholesky (select m *inducing points*); does *not* require forming K_{XX}
- Solve $P(LL^T + \sigma^2 I)P^T c = f_X$ stably by reformulating c as a scaled regularized least squares residual:

$$\text{minimize } \|f_X - Lw\|^2 + \sigma^2 \|w\|^2, \quad c = \sigma^{-2}(f_X - Lw)$$

- Compute $\log\det \tilde{K} = \log\det(L^T L + \sigma^2 I) + 2(n-m)\log\sigma$; similar cheap rearrangement for derivatives.
- Prediction and predictive variance are also cheap.

If K is not low rank, can still use *rank-structured* factorization.

Scaling GPs: Kernel approximation

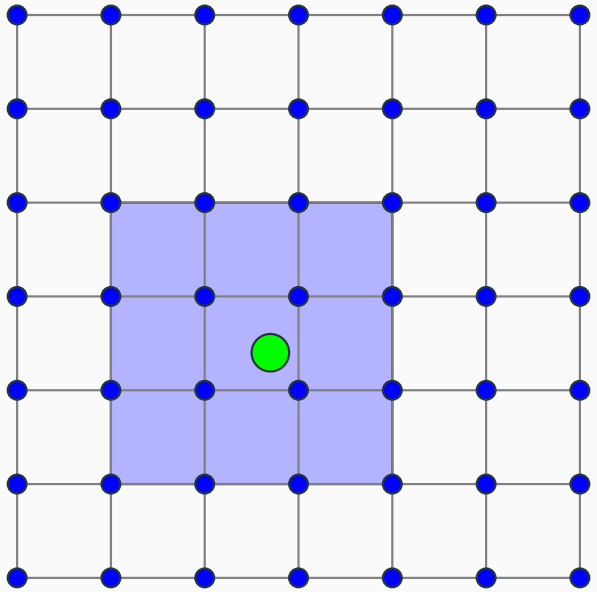
Basic ingredients for “black box” approach

- Fast MVMs with kernel matrices
- Krylov methods for linear solves and matrix functions
- Stochastic estimators: trace, diagonal, and other

Kernel approximations

- Low-rank approximation (via *inducing variables*)
  - Non-smooth kernels, small length scales $\implies$ large rank
  - Only semi-definite
- Sparse approximation
  - OK with SE kernels and short length scales
  - Less good with heavy tails or long length scales
  - May again lose definiteness
- More sophisticated: fast multipole, Fourier transforms
  - Same picture as in integral eq world (FMM, PFFT)
  - Main restriction: low dimensional spaces (2-3D)
- Kernel a model choice — how does approx affect results?

Example: Structured Kernel Interpolation (SKI)



Write $K_{XX} \approx W^T K_{UU} W$ where

- U is a uniform mesh of m points
- K_{UU} has Toeplitz or block Toeplitz structure
- Sparse W interpolates values from X to U

Apply K_{UU} via FFTs in $O(m \log m)$ time.
(Wilson and Nickisch, 2015)

Example: Structured Kernel Interpolation (SKI)

- SKI $\approx$ precorrected FFT method in integral equations (Phillips and White, 1997)
  - Except with less "precorrection" (just the diagonal)
- Many other methods transfered from integral equation literature: FFT-based, FMM-based, etc (Powell, Beatson, Gumarov, Duraswami, Ying, many others)
  - Mostly in the RBF / variational framework vs GP framework
  - Mostly low-dimensional
- What of high-dimensional problems?

Example: SKIP (SKI for Product kernels)

Can construct kernel from a product of 1D kernels, e.g.

$$k(x, y) = \exp(-\|x - y\|^2) = \prod_{j=1}^{d} \exp(-(x_j - y_j)^2)$$

Kernel matrix is a Hadamard products of 1D kernel matrices

$$K_{XX} = K_{XX}^{(1)} \odot K_{XX}^{(2)} \odot \ldots \odot K_{XX}^{(d)}$$

Observe

- $\left(K_{XX}^{(1)} \odot K_{XX}^{(2)}\right) v = \operatorname{diag}\left(K_{XX}^{(1)} D_v K_{XX}^{(2)}\right)$
- Can do this fast if $K_{XX}^{(1)}$ and $K_{XX}^{(2)}$ are low rank
- Apply idea recursively

(Gardner, Pleiss, Wu, Weinberger, Wilson, 2018)

Example: Additive kernels

Products of 1D are great, what about sums? For example:

$$k(x, y) = \frac{1}{J} \sum_{j=1}^{J} \phi(\eta_j^T (x - y))$$

If η drawn from an isotropic distribution, limit is

$$k_e(\|x - y\|) = \mathbb{E}_\eta \left[\phi(\eta^T (x - y)) \right].$$

Example:

$$\phi(z) = \cos(z), \eta \sim N(0, I_d) \quad \Longrightarrow \quad k_e(\tau) = \exp(-\|\tau\|^2/2).$$

Can do (non-Monte-Carlo) quadrature for faster convergence.

