PML Theory

Sam Laing 6283670
Albert Catalan 6443478

a) We have

$$p(y|X) = \int_{\forall f} p(y|f,X)\, p(f|X)\, df$$

we have $p(f) \sim \mathcal{N}(\mu, k) \Rightarrow p(f|X) = \mathcal{N}(f; \mu_x, K_{xx})$

$$\Rightarrow p(y|X) = \int_{\forall f} \mathcal{N}(y; f(X), \Lambda)\, \mathcal{N}(f; \mu_x, K_{xx})\, df$$

$$= \int_{\forall f} \mathcal{N}(y; \mu_x, \Lambda + K_{xx})\, \mathcal{N}\left(f; (\Lambda^{-1} + K_{xx})^{-1}(\Lambda^{-1} f(x) + \cancel{K_{xx}^{-1}}), (\Lambda^{-1} + K_{xx}^{-1})^{-1}\right) df$$

By the Result proved in last week's hw.

$$= \mathcal{N}(y; \mu_x, \Lambda + K_{xx}) \int_{\forall f} \mathcal{N}\left(f; (\Lambda^{-1} + K_{xx}^{-1})^{-1}(\Lambda^{-1} f(x) + K_{xx}^{-1}\mu_x), (\Lambda^{-1} + K_{xx}^{-1})^{-1}\right) df$$

$$= \mathcal{N}(y; \mu_x, \Lambda + K_{xx}) \cdot 1$$

(since we integrate over a distribution)

$\boxed{//}$

b) i) $y \mid x \sim N(\mu_x, \Lambda + K_{xx})$

$$\mathbb{E}_{y|x}\left[\log(p(y|x))\right]$$

$$= -\frac{1}{2}\left\{\mathbb{E}_{y|x}\left[(y-\mu_x)^T(K_{xx}+\Lambda)^{-1}(y-\mu_x)\right] + \mathbb{E}_{y|x}\left[\log\det(K_{xx}+\Lambda)\right] + n\log 2\pi\right\}$$

doesn't depend on y

$$= -\frac{1}{2}\operatorname{tr}\left[(K_{xx}+\Lambda)^{-1}(K_{xx}+\Lambda)\right]$$

By suggested identity

$$-\frac{1}{2}(\mu_x-\mu_x)^T(K_{xx}+\Lambda)^{-1}(\mu_x-\mu_x)$$

$$-\frac{1}{2}\log\det(K_{xx}+\Lambda) - n\log 2\pi$$

$$= -\frac{1}{2}\operatorname{tr}(\mathbb{1}) + 0 - \frac{1}{2}\log\det(K_{xx}+\Lambda) - \frac{n}{2}\log 2\pi$$

$$= -\frac{n}{2}(1+\log 2\pi) - \frac{1}{2}\log\det(K_{xx}+\Lambda)$$

(b) (iii) $\qquad K_{xx} = K_{xx}(\theta)$

$$\log p(y|X) = -\frac{1}{2}\Big[(y-\mu_x)^T (K_{xx}+\Lambda^{-1})^{-1}(y-\mu_x) + \log\det(K_{xx}+\Lambda^{-1}) + n\log 2\pi\Big]$$

Lets first prove the result & then prove the required identities:

$$\partial_\theta \log p(y|X) = \frac{1}{p(y|X)}\,\partial_\theta\, p(y|X)$$

we note that

$$\partial_\theta\Big[(y-\mu_x)^T(K_{xx}+\Lambda)^{-1}(y-\mu_x)\Big]$$

Chain rule

$$= \left(\partial_\theta (y-\mu_x)^T\right)(K_{xx}+\Lambda)^{-1}(y-\mu_x) + (y-\mu_x)^T\,\partial_\theta (K_{xx}+\Lambda)^{-1}(y-\mu_x)$$

$$+ (y-\mu_x)(K_{xx}+\Lambda)^{-1}\partial_\theta (y-\mu_x)$$

$$= (y-\mu_x)^T\,\partial_\theta (K_{xx}+\Lambda)^{-1}(y-\mu_x)$$

by result we will prove

$$= -(y-\mu_x)^T\left[(K_{xx}+\Lambda)^{-1}\frac{\partial K_{xx}}{\partial\theta}(K_{xx}+\Lambda)^{-1}\right](y-\mu_x)$$

~~and~~

$$\cancel{A)\ \partial_\theta \log p(y|X) = (y-\mu_x)^T(K_{xx}+\Lambda^{-1})\frac{\partial K_{xx}}{\partial\theta}(K_{xx}+\Lambda^{-1})^{-1}(y-\mu_x)}$$

using 2nd result

$$\Rightarrow \partial_\theta \log p(y|x) \overset{\circ}{=} \frac{1}{p(y|x)}\Big[-(y-\mu_x)^T(K_{xx}+\Lambda)^{-1}\frac{\partial K_{xx}}{\partial\theta}(K_{xx}+\Lambda)^{-1}(y-\mu_x)$$
$$+ \operatorname{tr}\Big(K_{xx}^{-1}\frac{\partial K_{xx}}{\partial\theta}\Big)\Big]$$

We now prove the two statements we made use of above:

•) let $K \equiv K_{xx}(\theta)$ have inverse $H \equiv H_{xx}(\theta)$

Then $KH = \mathbb{1}$. differentiating both sides (and noting that the product rule applies for matrices) gives:

$$(\partial_\theta K)\, H + K\,(\partial_\theta H) = 0$$

$$\Rightarrow \quad K\,(\partial_\theta H) = -(\partial_\theta K)\, H$$

multiplying both sides from the left by $K^{-1} = H$ then gives:

$$\partial_\theta (K_{xx}(\theta))^{-1} = -K_{xx}^{-1}\frac{\partial K_{xx}}{\partial\theta}K_{xx}^{-1} \quad \boxed{\text{Ⅲ}}$$

(this result holds $\forall$ matrices of full rank)

•) We observe that since $K_{xx}(\theta)$ is psd., $\exists$ $U(\theta)$ orthogonal, s.t. $D(\theta)$ positive diagonal such that:

$$K_{xx}(\theta) = U(\theta)\, D(\theta)\, U(\theta)^T$$

So we have $D(\theta) = \operatorname{diag}(\lambda_j(\theta))_{j=1}^n$ for continuous $(\lambda_j : \mathbb{R} \to \mathbb{R})_{j=1}^n$

then $\det(K_{xx}(\theta)) = \prod_{j=1}^{n} \lambda_j(\theta)$

$$\Rightarrow \partial_\theta \log(\det(K_{xx}(\theta))) = \partial_\theta \log \prod_{j=1}^{n} \lambda_j(\theta)$$

$$= \partial_\theta \sum_{j=1}^{n} \log \lambda_j(\theta)$$

$$(*) \quad = \sum_{j=1}^{n} \frac{1}{\lambda_j(\theta)} \lambda_j'(\theta) \quad \text{(product rule)}$$

we observe that $K_{xx}^{-1} = \mathcal{U} \mathcal{D}^{-1}(\theta) \mathcal{U}^T$
and $\mathcal{D}^{-1}(\theta) = \operatorname{diag}(1/\lambda_j(\theta))_{j=1}^{n}$

also $\frac{\partial K_{xx}}{\partial \theta} = \mathcal{U} \operatorname{diag}(\lambda_j'(\theta))_j \mathcal{U}^T$

we therefore recognize the expression $(*)$ as

$$\operatorname{tr}\left(\mathcal{U} \operatorname{diag}(1/\lambda_j(\theta))_{j=1}^{n} \mathcal{U}^T \cdot \mathcal{U} \operatorname{diag}(\lambda_j'(\theta))_{j=1}^{n} \mathcal{U}\right)$$

