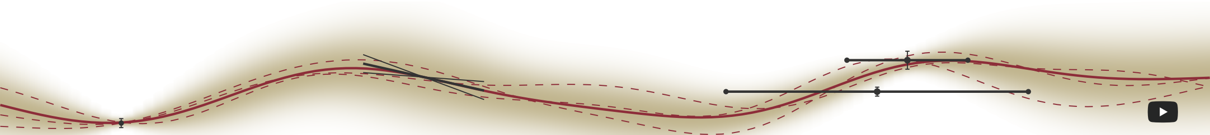
Numerics of Machine Learning
Lecture 03
Scaling Gaussian Processes to Large Datasets

Jonathan Wenger

3 November 2022

Eberhard Karls Universität Tübingen

Faculty of Science
Department of Computer Science
Chair for the Methods of Machine Learning



Where are we in the course?

- Last week: Textbook way of solving linear systems and GP regression
- This week: Modern way of solving large-scale linear systems for GP regression on large datasets
- Next week: Probabilistic numerics perspective on (approximate) GP regression

Today

- Gaussian processes on large datasets.
- Iterative methods as learning algorithms for the matrix inverse.
- Quadratic-time GP inference with (preconditioned) conjugate gradients.
- Linear-time GP inference via inducing point approaches.

Recap:
Gaussian Processes

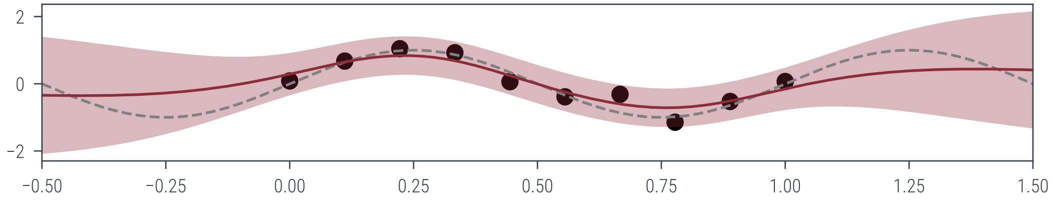
Recap: Gaussian Process Regression

An archetypical supervised machine learning model.

Goal: Learn an unknown function $f_* : \mathbb{R}^d \to \mathbb{R}$ from a training dataset of example input-output pairs.

Desiderata:

- Generalization to unseen data.
- Simplicity / interpretability.
- Know how much to trust the prediction.
- Fast training and inference.



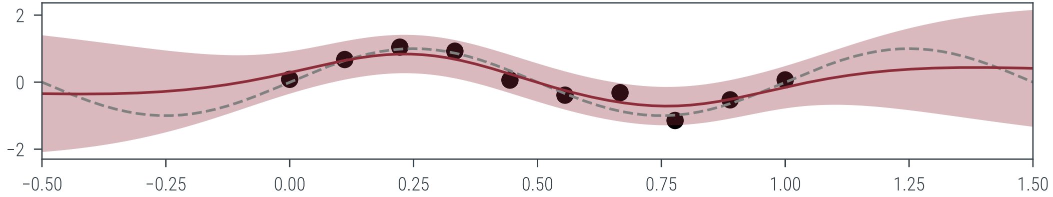
Recap: Gaussian Process Regression

An archetypical supervised machine learning model.

Goal: Learn an unknown function $f_* : \mathbb{R}^d \to \mathbb{R}$ from a training dataset of example input-output pairs.

$$f \sim \mathcal{GP}(\mu, k)$$
$$y \mid f(X) \sim \mathcal{N}(f(X), \sigma^2 I)$$
$$f \mid X, y \sim \mathcal{GP}(\mu_{\text{post}}, k_{\text{post}})$$

$$\mu_{\text{post}}(x) = \mu(x) + k(x, X)(k(X, X) + \sigma^2 I)^{-1}(y - \mu(X))$$
$$k_{\text{post}}(x_0, x_1) = k(x_0, x_1) - k(x_0, X)(k(X, X) + \sigma^2 I)^{-1} k(X, x_1)$$

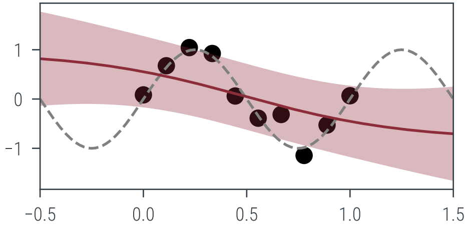


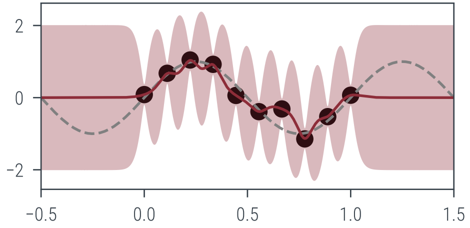
Recap: Model selection for Gaussian Processes

Finding the best kernel hyperparameters.

Model selection: Find kernel hyperparameters $\boldsymbol{\theta}$ to maximize the log-marginal likelihood:

$$
\begin{aligned}
\boldsymbol{\theta}_* &= \arg\max_{\theta} \mathcal{L}(\boldsymbol{\theta}) \\
&= \arg\max_{\theta} \log p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{\theta}) = \arg\max_{\theta} \log \int p(\boldsymbol{y} \mid f(\boldsymbol{X}) = \boldsymbol{z}, \boldsymbol{\theta}) p(f(\boldsymbol{X}) = \boldsymbol{z} \mid \boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{z} \\
&= \arg\max_{\theta} -\frac{1}{2} \underbrace{(\boldsymbol{y} - \boldsymbol{\mu})^\top (k_{\boldsymbol{\theta}}(\boldsymbol{X}, \boldsymbol{X}) + \sigma^2 I)^{-1} (\boldsymbol{y} - \boldsymbol{\mu})}_{\text{model fit}} - \frac{1}{2} \underbrace{\log\det(k_{\boldsymbol{\theta}}(\boldsymbol{X}, \boldsymbol{X}) + \sigma^2 I)}_{\text{model complexity / Occam factor}}
\end{aligned}
$$





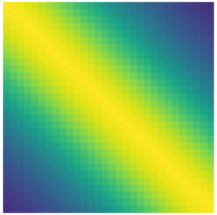
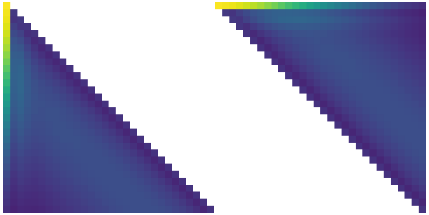
Recap: GP Regression via the Cholesky Decomposition

One numerical method to rule them all?

We need access to

- $v \mapsto (k(X, X) + \sigma^2 I)^{-1} v$ (evaluated $m + 1$ times) and
- $\log\det(k(X, X) + \sigma^2 I)$, as well as its gradient.

$\Longrightarrow$ **Cholesky decomposition**

$k(X, X) + \sigma^2 I =$  $=$  $= LL^\top$

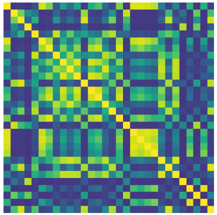
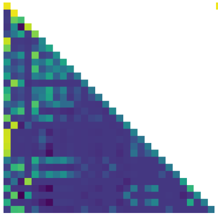
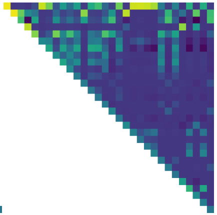
Recap: GP Regression via the Cholesky Decomposition

One numerical method to rule them all?