Scaling GPs: Black box approach

Function application via Lanczos

A computational kernel: $f(\tilde{K})b$

- Run Lanczos from starting vector $b/\|b\|$
- At n steps in exact arithmetic,

$$f(\tilde{K})b = Qf(T)Q^T b = \|b\|Qf(T)e_1$$

- Truncate at $k \ll n$ steps, use

$$f(\tilde{K})b \approx \|b\|Q_k f(T_k)e_1$$

- Error analysis hinges on quality of poly approx

$$\min_{f \in P_k} \max_{\lambda \in \Lambda(\tilde{K})} |f(\lambda) - \hat{f}(\lambda)|$$

- Compare: Chebyshev methods just use $[\lambda_{\mathsf{min}}, \lambda_{\mathsf{max}}]$

CG is a special case corresponding to $f(z) = z^{-1}$.

Tractable traces

CG solves systems with $\tilde{K}$; problem terms are

$$\mathcal{L}_{|K|} = -\frac{1}{2}\,\mathsf{tr}\left(\log \tilde{K}\right) \qquad \frac{\partial \mathcal{L}_{|K|}}{\partial \theta_i} = -\frac{1}{2}\,\mathsf{tr}\left(\tilde{K}^{-1}\frac{\partial \tilde{K}}{\partial \theta_i}\right)$$

Q: How do we parley fast MVMs into trace computations?

Tractable traces

Stochastic trace estimation trick:

- $z \in \mathbb{R}^n$ has independent random entries
- $\mathbb{E}[z_i] = 0$ and $\mathbb{E}[z_i^2] = 1$

Then

$$\mathbb{E}[z^T A z] = \sum_{i,j} a_{ij} \mathbb{E}[z_i z_j] = \mathbf{tr}(A).$$

NB: $\mathbb{E}[z \odot Az] = \mathbf{diag}(A)$.

Standard choices for the probe vector z:

- Hutchinson: $z_i = \pm 1$ with probability 0.5
- Gaussian: $z_i \sim N(0, 1)$

See Avron and Toledo review, JACM 2011.

Putting it together

For each probe vector z until error bars small enough:

- Run Lanczos from $z/\|z\|$
- Use Lanczos to estimate $\tilde{K}^{-1}z$ and $\log(\tilde{K})z$
- Dot products yield estimators:

$$\mathcal{L}_{|K|} = -\frac{1}{2}\mathbb{E}\left[z^T \log(\tilde{K})z\right]$$

$$\frac{\partial \mathcal{L}_{|K|}}{\partial \theta_i} = -\frac{1}{2}\mathbb{E}\left[(\tilde{K}^{-1}z)^T\left(\frac{\partial \tilde{K}}{\partial \theta_i}z\right)\right]$$

Cost per probe:

- One Lanczos process
- One matvec per parameter with derivative

Quite effective in practice! And amenable to preconditioning.

“There is No New Thing Under the Sun”

“Generalized Cross-Validation for Large-Scale Problems”
Golub and Von Matt (1997)

- Treats least squares and GCV (vs kernel methods and MLE)
- But the same Lanczos + stochastic trace estimator combo

Pivoted Cholesky preconditioning

Let $M = P(LL^T + \sigma^t I)P^T \approx \tilde{K}$ with $L \in \mathbb{R}^{n \times m},\ m \ll n$:

- *Preconditioned* CG: works (implicitly) with $M^{-1}\tilde{K}$
- Note $\log\det \tilde{K} = \log\det M + \log\det M^{-1}\tilde{K}$
- Know how to do fast direct solves and log det with M
- All boils down to generalized Lanczos with $(\tilde{K}, M)$
- Smooth kernels (e.g. SE) and long length scales $\implies$ convergence in few steps

Batching for performance

Generalized Lanczos per probe vector involves

- One matvec with $\tilde{K}$ per step
- One solve with M per step
- Barrier between steps
- Low *arithmetic intensity* (flops / memory access)
- Limited opportunities for parallelism

Idea: Lanczos for several probes in parallel

- Multiply $\tilde{K}$ or M^{-1} by *panel* of vectors / step
- Improves cache use and parallelism