(since $\mathcal{U}^T = \mathcal{U}^{-1}$)

$$= \operatorname{tr}\left(K_{xx}^{-1} \frac{\partial K_{xx}}{\partial \theta}\right)$$

$\blacksquare$

final6

June 12, 2023

1 Probabilistic Machine Learning

1.1 Exercise Sheet No. 6 — Source Separation

Submission by: * Sam, Laing, Matrikelnummer: 6283670 * Albert, Catalan, Matrikelnummer: 6443478

[1]:
```
import functools

import jax
import jax.numpy as jnp
import numpy as np


jax.config.update("jax_enable_x64", True)
```

[2]:
```
from matplotlib import pyplot as plt
from tueplots import bundles
from tueplots.constants.color import rgb


plt.rcParams.update(bundles.beamer_moml())
plt.rcParams.update({'figure.dpi': 200})
```

1.2 Exercise 6.2 (Coding Exercise)

The lecture covered an extended example, in which a GP was used to model the Keeling curve, i.e. the temporal evolution of the atmospheric CO_2 concentration at Mauna Loa Observatory in Hawaii. This model is useful for separating certain effects in the data like long- and mid-term trends as well as periodicity.

However, since all kernels used for modelling the data are in fact so-called stationary (or translation-invariant) kernels, i.e. $k(t_0, t_1) = k(|t_0 - t_1|)$, the model will not extrapolate well. Rather, its prediction will at some point return to the prior mean, which is just set to a constant.

In this exercise, we will attempt to fix this problem by injecting more prior knowledge into the model. Specifically, we will add a covariate, which we hypothesize to have causal influence on the atmospheric CO_2 concentration. By scrutinizing the behavior of the model on the dataset, we can gain insights into the predictive power of the covariate.

Disclaimer: This is a very crude data analysis, which is merely meant to illustrate how GPs can be augmented with additional input data. It should not be used to make any scientific statement about the dynamics of atmospheric CO_2 concentration.

1.2.1 Data

First, let's load and plot the data.

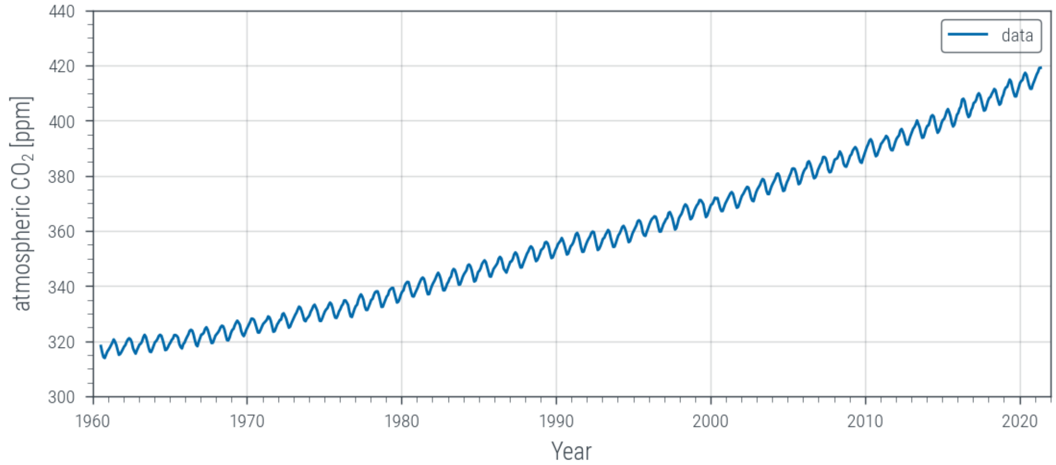
[3]:
```
data = np.load("Ex06_data.npz")
```

[4]:
```
from matplotlib import ticker


def setup_plot(ax):
    ax.set_xlabel("Year")
    ax.xaxis.set_major_locator(ticker.MultipleLocator(10))
    ax.xaxis.set_minor_locator(ticker.MultipleLocator(1))
    ax.set_xlim([1960, 2022])
    ax.grid(which="major", axis="both")


def plot_data(ax):
    ax.plot(
        data["date"],
        data["atmospheric_co2"],
        label="data",
        color=rgb.tue_blue,
    )
    ax.yaxis.set_major_locator(ticker.MultipleLocator(20))
    ax.yaxis.set_minor_locator(ticker.MultipleLocator(5))
    ax.set_ylabel("atmospheric CO$_2$ [ppm]")
    ax.set_ylim([300, 440])


setup_plot(plt.gca())
plot_data(plt.gca())

plt.legend()
```

[4]:
```
<matplotlib.legend.Legend at 0x7f97fd003f10>
```

```
findfont: Font family ['cursive'] not found. Falling back to DejaVu Sans.
findfont: Generic family 'cursive' not found because none of the following
```

```
families were found: Apple Chancery, Textile, Zapf Chancery, Sand, Script MT,
Felipa, Comic Neue, Comic Sans MS, cursive
```

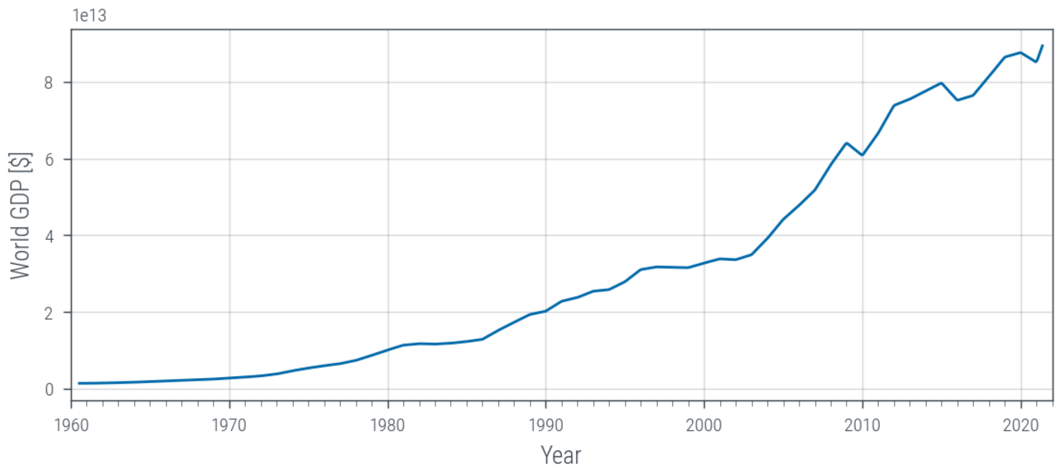


We will work under the hypothesis that the total gross domestic product (GDP) of the world economy has a causal effect on the atmospheric CO2 concentration.

The `data["world_gdp"]` array contains estimates of the world GDP over time. The raw data is made available by the World Bank and has been linearly interpolated to match the temporal resolution of the atmospheric CO_2 data.