We need access to

- $v \mapsto (k(X,X) + \sigma^2 I)^{-1} v$ (evaluated $m+1$ times) and
- $\log\det(k(X,X) + \sigma^2 I)$, as well as its gradient.

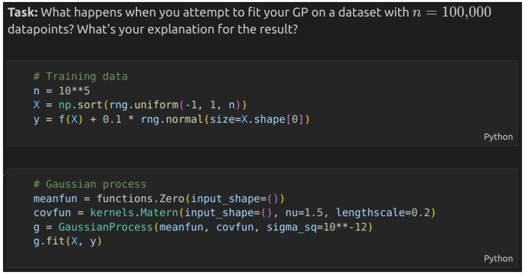
$\Longrightarrow$ Cholesky decomposition

$k(X,X) + \sigma^2 I =$  $=$   $= LL^\intercal$

Large-scale Gaussian Process Regression

Cholesky-based Gaussian process regression on a large dataset.

Homework: Train GP on dataset with $n = 100{,}000$



Task: What happens when you attempt to fit your GP on a dataset with $n = 100{,}000$ datapoints? What's your explanation for the result?

```python
# Training data
n = 10**5
X = np.sort(rng.uniform(-1, 1, n))
y = f(X) + 0.1 * rng.normal(size=X.shape[0])
```

Python

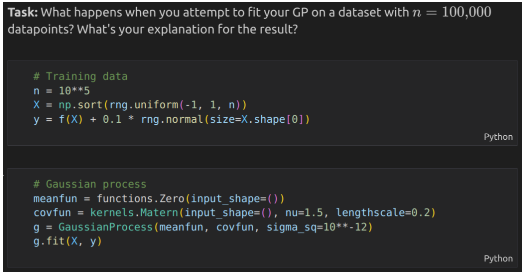
```python
# Gaussian process
meanfun = functions.Zero(input_shape=())
covfun = kernels.Matern(input_shape=(), nu=1.5, lengthscale=0.2)
g = GaussianProcess(meanfun, covfun, sigma_sq=10**-12)
g.fit(X, y)
```

Python

Large-scale Gaussian Process Regression

Cholesky-based Gaussian process regression on a large dataset.

Homework: Train GP on dataset with $n = 100{,}000$



Task: What happens when you attempt to fit your GP on a dataset with $n = 100{,}000$ datapoints? What's your explanation for the result?

```python
# Training data
n = 10**5
X = np.sort(rng.uniform(-1, 1, n))
y = f(X) + 0.1 * rng.normal(size=X.shape[0])
```

Python

```python
# Gaussian process
meanfun = functions.Zero(input_shape=())
covfun = kernels.Matern(input_shape=(), nu=1.5, lengthscale=0.2)
g = GaussianProcess(meanfun, covfun, sigma_sq=10**-12)
g.fit(X, y)
```

Python

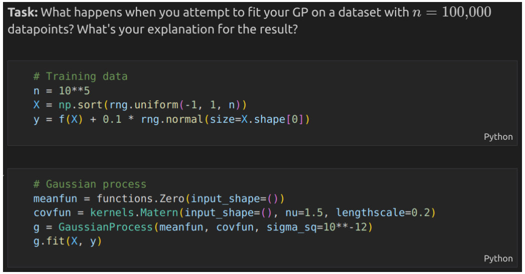
Memory:

```
MemoryError: Unable to allocate 74.5GiB for
an array with shape (100000, 100000) and data
type float64
```

Large-scale Gaussian Process Regression

Cholesky-based Gaussian process regression on a large dataset.

Homework: Train GP on dataset with $n = 100{,}000$



Task: What happens when you attempt to fit your GP on a dataset with $n = 100{,}000$ datapoints? What's your explanation for the result?

```
# Training data
n = 10**5
X = np.sort(rng.uniform(-1, 1, n))
y = f(X) + 0.1 * rng.normal(size=X.shape[0])
```

Python

```
# Gaussian process
meanfun = functions.Zero(input_shape=())
covfun = kernels.Matern(input_shape=(), nu=1.5, lengthscale=0.2)
g = GaussianProcess(meanfun, covfun, sigma_sq=10**-12)
g.fit(X, y)
```

Python

Memory:

`MemoryError: Unable to allocate 74.5GiB for an array with shape (100000, 100000) and data type float64`

Time: Modern CPU $\approx 10^9 \frac{\text{flops}}{\text{s}}$:

$$\frac{\#\text{flops}}{10^9 \frac{\text{flops}}{\text{s}}} \simeq \frac{1}{3}\frac{(10^5)^3}{10^9}\text{s} = \frac{1}{3}10^{15-9}\text{s} = \frac{1}{3}10^6\text{s} \approx 83\text{h}$$

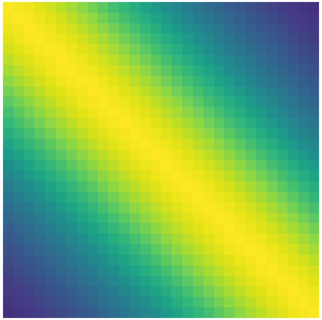
A Cholesky decomposition is prohibitive both in time and space for large datasets.

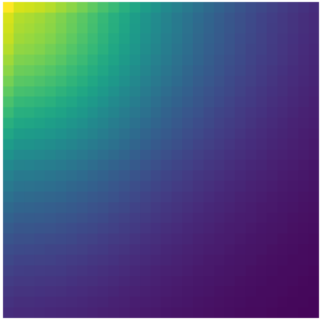
Gaussian Process Approximation in $\mathcal{O}(in^2)$: Iterative Methods

Iterative Approximation of the Kernel Matrix via Cholesky

The Cholesky decomposition computes a rank-i approximation to the kernel matrix.

$$k(X, X) + \sigma^2 I = \left(\vphantom{\Big|} \right) \approx \left(\vphantom{\Big|} \right) = L_1 L_1^\top$$

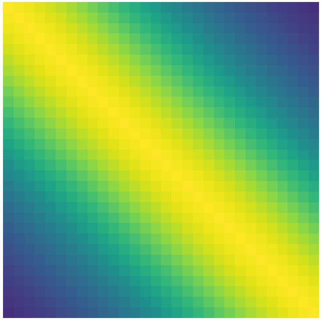




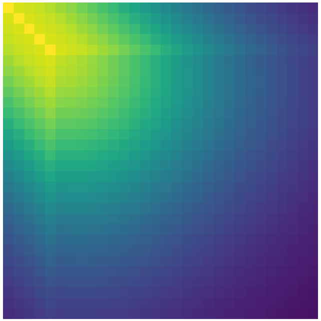
Iterative Approximation of the Kernel Matrix via Cholesky

The Cholesky decomposition computes a rank-*i* approximation to the kernel matrix.

$$k(X, X) + \sigma^2 I = \left(\right.$$



$$\left. \right) \approx \left(\right.$$

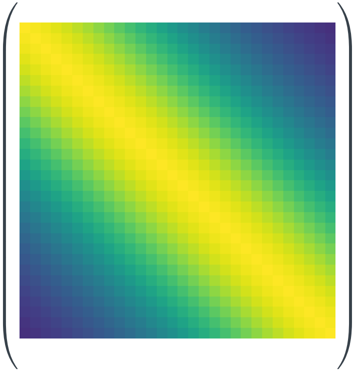


$$\left. \right) = L_5 L_5^\top$$

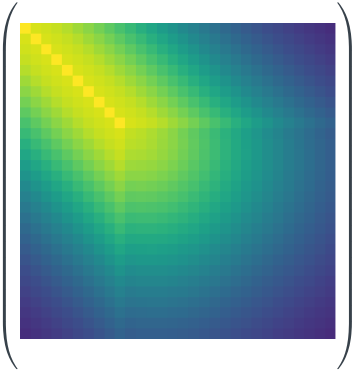
Iterative Approximation of the Kernel Matrix via Cholesky

The Cholesky decomposition computes a rank-*i* approximation to the kernel matrix.