The whole package

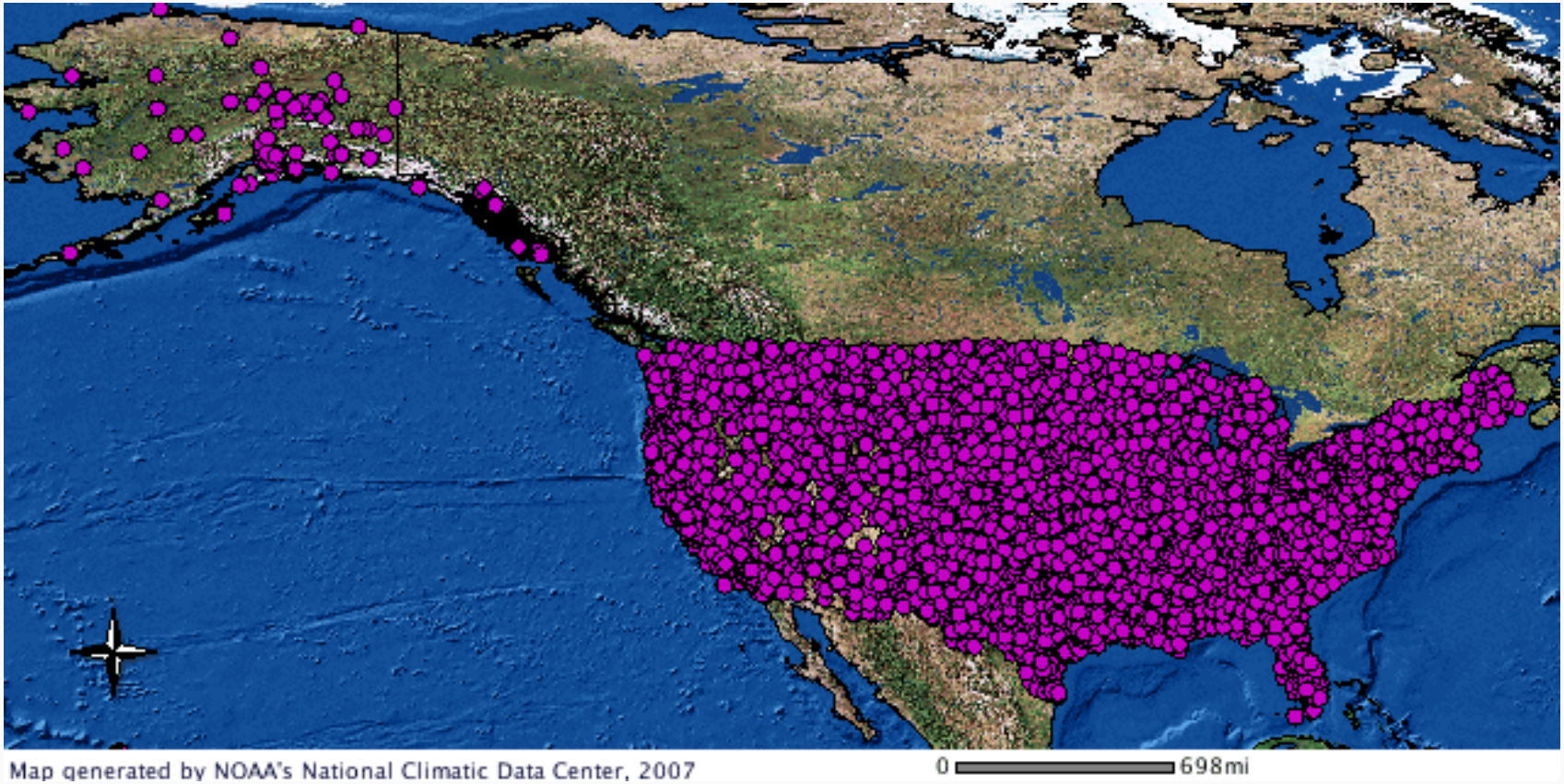
So we have

- Stochastic estimators + Krylov iterations
- Preconditioning to reduce steps to convergence
- Blocking to reduce time per step
- GPU acceleration speeds things up further

For all the tricks together: `https://gpytorch.ai`

Examples

Example: Rainfall



Example: Rainfall

Method	n	m	MSE	Time [min]
Lanczos	528k	3M	0.613	14.3
Scaled eigenvalues	528k	3M	0.621	15.9
Exact	12k	-	0.903	11.8

- Data: Hourly precipitation data at 5500 weather stations
- Aggregate into daily precipitation
- Total data: $628K$ entries
- Train on $100K$ data points, test on remainder
- Use SKI with 100 points per spatial dim, 300 in time
- Comparison: scaled eigenvalues approx, exact solve