[5]:
```python
setup_plot(plt.gca())

plt.plot(
    data["date"],
    data["world_gdp"],
    label="World GDP",
    color=rgb.tue_blue,
)

plt.ylabel("World GDP [$]")
```

[5]:
```
Text(0, 0.5, 'World GDP [$]')
```

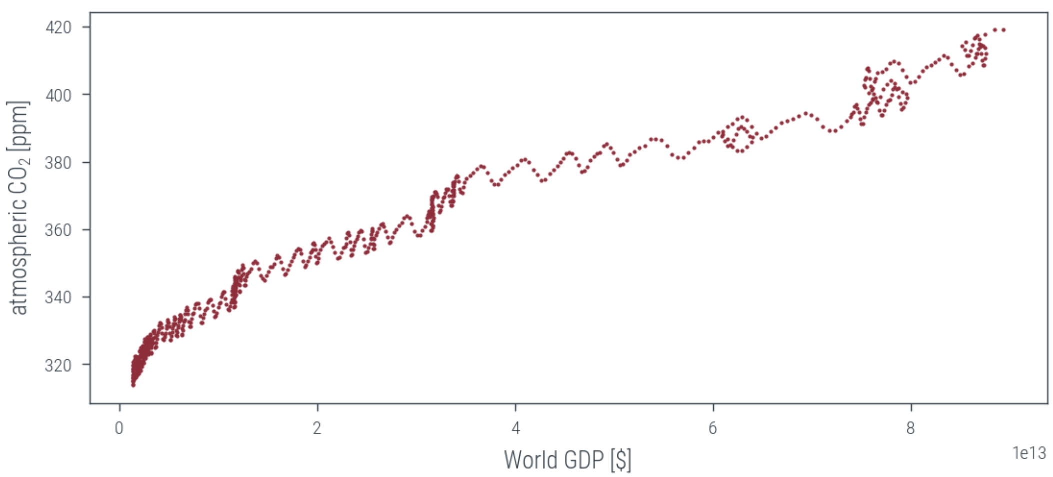


Comparing the plots of the atmospheric CO_2 concentration and the world GDP over time, there seems to be a relationship between the two.

To emphasize this, we plot the two against one another.

[6]:
```
plt.plot(
    data["world_gdp"],
    data["atmospheric_co2"],
    ".",
    markersize=1,
)

plt.xlabel("World GDP [$]")
plt.ylabel("atmospheric CO$_2$ [ppm]")
```

[6]:
```
Text(0, 0.5, 'atmospheric CO$_2$ [ppm]')
```



This plot suggests that a globally linear relationship between the world GDP and the atmospheric CO_2 concentration with some local deviations might explain the data reasonably well.

1.2.2 Building a Prior

We will now build a non-stationary GP prior f for explaining this phenomenon.