$k(X, X) + \sigma^2 I =$



$\approx$



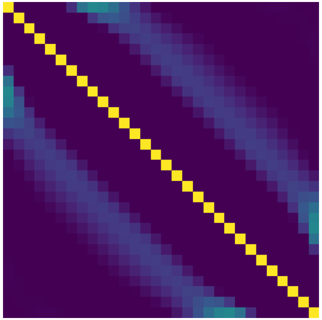
$= L_{10} L_{10}^{\mathsf{T}}$

Cholesky can be seen as an iterative learning algorithm for the kernel matrix.

Iterative Approximation of the Inverse Kernel Matrix

We need the inverse of the kernel matrix for Gaussian process inference.

$$(k(X,X)+\sigma^2 I)^{-1} = \left(\vphantom{\Big|} \right) \approx \left(? \right) = C_i$$



Can we approximate the linear solves $v \mapsto (k(X,X)+\sigma^2 I)^{-1} v \approx C_i v$?

Learning to Invert the Kernel Matrix

The Cholesky decomposition as a learning algorithm for the inverse kernel matrix.

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Algorithm 1 Cholesky Decomposition

Input: spd matrix A
Output: lower triangular L_i, s.t. $L_i L_i^\intercal \approx A$

```
procedure CHOLESKY($A$)
    $A' \leftarrow A$
    for $i \in \{1, \dots, n\}$ do



        $l_i \leftarrow A'_{:i} / \sqrt{A'_{ii}} = A'(e_i / \|e_i\|_{A'})$

        $A' \leftarrow A' - l_i l_i^\intercal = A - L_i L_i^\intercal$    // Matrix residual
        $L_i = (L_{i-1} \quad l_i)$    // Cholesky factor
    end for
    return $L_i$
end procedure
```

Goal: (Low-rank) Approximation $C_i \approx A^{-1}$

Observation: Matrix approx. $\rightarrow$ inverse approx.?

$$L_i L_i^\intercal \approx A$$

$$\underbrace{(A^{-1} L_i)(A^{-1} L_i)^\intercal}_{=C_i} \approx A^{-1}$$

Consider last column $(A^{-1} L_i)_{:i} = A^{-1} l_i$:

$$A^{-1} l_i = A^{-1} A' \frac{e_i}{\|e_i\|_{A'}} = A^{-1}(A - L_{i-1} L_{i-1}^\intercal) \frac{e_i}{\|e_i\|_{A'}}$$

$$= (I - C_{i-1} A) \frac{e_i}{\|e_i\|_{A'}}$$

Learning to Invert the Kernel Matrix

The Cholesky decomposition as a learning algorithm for the inverse kernel matrix.

Algorithm 2 Cholesky with Inverse Approximation

Input: spd matrix A

Output: lower triangular L_i, s.t. $L_i L_i^\intercal \approx A$, low-rank $C_i \approx A^{-1}$

```
procedure CHOLESKY($A$)
    $A' \leftarrow A$, $C_0 = 0$
    for $i \in \{1, \dots, n\}$ do
        $s_i \leftarrow e_i$                                                  // Action
        $d_i \leftarrow (I - C_{i-1}A)s_i$
        $\eta_i \leftarrow s_i^\intercal A d_i = e_i^\intercal A' e_i = \|e_i\|_{A'}^2$,   // Norm. constant
        $l_i \leftarrow A \frac{1}{\sqrt{\eta_i}} d_i$                               // Matrix observation
        $C_i \leftarrow C_{i-1} + \frac{1}{\eta_i} d_i d_i^\intercal$                        // Inverse estimate
        $A' \leftarrow A - L_i L_i^\intercal = A(A^{-1} - C_i)A = A(I - C_i A)$
        $L_i = \begin{pmatrix} L_{i-1} & l_i \end{pmatrix}$
    end for
    return $L_i$, $C_i$
end procedure
```

Goal: (Low-rank) Approximation $C_i \approx A^{-1}$

Observation: Matrix approx. $\rightarrow$ inverse approx.?

$$L_i L_i^\intercal \approx A$$

$$\underbrace{(A^{-1}L_i)(A^{-1}L_i)^\intercal}_{=C_i} \approx A^{-1}$$

Consider last column $(A^{-1}L_i)_{:i} = A^{-1} l_i$:

$$A^{-1} l_i = A^{-1} A' \frac{e_i}{\|e_i\|_{A'}} = A^{-1}(A - L_{i-1}L_{i-1}^\intercal)\frac{e_i}{\|e_i\|_{A'}}$$

$$= (I - C_{i-1}A)\frac{e_i}{\|e_i\|_{A'}} = \frac{1}{\sqrt{\eta_i}} d_i$$

Computational complexity: $\#\text{flops} \in \mathcal{O}(in^2)$

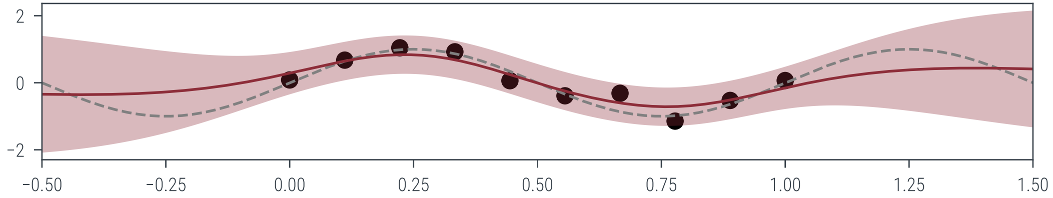
Cholesky can be seen as an iterative learning algorithm for the kernel matrix and its inverse.

Gaussian Process Inference via the Partial Cholesky

Performing Gaussian process inference with a learned inverse approximation via the partial Cholesky decomposition.

$$f \sim \mathcal{GP}(\mu, k)$$
$$y \mid f(X) \sim \mathcal{N}(f(X), \sigma^2 I)$$
$$f \mid X, y \sim \mathcal{GP}(\mu_{\text{post}}, k_{\text{post}})$$

$$\mu_{\text{post}}(x) = \mu(x) + k(x, X)(k(X, X) + \sigma^2 I)^{-1}(y - \mu(X))$$
$$k_{\text{post}}(x_0, x_1) = k(x_0, x_1) - k(x_0, X)(k(X, X) + \sigma^2 I)^{-1} k(X, x_1)$$

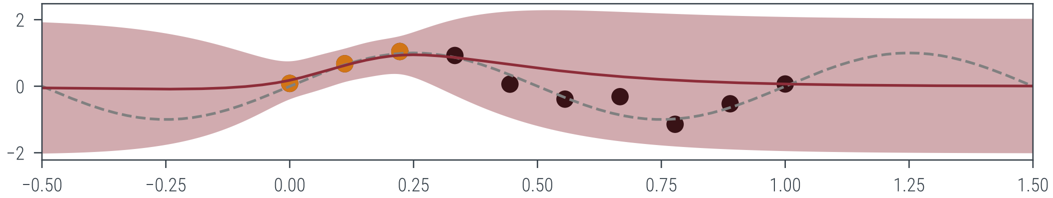


Gaussian Process Inference via the Partial Cholesky

Performing Gaussian process inference with a learned inverse approximation via the partial Cholesky decomposition.

