UNIVERSITÀ DEGLI STUDI FIRENZE

DISIA
DIPARTIMENTO DI STATISTICA
INFORMATICA, APPLICAZIONI
“GIUSEPPE PARENTI”

A NEW PROGRAMMING INTERFACE FOR GAUSSIAN PROCESS REGRESSION

Giacomo Petrillo, University of Florence
Department of Statistics, Computer Science and Applications (DiSIA)
April 8, 2022

Contents

A concrete problem: fitting parton distribution functions

Parton distribution functions (PDFs)

- Functions $f : (0,1) \to \mathbb{R}$
- Used in particle Physics to characterize protons
- Sort of a velocity distribution of quarks inside the proton
- Must be obtained from **indirect** data
- **No parametric form**

(Thanks to Alessandro Candido from the NNPDF group in Milan for this example)

The fitting task
(simplified version)

- Eight unknown functions: $f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8 : (0,1) \to \mathbb{R}$
- Constraint #1: $\displaystyle\sum_{i=1}^{8} \int_0^1 f_i(x)\,\mathrm{d}x = 1$
- Constraint #2: $\displaystyle\sum_{i=1}^{8} \int_0^1 x \cdot f_i(x)\,\mathrm{d}x = 1$
- Data #1 (linear link): $\mathbf{y}^{(1)} = \displaystyle\sum_{i=1}^{8} M_i^{(1)} f_i(\mathbf{x}_i^{(1)}) + \boldsymbol{\epsilon}^{(1)}$
- Data #2 (quadratic link): $\mathbf{y}^{(2)} = \displaystyle\sum_{i=1}^{8} f_i(\mathbf{x}_i^{(2)})^\top M_i^{(2)} f_i(\mathbf{x}_i^{(2)}) + \boldsymbol{\epsilon}^{(2)}$

Problems

- Good uncertainty quantification required
- Currently solved with a Monte Carlo of neural networks— computationally burdensome, hinders progress
- How to impose the integral constraints efficiently?
- How to keep into account all the correlations without sampling?
- How to regularize i.e. avoid functions which make no sense?

Solution
(on paper)

- Adopt Bayesian inference (easier uncertainty quantification)
- $\Rightarrow$ Must define an **a priori probability distribution** on the space of functions on $(0,1)$
- Then we obtain an **a posteriori distribution** with Bayes' theorem plugging data and constraints
- $\{f : (0,1) \to \mathbb{R}\}$ is a real vector space, so we can use a multivariate Normal as prior
- Infinite dimensional multivariate Normals are **Gaussian processes**

- I won't actually show you the solution (I don't have the data)
- Just remember this is the kind of problem that the program is designed to make easy

Gaussian processes

Gaussian process

Definition

- A (zero-mean) Normal distribution is characterized by its **covariance matrix** V
- $p(\mathbf{y}) \propto \exp\left(-\frac{1}{2}\mathbf{y}^\top V^{-1}\mathbf{y}\right)$
- $V_{ij} = \mathrm{Cov}[y_i, y_j]$
- In an infinite-dimensional space, this is called **covariance function** or **kernel**:
- $k(x, x') \equiv \mathrm{Cov}[f(x), f(x')]$

Gaussian process

Properties

- Normality is preserved under marginalization
- i.e., looking only at a certain subvector, it's still a Normal distribution
- The covariance matrix is the corresponding submatrix
- $\Rightarrow$ even if the space is infinite-dimensional, we need to **compute the kernel only on the finite set of points we actually use**
- Normality is preserved by linear transformations
- $\mathrm{Cov}[A\mathbf{y}] = AVA^\top$

Gaussian process

Inference (1/2)

- Inference means we observe some values of the function, and we want the probability distribution of other unseen values.
- We have $\mathbf{y} \equiv f(\mathbf{x})$
- We want $\mathbf{y}^* \equiv f(\mathbf{x}^*)$
- Thus we want the conditional probability $p(\mathbf{y}^* \mid \mathbf{y})$
- Normality is preserved by conditioning
- So $p(\mathbf{y}^* \mid \mathbf{y})$ is still Normal