[7]:
```
import dataclasses
from collections.abc import Callable


@dataclasses.dataclass
class Gaussian:
    # Gaussian distribution with mean mu and covariance Sigma
    mu: jnp.ndarray  # shape (D,)
    Sigma: jnp.ndarray  # shape (D,D)

    @functools.cached_property
    def L(self):
        """Cholesky decomposition of the covariance matrix"""
        return jnp.linalg.cholesky(self.Sigma)

    @functools.cached_property
    def L_factor(self):
        """Cholesky factorization of the covariance matrix
        (for use in jax.scipy.linalg.cho_solve)"""
        return jax.scipy.linalg.cho_factor(self.Sigma, lower=True)

    @functools.cached_property
    def logdet(self):
        """log-determinant of the covariance matrix
        e.g. for computing the log-pdf
        """
        return 2 * jnp.sum(jnp.log(jnp.diag(self.L)))

    @functools.cached_property
    def prec(self):
        """precision matrix.
        you probably don't want to use this directly, but rather prec_mult
        """
        return jnp.linalg.inv(self.Sigma)

    def prec_mult(self, x):
        """precision matrix multiplication
        implements Sigma^{-1} @ x. For numerical stability, we use the Cholesky␣
↪factorization
        """
        return jax.scipy.linalg.cho_solve(self.L_factor, x)
```

```python
    @functools.cached_property
    def mp(self):
        """precision-adjusted mean"""
        return self.prec_mult(self.mu)

    def log_pdf(self, x):
        """log N(x;mu,Sigma)"""
        return (
            -0.5 * (x - self.mu) @ self.prec_mult(x - self.mu)
            - 0.5 * self.logdet
            - 0.5 * len(self.mu) * jnp.log(2 * jnp.pi)
        )

    def __mult__(self, other):
        """
        Products of Gaussian pdfs are Gaussian pdfs!
        Multiplication of two Gaussian PDFs  (not RVs!)
        other: Gaussian RV
        """
        Sigma = jnp.linalg.inv(self.prec + other.prec)
        mu = Sigma @ (self.mp + other.mp)
        return Gaussian(mu=mu, Sigma=Sigma)

    def __rmatmul__(self, A):
        """Linear maps of Gaussian RVs are Gaussian RVs
        A: linear map, shape (N,D)
        """
        return Gaussian(mu=A @ self.mu, Sigma=A @ self.Sigma @ A.T)

    @functools.singledispatchmethod
    def __add__(self, other):
        """Affine maps of Gaussian RVs are Gaussian RVs
        shift of a Gaussian RV by a constant.
        We implement this as a singledispatchmethod, because jnp.ndarrays can␣
↪not be dispatched on,
        and register the addition of two RVs below
        """
        other = jnp.asarray(other)
        return Gaussian(mu=self.mu + other, Sigma=self.Sigma)

    def condition(self, A, y, Lambda):
        """Linear conditionals of Gaussian RVs are Gaussian RVs
        Conditioning of a Gaussian RV on a linear observation
        A: observation matrix, shape (N,D)
        y: observation, shape (N,)
        Lambda: observation noise covariance, shape (N,N)
```

```
        """
        Gram = A @ self.Sigma @ A.T + Lambda
        L = jax.scipy.linalg.cho_factor(Gram, lower=True)
        mu = self.mu + self.Sigma @ A.T @ jax.scipy.linalg.cho_solve(L, y - A @␣
↪self.mu)
        Sigma = self.Sigma - self.Sigma @ A.T @ jax.scipy.linalg.cho_solve(
            L, A @ self.Sigma
        )
        return Gaussian(mu=mu, Sigma=Sigma)

    @functools.cached_property
    def std(self):
        # standard deviation
        return jnp.sqrt(jnp.diag(self.Sigma))

    def sample(self, key, num_samples=1):
        """
         sample from the Gaussian
        # alternative implementation: works because the @ operator contracts on␣
↪the second-to-last axis on the right
        # return (self.L @ jax.random.normal(key, shape=(num_samples, self.mu.
↪shape[0], 1)))[...,0] + self.mu
        # or like this, more explicit, but not as easy to read
        # return jnp.einsum("ij,kj->ki", self.L, jax.random.normal(key,␣
↪shape=(num_samples, self.mu.shape[0]))) + self.mu
        # or the scipy version:
        """
        return jax.random.multivariate_normal(
            key, mean=self.mu, cov=self.Sigma, shape=(num_samples,),␣
↪method="svd"
        )


@Gaussian.__add__.register
def _add_gaussians(self, other: Gaussian):
    # sum of two Gaussian RVs
    return Gaussian(mu=self.mu + other.mu, Sigma=self.Sigma + other.Sigma)


@dataclasses.dataclass
class GaussianProcess:
    # mean function
    m: Callable[[jnp.ndarray], jnp.ndarray]
    # covariance function
    k: Callable[[jnp.ndarray, jnp.ndarray], jnp.ndarray]

    def __call__(self, x):
```

```
        return Gaussian(mu=self.m(x), Sigma=self.k(x[:, None, :], x[None, :, :
↪]))

    def condition(self, y, X, sigma):
        return ConditionalGaussianProcess(
            self, y, X, Gaussian(mu=jnp.zeros_like(y), Sigma=sigma * jnp.
↪eye(len(y)))
        )

    def plot(
        self,
        ax,
        x,
        color=rgb.tue_gray,
        mean_kwargs={},
        std_kwargs={},
        num_samples=0,
        rng_key=None,
    ):
        gp_x = self(x)
        ax.plot(x[:, 0], gp_x.mu, color=color, **mean_kwargs)
        ax.fill_between(
            x[:, 0],
            gp_x.mu - 2 * gp_x.std,
            gp_x.mu + 2 * gp_x.std,
            color=color,
            **std_kwargs
        )
        if num_samples > 0:
            ax.plot(
                x[:, 0],
                gp_x.sample(rng_key, num_samples=num_samples).T,
                color=color,
                alpha=0.2,
            )

    def plot_shaded(
        self,
        ax,
        x,
        color=rgb.tue_gray,
        yrange=None,
        yres=1000,
        mean_kwargs={},
        std_kwargs={},
        num_samples=0,
        rng_key=None,
```

```
    ):
        if yrange is None:
            yrange = ax.get_ylim()

        gp_x = self(x)
        ax.plot(x[:, 0], gp_x.mu, color=color, **mean_kwargs)

        yy = jnp.linspace(*yrange, yres)[:, None]
        ax.imshow(
            gp_shading(yy, gp_x.mu, gp_x.std),
            extent=[x[0, 0], x[-1, 0], *yrange],
            **std_kwargs,
            aspect="auto",
            origin="lower"
        )

        ax.plot(x[:, 0], gp_x.mu - 2 * gp_x.std, color=color, lw=0.25)
        ax.plot(x[:, 0], gp_x.mu + 2 * gp_x.std, color=color, lw=0.25)
        if num_samples > 0:
            ax.plot(
                x[:, 0],
                gp_x.sample(rng_key, num_samples=num_samples).T,
                color=color,
                alpha=0.2,
            )

def gp_shading(yy, mu, std):
        return jnp.exp(-((yy - mu) ** 2) / (2 * std**2)) # / (std * jnp.sqrt(2␣
 ↪* jnp.pi))

class ConditionalGaussianProcess(GaussianProcess):
    """
    A Gaussian process conditioned on data.
    Implented as a proper python class, which allows inheritance from the␣
 ↪GaussianProcess superclass:
    A conditional Gaussian process contains a Gaussian process prior, provided␣
 ↪at instantiation.
    """

    def __init__(self, prior, y, X, epsilon: Gaussian):
        self.prior = prior
        self.y = jnp.atleast_1d(y)  # shape: (n_samples,)
        self.X = jnp.atleast_2d(X)  # shape: (n_samples, n_features)
        self.epsilon = epsilon
        # initialize the super class
        super().__init__(self._mean, self._covariance)
```

```python
    @functools.cached_property
    def predictive_covariance(self):
        return self.prior.k(self.X[:, None, :], self.X[None, :, :]) + self.
↪epsilon.Sigma

    @functools.cached_property
    def predictive_mean(self):
        return self.prior.m(self.X) + self.epsilon.mu

    @functools.cached_property
    def predictive_covariance_cho(self):
        return jax.scipy.linalg.cho_factor(self.predictive_covariance)

    @functools.cached_property
    def representer_weights(self):
        return jax.scipy.linalg.cho_solve(
            self.predictive_covariance_cho,
            self.y - self.predictive_mean,
        )

    def _mean(self, x):
        x = jnp.asarray(x)
        return (
            self.prior.m(x)
            + self.prior.k(x[..., None, :], self.X[None, :, :])
            @ self.representer_weights
        )

    @functools.partial(jnp.vectorize, signature="(d),(d)->()", excluded={0})
    def _covariance(self, a, b):
        return self.prior.k(a, b) - self.prior.k(
            a, self.X
        ) @ jax.scipy.linalg.cho_solve(
            self.predictive_covariance_cho,
            self.prior.k(self.X, b),
        )

    def _m_proj(self, x, projection, projection_mean):
        x = jnp.asarray(x)

        if projection_mean is None:
            projection_mean = self.prior.m

        return (
            projection_mean(x)
            + projection(x[..., None, :], self.X[None, :, :]) @ self.
↪representer_weights
```

```
        )

    @functools.partial(jnp.vectorize, signature="(d),(d)->()", excluded={0, 3})
    def _cov_proj(self, a, b, projection):
        return projection(a, b) - projection(a, self.X) @ jax.scipy.linalg.
↪cho_solve(
            self.predictive_covariance_cho,
            projection(b, self.X),
        )

    def project(self, k_proj, m_proj=None):
        return GaussianProcess(
            lambda x: self._m_proj(x, k_proj, m_proj),
            lambda x0, x1: self._cov_proj(x0, x1, k_proj),
        )
```

[8]:
```
# Let's define some commonly used building blocks for GP models
def zero_mean(x):
    return jnp.zeros_like(x[..., 0])

def gaussian_kernel(x0, x1, ell=1.0, theta=1.0):
    return theta**2 * jnp.exp(-jnp.sum((x0 - x1)**2, axis=-1) / (2. * ell**2))

def rational_quadratic_kernel(x0, x1, alpha=1.0, ell=1.0, theta=1.0):
    return theta**2 * (
        1 + jnp.sum((x0 - x1) ** 2, axis=-1) / (2 * alpha * ell**2)
    ) ** (-alpha)

def matern_1_2_kernel(x0, x1, ell=1.0, theta=1.0):
    return theta**2 * jnp.exp(
        -jnp.linalg.norm(x0 - x1, 2, axis=-1) / ell
    )
```

[9]:
```
from matplotlib.colors import LinearSegmentedColormap

cmap_rw = LinearSegmentedColormap.from_list(
    "rw", [(1, 1, 1), rgb.tue_red], N=1024)
cmap_dw = LinearSegmentedColormap.from_list(
    "dw", [(1, 1, 1), rgb.tue_dark], N=1024)
cmap_bw = LinearSegmentedColormap.from_list(
    "bw", [(1, 1, 1), rgb.tue_blue], N=1024)
cmap_gw = LinearSegmentedColormap.from_list(
    "gw", [(1, 1, 1), rgb.tue_green], N=1024)
cmap_bwr = LinearSegmentedColormap.from_list(
    "bwr", [rgb.tue_blue, (1, 1, 1), rgb.tue_red], N=1024
)
```

Our prior will be a modification of the model from the lecture, which replaces the long-term trend by a GDP-based additive component. To inform the GP about the current value of the GDP, we will construct a bivariate model with inputs

$$f(t, \mathrm{GDP}(t)). \tag{1}$$

This way, we can still interpret our GP as a univariate function of time if we know the (approximate) GPD at every evaluation point t.