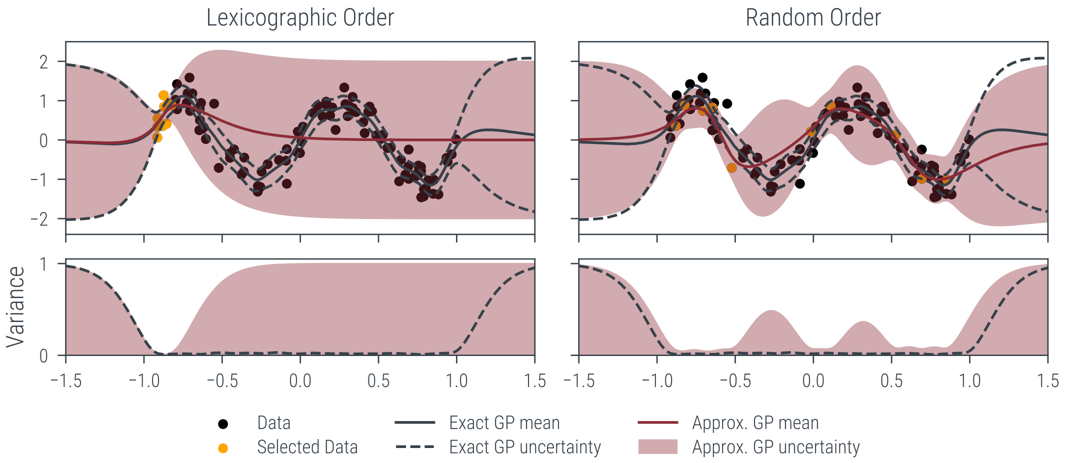
$$f \sim \mathcal{GP}(\mu, k)$$
$$y \mid f(X) \sim \mathcal{N}(f(X), \sigma^2 I)$$
$$f \mid X, y \sim \mathcal{GP}(\mu_{\text{post}}, k_{\text{post}})$$

$$\mu_{\text{post}}(x) = \mu(x) + k(x, X) C_i (y - \mu(X))$$
$$k_{\text{post}}(x_0, x_1) = k(x_0, x_1) - k(x_0, X) C_i k(X, x_1)$$



Interpreting the Pivoting Strategy as Active Learning

In each iteration the partial Cholesky selects a datapoint as a pivot via its action.



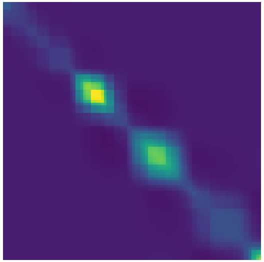
The selection of datapoints, i.e. choice of actions s_i, matters a lot for convergence.

Can we find better actions?

Why restrict ourselves to just unit vectors to probe the matrix residual?

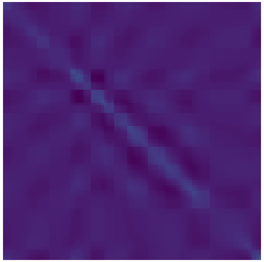
Partial Cholesky

$$A'e_i = A(I - C_{i-1}A)s_i = Ad_i$$

$$= \Bigg($$  $$\Bigg) \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Other Method?

$$A'e_i = A(I - C_{i-1}A)s_i = Ad_i$$

$$= \Bigg($$  $$\Bigg) \begin{pmatrix} * \\ \vdots \\ * \\ * \\ * \\ \vdots \\ * \end{pmatrix}$$

Can we learn the kernel matrix (inverse) in a more efficient way via different actions?

How to rapidly compute linear solves with a (kernel) matrix: **Method of Conjugate Gradients**

Method of Conjugate Gradients

Efficiently solving linear systems with positive definite system matrix via matrix-vector multiplies.

Goal: Approximately solve linear system $Ax = b$ with few matrix-vector multiplies.

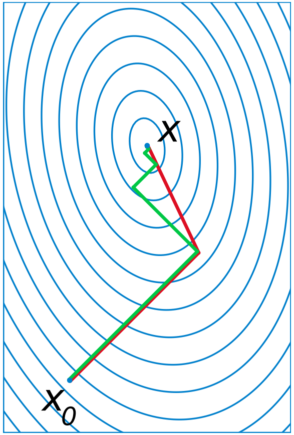
Idea: Rephrase as quadratic optimization problem and optimize. Let

$$f(x) = \frac{1}{2} x^\top A x - b^\top x$$

then $\nabla f(x) = 0 \iff Ax = b \iff r(x) := b - Ax = 0.$

Question: How should we optimize?

1. Gradient descent: Follow $\mathbf{d}_i = r(x_i) = -\nabla f(x_i)$ s.t. $\langle \mathbf{d}_i, \mathbf{d}_j \rangle = 0$.



Oleg Alexandrov, commons.wikimedia.org/w/index.php?curid=2267598

Method of Conjugate Gradients

Efficiently solving linear systems with positive definite system matrix via matrix-vector multiplies.

Goal: Approximately solve linear system $Ax = b$ with few matrix-vector multiplies.

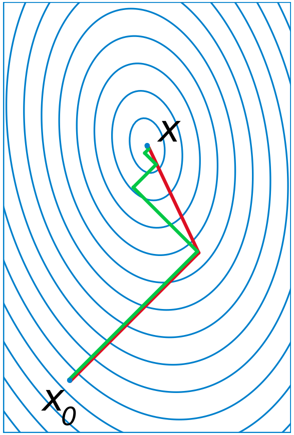
Idea: Rephrase as quadratic optimization problem and optimize. Let

$$f(x) = \frac{1}{2}x^\top Ax - b^\top x$$

then $\nabla f(x) = 0 \iff Ax = b \iff r(x) := b - Ax = 0$.

Question: How should we optimize?

1. Gradient descent: Follow $d_i = r(x_i) = -\nabla f(x_i)$ s.t. $\langle d_i, d_j \rangle = 0$.
2. Conjugate direction method: Follow d_i s. t. $\langle d_i^\top d_j \rangle_A = d_i^\top A d_j = 0$ for $i \neq j$.
   $\implies$ convergence in at most n steps.



Oleg Alexandrov, commons.wikimedia.org/w/index.php?curid=2267598

Method of Conjugate Gradients

Efficiently solving linear systems with positive definite system matrix via matrix-vector multiplies.

Goal: Approximately solve linear system $Ax = b$ with few matrix-vector multiplies.

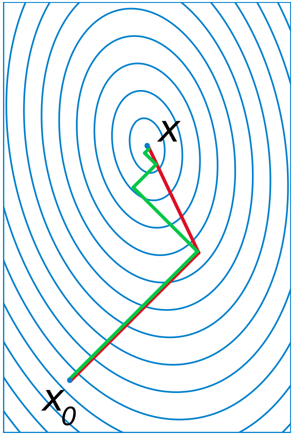
Idea: Rephrase as quadratic optimization problem and optimize. Let

$$f(x) = \frac{1}{2} x^\intercal A x - b^\intercal x$$

then $\nabla f(x) = 0 \iff Ax = b \iff r(x) := b - Ax = 0$.

