Gaussian Process Regression

At times you don't care about the underlying model for your data points and just want a model that describes the data. One such fitting technique is know as Gaussian process regression (also know as kriging). This kind of regression assumes all the data points are drawn from a common covariance function. This function is used to generate an (infinite) set of functions and only keeps the ones that pass through the observed data.

Packages being used

- `pymc3` : has a Gaussian process regression function

Relevant documentation

- `pymc3` : https://docs.pymc.io/en/stable/pymc-examples/examples/gaussian_processes/GP-MeansAndCovs.html, https://docs.pymc.io/en/stable/pymc-examples/examples/gaussian_processes/GP-Marginal.html

In [1]:
```
import numpy as np
import pymc3 as pm
import theano.tensor as tt
from scipy import interpolate
import seaborn
from matplotlib import pyplot as plt
import mpl_style
%matplotlib inline
plt.style.use(mpl_style.style1)
seaborn.axes_style(mpl_style.style1);
```

```
WARNING (theano.tensor.blas): Using NumPy C-API based implementation for BLAS functions.
```

The squared exponential covariance (or Radial-basis function or Exponential Quadratic)

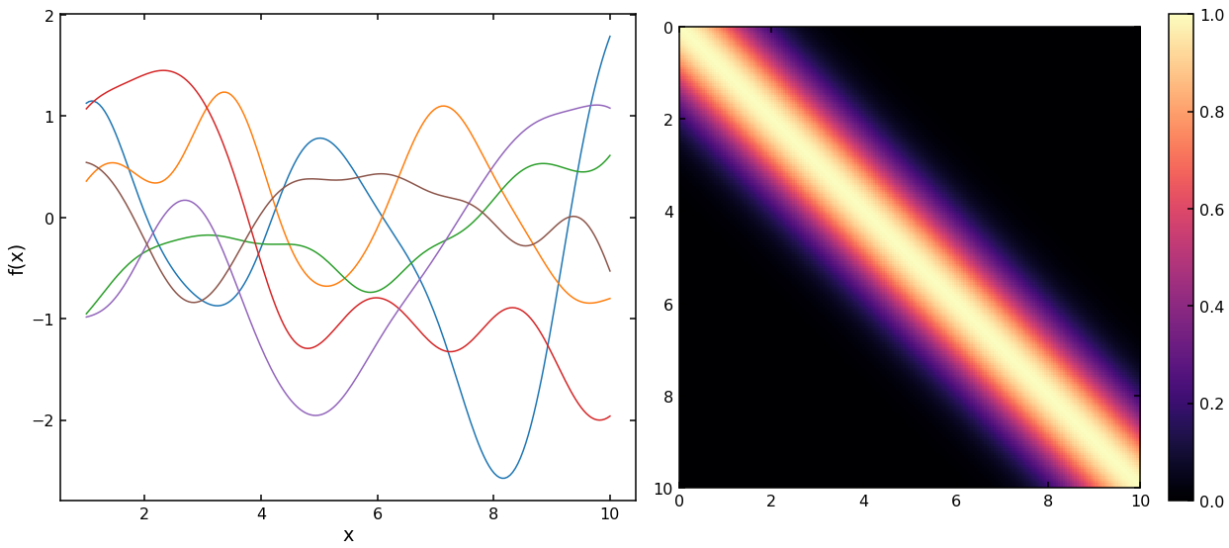
As an example we will use the squared exponential covariance function:

$$\mathrm{Cov}\left(x_1, x_2; h\right) = \exp\left(\frac{-(x_1 - x_2)^2}{2h^2}\right)$$

Lets using this function to draw some *unconstrained* functions:

In [2]:
```
h = 1
cov = pm.gp.cov.ExpQuad(1, h) + pm.gp.cov.WhiteNoise(1e-6)

x = np.linspace(1, 10, 200)[:, None]
K = cov(x).eval()

plt.figure(1, figsize=(18, 8))
```

```
plt.subplot(121)
plt.plot(x, pm.MvNormal.dist(mu=np.zeros(K.shape[0]), cov=K, shape=K.shape[0]
plt.xlabel('x')
plt.ylabel('f(x)')

plt.subplot(122)
plt.imshow(K, interpolation='none', origin='upper', extent=[0, 10, 10, 0])
plt.colorbar()
plt.tight_layout();
```



Constrain the model

Assume we have some data points, we can use Gaussian process regression to only pick the models that pass through those points:

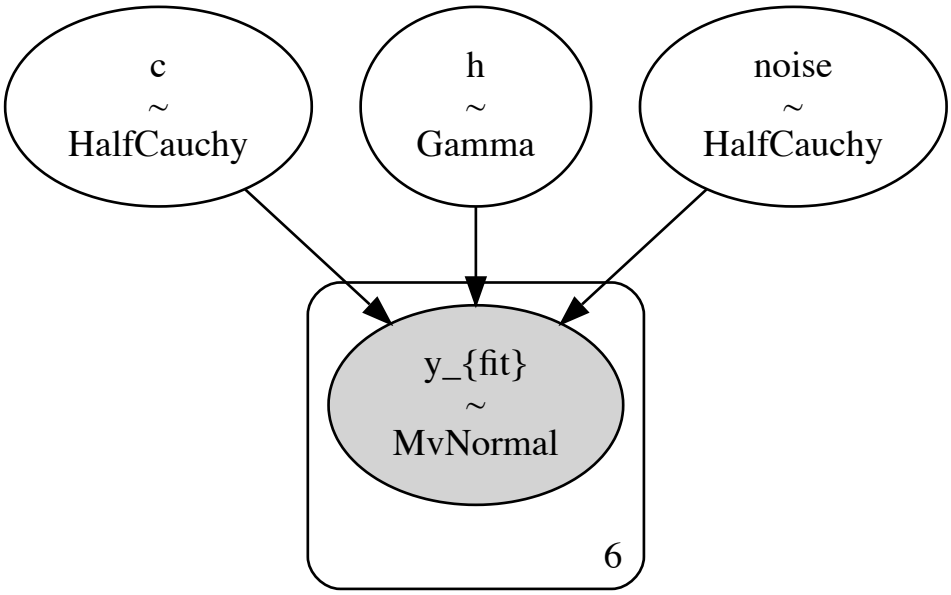
In [3]:
```
x1 = np.array([1, 3, 5, 6, 7, 8])
y1 = x1 * np.sin(x1)
```

Build the PYMC model

We will define priors for the length scale `h` and the leading scaling coefficient `c`. We will assume there is a small level of equal but unknown noise associated with each data point.