[10]:
```
X = jnp.stack(
    (data["date"], data["world_gdp"]),
    axis=-1,
)
y = jnp.asarray(data["atmospheric_co2"])

X_train = X[:-200, :]
y_train = y[:-200]
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun for
more info.)
```

[11]:
```
def decompose_t_gdp(t_gdp):
    """This function will come in handy when implementing mean functions and␣
↪kernels for the GP."""
    t_gdp = jnp.asarray(t_gdp)
    return t_gdp[..., 0], t_gdp[..., 1]
```

Like the model in the lecture, our GP prior will be an additive combination of simpler GP priors with different responsibilities:

$$f(t, \mathrm{GDP}(t)) = f_{\mathrm{const}}(t) + f_{\mathrm{GDP}}(t, \mathrm{GDP}(t)) + f_{\mathrm{period}}(t) + f_{\text{mid-term}}(t) + f_{\mathrm{weather}}(t). \tag{2}$$

First, we extend the model from the lecture by allowing the GP to learn the constant component f_{const} of the posterior. We can achieve this this by constructing a parametric GP with one constant feature function $\phi(t) = 1$, i.e.

$$f_{\mathrm{const}}(t) = \phi(t) w_{\mathrm{const}}, \tag{3}$$

where $w_{\mathrm{const}} \sim \mathcal{N}(\mu_{\mathrm{const}}, \theta^2_{\mathrm{const}})$. The constant mean function in the model from the lecture was a limiting case of this component with $\theta^2_{\mathrm{const}} = 0$.

Task: Implement the mean and kernel of f_{const}.

[37]:
```
# we set the mean to a constant function at the data minimum
mu_const = np.min(y_train)
theta_const = jnp.std(y_train)


def constant_mean(t_gdp, mu=mu_const):
    t, _ = decompose_t_gdp(t_gdp)
    return mu * jnp.ones_like(t)
```

```python
def constant_kernel(t_gdp0, t_gdp1, theta=theta_const):
    t0, _ = decompose_t_gdp(t_gdp0)
    t1, _ = decompose_t_gdp(t_gdp1)
    return theta**2 * jnp.ones_like(t1)
```

As noted above, the data suggests a globally linear relationship with some local deviations between atmospheric CO_2 concentration and world GDP. The component f_{GDP} models this assumption using a GP:

$$f_{\text{GDP}}(t, \text{GDP}(t)) := \text{GDP}(t) \cdot (c_{\text{GDP}} + g_{\text{GDP}}(t)), \tag{4}$$

with

$$\begin{aligned} c_{\text{GDP}} &\sim \mathcal{N}(\mu_{\text{GDP,c}}, \theta^2_{\text{GDP,c}}), \qquad \text{and} \\ g_{\text{GDP}} &\sim \mathcal{GP}(0, \theta^2_{\text{GDP,g}} \cdot k_{\text{RQ}}), \end{aligned}$$

where k_{RQ} is a rational quadratic kernel with lengthscale $\ell_{\text{GDP,g}}$ and shape parameter $\alpha_{\text{GDP,g}}$.

Task: Implement the mean and kernel of f_{GDP}.

[13]:
```python
mu_c_gdp = np.mean((y_train - mu_const) / X_train[..., 1])  # ppm / dollar
theta_c_gdp = np.std((y_train - mu_const) / X_train[..., 1])  # ppm / dollar

alpha_g_gdp = 1.  # unitless
ell_g_gdp = 10.  # years
theta_g_gdp = mu_c_gdp / 2. - theta_c_gdp  # ppm / dollar

def gdp_mean(t_gdp, mu_c=mu_c_gdp):
    _, gdp = decompose_t_gdp(t_gdp)
    return mu_c * gdp

def gdp_kernel(
    t_gdp0,
    t_gdp1,
    theta_c=theta_c_gdp,
    alpha_g=alpha_g_gdp,
    ell_g=ell_g_gdp,
    theta_g=theta_g_gdp,
):
    t0, gdp0 = decompose_t_gdp(t_gdp0)
    t1, gdp1 = decompose_t_gdp(t_gdp1)

    k_rq = rational_quadratic_kernel(
            t0[..., None],
```

```
            t1[..., None],
            alpha=alpha_g,
            ell=ell_g)
    return theta_c**2 + theta_g**2 *k_rq
```

The periodic and mid-term components are identical to those in the model from the lecture.

[14]:
```
theta_periodic = 5.0  # ppm
ell_decay_periodic = 50.0  # years
ell_periodic = 1.0  # years

periodic_mean = zero_mean

def periodic_kernel(
    t_gdp0,
    t_gdp1,
    period=1.0,
    ell_periodic=ell_periodic,
    ell_decay=ell_decay_periodic,
    theta=theta_periodic,
):
    t0, _ = decompose_t_gdp(t_gdp0)
    t1, _ = decompose_t_gdp(t_gdp1)

    return (
        theta**2
        * jnp.exp(
            -2
            * jnp.sin(jnp.pi * (t0 - t1) / period) ** 2
            / ell_periodic**2
        )
        * gaussian_kernel(
            t0[..., None],
            t1[..., None],
            ell=ell_decay,
        )
    )
```

[15]:
```
theta_mid_term = 1.0  # ppm
ell_mid_term = 1.0  # years
alpha_mid_term = 1.0  # unitless

mid_term_trend_mean = zero_mean

def mid_term_trend_kernel(t_gdp0, t_gdp1, ell=ell_mid_term,␣
 ↪alpha=alpha_mid_term, theta=theta_mid_term):
    t0, _ = decompose_t_gdp(t_gdp0)
```

```
    t1, _ = decompose_t_gdp(t_gdp1)

    return rational_quadratic_kernel(
        t0[..., None],
        t1[..., None],
        alpha=alpha,
        ell=ell,
        theta=theta,
    )
```

The local weather component is a slight modification of the corresponding term in the model from the lecture. Instead of the Gaussian kernel, we use a Matérn-$\frac{1}{2}$ kernel here and we also drop the white-noise kernel, since we model measurement noise explicitly.

[16]:
```
theta_weather = 0.1  # ppm
ell_weather = 0.1  # years

weather_mean = zero_mean

def weather_kernel(t_gdp0, t_gdp1, ell=ell_weather, theta=theta_weather):
    t0, _ = decompose_t_gdp(t_gdp0)
    t1, _ = decompose_t_gdp(t_gdp1)

    return matern_1_2_kernel(
        t0[..., None],
        t1[..., None],
        ell=ell,
        theta=theta,
    )
```

We now combine these building blocks into a common model.