Question: How should we optimize?

1. Gradient descent: Follow $d_i = r(x_i) = -\nabla f(x_i)$ s.t. $\langle d_i, d_j \rangle = 0$.
2. Conjugate direction method: Follow d_i s. t. $\langle d_i^\intercal d_j \rangle_A = d_i^\intercal A d_j = 0$ for $i \neq j$. $\implies$ convergence in at most n steps.
3. Conjugate gradient method: First step $d_0 = r(x_0)$.



Oleg Alexandrov, commons.wikimedia.org/w/index.php?curid=2267598

Algorithm: Method of Conjugate Gradients

We can interpret CG as a learning algorithm for the matrix inverse as well.

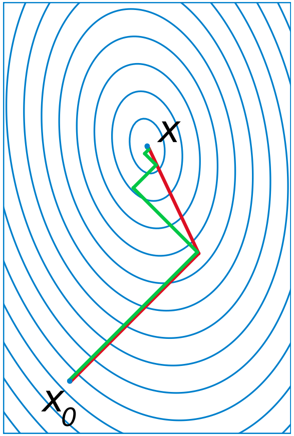
Algorithm 3 Conjugate Gradient Method

Input: spd matrix A, vector b, initial guess x_0
Output: approximate solution $x_i \approx A^{-1}b$

```
procedure CG($A, b, x_0$)
    while $\|r_i\|_2 > \max(\delta_{\text{rtol}}\|b\|_2, \delta_{\text{atol}})$ do
        $r_{i-1} \leftarrow b - Ax_{i-1}$    // Residual

        $d_i \leftarrow r_{i-1} - \frac{r_{i-1}^\top A d_{i-1}}{d_{i-1}^\top A d_{i-1}} d_{i-1}$    // Search direction

        $x_i \leftarrow x_{i-1} + \frac{r_{i-1}^\top r_{i-1}}{d_i^\top A d_i} d_i$    // Solution estimate
    end while
    return $x_i$
end procedure
```



Oleg Alexandrov, commons.wikimedia.org/w/index.php?curid=2267598

Algorithm: Method of Conjugate Gradients

We can interpret CG as a learning algorithm for the matrix inverse as well.

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Algorithm 3 Conjugate Gradient Method

Input: spd matrix A, vector b, initial guess x_0

Output: approximate solution $x_i \approx A^{-1}b$

```
procedure CG($A, b, x_0$)
    while $\|r_i\|_2 > \max(\delta_{\text{rtol}}\|b\|_2, \delta_{\text{atol}})$ do
        $r_{i-1} \leftarrow b - Ax_{i-1}$                                        // Residual

        $d_i \leftarrow r_{i-1} - \frac{r_{i-1}^\top A d_{i-1}}{d_{i-1}^\top A d_{i-1}} d_{i-1}$          // Search direction

        $x_i \leftarrow x_{i-1} + \frac{r_{i-1}^\top r_{i-1}}{d_i^\top A d_i} d_i$                       // Solution estimate
    end while
    return $x_i$
end procedure
```

Algorithm 4 CG with Inverse Approximation

Input: spd matrix A, vector b, initial guess x_0

Output: approximate solution $x_i \approx A^{-1}b$, low-rank $C_i \approx A^{-1}$

```
procedure CG($A, b, x_0$)
    while $\|r_i\|_2 > \max(\delta_{\text{rtol}}\|b\|_2, \delta_{\text{atol}})$ do
        $r_{i-1} \leftarrow b - Ax_{i-1}$                                        // Residual
        $s_i \leftarrow r_{i-1}$                                                 // Action
        $\alpha_i \leftarrow s_i^\top r_{i-1}$                                         // Observation
        $d_i \leftarrow (I - C_{i-1}A)s_i$                                   // Search direction
        $\eta_i \leftarrow s_i^\top A d_i = d_i^\top A d_i$                                // Norm. constant
        $C_i \leftarrow C_{i-1} + \frac{1}{\eta_i} d_i d_i^\top$                            // Inverse estimate
        $x_i \leftarrow x_{i-1} + \frac{\alpha_i}{\eta_i} d_i = C_i b$                       // Solution estimate
    end while
    return $x_i, C_i$
end procedure
```

Algorithm: Method of Conjugate Gradients

We can interpret CG as a learning algorithm for the matrix inverse as well.

Algorithm 2 Cholesky with Inverse Approximation

Input: spd matrix A

Output: lower triangular L_i, s.t. $L_i L_i^\intercal \approx A$, low-rank $C_i \approx A^{-1}$

```
procedure CHOLESKY($A$)
    $A' \leftarrow A$, $C_0 = 0$
    for $i \in \{1, \dots, n\}$ do
        $s_i \leftarrow e_i$                                                   // Action
        $d_i \leftarrow (I - C_{i-1}A)s_i$
        $\eta_i \leftarrow s_i^\intercal A d_i = e_i^\intercal A' e_i = \|e_i\|^2_{A'}$      // Norm. constant
        $l_i \leftarrow A \frac{1}{\sqrt{\eta_i}} d_i$                                  // Matrix observation
        $C_i \leftarrow C_{i-1} + \frac{1}{\eta_i} d_i d_i^\intercal$                         // Inverse estimate
        $A' \leftarrow A - L_i L_i^\intercal = A(A^{-1} - C_i)A = A(I - C_i A)$
        $L_i = (L_{i-1} \quad l_i)$
    end for
    return $L_i$, $C_i$
end procedure
```

Algorithm 4 CG with Inverse Approximation

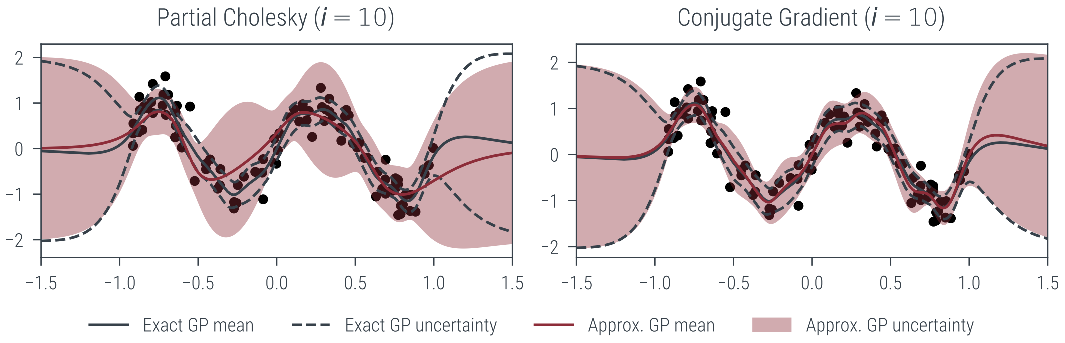
Input: spd matrix A, vector b, initial guess x_0

Output: approximate solution $x_i \approx A^{-1}b$, low-rank $C_i \approx A^{-1}$

```
procedure CG($A, b, x_0$)
    while $\|r_i\|_2 > \max(\delta_{\text{rtol}}\|b\|_2, \delta_{\text{atol}})$ do
        $r_{i-1} \leftarrow b - Ax_{i-1}$                                // Residual
        $s_i \leftarrow r_{i-1}$                                         // Action
        $\alpha_i \leftarrow s_i^\intercal r_{i-1}$                                  // Observation
        $d_i \leftarrow (I - C_{i-1}A)s_i$                               // Search direction
        $\eta_i \leftarrow s_i^\intercal A d_i = d_i^\intercal A d_i$                           // Norm. constant
        $C_i \leftarrow C_{i-1} + \frac{1}{\eta_i} d_i d_i^\intercal$                         // Inverse estimate
        $x_i \leftarrow x_{i-1} + \frac{\alpha_i}{\eta_i} d_i = C_i b$                      // Solution estimate
    end while
    return $x_i$, $C_i$
end procedure
```

Comparing the Partial Cholesky and CG for GP Inference

How we observe the kernel matrix influences the approximate posterior.