In [4]:
```
X = x1[:, None]
with pm.Model() as model:
    h = pm.Gamma("h", alpha=2, beta=1)
    c = pm.HalfCauchy("c", beta=5)
    cov = c**2 * pm.gp.cov.ExpQuad(1, ls=h)
    gp = pm.gp.Marginal(cov_func=cov)
    noise = pm.HalfCauchy("noise", beta=0.1)
    y_fit = gp.marginal_likelihood("y_{fit}", X=X, y=y1, noise=noise)

display(model)
display(pm.model_to_graphviz(model))
```

$$
\begin{aligned}
\text{h\_log\_\_} &\sim \text{TransformedDistribution} \\
\text{c\_log\_\_} &\sim \text{TransformedDistribution} \\
\text{noise\_log\_\_} &\sim \text{TransformedDistribution} \\
\text{h} &\sim \text{Gamma} \\
\text{c} &\sim \text{HalfCauchy} \\
\text{noise} &\sim \text{HalfCauchy} \\
\text{y\_\{fit\}} &\sim \text{MvNormal}
\end{aligned}
$$



Find the maximum of the likelihood using the `find_MAP` function.

In [5]:
```
with model:
    mp = pm.find_MAP()

display('Best fit kernel: {0:.2f}**2 * ExpQuad(ls={1:.2f})'.format(mp['c'], m
```

100.00% [35/35 00:00<00:00 logp = -16.448, ||grad|| = 0.0003648]

```
'Best fit kernel: 4.16**2 * ExpQuad(ls=1.58)'
```

Use the fit to interpolate to new `X` values

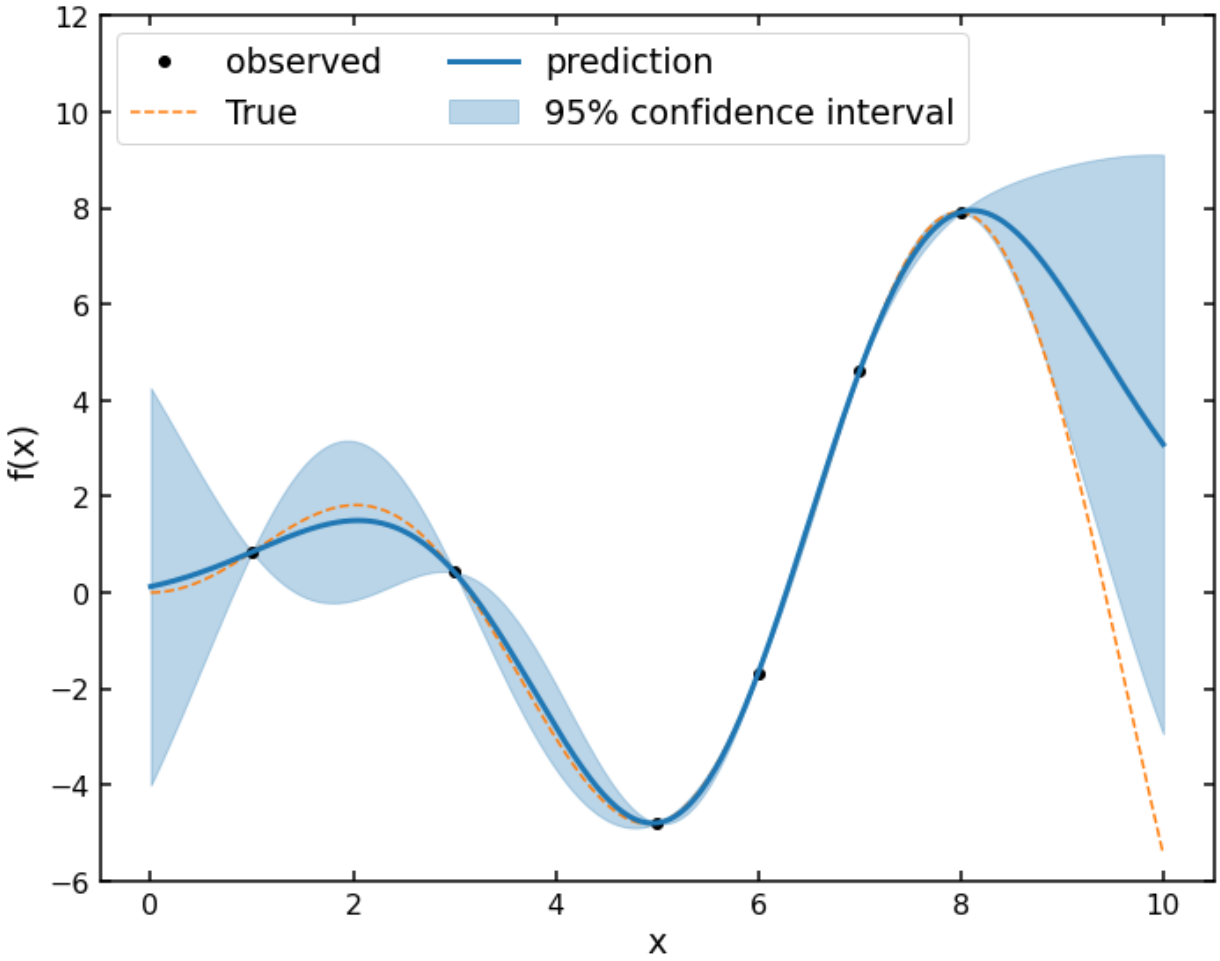
This `MAP` fit can be used to interpolate and extrapolate to a new grid of points. PYMC offers the `predict` method to make this easier.

In [6]:
```
n_new = 500
X_new = np.linspace(0, 10, n_new)

mu, var = gp.predict(X_new[:,None], point=mp, diag=True)
sd = np.sqrt(var)
```

Let's plot the result:

In [7]:
```
plt.figure(2, figsize=(10, 8))
plt.plot(x1, y1, 'ok', label='observed')
```

```
plt.plot(
    X_new,
    X_new * np.sin(X_new),
    '--',
    color='C1',
    label='True'
)

plt.plot(
    X_new.flatten(),
    mu,
    color='C0',
    lw=3,
    zorder=3,
    label='prediction'
)

# plot 95% best fit region
plt.fill_between(
    X_new.flatten(),
    mu - 1.96*sd,
    mu + 1.96*sd,
    color='C0',
    alpha=0.3,
    zorder=1,
    label='95% confidence interval'
)

# labels and legend
plt.xlabel('x')
plt.ylabel('f(x)')
plt.ylim(-6, 12)
plt.legend(loc='upper left', ncol=2)
plt.tight_layout();
```



Sampling from the data

Sometimes you want to know the values of the covariance function and draw samples from the posterior distribution. We can do easily do this within PYMC with the `sample` method:

In [10]:
```
with model:
    trace = pm.sample(2000, target_accept=0.99)
    display(pm.summary(trace))
    pm.plot_trace(trace, figsize=(12, 9));
```