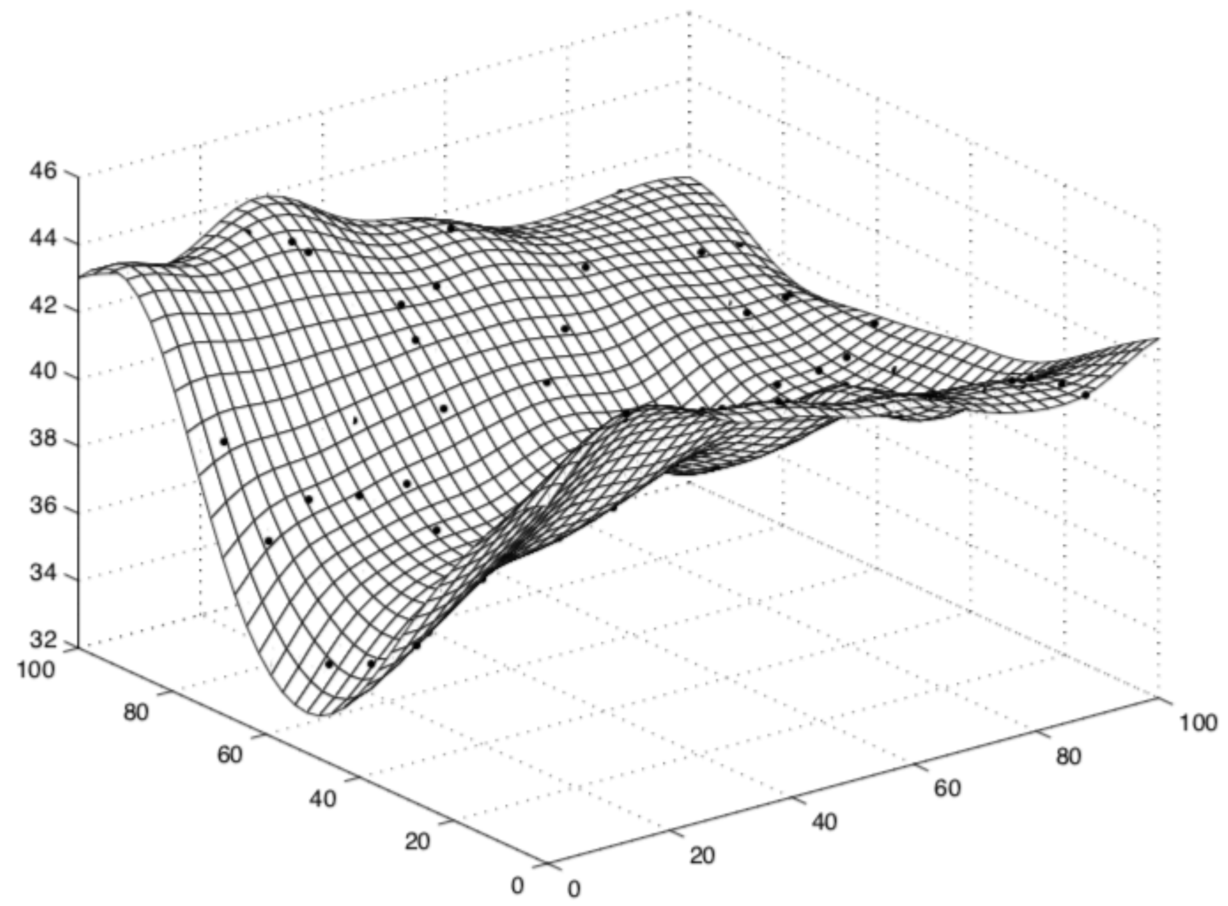
Gaussian process

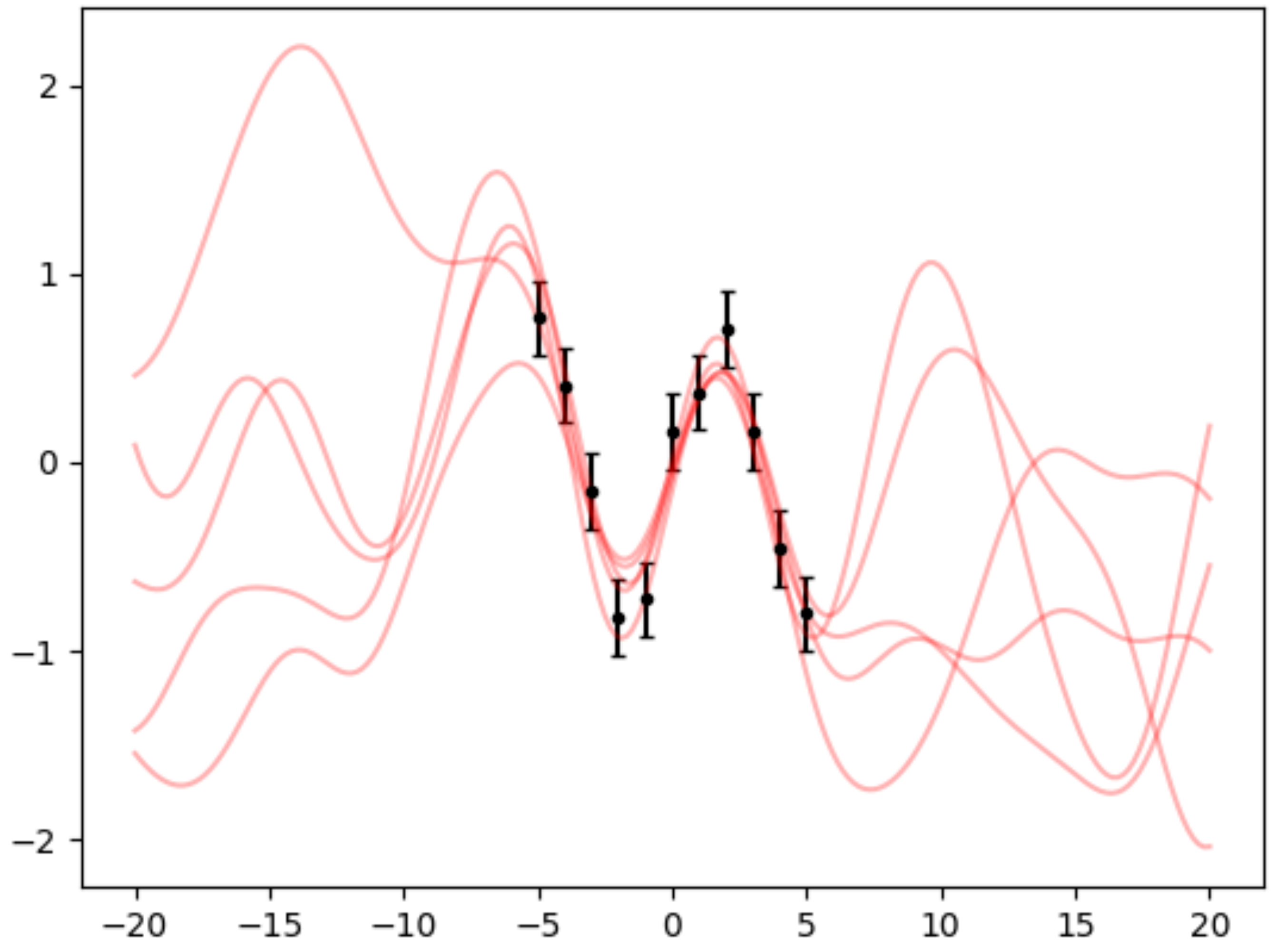
Inference (2/2)