The method of conjugate gradients seems to converge faster. But how fast?

Numerics Interlude

How much should I trust the output of a numerical algorithm?

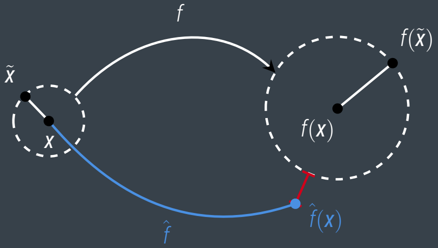
Machine precision: unavoidable rounding error in floating point arithmetic $\tilde{x} = \mathrm{fl}(x)$

Condition number: unavoidable error amplification by f

Condition number of a matrix $\kappa_2(A) = \|A^{-1}\|_2 \|A\|_2 = \frac{|\lambda_{\max}|}{|\lambda_{\min}|}$

Stability: error from my specific choice of algorithm $\hat{f}$

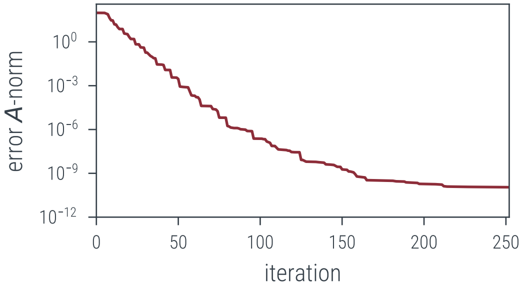
An algorithm is stable iff $\hat{f}$ behaves like $\mathrm{fl} \circ f \circ \mathrm{fl}$.



Convergence Behavior of CG

The spectrum of the matrix determines the convergence speed.

$n = 10^3 \quad \kappa(A) \approx 7 \cdot 10^5$



Theorem (Convergence Rate of CG)

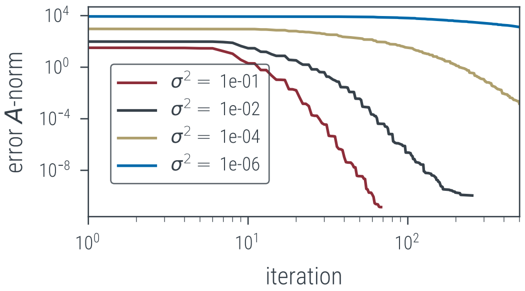
$$\|x - x_i\|_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^i \|x - x_0\|_A$$

CG converges fast for a small condition number.

Fast Convergence in all Cases?

Things can go wrong. Especially for kernel matrices.

$n = 10^3 \quad A = K + \sigma^2 I$



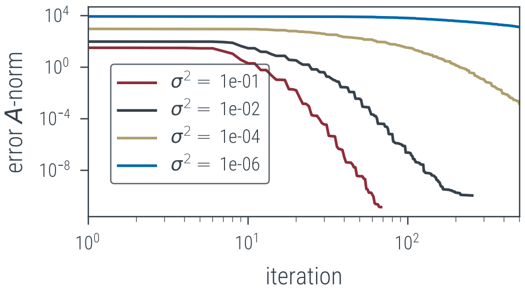
Theorem (Convergence Rate of CG)

$$\|x - x_i\|_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^i \|x - x_0\|_A$$

Fast Convergence in all Cases?

Things can go wrong. Especially for kernel matrices.

$n = 10^3 \quad A = K + \sigma^2 I$



Theorem (Convergence Rate of CG)

$$\|x - x_i\|_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^i \|x - x_0\|_A$$

$$K + \sigma^2 I = Q \Lambda Q^\intercal + \sigma^2 I = Q \Lambda Q^\intercal + \sigma^2 I Q Q^\intercal = Q(\underbrace{\Lambda + \sigma^2 I}_{\operatorname{diag}(\lambda_i(K) + \sigma^2)}) Q^\intercal \implies \kappa(K + \sigma^2 I) = \frac{\lambda_{\max}(K) + \sigma^2}{\lambda_{\min}(K) + \sigma^2}$$

If observation noise is small, close datapoints can significantly affect matrix conditioning.

Preconditioning

How to encode and leverage structural prior knowledge about matrices.

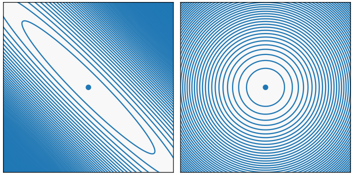
Preconditioner: Computationally tractable approximation $P \approx A$.

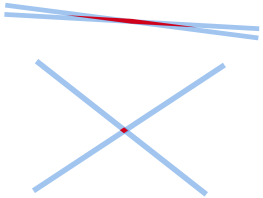
- Computing and storing P is cheap.
- Linear solves $v \mapsto P^{-1}v$ are efficient.
- Derived properties, such as the determinant are known.

Idea: Solve equivalent linear system $P^{-1}Ax = P^{-1}b$ such that

$$\kappa(P^{-1}A) \ll \kappa(A).$$

Intuition: Prior knowledge about A and A^{-1}.





Preconditioning accelerates and stabilizes linear solves via CG.

Making use of prior information for fast linear system solves: **Preconditioning**

Preconditioned Conjugate Gradients

Preconditioners accelerate convergence of CG.

Algorithm 5 Preconditioned CG

Input: spd matrix A, vector b, initial guess x_0, preconditioner P

Output: approximate solution $x_i \approx A^{-1}b$, low-rank $C_i \approx A^{-1}$

```
procedure CG($A, b, x_0, P$)
    while $\|r_i\|_2 > \max(\delta_{\mathrm{rtol}}\|b\|_2, \delta_{\mathrm{atol}})$ do
        $r_{i-1} \leftarrow b - Ax_{i-1}$                                  // Residual
        $s_i \leftarrow P^{-1} r_{i-1}$                                     // Action
        $\alpha_i \leftarrow s_i^\top r_{i-1} = r_{i-1}^\top (P^{-\top}Ax - P^{-\top}b)$   // Obs.
        $d_i \leftarrow (I - C_{i-1}A)s_i$                                  // Search direction
        $\eta_i \leftarrow s_i^\top A d_i = d_i^\top A d_i$                 // Norm. constant
        $C_i \leftarrow C_{i-1} + \frac{1}{\eta_i} d_i d_i^\top$            // Inverse estimate
        $x_i \leftarrow x_{i-1} + \frac{\alpha_i}{\eta_i} d_i$              // Solution estimate
    end while
    return $x_i, C_i$
end procedure
```

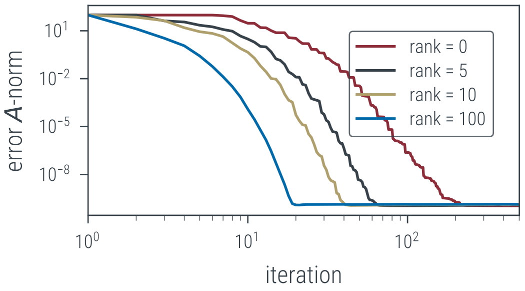
Preconditioned Conjugate Gradients

Preconditioners accelerate convergence of CG.