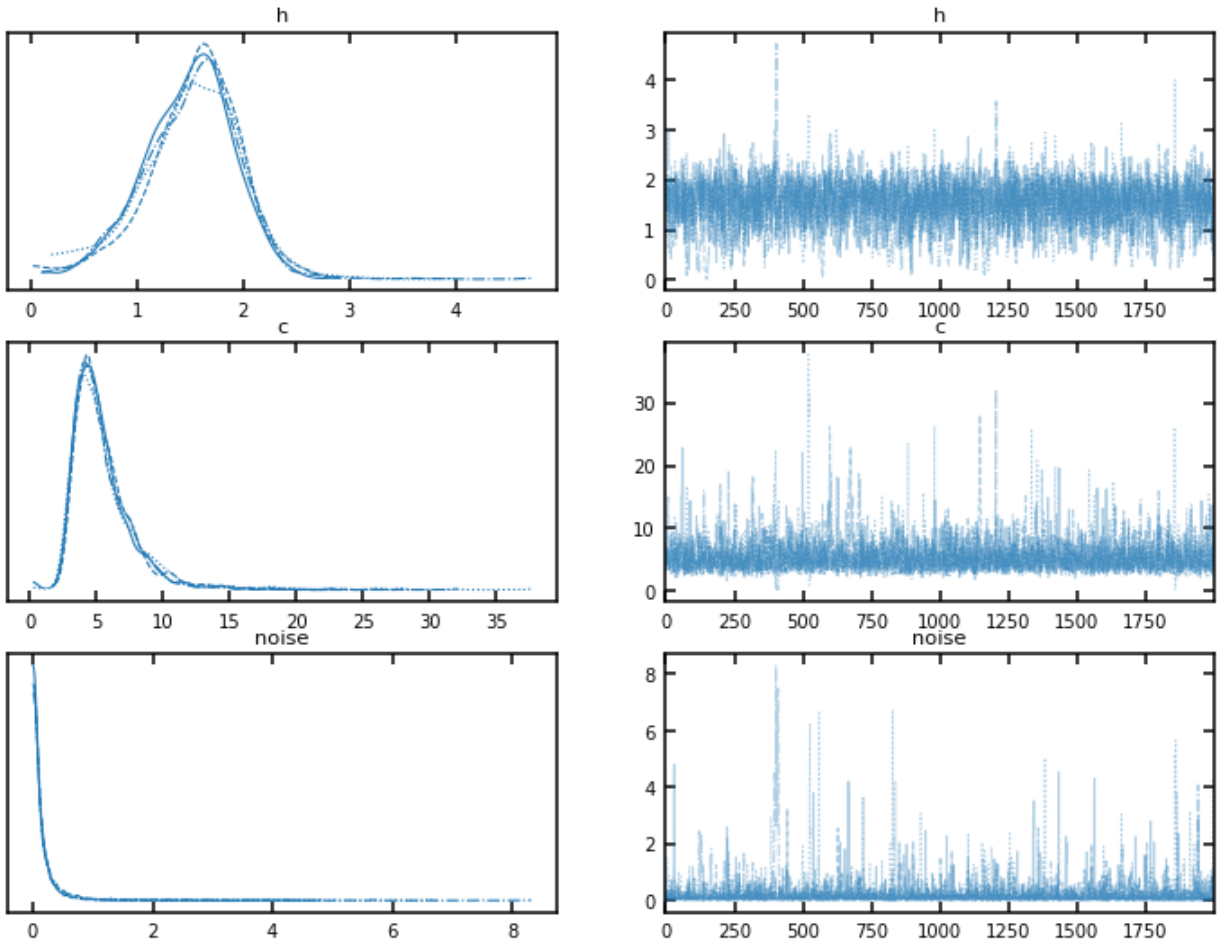
```
/tmp/ipykernel_24954/3990587379.py:2: FutureWarning: In v4.0, pm.sample will r
eturn an `arviz.InferenceData` object instead of a `MultiTrace` by default. Yo
u can pass return_inferencedata=True or return_inferencedata=False to be safe
and silence this warning.
  trace = pm.sample(2000, target_accept=0.99)
Auto-assigning NUTS sampler...
Initializing NUTS using jitter+adapt_diag...
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [noise, c, h]
```

100.00% [12000/12000 00:26<00:00 Sampling 4 chains, 0 divergences]

```
Sampling 4 chains for 1_000 tune and 2_000 draw iterations (4_000 + 8_000 draw
s total) took 26 seconds.
The number of effective samples is smaller than 25% for some parameters.
```

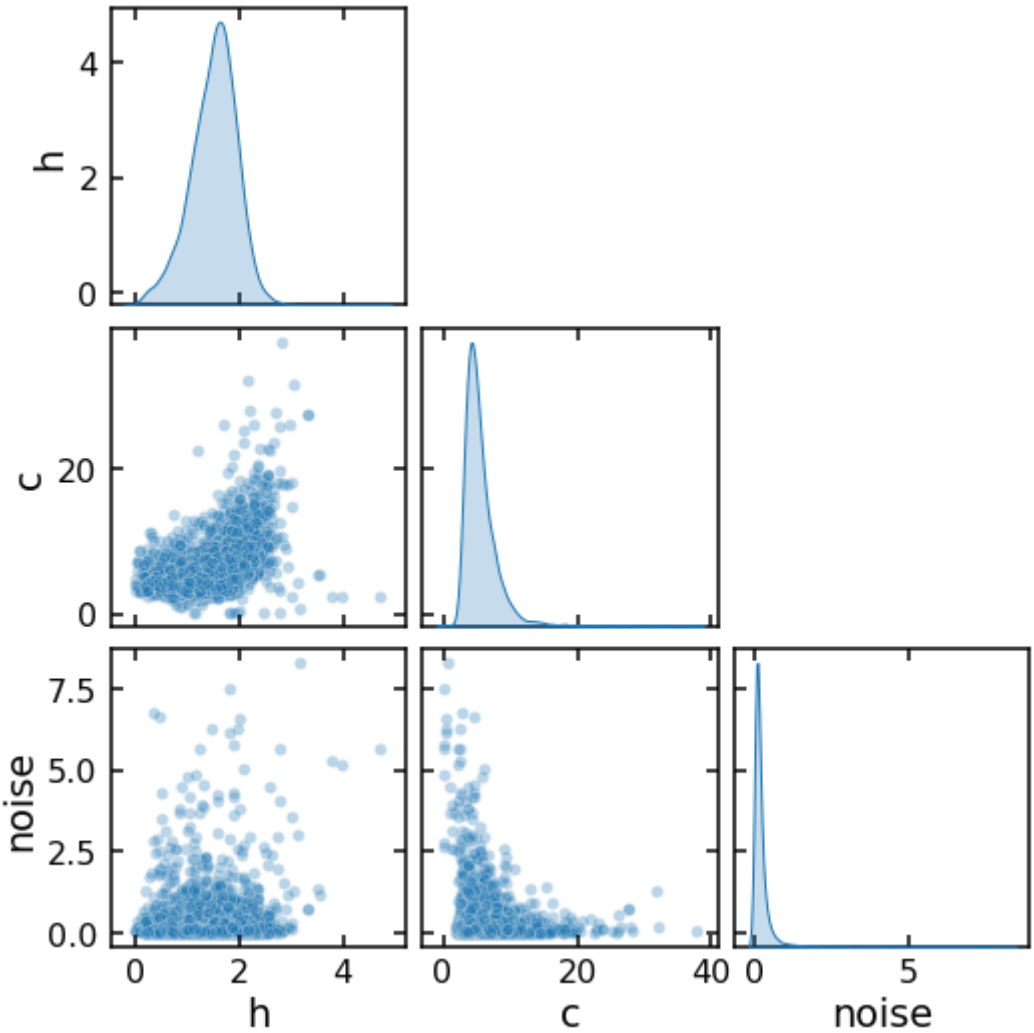
	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
h	1.510	0.452	0.618	2.315	0.011	0.007	1940.0	1408.0	1.0
c	5.513	2.515	2.386	9.665	0.055	0.040	2677.0	2351.0	1.0