- $p(\mathbf{y}^* \mid \mathbf{y})$ is Normal $\Rightarrow$ we compute its mean and covariance matrix
- Consider the covariance matrix of the joint vector $(\mathbf{y}, \mathbf{y}^*)$ in block form
- $\begin{pmatrix} \mathrm{Cov}[\mathbf{y}] & \mathrm{Cov}[\mathbf{y}, \mathbf{y}^*] \\ \mathrm{Cov}[\mathbf{y}^*, \mathbf{y}] & \mathrm{Cov}[\mathbf{y}^*] \end{pmatrix} = \begin{pmatrix} V_{yy} & V_{yy^*} \\ V_{yy^*}^\top & V_{y^*y^*} \end{pmatrix}$
- Mean: $E[\mathbf{y}^* \mid \mathbf{y}] = V_{y^*y} V_{yy}^{-1} \mathbf{y}$
- Covariance: $\mathrm{Cov}[\mathbf{y}^* \mid \mathbf{y}] = V_{y^*y^*} - V_{yy^*}^\top V_{yy}^{-1} V_{yy^*}$ (Schur complement of V_{yy})

Gaussian process

Result





Gaussian process

Computational aspects

- As highlighted, we have to invert the covariance matrix: V_{yy}^{-1}
- Let n be the length of $\mathbf{y}$, i.e., number of data points
- Computing V_{yy} is $O(n^2)$ (evaluate $k(x, x')$ on all pairs)
- Inverting (decomposing) V_{yy} is $O(n^3)$
- $O(n^3)$ is the **bottleneck**, #datapoints must be < 1000-5000

Gaussian process

Algorithm

- Structure of a Gaussian process program:
- Input: the kernel function $k(x, x')$
- Input: the points $\mathbf{x}$, $\mathbf{x}^*$
- Input: the data $\mathbf{y}$
- Step 1: build the covariance matrix of $(\mathbf{y}, \mathbf{y}^*)$
- Step 2: decompose the covariance matrix
- Output: $E[\mathbf{y}^* \mid \mathbf{y}]$ and $\mathrm{Cov}[\mathbf{y}^* \mid \mathbf{y}]$

Gaussian process

User interface (1/3)

- In the example we had:
  - 8 functions $f_1, \ldots, f_8$
  - sum/integral constraints $\sum_{i=1}^{8} \int \ldots = 1$
  - function → data mappings $M^{(1)}$, $M^{(2)}$
- How do you specify $k(x, x')$, $\mathbf{x}$ and $\mathbf{x}^*$ in this case?

Gaussian process

User interface (2/3)

- Eight functions is equivalent to one function with an additional input: $f_i(x) \equiv f(x, i)$

- Integrals are **linear transformations** in the space of functions, so still part of the Gaussian process

- Finite linear transformation as well like $M^{(1)}$

- $M^{(2)}$ is nonlinear, can be done but won't explain now

Gaussian process

User interface (3/3)

- The user wants to talk in terms of individual functions, integrals, transformations...
- ...The program behind the scenes must build this:

$$\left(\begin{array}{c|c|c|c} V_{\text{integ-integ}} & V_{\text{integ-M1}} & V_{\text{integ-M2}} & V_{\text{integ-}x^*} \\ \hline V_{\text{integ-M1}}^\top & V_{\text{M1-M1}} & V_{\text{M1-integ}} & V_{\text{M1-}x^*} \\ \hline V_{\text{integ-M2}}^\top & V_{\text{M1-M2}}^\top & V_{\text{M2-M2}} & V_{\text{M2-}x^*} \\ \hline V_{\text{integ-}x^*}^\top & V_{\text{M1-}x^*}^\top & V_{\text{M2-}x^*}^\top & V_{x^* x^*} \end{array}\right)$$