[17]:
```
def model_mean(t_gdp, parameters):
    return (
        constant_mean(t_gdp, mu=parameters["mu_const"])
        + gdp_mean(t_gdp, mu_c=parameters["mu_c_gdp"])
        + periodic_mean(t_gdp)
        + mid_term_trend_mean(t_gdp)
        + weather_mean(t_gdp)
    )


def model_kernel(t_gdp0, t_gdp1, parameters):
    return (
        constant_kernel(
            t_gdp0,
            t_gdp1,
            theta=parameters["theta_const"],
```

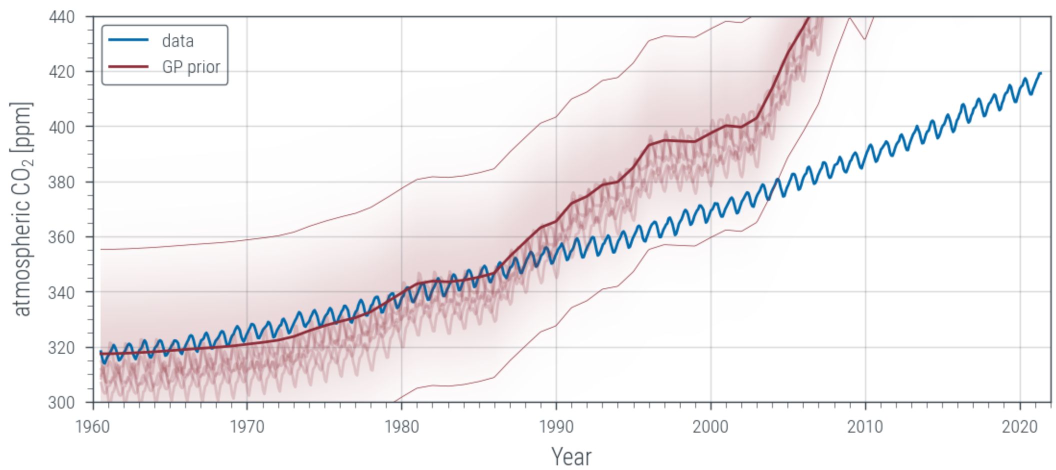
```
        )
        + gdp_kernel(
            t_gdp0,
            t_gdp1,
            theta_c=theta_c_gdp,  # parameters["theta_c_gdp"],
            alpha_g=parameters["alpha_g_gdp"],
            ell_g=parameters["ell_g_gdp"],
            theta_g=theta_g_gdp,  # parameters["theta_g_gdp"],
        )
        + periodic_kernel(
            t_gdp0,
            t_gdp1,
            ell_periodic=parameters["ell_periodic"],
            ell_decay=parameters["ell_decay_periodic"],
            theta=parameters["theta_periodic"],
        )
        + mid_term_trend_kernel(
            t_gdp0,
            t_gdp1,
            ell=parameters["ell_mid_term"],
            alpha=parameters["alpha_mid_term"],
            theta=parameters["theta_mid_term"],
        )
        + weather_kernel(
            t_gdp0,
            t_gdp1,
            ell=parameters["ell_weather"],
            theta=parameters["theta_weather"],
        )
    )


# initial guesses for the parameters:
init_params = {
    "mu_const": mu_const,
    "theta_const": theta_const,
    "mu_c_gdp": mu_c_gdp,
    "theta_c_gdp": theta_c_gdp,
    "alpha_g_gdp": alpha_g_gdp,
    "ell_g_gdp": ell_g_gdp,
    "theta_g_gdp": theta_g_gdp,
    "ell_periodic": ell_periodic,
    "ell_decay_periodic": ell_decay_periodic,
    "theta_periodic": theta_periodic,
    "ell_mid_term": ell_mid_term,
    "alpha_mid_term": alpha_mid_term,
    "theta_mid_term": theta_mid_term,
```

```
    "ell_weather": ell_weather,
    "theta_weather": theta_weather,
    "sigma": 0.2,
}


gp = GaussianProcess(
    functools.partial(model_mean, parameters=init_params),
    functools.partial(model_kernel, parameters=init_params),
)
```

Let's visualize the prior.

[18]:
```
setup_plot(plt.gca())
plot_data(plt.gca())
gp.plot_shaded(
    plt.gca(),
    X,
    yres=1000,
    color=rgb.tue_red,
    mean_kwargs={"label": "GP prior"},
    std_kwargs={"alpha": 0.2, "cmap": cmap_rw},
    num_samples=5,
    rng_key=jax.random.PRNGKey(3),
)
plt.legend()
```

[18]: <matplotlib.legend.Legend at 0x7f97fc986590>

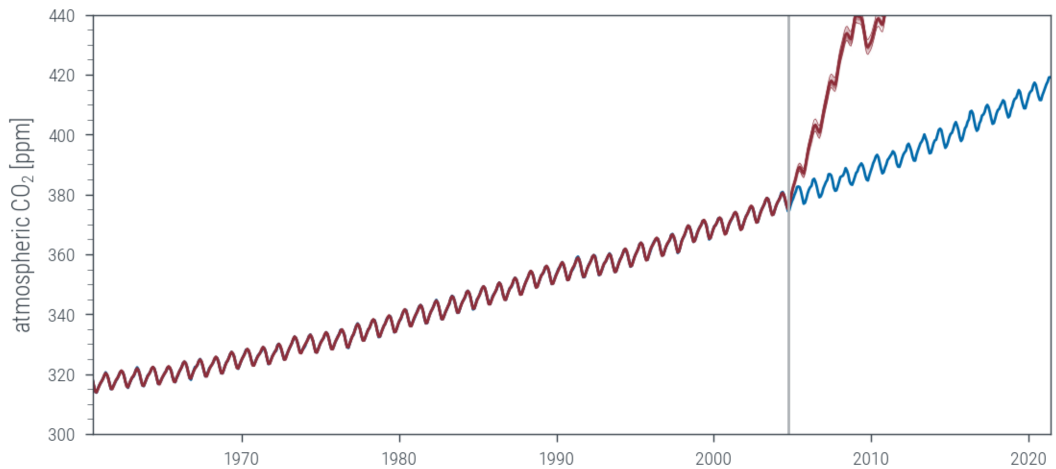


Compared to the model from the lecture, it is arguably much more plausible that the data was generated by this model.

1.2.3 Posterior

[19]:
```
gp_posterior = gp.condition(y_train, X_train, sigma=init_params["sigma"]**2)

plot_data(plt.gca())
gp_posterior.plot_shaded(
    plt.gca(),
    X,
    yres=1000,
    color=rgb.tue_red,
    mean_kwargs={"label": "GP posterior mean"},
    std_kwargs={"alpha": 0.2, "cmap": cmap_rw},
    num_samples=5,
    rng_key=jax.random.PRNGKey(3),
)
plt.axvline(X_train[-1, 0], color=rgb.tue_gray)
```

[19]: `<matplotlib.lines.Line2D at 0x7f97d43cd240>`



The posterior belief also seems to extrapolate reasonably well.