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
noise	0.200	0.441	0.000	0.582	0.012	0.010	3693.0	3292.0	1.0



In [11]:

```
df = pm.trace_to_dataframe(trace)

def hide_current_axis(*args, **kwargs):
    plt.gca().set_visible(False)

g = seaborn.pairplot(
    df,
    diag_kind='kde',
    plot_kws={'alpha': 0.3}
)
g.map_upper(hide_current_axis)
seaborn.despine(top=False, right=False)
```



Notice the trade off between the noise level and the `c` parameter. When `c` becomes small the noise becomes large, i.e. it models all the points as coming from a flat line with high noise.

Now that we have sampled from the distribution we can interpolate/extrapolate the fitted function. To do this we have to pass the new `X` values into the model as a `conditional`:

In [12]:
```
with model:
    f_new = gp.conditional('f_new', Xnew=X_new[:, None])
```

Finally we can draw samples of these conditional fits from the posterior using `sample_posterior_predictive`:

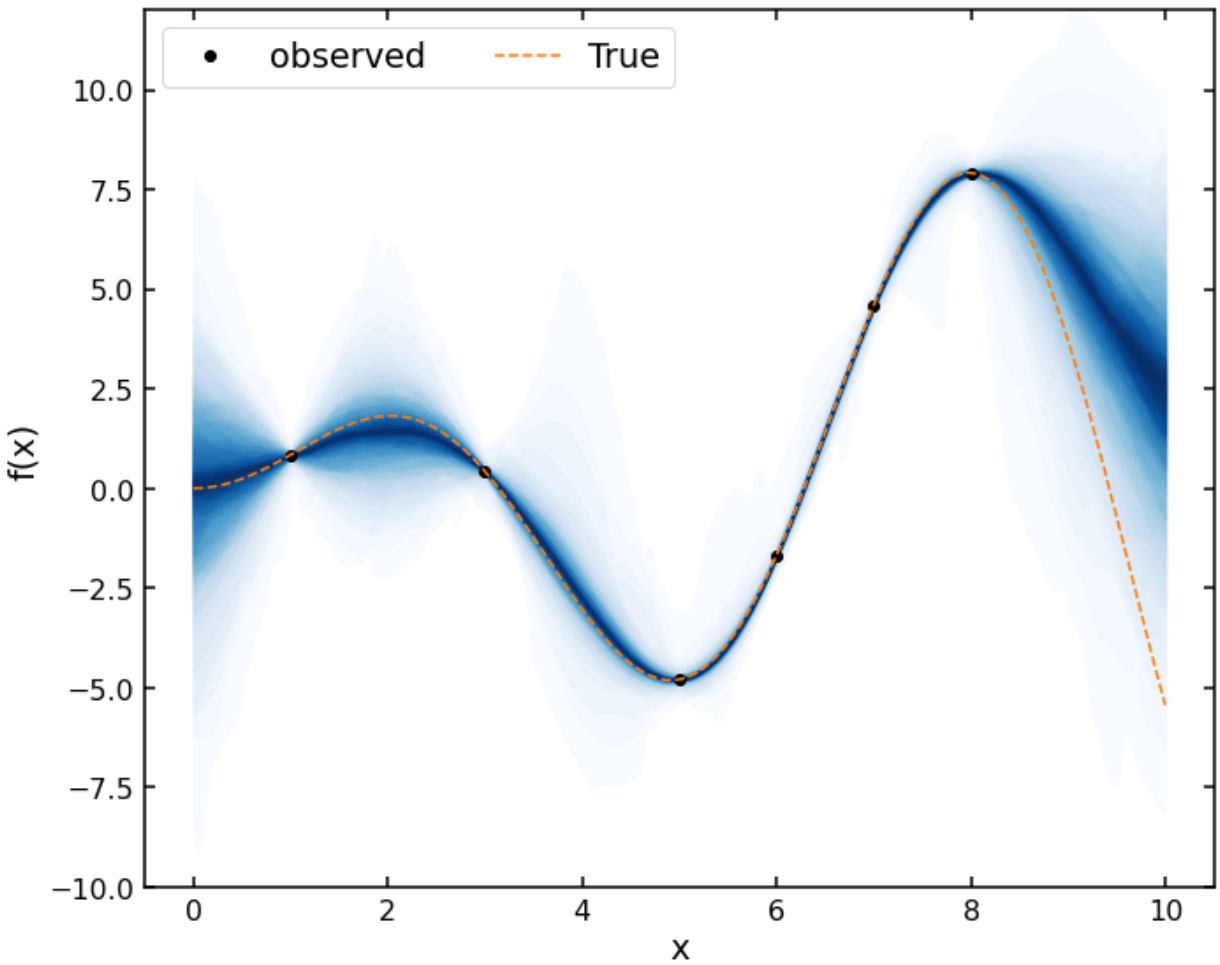
In [15]:
```
with model:
    ppc = pm.sample_posterior_predictive(trace, samples=200, var_names=['f_ne
```

100.00% [200/200 00:08<00:00]

Finally we can make a plot of these samples using the `plot_gp_dist` utility function:

In [16]:
```
plt.figure(6, figsize=(10, 8))
ax = plt.gca()

plt.plot(x1, y1, 'ok', label='observed')

pm.gp.util.plot_gp_dist(
```

```
    ax,
    ppc['f_new'],
    X_new,
    plot_samples=False,
    palette="Blues",
    fill_alpha=1
)

plt.plot(
    X_new,
    X_new * np.sin(X_new),
    '--',
    color='C1',
    label='True'
)

plt.xlabel('x')
plt.ylabel('f(x)')
plt.ylim(-10, 12)
plt.legend(loc='upper left', ncol=2)
plt.tight_layout();
```



We can see that this is slightly different than previous plot, this is due the the nature of the fitting process used. In the first case we used the maximum of the likelihood distribution as a point estimate of the best fit, and in the second case we sampled directly from the likelihood.