- Where the terms are $V_{\text{M1-}x^*} = M^{(1)} V_{x^{(1)} x^*}$,

$$V_{\text{M1-M2}} = M^{(1)} V_{x^{(1)} x^{(2)}} \tilde{M}^{(2)}\text{, etc.}$$

The program

The program

lsqfitgp

- A Python module (relies on functionalities too advanced for R)
- The core functionality is complete
- Install: `$ pip install lsqfitgp`
- Manual: https://lsqfitgp.readthedocs.io
- Released as open source
- The idea for the interface is taken from **lsqfit**, a program by G. P. Lepage, a theoretical Physicist at Cornell

The program

Features

Done:

- Finite transformations
- Derivatives/integrals
- Nonlinear finite transformations
- Interface

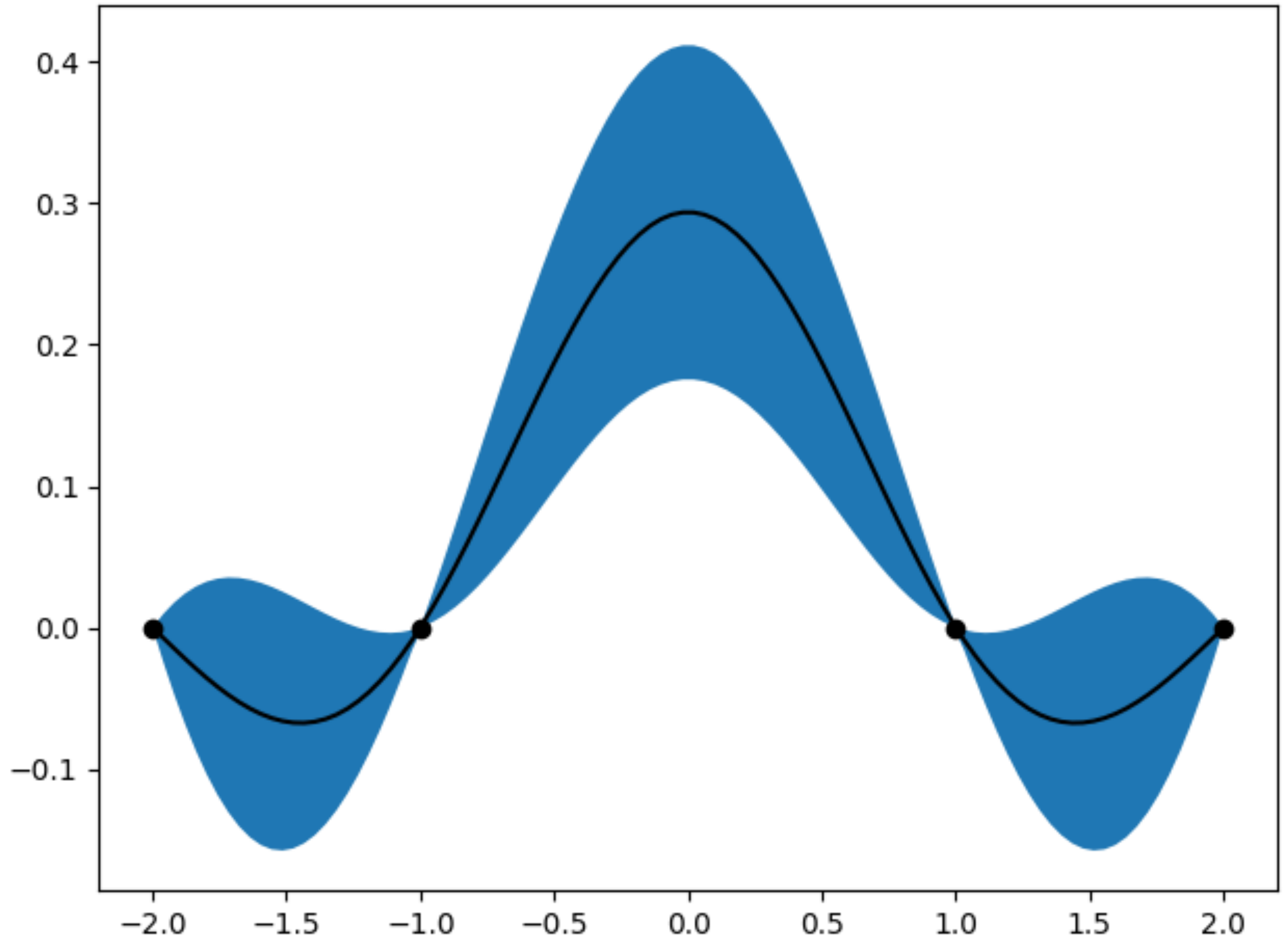
To do:

- Other infinite transformations (Fourier, Taylor)
- Fast decomposition of V_{yy}

Example 1

Constraint: local maximum in 0

```python
import lsqfitgp as lgp
import numpy as np
import gvar

gp = lgp.GP(lgp.ExpQuad())

gp.addx([-2, -1, 1, 2], 'data')
gp.addx(0, 'first', deriv=1)
gp.addx(0, 'second', deriv=2)

xplot = np.linspace(-2, 2, 100)
gp.addx(xplot, 'plot')

yplot = gp.predfromdata({
    'data'  : [0, 0, 0, 0],
    'first' : 0,
    'second': gvar.gvar(-1, 0.3) # -1 +/- 0.3
}, 'plot')
```



```
yplot = array([-3(60)e-17, -0.007(17), -0.015(33), ...])
```

Example 1

lsqfitgp vs. pymc3

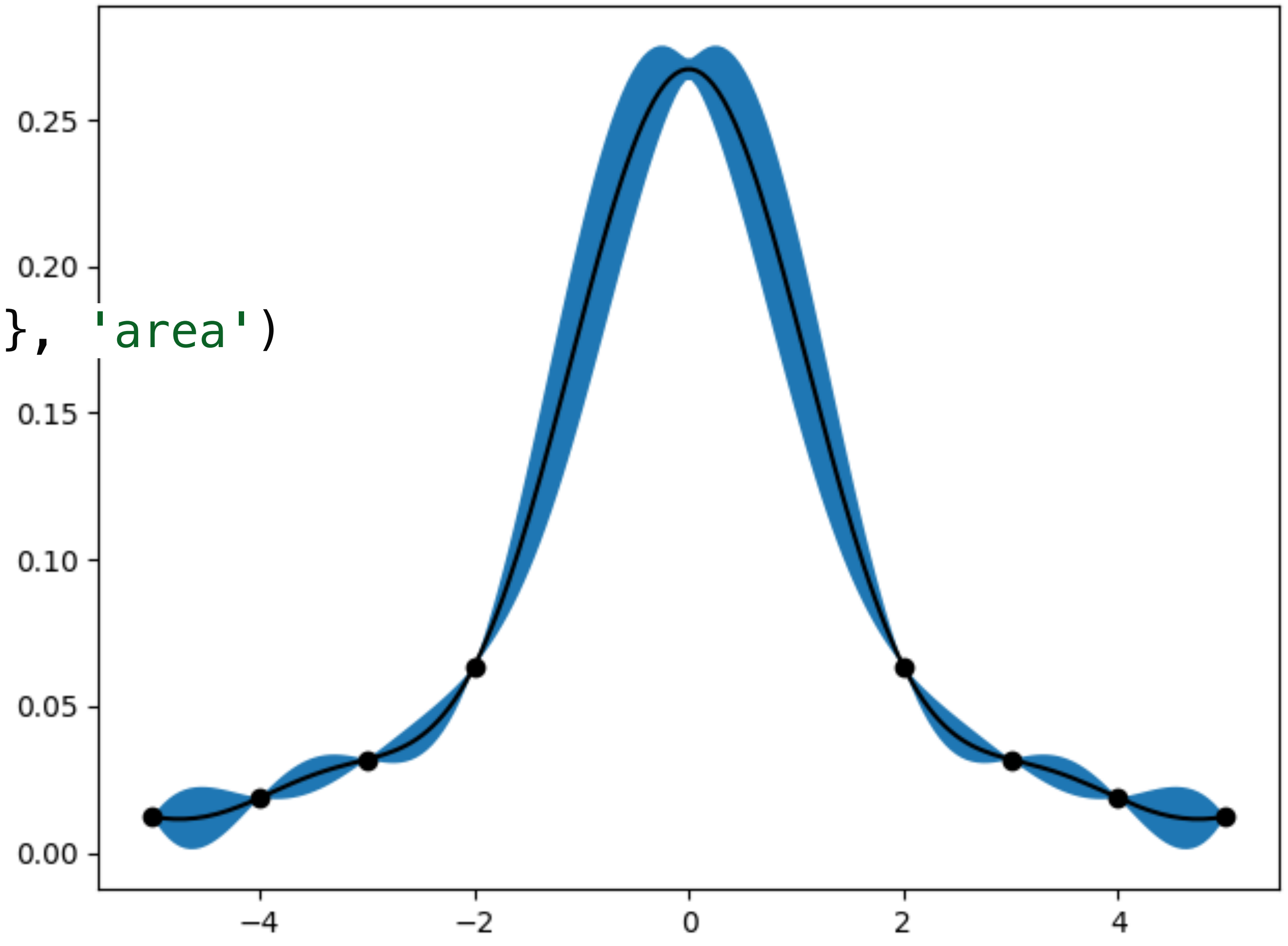
```python
import lsqfitgp as lgp
import numpy as np
import gvar

gp = lgp.GP(lgp.ExpQuad())

gp.addx([-2, -1, 1, 2], 'data')
gp.addx(0, 'first', deriv=1)
gp.addx(0, 'second', deriv=2)

xplot = np.linspace(-2, 2, 100)
gp.addx(xplot, 'plot')

yplot = gp.predfromdata({
    'data'  : [0, 0, 0, 0],
    'first' : 0,
    'second': gvar.gvar(-1, 0.3) # -1 +/- 0.3
}, 'plot')
```

```python
import pymc3 as pm
import numpy as np
import theano.tensor as tt
from matplotlib import pyplot as plt

Xysigma = np.array([
    # x, deriv,  y, sigma
    [-2,     0,  0,   0  ],
    [-1,     0,  0,   0  ],
    [ 1,     0,  0,   0  ],
    [ 2,     0,  0,   0  ],
    [ 0,     1,  0,   0  ],
    [ 0,     2, -1,   0.3],
])

X = Xysigma[:, :2]
y = Xysigma[:, 2]
sigma = Xysigma[:, 3]

Xplot = np.stack([
    np.linspace(-2, 2, 100),
    np.zeros(100),
], axis=1)

# Probabilist's Hermite polynomials
def H1(x):
    return x
def H2(x):
    return (x - 1) * (x + 1)
def H3(x):
    return x * (x - tt.sqrt(3)) * (x + tt.sqrt(3))
def H4(x):
    return (x ** 2 - 6) * x ** 2 + 3