Algorithm 6 Preconditioned CG

Input: spd matrix A, vector b, initial guess x_0, preconditioner P
Output: approximate solution $x_i \approx A^{-1}b$, low-rank $C_i \approx A^{-1}$

```
procedure CG($A, b, x_0, P$)
    while $\|r_i\|_2 > \max(\delta_{\text{rtol}}\|b\|_2, \delta_{\text{atol}})$ do
        $r_{i-1} \leftarrow b - Ax_{i-1}$                                        // Residual
        $s_i \leftarrow P^{-1} r_{i-1}$                                          // Action
        $\alpha_i \leftarrow s_i^\top r_{i-1} = r_{i-1}^\top (P^{-\top}Ax - P^{-\top}b)$   // Obs.
        $d_i \leftarrow (I - C_{i-1}A)s_i$                                        // Search direction
        $\eta_i \leftarrow s_i^\top A d_i = d_i^\top A d_i$                        // Norm. constant
        $C_i \leftarrow C_{i-1} + \frac{1}{\eta_i} d_i d_i^\top$                   // Inverse estimate
        $x_i \leftarrow x_{i-1} + \frac{\alpha_i}{\eta_i} d_i$                     // Solution estimate
    end while
    return $x_i, C_i$
end procedure
```



Low-rank-plus-diagonal preconditioner:

$$\hat{K} \approx \hat{P}_\ell := P_\ell + \sigma^2 I = \textsc{Cholesky}(K, \text{rank} = \ell) + \sigma^2 I$$

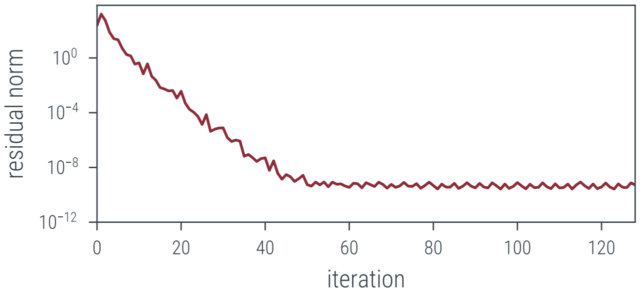
Memory: $\mathcal{O}(n\ell)$

Inverse via matrix inv. lemma: $\mathcal{O}(n\ell^2)$

Large-scale Linear Solve

Solving a large-scale linear system ($n = 100000$) with preconditioned CG.

Matrix size: $n = 10^5$ Preconditioner: Cholesky($\ell = 20$) Time $\approx (1.5 + 6)$ min (Intel i7, 32GB RAM)



Note: At runtime track residual norm $\|r_i\|_2 = \|A(x - x_i)\|_2 = \|x - x_i\|_{A^\top A}$ since $\|x - x_i\|_A$ is unavailable.

Preconditioning can significantly accelerate a CG solve. Precomputation cost amortizes across solves.

What about hyperparameter optimization?
Stochastic Trace Estimation

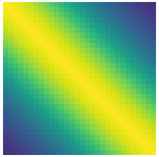
Hyperparameter Optimization via Iterative Methods

How can we compute the quantities necessary for hyperparameter optimization?

Goal: Find kernel hyperparameters $\boldsymbol{\theta}$, which maximize log-marginal likelihood $\mathcal{L}(\boldsymbol{\theta})$. $\to$ gradient-based hyperparameter optimization

Need to: Evaluate log-marginal likelihood and its derivative repeatedly.

- log-marginal likelihood $\mathcal{L}(\boldsymbol{\theta}) = -\frac{1}{2}(y-\boldsymbol{\mu})^\intercal \hat{K}^{-1}(y-\boldsymbol{\mu}) - \frac{1}{2}\log\det(\hat{K}) - \frac{n}{2}\log(2\pi)$
- derivative $\frac{\partial}{\partial\theta}\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2}(y-\boldsymbol{\mu})^\intercal \hat{K}^{-1}\frac{\partial\hat{K}}{\partial\theta}\hat{K}^{-1}(y-\boldsymbol{\mu}) - \frac{1}{2}\operatorname{tr}(\hat{K}^{-1}\frac{\partial\hat{K}}{\partial\theta})$

$\hat{K} =$ 

$n \times n$

Challenge: Computationally costly operations with the kernel matrix.

- linear solves $v \mapsto \hat{K}^{-1}v \to$ iterative methods
- matrix traces $\log\det(\hat{K}) = \operatorname{tr}(\log(\hat{K}))$ and $\operatorname{tr}(\hat{K}^{-1}\frac{\partial\hat{K}}{\partial\boldsymbol{\theta}_i})$

Can we also compute matrix traces via matrix-vector multiplication?

Matrix Trace Estimation

Computing traces of large matrices via matrix-vector multiplication.

Definition: Trace of a matrix

$$\operatorname{tr}(A) = \sum_{i=1}^{n} A_{ii} = \sum_{i=1}^{n} e_i^\top A e_i = \sum_{i=1}^{n} \lambda_i(A)$$

Problem: Can only afford $\ell \ll n$ matrix-vector multiplies.

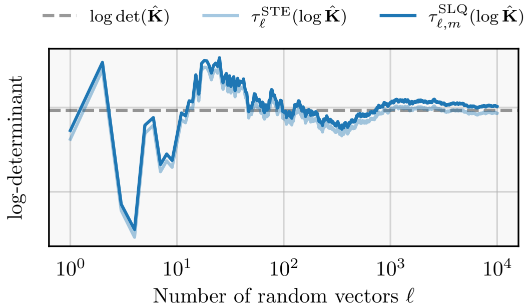
Observation: For orthogonal $Z \in \mathbb{R}^{n \times n}$, it holds that $\operatorname{tr}(A) = \operatorname{tr}(AZZ^\top) = \operatorname{tr}(Z^\top AZ) = \sum_{i=1}^{n} z_i^\top A z_i$.

Idea: Draw ℓ random vectors z_i, s.t. $\mathbb{E}[z_i] = 0$ and $\operatorname{Cov}(\sqrt{n} z_i) = I$, then

$$\begin{aligned}\operatorname{tr}(A) &= \operatorname{tr}(A \operatorname{Cov}(\sqrt{n} z_i)) = n \operatorname{tr}(A \mathbb{E}[z_i z_i^\top]) = n \operatorname{tr}(\mathbb{E}[A z_i z_i^\top]) \\ &= n \mathbb{E}[\operatorname{tr}(A z_i z_i^\top)] = n \mathbb{E}[\operatorname{tr}(z_i^\top A z_i)] = n \mathbb{E}[z_i^\top A z_i] \approx \frac{n}{\ell} \sum_{i=1}^{\ell} z_i^\top A z_i\end{aligned}$$

Stochastic Trace Estimation

Computing matrix traces $\operatorname{tr}(f(\hat{K}))$ via matrix-vector multiplication (Ubaru et al., 2017).



$$
\begin{aligned}
\operatorname{tr}(f(\hat{K})) &= n\mathbb{E}[z_i^\top f(\hat{K}) z_i] \\
&\approx \tau_\ell^{\mathrm{STE}}(f(\hat{K})) = \frac{n}{\ell}\sum_{i=1}^{\ell} z_i^\top f(\hat{K}) z_i \\
&\approx \tau_{\ell,m}^{\mathrm{SLQ}}(f(\hat{K}))
\end{aligned}
$$