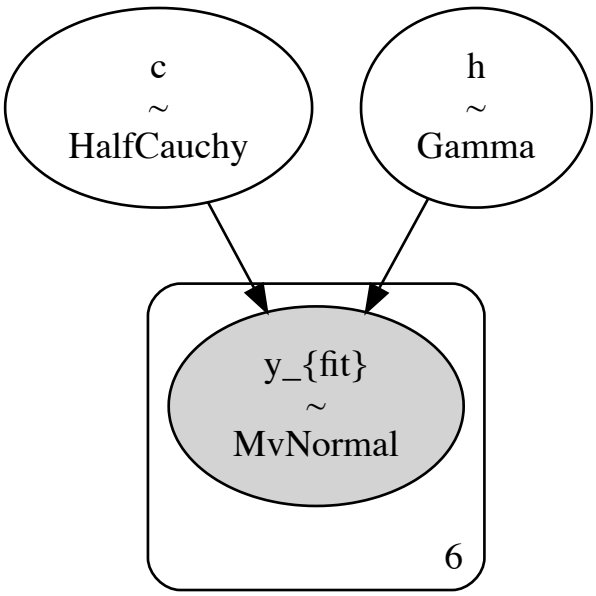
Noisy data

Let's add some noise to the data. We will assume each data point has independent errorbars. These values can be passed directly into the `marginal_likelihood` function instead of the prior we were using before.

In [17]:
```
dy = 0.5 + np.random.random(y1.shape)
y_noise = np.random.normal(0, dy)
y2 = y1 + y_noise
```

In [18]:
```
with pm.Model() as model_noise:
    h = pm.Gamma("h", alpha=2, beta=1)
    c = pm.HalfCauchy("c", beta=5)
    cov = c**2 * pm.gp.cov.ExpQuad(1, ls=h)
    gp = pm.gp.Marginal(cov_func=cov)
    y_fit = gp.marginal_likelihood("y_{fit}", X=X, y=y2, noise=dy)

display(model_noise)
display(pm.model_to_graphviz(model_noise))
```

$$
\begin{aligned}
\text{h\_log\_\_} &\sim \text{TransformedDistribution} \\
\text{c\_log\_\_} &\sim \text{TransformedDistribution} \\
\text{h} &\sim \text{Gamma} \\
\text{c} &\sim \text{HalfCauchy} \\
\text{y\_\{fit\}} &\sim \text{MvNormal}
\end{aligned}
$$



In [19]:
```
with model_noise:
    mp_noise = pm.find_MAP()

display('Best fit kernel: {0:.2f}**2 * ExpQuad(ls={1:.2f})'.format(mp_noise['
```

100.00% [11/11 00:00<00:00 logp = -19.271, ||grad|| = 4.2433e-07]

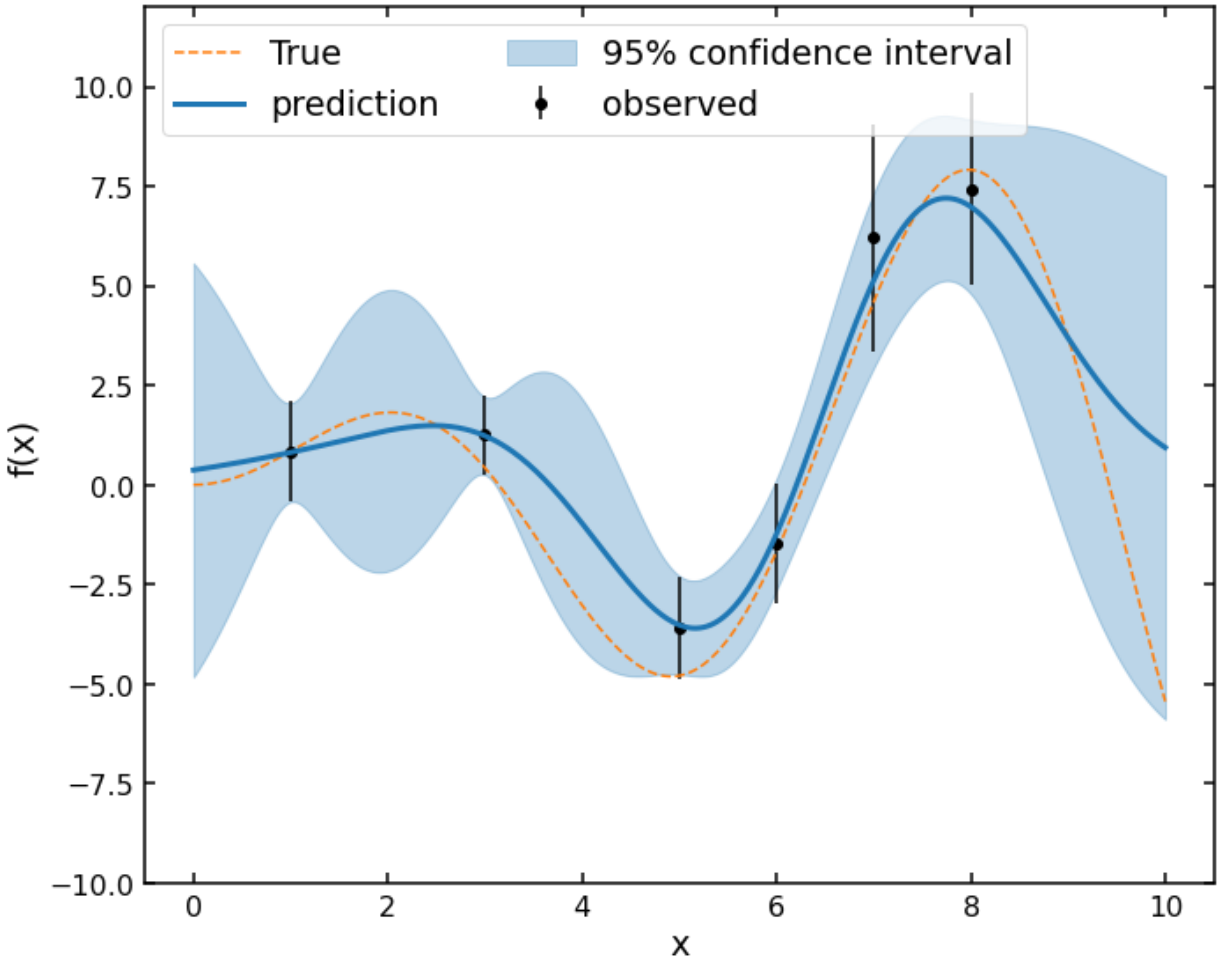
```
'Best fit kernel: 3.58**2 * ExpQuad(ls=1.12)'
```

Plot the results

As before we can interpolate and extrapolate to new points.