def expquad00(x, xs):
    return tt.exp(-1/2 * (x - xs) ** 2)
def expquad01(x, xs):
    return H1(x - xs) * expquad00(x, xs)
def expquad02(x, xs):
    return H2(x - xs) * expquad00(x, xs)
def expquad11(x, xs):
    return -H2(x - xs) * expquad00(x, xs)
def expquad12(x, xs):
    return -H3(x - xs) * expquad00(x, xs)
def expquad22(x, xs):
    return H4(x - xs) * expquad00(x, xs)

class MyKernel(pm.gp.cov.Covariance):

    def __init__(self):
        super(MyKernel, self).__init__(2)

    def diag(self, X):
        return tt.choose(
            X[:, 1].astype(int),
            [
                expquad00(X[:, 0], X[:, 0]),
                expquad11(X[:, 0], X[:, 0]),
                expquad22(X[:, 0], X[:, 0]),
            ]
        )

    def full(self, X, Xs=None):
        if Xs is None:
            Xs = X
        return tt.choose(
            (3 * X[:, None, 1] + Xs[None, :, 1]).astype(int),
            [
                expquad00(X [:, None, 0], Xs[None, :, 0]),
                expquad01(X [:, None, 0], Xs[None, :, 0]),
                expquad02(X [:, None, 0], Xs[None, :, 0]),
                expquad01(Xs[None, :, 0], X [:, None, 0]),
                expquad11(X [:, None, 0], Xs[None, :, 0]),
                expquad12(X [:, None, 0], Xs[None, :, 0]),
                expquad02(Xs[None, :, 0], X [:, None, 0]),
                expquad12(Xs[None, :, 0], X [:, None, 0]),
                expquad22(X [:, None, 0], Xs[None, :, 0]),
            ]
        )

with pm.Model() as model:
    cov_func = MyKernel()
    gp = pm.gp.Marginal(cov_func=cov_func)
    y_data = gp.marginal_likelihood('y_data', X=X, y=y, noise=sigma)

mu, var = gp.predict(Xplot, point=[], diag=True)
```