Problems:

- Worst-case convergence in the number of random vectors is $\mathcal{O}(\ell^{-\frac{1}{2}})$
- Introduces stochasticity into hyperparameter optimization

$\implies$ slows down training

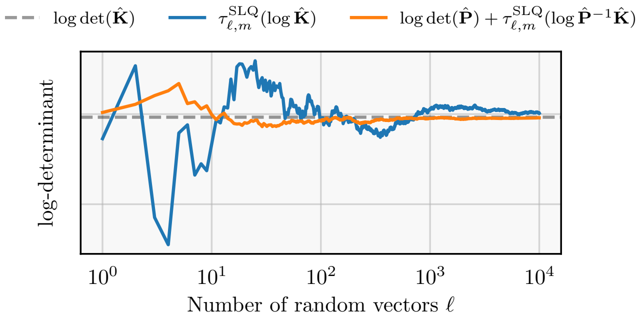
Preconditioned Log-Determinant Estimation

Wenger, Pleiss, Hennig, Cunningham, Gardner. *Preconditioning for Scalable Gaussian Process Hyperparameter Optimization*, ICML, 2022

Idea: Decompose log-determinant into deterministic and stochastic approximation.

$$\log\det(\hat{K}) = \log\det(\hat{P}_\ell \hat{P}_\ell^{-1}\hat{K}) = \underbrace{\log\det(\hat{P}_\ell)}_{\text{known}} + \underbrace{\operatorname{tr}(\log(\hat{K}) - \log(\hat{P}_\ell))}_{\approx \text{ stochastic trace estimate}}$$

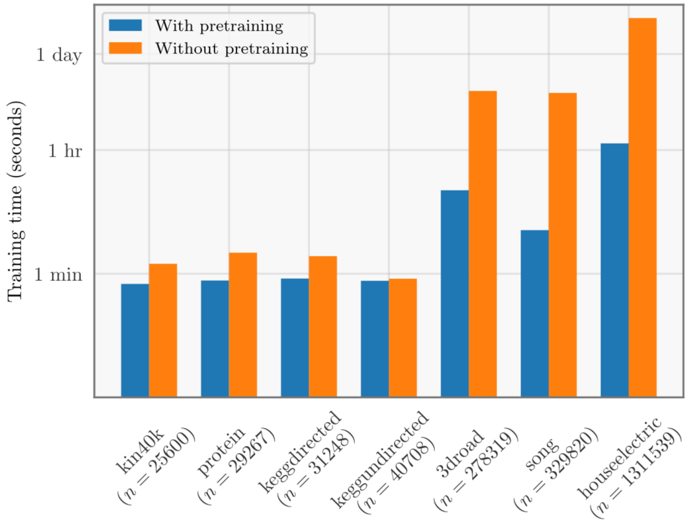
The better the preconditioner, the smaller the stochastic approximation $\Rightarrow$ variance reduction



- Backward pass analogously via automatic differentiation.
- If we compute a preconditioner for CG, we can simply reuse it at negligible overhead.
- If $\hat{P}_\ell \to \hat{K}$ at rate $g(\ell)$, then the STE only requires $\mathcal{O}(\ell^{-\frac{1}{2}} g(\ell))$ random vectors.

CG-based GP inference on Large-Scale Data with GPyTorch

Wang, Pleiss, Gardner, Tyree, Weinberger, Wilson. *Exact Gaussian Processes on a Million Data Points*, NeurIPS, 2019



Observations

- Iterative linear solvers are learning algorithms for the kernel matrix inverse.
- The solver actions significantly affect convergence speed.
- Choosing solver actions can be interpreted as active learning.
- Convergence can be improved through preconditioning, which is a form of prior information.

Fast numerical algorithms for Gaussian processes need “domain expertise”.

Can we approximate in linear time $\mathcal{O}(i^2 n)$?

Sparse Gaussian Processes

Stochastic Variational Gaussian Processes

Titsias. *Variational learning of inducing variables in sparse Gaussian processes*, AISTATS, 2009.
Hensman, Fusi, Lawrence. *Gaussian Processes for Big Data*, UAI, 2013

Observation: Datasets often contain similar data.
$\rightarrow$ Summarize training data via inducing inputs $Z \in \mathbb{R}^{n \times i}$.

Idea: Instead of approximating the quantities needed for inference, approximate posterior directly.

Define variational family $q_{Z,\boldsymbol{\mu},\boldsymbol{\Sigma}} \sim \mathcal{GP}(\mu_Z, k_Z)$, where

$$\begin{aligned} \mu_Z(x) &= k(x,Z)k(Z,Z)^{-1}\boldsymbol{\mu} \\ k_Z(x_0,x_1) &= k(x_0,x_1) - k(x_0,Z)k(Z,Z)^{-1}k(Z,x_1) + \underbrace{k(x_0,Z)k(Z,Z)^{-1}\boldsymbol{\Sigma}k(Z,Z)^{-1}k(Z,x_1)}_{\text{correction term}} \end{aligned}$$

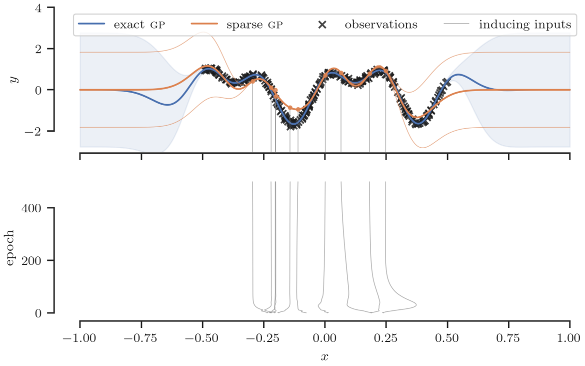
and optimize parameters $(Z, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ by minimizing objective $D_{\mathrm{KL}}(q_{Z,\boldsymbol{\mu},\boldsymbol{\Sigma}} || f_{\text{posterior}})$.

Computational complexity: $\mathcal{O}(i^2 n)$

Stochastic Variational Gaussian Processes

Titsias. *Variational learning of inducing variables in sparse Gaussian processes*, AISTATS, 2009.
Hensman, Fusi, Lawrence. *Gaussian Processes for Big Data*, UAI, 2013

Idea: Instead of approximating the quantities needed for inference, approximate posterior directly.



Source: https://tiao.io/post/sparse-variational-gaussian-processes/

Can we design a method where we can trust the UQ *no matter how much computation we've done*?

Summary

- **Scaling GPs to large datasets requires approximation.**
- Iterative methods enable posterior approximation and hyperparameter optimization in $\mathcal{O}(n^2)$.
- **Iterative methods** are **active learning algorithms**.
- Preconditioning, i.e. prior information, accelerates convergence.
- Sparse GP approximations enable inference in $\mathcal{O}(n)$ at the expense of uncertainty quantification.

Please cite this course, as

```
@techreport{NoML22,
     title = {Numerics of Machine Learning},
     author = {N. Bosch and J. Grosse
     and P. Hennig and A. Kristiadi
     and M. Pförtner and J. Schmidt
     and F. Schneider and L. Tatzel
     and J. Wenger},
     series = {Lecture Notes in Machine Learning},
     year = {2022},
     institution = {Tübingen AI Center},
}
```

Next week: A probabilistic view on iterative GP approximation.