1.2.4 Hyperparameter Optimization

[ ]:
```
import jaxopt


def NegEvidence(parameters):
    gp = GaussianProcess(
        functools.partial(model_mean, parameters=parameters),
        functools.partial(model_kernel, parameters=parameters),
    )
```

```
    noise = Gaussian(
        jnp.zeros_like(y_train),
        parameters["sigma"]**2 * jnp.eye(y_train.size),
    )

    predictive = gp(X_train) + noise

    return -predictive.log_pdf(y_train)

print(NegEvidence(init_params))

optim = jaxopt.ScipyMinimize(
    fun=jax.value_and_grad(NegEvidence),
    method="CG",
    value_and_grad=True,
    maxiter=100,
    jit=True,
    options={"disp": True},
)

opt_params, _ = optim.run(init_params)
```

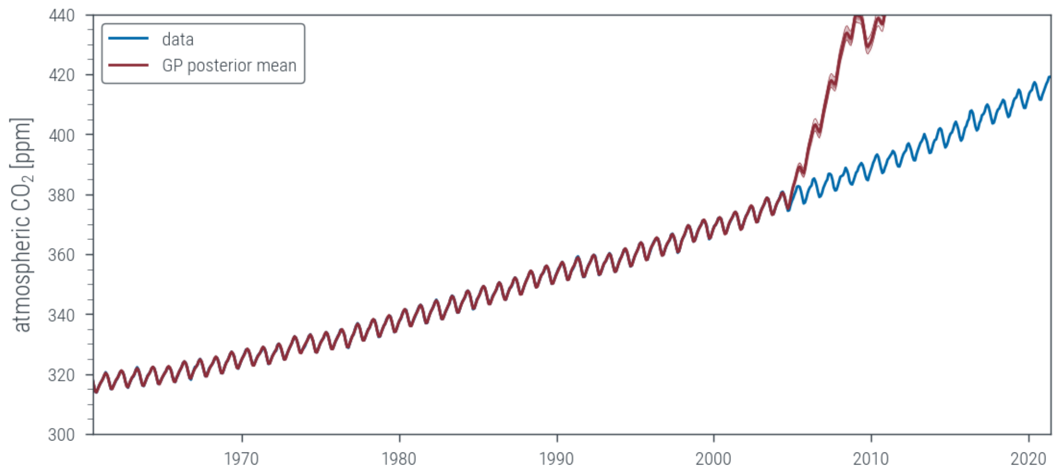
[ ]:
```
init_params
```

[ ]:
```
opt_params
```

[39]:
```
gp = GaussianProcess(
    functools.partial(model_mean, parameters=opt_params),
    functools.partial(model_kernel, parameters=opt_params),
)
```

[40]:
```
gp_posterior = gp.condition(y_train, X_train, sigma=opt_params["sigma"]**2)

plot_data(plt.gca())
gp_posterior.plot_shaded(
    plt.gca(),
    X,
    yres=1000,
    color=rgb.tue_red,
    mean_kwargs={"label": "GP posterior mean"},
    std_kwargs={"alpha": 0.2, "cmap": cmap_rw},
    num_samples=5,
    rng_key=jax.random.PRNGKey(3),
)
plt.legend()
```

[40]: <matplotlib.legend.Legend at 0x7f9812231c00>



1.2.5 Source Separation

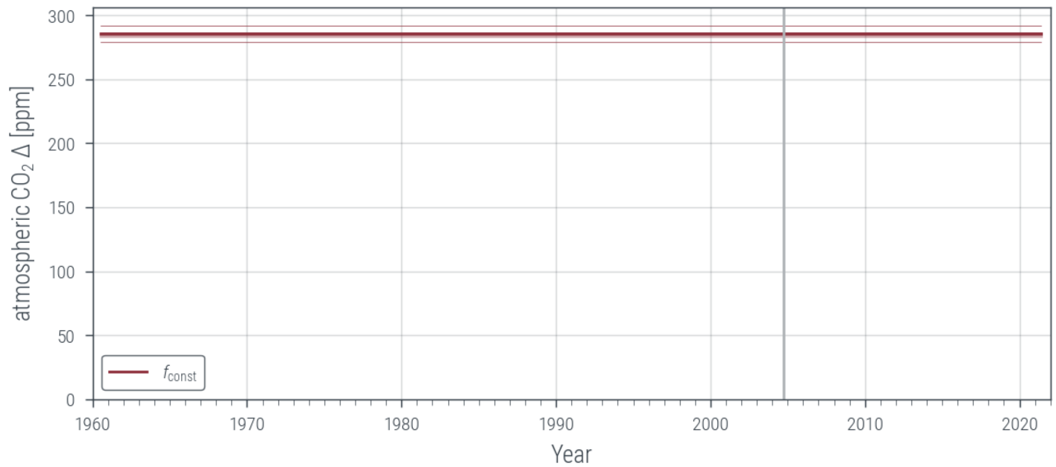
Task: Visualize the individual posterior belief over all additive components (f_{const}, f_{GDP}, f_{periodic}, $f_{\text{mid-term}}$, f_{weather}) using source separation.

[31]:
```
def plot_component(ax, gp_component: GaussianProcess, component_label: str):
    setup_plot(ax)
    ax.set_ylabel("atmospheric CO$_2$ $\Delta$ [ppm]")
    gp_component.plot_shaded(
        ax,
        X,
        yres=1000,
        color=rgb.tue_red,
        mean_kwargs={"label": component_label},
        std_kwargs={"alpha": 0.2, "cmap": cmap_rw},
        num_samples=5,
        rng_key=jax.random.PRNGKey(3),
    )
    ax.axvline(X_train[-1, 0], color=rgb.tue_gray)
```

[41]:
```
m_constant = functools.partial(
    constant_mean, mu=opt_params['mu_const']
)

k_constant = functools.partial(
    constant_kernel,
    theta=opt_params['theta_const'],
)

gp_constant =  gp_posterior.project(k_constant, m_constant)
```

```
plot_component(plt.gca(), gp_constant, "$f_\mathrm{const}$")
plt.legend()
```

[41]: <matplotlib.legend.Legend at 0x7f97b426b7f0>



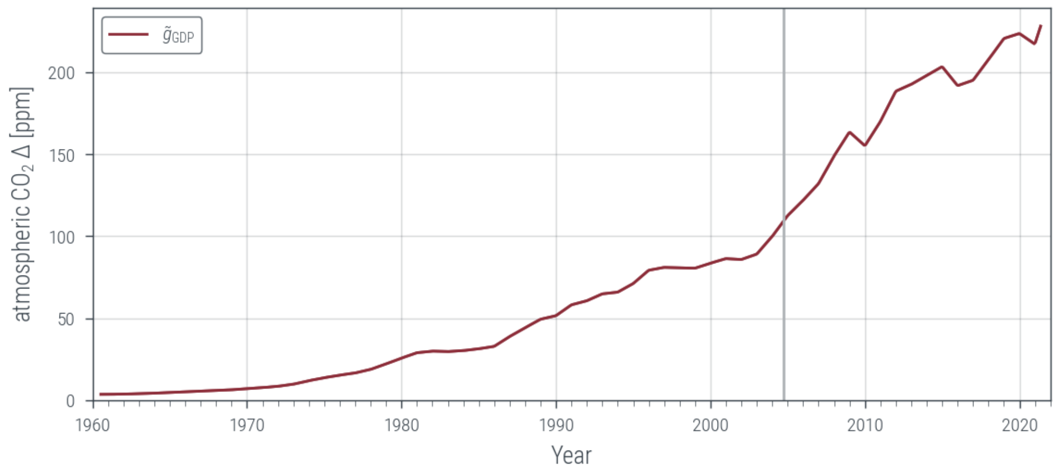
Task: Plot the posterior belief over

$$\tilde{g}_{\mathrm{GDP}}(t, \mathrm{GDP}(t)) := \mathrm{GDP}(t) \cdot g_{\mathrm{GDP}}(t)$$

using source separation.