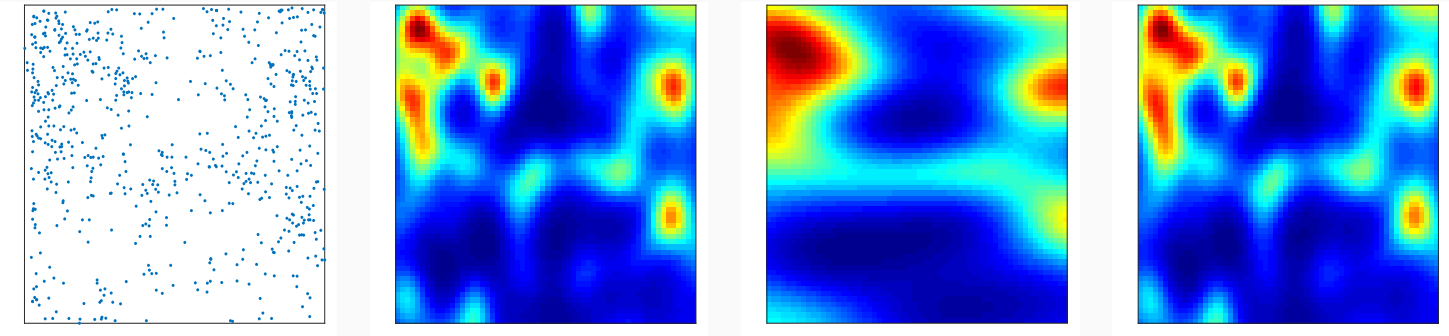
NB: This is with an older MATLAB code (GPML), not GPyTorch

Example: Hickory data

Can build other stochastic processes via GPs

- Example: Log-Gaussian Cox process model
  - Models count data (e.g. events in spatial bins)
  - Poisson conditional on intensity function
  - Log intensity drawn from a GP
- Laplace approximation for posterior
- Data set is point pattern of 703 hickory trees in Michigan

Example: Hickory data



(a) Points (b) Exact (c) Scaled eigs (d) Lanczos

Figure 1: Prediction by different methods on the Hickory dataset.

Method	s_f	ℓ_1	ℓ_2	$-\log p(y\|\theta)$	Time [s]
Exact	0.696	0.063	0.085	1827.56	465.9
Lanczos	0.693	0.066	0.096	1828.07	21.4
Scaled eigs	0.543	0.237	0.112	1851.69	2.5

Table 1: Hyper-parameters recovered by different methods

GPyTorch boost

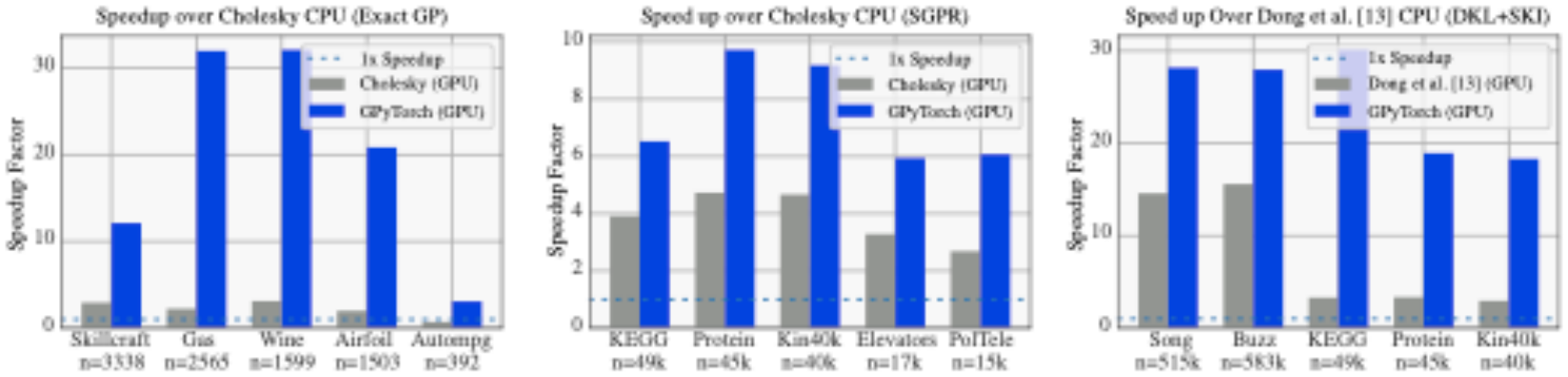


Figure 1: Speedup of GPU-accelerated inference engines. BBMM is in blue, and competing GPU methods are in gray. **Left:** Exact GPs. **Middle:** SGPR [21, 45] – speedup over CPU Cholesky-based inference engines. **Right:** SKI+DKL [50, 52] – speedup over CPU inference of Dong et al. [13].

For more...

Delbridge, B., Wilson. Randomly projected additive Gaussian processes for regression. ICML 2020.

Eriksson, Dong, Lee, B., Wilson. Scaling Gaussian Process Regression with Derivatives. NeurIPS 2018.

Gardner, Pleiss, Weinberger, B., Wilson. GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration. NeurIPS 2018.

K. Dong, D. Eriksson, H. Nickisch, D. Bindel, and A. G. Wilson, Scalable Log Determinants for Gaussian Process Kernel Learning. NeurIPS 2017.

GPyTorch: `https://gpytorch.ai`