In [20]:
```
mu_noise, var_noise = gp.predict(X_new[:,None], point=mp_noise, diag=True)
sd_noise = np.sqrt(var_noise)
```

In [21]:
```
plt.figure(3, figsize=(10, 8))
plt.errorbar(x1, y2, yerr=1.96*dy, fmt='ok', label='observed')

plt.plot(
    X_new,
    X_new * np.sin(X_new),
    '--',
    color='C1',
    label='True'
)

plt.plot(
    X_new.flatten(),
    mu_noise,
    color='C0',
    lw=3,
    zorder=3,
    label='prediction'
)

# plot 95% best fit region
plt.fill_between(
    X_new.flatten(),
    mu_noise - 1.96*sd_noise,
    mu_noise + 1.96*sd_noise,
    color='C0',
    alpha=0.3,
    zorder=1,
    label='95% confidence interval'
)

# labels and legend
plt.xlabel('x')
plt.ylabel('f(x)')
plt.ylim(-10, 12)
plt.legend(loc='upper left', ncol=2)
plt.tight_layout();
```



A Cubic Spline

So far we have been using the `ExpQuad` kernel, but it there are others that can be used. You may have noticed that this method of fitting provides smooth curves that pass through the data points very similar to how a spline fit does. As it turns out, a spline fit is just a special case of a Gaussian process fit. To recreate a cubic spline we can use the following kernel:

$$\text{Cov}(x_1, x_2) = 1 + |x_1 - x_2| \frac{\min(x_1, x_2)^2}{2} + \frac{\min(x_1, x_2)^3}{3}$$

Under the condition that all values of x_1 and x_2 are between the values of 0 and 1. This normalization ensure that the covariance matrix is positive definite. So unlike other kernels we will needed to know the range we plan to extrapolate onto before doing our fit.

This kernel is not built into `pymc3` so we will have to write a custom kernel for it:

In [22]:
```
class CubicSpline(pm.gp.cov.Covariance):
    def __init__(self, dim, x_min=0, x_max=1):
        super(CubicSpline, self).__init__(1, None)
        self.x_min = x_min
        self.x_max = x_max

    def norm(self, X):
        d = self.x_max - self.x_min
        return (X - self.x_min) / d

    def diag(self, X):
```

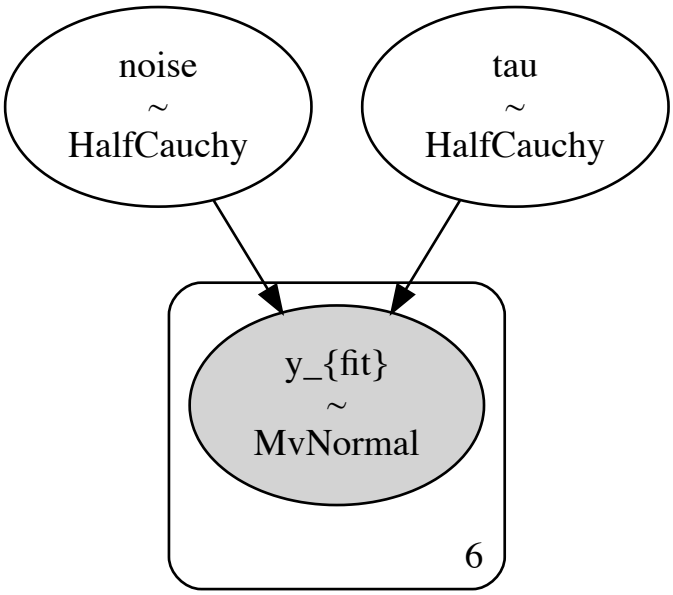
```
        X, _ = self._slice(X, Xs=None)
        X = self.norm(X)
        Xt = tt.flatten(X)
        return 1 + (Xt**3) / 3

    def full(self, X, Xs=None):
        X, Xs = self._slice(X, Xs)
        if Xs is None:
            Xs = X
        X = self.norm(X)
        Xs = self.norm(Xs)
        d = tt.abs_(X - tt.transpose(Xs))
        v = tt.minimum(X, tt.transpose(Xs))
        k = 1 + (0.5 * d * v**2) + ((v**3) / 3)
        return k
```

Now we can fit for this kernel's coefficient. The prior for the noise is pushed to lower values to ensure the fit does not treat the data as "noise only."

In [23]:

```
with pm.Model() as model_cube:
    tau = pm.HalfCauchy('tau', beta=5)
    # we know we will be interpolating onto points between 0 and 10, so inita
    cov = CubicSpline(1, x_min=0, x_max=10) * tau**2
    gp = pm.gp.Marginal(cov_func=cov)
    noise = pm.HalfCauchy('noise', beta=0.1)
    y_fit = gp.marginal_likelihood("y_{fit}", X=x1[:, None], y=y1, noise=nois

display(model_cube)
display(pm.model_to_graphviz(model_cube))
```

$$
\begin{aligned}
\text{tau\_log\_\_} &\sim \text{TransformedDistribution} \\
\text{noise\_log\_\_} &\sim \text{TransformedDistribution} \\
\text{tau} &\sim \text{HalfCauchy} \\
\text{noise} &\sim \text{HalfCauchy} \\
\text{y\_\{fit\}} &\sim \text{MvNormal}
\end{aligned}
$$



In [24]:

```
with model_cube:
    mp_cube = pm.find_MAP()
```

```
display('{0:.2f}**2 * CubicSpline'.format(mp_cube['tau']))
```

100.00% [30/30 00:00<00:00 logp = -27.037, ||grad|| = 0.045779]

```
'96.42**2 * CubicSpline'
```

In [25]:

```
mu_cube, var_cube = gp.predict(X_new[:,None], point=mp_cube, diag=True)
sd_cube = np.sqrt(var_cube)
```