Example 2

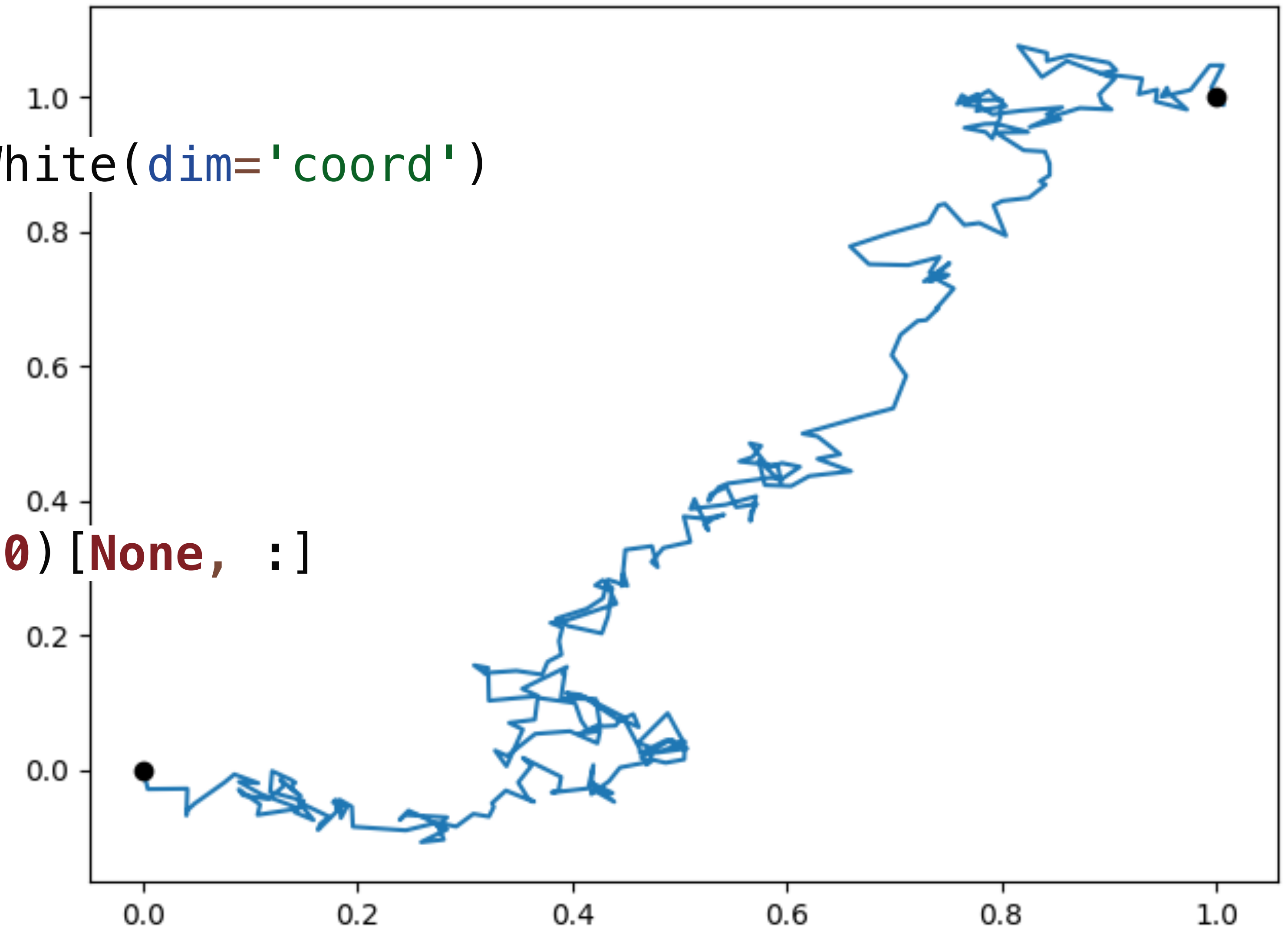
Constrained area

```
function = lambda x: 1/np.pi * 1/(1 + x**2)

x = np.array([-5, -4, -3, -2, 2, 3, 4, 5])
y = function(x)

gp = lgp.GP(lgp.ExpQuad(scale=2))

gp.addx(x, 'datapoints', deriv=1)
gp.addx(-5, 'left')
gp.addx(5, 'right')
gp.addtransf({'left': -1, 'right': 1}, 'area')

xplot = np.linspace(-5, 5, 200)
gp.addx(xplot, 'plot', deriv=1)

yplot = gp.predfromdata({
    'datapoints': y,
    'area'      : 0.87,
}, 'plot')
```



Example 3

Multioutput



```
k = lgp.Wiener(dim='time') * lgp.White(dim='coord')
gp = lgp.GP(k)

x = np.empty((2, 300), dtype=[
    ('time' , float),
    ('coord', int  ),
])
x['time'] = np.linspace(0, 0.1, 300)[None, :]
x['coord'] = np.arange(2)[:, None]
gp.addx(x, 'walk')

end = np.empty(2, dtype=x.dtype)
end['time'] = np.max(x['time'])
end['coord'] = np.arange(2)
gp.addx(end, 'endpoint')

path = gp.predfromdata({'endpoint': [1, 1]}, 'walk')

x, y = next(gvar.raninter(path))
```

Numerical linear algebra TODOs

Decoming the V matrix

(inefficiently)

- V is positive semi-definite
- V is often very degenerate (numerically)
- Quite robust general solution: diagonalize V, "pump" or "cut" low eigenvalues—$O(n^3)$
- Faster: estimate max eigenvalue with Gershgorin, add epsilon to the diagonal, do Cholesky—$O(n^3)$
- $O(n^2)$ but slow: low-rank approximation with iterative method
- These things are implemented

Decomposing the V matrix
(efficiently)

- Two main routes: **approximate** algorithms, **exact** algorithms that work only for special matrices

- Approximate:
    - Assume a sparsity structure for the inverse with a DAG
    - Hierarchical (not very successful)

- Exact:
    - V is Toeplitz: $O(n \log^2 n)$
    - Sparse V
    - V is a Kronecker product
    - V is semiseparable: $O(n)$
    - V^{-1} is exactly sparse (Markov process)

These things are missing but I mostly know how to do them

Decomposing the V matrix

One block at a time (1/3)

- Often the V matrix has a natural block form suggested by the problem:

$$\left(\begin{array}{c|c|c} V_{11} & V_{12} & V_{13} \\ \hline V_{12}^\top & V_{22} & V_{23} \\ \hline V_{13}^\top & V_{23}^\top & V_{33} \end{array}\right)$$

- What can we do if V_11 is, say, Toeplitz, but the rest is not?

- ⇒ Blockwise Gaussian elimination (M = Schur complement of A)

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)^{-1} = \left(\begin{array}{c|c} A^{-1} + A^{-1}BM^{-1}CA^{-1} & -A^{-1}BM^{-1} \\ \hline -M^{-1}CA^{-1} & M^{-1} \end{array}\right)$$

Decomposing the V matrix

One block at a time (2/3)

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)^{-1} = \left(\begin{array}{c|c} A^{-1} + A^{-1}BM^{-1}CA^{-1} & -A^{-1}BM^{-1} \\ \hline -M^{-1}CA^{-1} & M^{-1} \end{array}\right)$$

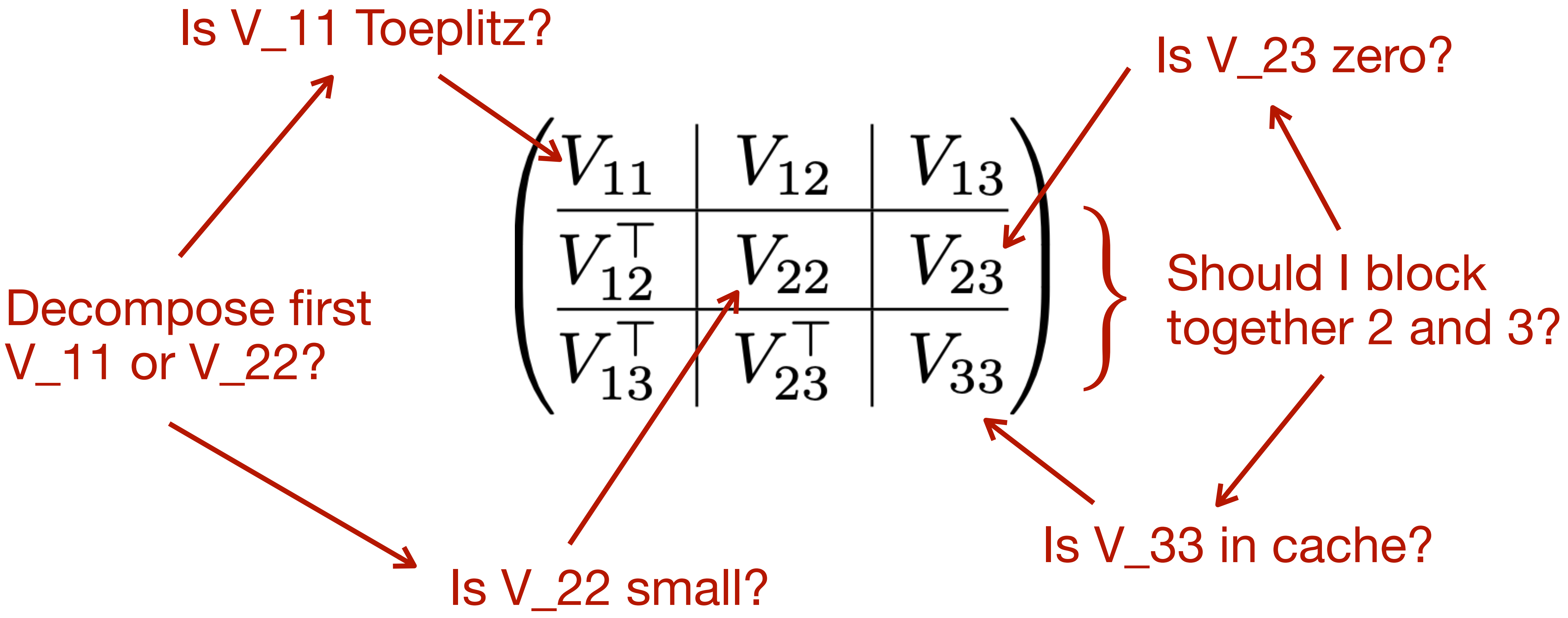
$$\left(\begin{array}{c|c|c} V_{11} & V_{12} & V_{13} \\ \hline V_{12}^\top & V_{22} & V_{23} \\ \hline V_{13}^\top & V_{23}^\top & V_{33} \end{array}\right) \qquad M = D - CA^{-1}B$$

- First decompose V_11 with fast algorithm, then do the rest
- Other use case: cache decomposition to add rows later (Bayesian optimization: inference->new data->inference->new data...)

Decomposing the V matrix

One block at a time (3/3)

The strategy should be decided automatically by the program



Is V_11 Toeplitz?
Is V_23 zero?
Decompose first V_11 or V_22?
Should I block together 2 and 3?
Is V_22 small?
Is V_33 in cache?

$$\left(\begin{array}{c|c|c} V_{11} & V_{12} & V_{13} \\ \hline V_{12}^\top & V_{22} & V_{23} \\ \hline V_{13}^\top & V_{23}^\top & V_{33} \end{array}\right)$$

Thanks for the attention

Try it:

```
$ pip install lsqfitgp
```