[42]:
```
k_g_tilde= functools.partial(
    gdp_kernel,
    alpha_g=opt_params['alpha_g_gdp'],
    ell_g=opt_params['ell_g_gdp'],
    theta_g=opt_params['theta_g_gdp'],
    theta_c=opt_params['theta_c_gdp'],
)

m_g_tilde = functools.partial(
    gdp_mean, mu_c=opt_params['mu_c_gdp']
    # I choose mu_c_gdp instead of 0 to get nonzero posterior.
)


gp_g_tilde =  gp_posterior.project(k_g_tilde, m_g_tilde)

plot_component(plt.gca(), gp_g_tilde, r"$\tilde{g}_\mathrm{GDP}$")
plt.legend()
```

[42]: <matplotlib.legend.Legend at 0x7f9794796890>

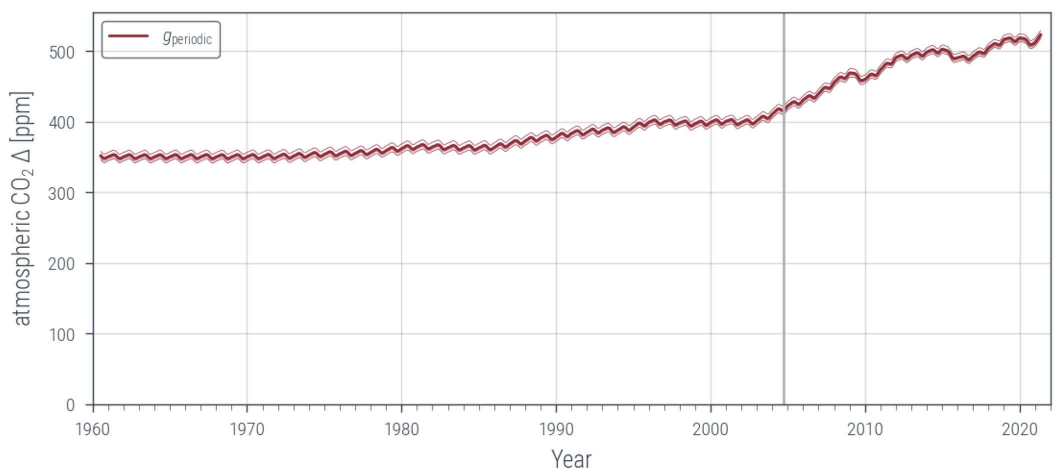


1.2.6 More plots!

To analyze the model we will also show the posterior belief over $f_{periodic}$, $f_{mid-term}$, $f_{weather}$

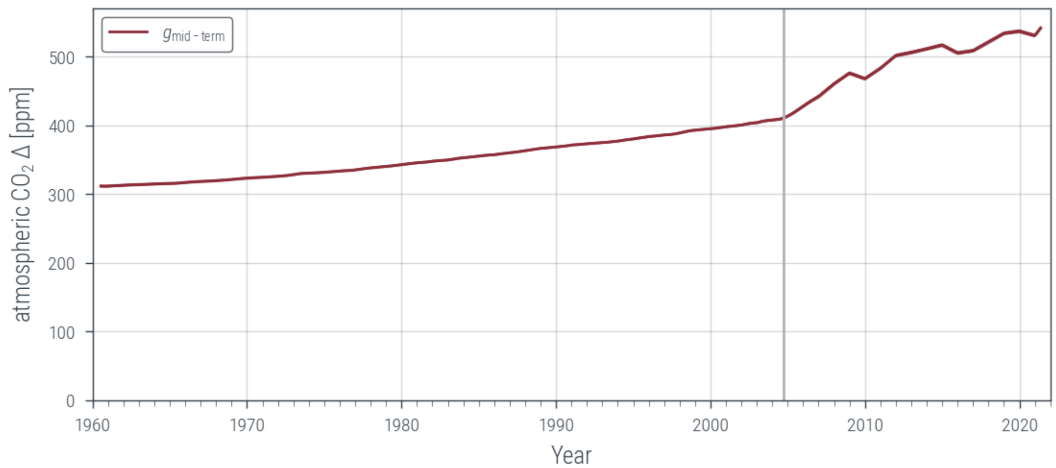
[43]:
```
k_periodic = functools.partial(periodic_kernel,
                               ell_periodic=opt_params['ell_periodic'],
                               ell_decay=opt_params['ell_decay_periodic'],
                               theta=opt_params['theta_periodic'],)

gp_periodic = gp_posterior.project(k_periodic)


plot_component(plt.gca(), gp_periodic, r"${g}_\mathrm{periodic}$")
plt.legend()
```

[43]: <matplotlib.legend.Legend at 0x7f979452a410>



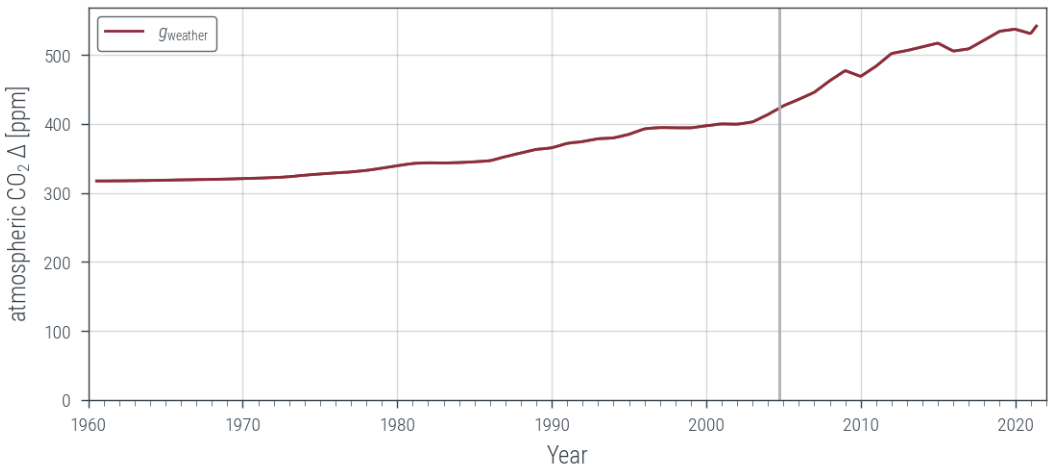
[44]:
```
k_mid_term = functools.partial(mid_term_trend_kernel,
                               ell=opt_params['ell_mid_term'],
                               alpha=opt_params['alpha_mid_term'],
                               theta=opt_params['theta_mid_term'])

gp_mid_term = gp_posterior.project(k_mid_term)


plot_component(plt.gca(), gp_mid_term, r"${g}_\mathrm{mid-term}$")
plt.legend()
```

[44]: <matplotlib.legend.Legend at 0x7f97944b27d0>



[45]:
```
k_weather= functools.partial(weather_kernel,
                             ell=opt_params['ell_weather'],
                             theta=opt_params['theta_weather'])

gp_weather = gp_posterior.project(k_weather)


plot_component(plt.gca(), gp_weather, r"$g_\mathrm{weather}$")
plt.legend()
```

[45]: <matplotlib.legend.Legend at 0x7f979425f7c0>



1.2.7 Discussion

Interpret the results. Is this a good model of the data? What could be improved and how?

If there is enough time in the tutorial, we'll discuss your thoughts.

We are trying to model an extremely complex problem relying on a strong assumption on the data, which feels dangerous (I guess that is how prior knowledge feels?).

As can be seen, our model overexplains the amount of atmospheric CO_2. An idea that comes to mind is to weight the GP terms in the sum.