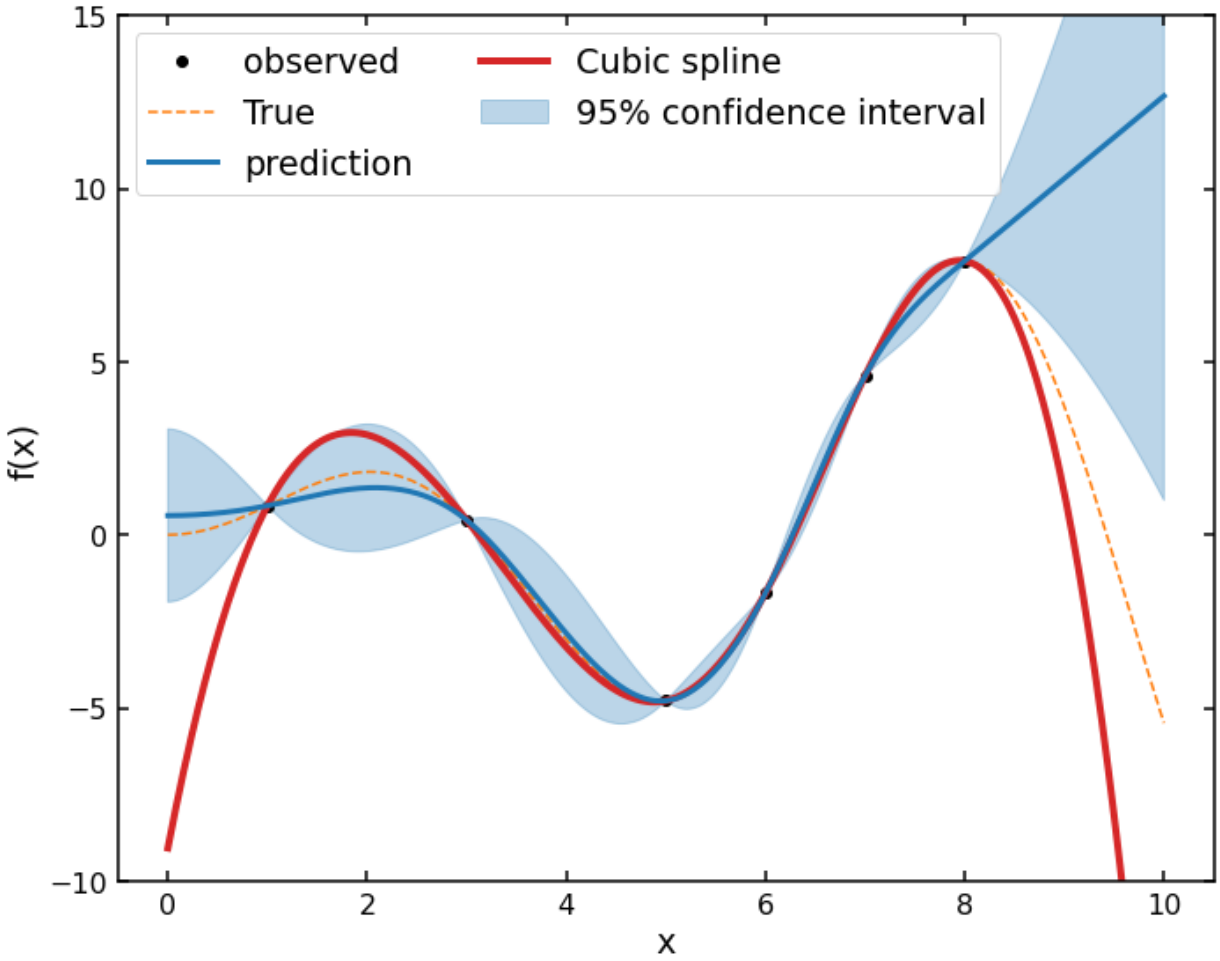
We will also fit a cubic spline to the data and compare it to the Gaussian process fit.

In [26]:

```
tck = interpolate.splrep(x1, y1, k=3)
y_new = interpolate.splev(X_new, tck)
```

In [27]:

```
plt.figure(4, figsize=(10, 8))
plt.plot(x1, y1, 'ok', label='observed')

plt.plot(
    X_new,
    X_new * np.sin(X_new),
    '--',
    color='C1',
    label='True'
)

plt.plot(
    X_new.flatten(),
    mu_cube,
    color='C0',
    lw=3,
    zorder=3,
    label='prediction'
)

# plot 95% best fit region
plt.fill_between(
    X_new.flatten(),
    mu_cube - 1.96*sd_cube,
    mu_cube + 1.96*sd_cube,
    color='C0',
    alpha=0.3,
    zorder=1,
    label='95% confidence interval'
)

plt.plot(X_new, y_new, color='C3', label='Cubic spline', lw=4)

# labels and legend
plt.xlabel('x')
plt.ylabel('f(x)')
plt.ylim(-10, 15)
plt.legend(loc='upper left', ncol=2)
plt.tight_layout();
```



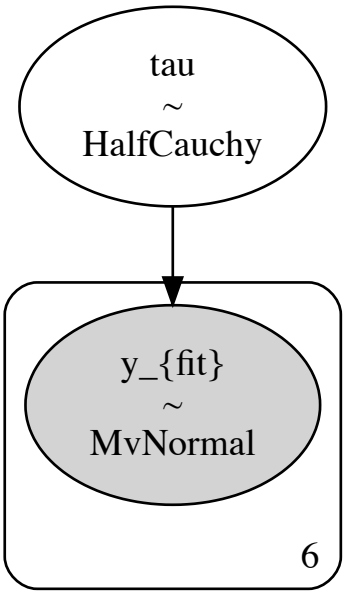
Notice that aside from the end points the Gaussian process the the spline give the same result. Additionally we now have an errorbar estimate for a spline fit!

Adding noise to the spline fit

Now that we have this working we can include measurement errors on each of the data points like we did before.

In [28]:
```
with pm.Model() as model_cube_err:
    tau = pm.HalfCauchy('tau', beta=5)
    cov = CubicSpline(1, x_min=0, x_max=10) * tau**2
    gp = pm.gp.Marginal(cov_func=cov)
    y_fit = gp.marginal_likelihood("y_{fit}", X=x1[:, None], y=y2, noise=dy)

display(model_cube_err)
display(pm.model_to_graphviz(model_cube_err))
```

$$
\begin{aligned}
\text{tau\_log\_\_} &\sim \text{TransformedDistribution} \\
\text{tau} &\sim \text{HalfCauchy} \\
\text{y\_\{fit\}} &\sim \text{MvNormal}
\end{aligned}
$$



In [29]:
```
with model_cube_err:
    mp_cube_err = pm.find_MAP()

display('{0:.2f}**2 * CubicSpline'.format(mp_cube_err['tau']))
```

100.00% [9/9 00:00<00:00 logp = -53.361, ||grad|| = 4.3732]

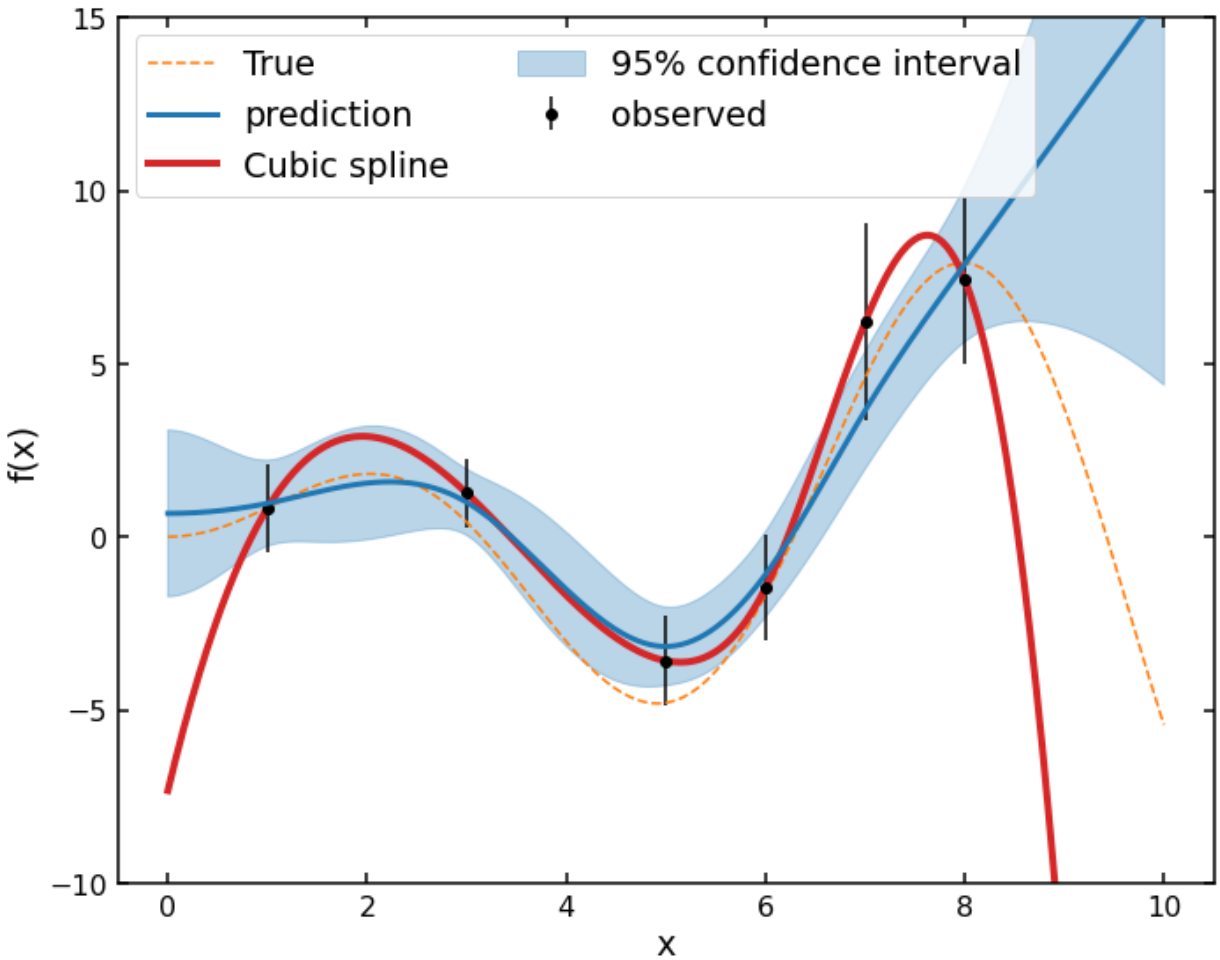
```
'72.48**2 * CubicSpline'
```

In [30]:
```
mu_cube_err, var_cube_err = gp.predict(X_new[:,None], point=mp_cube_err, diag
sd_cube_err = np.sqrt(var_cube_err)
```

In [31]:
```
tck_noise = interpolate.splrep(x1, y2, k=3)
y_new_noise = interpolate.splev(X_new, tck_noise)
```

In [32]:
```
plt.figure(5, figsize=(10, 8))
plt.errorbar(x1, y2, yerr=1.96*dy, fmt='ok', label='observed')

plt.plot(
    X_new,
    X_new * np.sin(X_new),
    '--',
    color='C1',
    label='True'
)

plt.plot(
    X_new.flatten(),
    mu_cube_err,
    color='C0',
    lw=3,
    zorder=3,
    label='prediction'
)

# plot 95% best fit region
plt.fill_between(
    X_new.flatten(),
```

```
    mu_cube_err - 1.96*sd_cube_err,
    mu_cube_err + 1.96*sd_cube_err,
    color='C0',
    alpha=0.3,
    zorder=1,
    label='95% confidence interval'
)

plt.plot(X_new, y_new_noise, color='C3', label='Cubic spline', lw=4)

# labels and legend
plt.xlabel('x')
plt.ylabel('f(x)')
plt.ylim(-10, 15)
plt.legend(loc='upper left', ncol=2)
plt.tight_layout();
```



Other notes

- There are many covariance kernels you can pick;
  - `Constant` : a constant value that can be multiplied or added to any of the other kernels
  - `WhiteNoise` : a white noise kernel
  - `ExpQud` : exponentiated quadratic, smooth kernel parameterized by a length-scale
  - `RatQuad` : rational quadratic, a (infinite sum) mixture of different `ExpQud` 's each with different length-scales
  - `Exponential` : Similar to `ExpQud` but without the square in the exponent.

    - `Marten52`: Marten 5/2 non-smooth generalization of `RBF`, parameterized by length-scale and smoothness
    - `Marten32`: Marten 3/2 non-smooth generalization of `RBF`, parameterized by length-scale and smoothness
    - `Cosine`: periodic kernel built with `cos`
    - `Linear`: a non-stationary kernel that can be used to fit a line
    - `Polynomial`: a non-stationary kernel commonly polynomial like fit
    - `Periodic`: periodic function kernel, parameterized by a length-scale and periodicity
- There are also three mean functions to choose from:
    - `Zero`: The mean is all zeros (this is the default)
    - `Constant`: The mean is a constant value (i.e. a global y-offset)
    - `Linear`: The mean is a linear function (i.e. a polynomial)
- See https://docs.pymc.io/en/stable/pymc-examples/examples/gaussian_processes/GP-MeansAndCovs.html for examples of each kernel and mean function

- All `X` positions must be unique
- The computational complexity is $O(N^3)$ where N is the number of data point. If you have a large number of data points you can use PYMC3's variational inference methods (https://docs.pymc.io/api/inference.html#variational) that replace the (complex) posterior with simpler approximations that are faster to compute.

In [ ]: