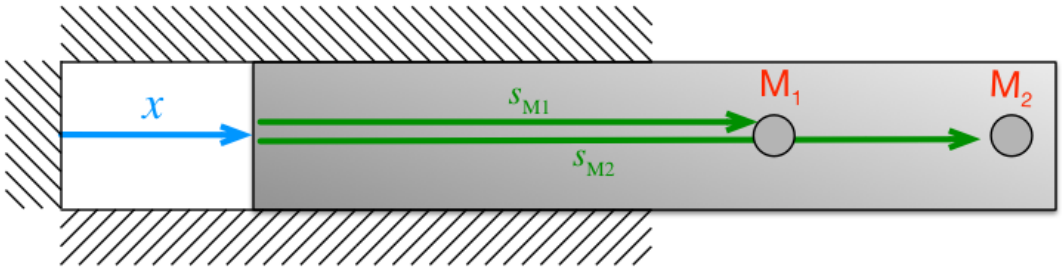
APPENDIX C: Bayesian inverse kinematics basics

This Appendix describes a simple Bayesian inverse kinematics (IK) model to introduce the computational details of the main manuscript's Bayesian IK model.

Consider the slider mechanism pictured below. This mechanism consists of a single rigid slider which moves to a global position x, and to which two markers ("M1" and "M2") are rigidly fixed. Let's assume that we know that the local marker positions are: s_{M1} = 35 mm and s_{M2} = 45 mm.



Let's also assume that we have imperfectly measured the global marker positions (x'_{M1} and x'_{M2}) as 46.0 and 55.0 mm, respectively. The IK problem is to find the slider position x given the marker measurements x'_{M1} and x'_{M2}.

Approach 1: Least-squares

The true global marker positions can be expressed in terms of our unknown variable x as follows:

$$\begin{aligned} x_{\mathrm{M1}} &= x + 35 \\ x_{\mathrm{M2}} &= x + 45 \end{aligned}$$

The sum of the the squared measurement errors is:

$$f(x) = (x_{\mathrm{M1}} - 46)^2 + (x_{\mathrm{M2}} - 55)^2$$

which reduces to:

$$\begin{aligned} f(x) &= (x - 11)^2 + (x - 10)^2 \\ &= (x^2 - 22x + 121) + (x^2 - 20x + 100) \\ &= 2x^2 - 42x + 221 \end{aligned}$$

The minimum value of this function is the least-squares (LS) solution, and is given when the derivative is zero:

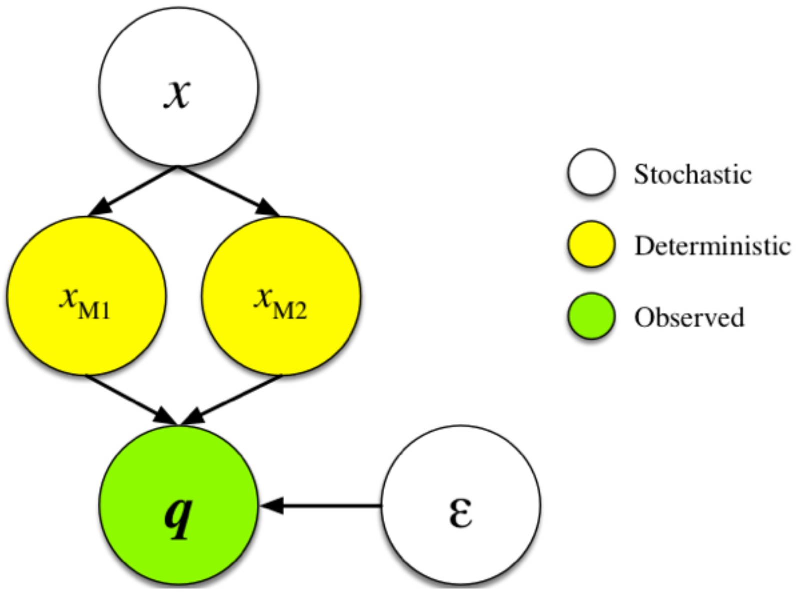
$$\frac{d}{dx} f(x) = 4x - 42 = 0$$
$$x = \frac{42}{4}$$

so our least-squares estimate is: x = 10.5 mm, and the (minimized) measurement error is 0.5 mm for each marker.

Approach 2: Bayesian

In contrast to the least-squares (LS) approach, which minimizes measurement error, **the Bayesian approach maximizes model parameters' mean posterior probabilities** given observed (noisy) data. A roughly equivalent interpretation is that the Bayesian approach maximizes the probability of observing the given data. This is done by optimizing the parameters which characterize a stochastic system.

To implement Bayesian IK we start with a model of our measurements as depicted below.



In this model there are two stochastic variables: x and ϵ, where x is the slider displacement we wish to estimate, and where ϵ is measurement error. Once the numerical value of x is known, then the true values of the deterministic variables (x_{M1} and x_{M2}) are also known. Similarly, once the random measurement error values ϵ are known, then the values of the (modeled) observed variables q are known. Here q is a generalized observation vector containing the values of all observed variables, in this case: x'_{M1} and x'_{M2}.

Let's start to implement this model in Python by defining known values: the true local positions and the observed measurment values.

In [1]:

```
%matplotlib notebook

import numpy as np
from matplotlib import pyplot

### local positions:
sM1 = 35.0
sM2 = 45.0
### measurements:
xpM1 = 46.0
xpM2 = 55.0
```

Let's next implement a function which will generate random datasets q according to our measurement model. Let's arbitrarily set the current slider position to x = 8 mm, and let's assume that our measurement error ϵ is Gaussian with a standard deviation of 0.5 mm.

In [2]:

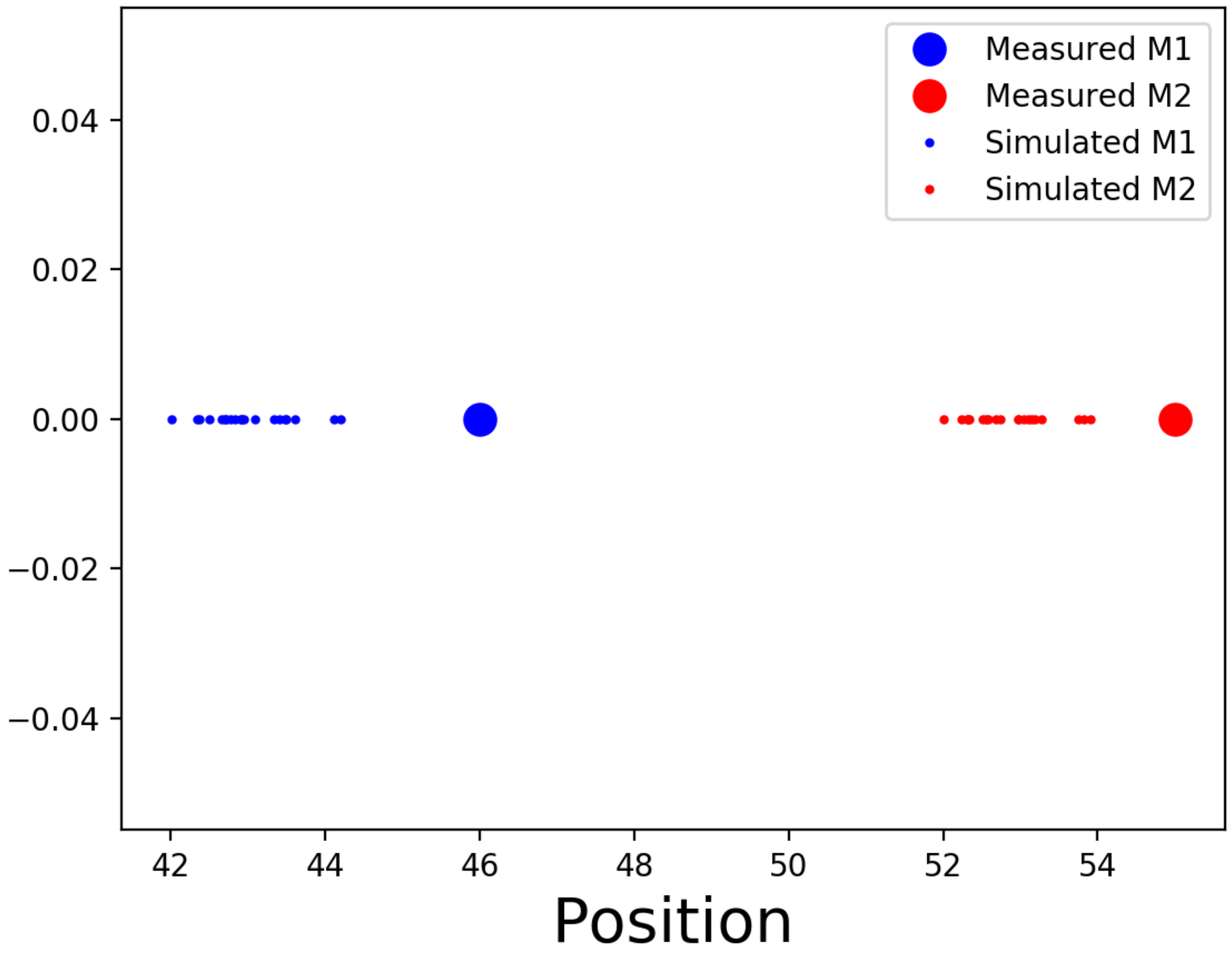
```
import random

noise_sd = 0.5  #mm

#define measurement model:
def random_measurement(x):
    xM1 = x + sM1 + random.gauss(0, noise_sd)
    xM2 = x + sM2 + random.gauss(0, noise_sd)
    return [xM1, xM2]

#generate random data (20 observations):
random.seed(0)
x = 8.0
X = np.array([random_measurement(x)  for i in range(20)])
print( X )
```

```
[[ 43.4708577   52.30171095]
 [ 42.66014278  53.18525178]
 [ 42.49182555  52.96393999]
 [ 43.08959824  52.58445039]
 [ 42.34548132  53.09694387]
 [ 43.49662485  52.67650918]
 [ 42.83316601  53.82283588]
 [ 42.72055512  52.74292167]
 [ 44.20205967  52.2344587 ]
 [ 43.39823292  51.99817574]
 [ 42.70151863  53.75184044]
 [ 43.61071821  52.54943992]
 [ 42.77315063  53.04011652]
 [ 42.37094835  53.27611   ]
 [ 44.11378865  52.32237925]
 [ 42.00923346  53.14412187]
 [ 42.94043834  53.90216497]
 [ 42.91981891  52.97467014]
 [ 42.90456306  52.50469688]
 [ 43.33651499  52.33795877]]
```

This represents 20 different sets of measurements we could expect if our model is accurate. The results can be visualized as follows:

In [3]:

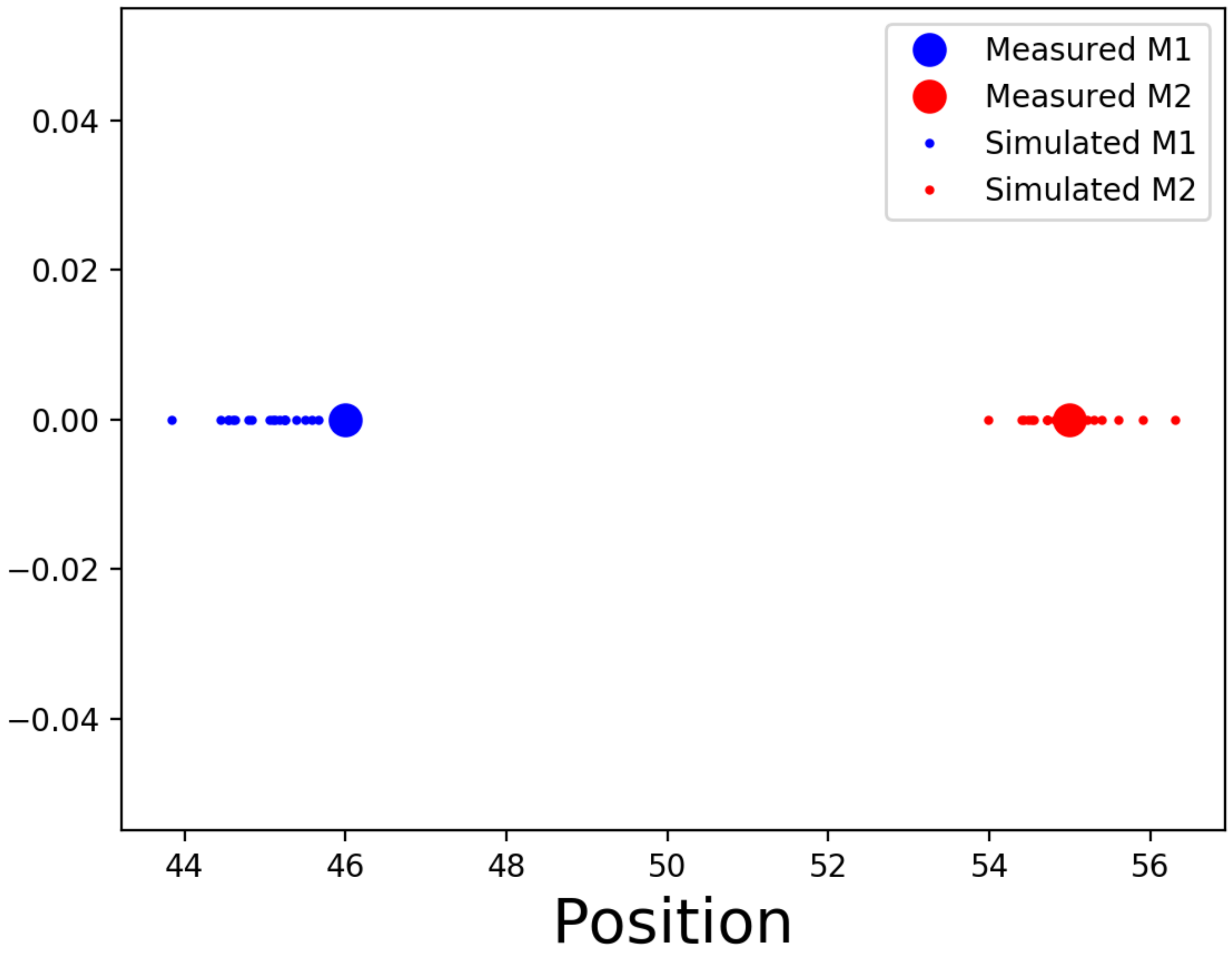
```
def myplot(X):
    ax = pyplot.axes()
    ax.plot(xpM1, 0, 'bo', markersize=10, label='Measured M1')
    ax.plot(xpM2, 0, 'ro', markersize=10, label='Measured M2')
    ax.plot(X[:,0], [0]*X.shape[0], 'b.', markersize=4, label='Simulated M1')
    ax.plot(X[:,1], [0]*X.shape[0], 'r.', markersize=4, label='Simulated M2')
    ax.legend()
    ax.set_xlabel('Position', size=20)

pyplot.figure()
myplot(X)
```



For the current position (x=8) and assumed measurement error (ϵ=0.5) the simulated data do not overlap with the measured data, implying that the current value of x is unlikely to have produced the observed (measured) data. Let's change x to 10 and try again.

In [4]:

```
X = np.array([random_measurement(10)  for i in range(20)])
pyplot.figure()
myplot(X)
```



There is now much greater overlap between the simulated and measured data, implying that x=10 is more likely to have produced the observed data than x=8. To find the optimum value for x using Bayesian inference, let's first implement our observations model in PyMC.

In [5]:

```
import pymc

tau = 1 / 0.5**2    #measurement precision ( 1 / SD^2 )
x   = pymc.Uniform("x", -20, 20)  #prior distribution for x

@pymc.deterministic
def observations_model(x=x):
    xM1 = x + sM1  #global position of Marker 1
    xM2 = x + sM2  #global position of Marker 2
    return [xM1, xM2]
q = pymc.Normal("q", observations_model, tau)
```

In the code snippet above a variable "`tau`" is created to specify our assumed measurement precision, where precision (τ) is defined as the inverse of variance (σ^2):

$$\tau \equiv \frac{1}{\sigma^2}$$

Since we have assumed a standard deviation of 0.5 mm, our assumed precision is: τ=4.

Next we specify a prior distribution for the slider position x. The prior distribution used above is uniform between $x = -20$ and $x = +20$, implying that the slider can be anywhere in that range with equal probability. Note that the variable "`x`" is a stochastic PyMC variable. This means that we can generate random values for this variable based on its current distribution as follows:

In [6]:

```
print( x.random() )
print( x.random() )
print( x.random() )
print( x.random() )
print( x.random() )
```

```
-11.322780354906413
17.681369915052528
1.7715768482792242
-19.929250130439403
-12.219761095987046
```

You can see that these random values span the range [-20, 20]. Every time we call `random()`, PyMC changes `x`'s "`value`" attribute as follows.

In [7]:

```
print( x.value )  #the final value from above
print( x.value )  #the value hasn't changed yet
x.random()
print( x.value )  #now it has
```

```
-12.219761095987046
-12.219761095987046
13.85512994873043
```

The last step in the code snippet above is to create a PyMC observations model. First we create a function called `observations_model` which computes the expected marker positions based on the value of `x` and returns both values. Next we tell PyMC that the result of `observations_model` should be regarded as stochastic, with a normal (Gaussian) distribution that has a mean of the true marker positions (given `x`) and a precision of `tau`. The "`@pymc.deterministic`" statement is a function decorator which alerts PyMC to regard this function as a deterministic variable, and to use `x`'s value inside that function. Just like `x`, `q` is now a PyMC stochastic variable and we can generate random `q` values as follows.

In [8]:

```
x.set_value(0)      #set the slider position
print( q.random() ) #random marker positions given x=0
print( q.random() )
print( q.random() )
print( q.random() )
print( q.random() )
```

```
[ 34.77644934  44.59803815]
[ 34.72185988  45.36694377]
[ 35.02122191  44.94128098]
[ 35.39952566  44.97303196]
[ 35.27585701  44.84364442]
```

If we instead choose the least-squares solution $x = 10.5$ we can generate random data which looks much more like our observed values of: $x'_{\mathrm{M1}} = 46$ and $x'_{\mathrm{M2}} = 55$.

In [9]:

```
x.set_value(10.5)
print( q.random() )
print( q.random() )
print( q.random() )
print( q.random() )
print( q.random() )
```

```
[ 45.12567859  55.87785164]
[ 45.42097263  56.38726659]
[ 44.74580806  55.22009218]
[ 45.64699282  55.63735726]
[ 45.48082171  55.29616333]
```

Instead of directly altering x in this manner, and then finding its optimal value in a numerical optimization sense, PyMC instead uses the observed data to alter x's current distribution, and then examine x's and q's posterior distributions in attempts to maximize their maxima. In other words, rather than searching for a singla optimium value for x, PyMC recognizes that both x and q are probabilistic, so attempts to mutually maximize their posterior probabilities, given the observed data. In order to achieve that we first have to tell PyMC what our observations were. This can be done by slightly altering our q variable as follows:

In [10]:

```
qobs = pymc.Normal("qobs", observations_model, tau, value=[46,55], observed=Tr
ue)
```

Next, maximizing the posterior distributions can be achieved with a Markov Chain Monte Carlo (MCMC) simulation:

In [11]:

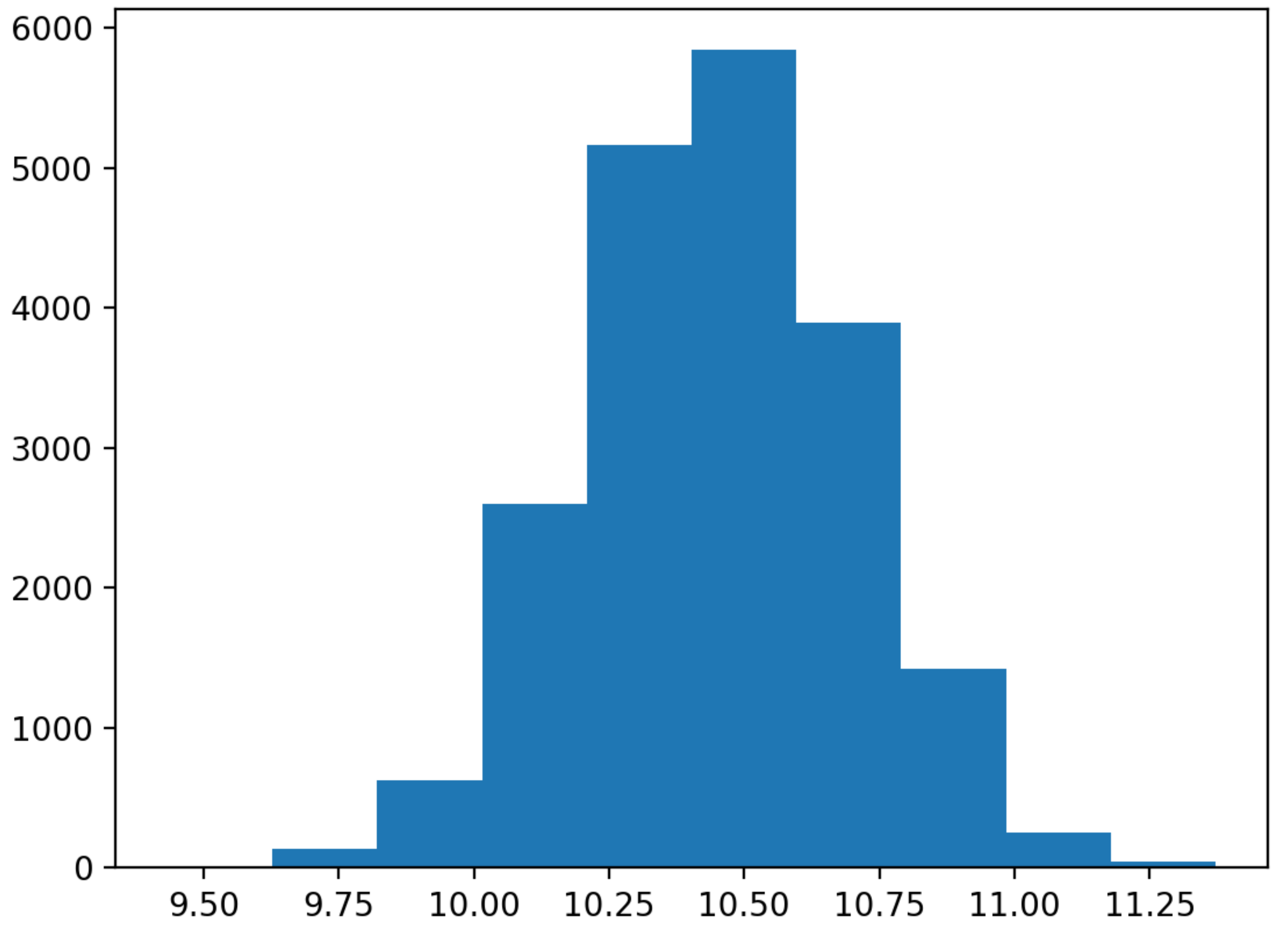
```
mcmc = pymc.MCMC([qobs, x])
mcmc.sample(40000, 20000)
```

```
 [-----------------100%-----------------] 40000 of 40000 complete
in 1.5 sec
```

The first line initializes a model for MCMC simulation using all stochastic variables. The second line iteratively samples from their posterior distributions, discarding the first 20,000 iterations, and retaining the final 20,000 iterations for the final posterior distributions. Let's examine our posterior distribution for x.

In [12]:

```
X = mcmc.trace('x')[:]
pyplot.figure()
pyplot.hist(X);
```



This posterior distribution spans the least-squares solution (x=10.5), and the maximum a posteriori likelihood (MAP) is very close to the least-square solution:

In [13]:

```
print( X.mean() )  #MAP
```

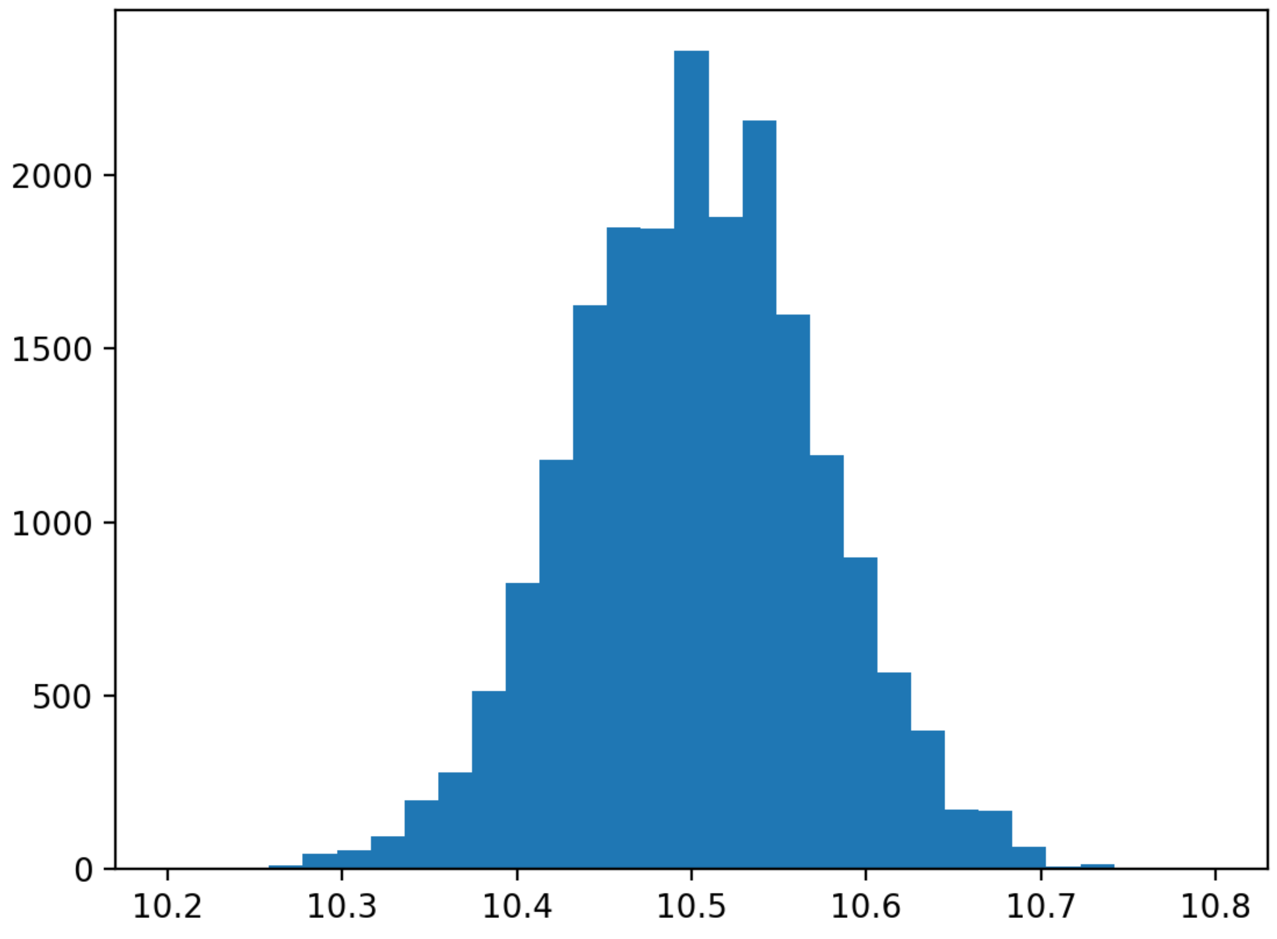
```
10.4486464775
```

However, as depcited in the posterior distribution above there is also a fair bit of uncertainty in our estimate for x. Surely values of x=10 or x=11.0 aren't the best choices, but this result suggests that they are still plausible, based on our modeling assumptions. So let's change the assumptions and examine the effects. Let's first try increasing the presumed measurement precision (from SD=0.5 mm to SD=0.1 mm).

In [14]:

```
tau  = 1 / 0.1**2    #measurement precision ( 1 / SD^2 )
x    = pymc.Uniform("x", -20, 20)  #prior distribution for x
@pymc.deterministic
def observations_model(x=x):
    xM1 = x + sM1  #global position of Marker 1
    xM2 = x + sM2  #global position of Marker 2
    return [xM1, xM2]
qobs = pymc.Normal("qobs", observations_model, tau, value=[46,55], observed=True)

mcmc = pymc.MCMC([qobs, x])
mcmc.sample(40000, 20000)

pyplot.figure()
X = mcmc.trace('x')[:]
pyplot.hist(X, range=(10.2,10.8), bins=31);
```

```
 [-----------------100%-----------------] 40000 of 40000 complete
in 1.4 sec
```

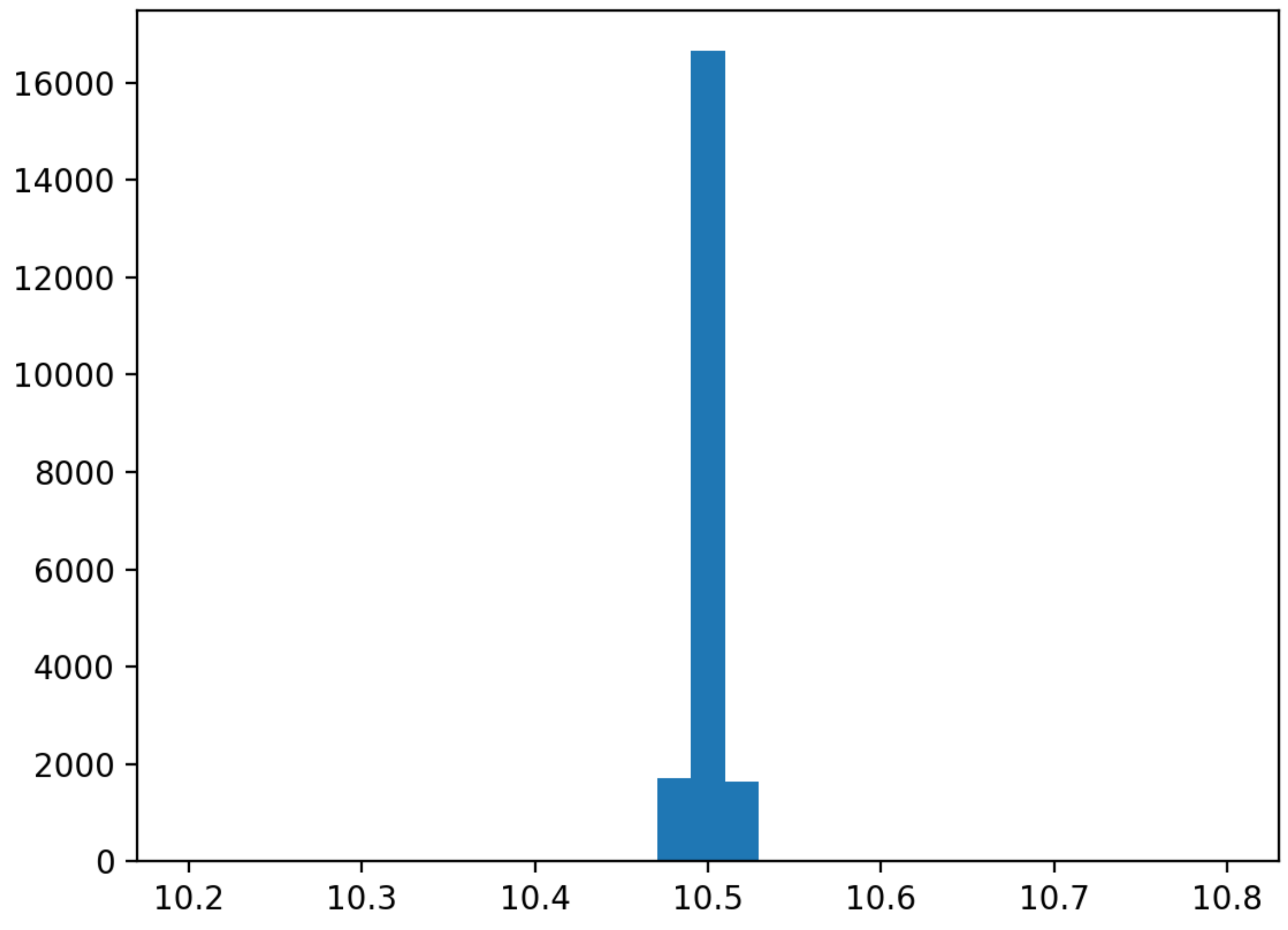


Greater assumed measurement precision has narrowed the posterior. Let's next try a presumption of extremely precise measurements (SD=0.01 mm).

In [15]:

```
tau  = 1 / 0.01**2    #measurement precision ( 1 / SD^2 )
x    = pymc.Uniform("x", -20, 20)  #prior distribution for x
@pymc.deterministic
def observations_model(x=x):
    xM1 = x + sM1  #global position of Marker 1
    xM2 = x + sM2  #global position of Marker 2
    return [xM1, xM2]
qobs = pymc.Normal("qobs", observations_model, tau, value=[46,55], observed=Tr
ue)

mcmc = pymc.MCMC([qobs, x])
mcmc.sample(40000, 20000)

pyplot.figure()
X = mcmc.trace('x')[:]
pyplot.hist(X, range=(10.2,10.8), bins=31);
```

```
 [-----------------100%-----------------] 40000 of 40000 complete
in 1.4 sec
```



Clearly our assumptions regarding τ have affected our posterior distribution. In fact, it has become clear that **the least-squares solution is equivalent to the Bayesian solution under an assumption of infinite measurement precision**. An equivalent interpretation is that as the presumed measurement precision increases, the Bayesian solution converges to x=10.5, with no uncertainty, just like the least-squares solution.

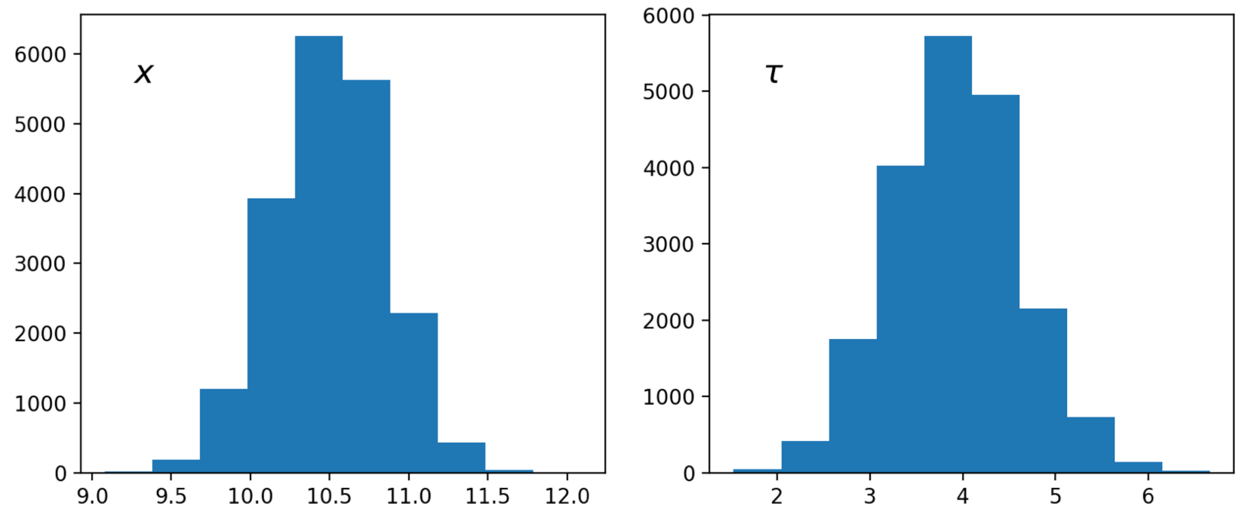
This seems somewhat strange: the least-squares solution has minimized measurement error, so how can that be equivalent to an assumption of no measurement error? This apparent paradox is resolved by the opposite perspectives on measurements that the two approaches adopt. The least-squares approach regards the measured data as hard, and minimizes an explicit function of those measurements. Contrastingly, the Bayesian approach regards the measured data as soft -- just one manifestation of an infinite set of plausible measurements. Thus setting infinite precision forces the Bayesian approach to regard the data as hard, and consequently to assume zero variability in its estimate of x.

This raises an important point: what actual value of precision should we use? Most measurement device manufacturers provide technical specifications regarding measurement accuracy values, so one option is to use those precision data. A second, more Bayesian option, is to regard precision itself as stochastic, and to then let Bayesian inference decide the most likely precision, based on the data. This can be implemented as follows:

In [16]:

```
x    = pymc.Uniform("x", -20, 20)  #prior distribution for x
tau  = pymc.Normal("tau", 4, 2, value=4)  #prior distribution for tau

@pymc.deterministic
def observations_model(x=x):
    xM1 = x + sM1  #global position of Marker 1
    xM2 = x + sM2  #global position of Marker 2
    return [xM1, xM2]
qobs = pymc.Normal("qobs", observations_model, tau, value=np.array([46,55]), o
bserved=True)

mcmc = pymc.MCMC([qobs, x, tau])   #now tau also appears in the collection of
stochastic variables
mcmc.sample(40000, 20000)

pyplot.figure(figsize=(10,4))
X   = mcmc.trace('x')[:]
TAU = mcmc.trace('tau')[:]
ax  = pyplot.subplot(121);  ax.hist(X);   ax.text(0.1, 0.85, r'$x$', size=16,
transform=ax.transAxes)
ax  = pyplot.subplot(122);  ax.hist(TAU); ax.text(0.1, 0.85, r'$\tau$', size=1
6, transform=ax.transAxes)

print( X.mean() )
print( TAU.mean() )
```

```
 [-----------------100%-----------------] 40000 of 40000 complete
in 2.9 sec
```



```
10.5030225838
3.92712295259
```

Our new definition of "`tau`" implies that we are reasonably certain that the true measurement precision is 4, but it might also be slightly larger or slightly smaller. This flexibility permits Bayesian inference to simultaneous find the most likely range of values for both x and τ. The final results depicted above suggest the following:

- The observed measurements tell us that the true value of x is most likely very close to 10.5, but due to measurement errors we cannot be completely confident in this value. The true value of x almost certainly lies in the range [9.5, 11.5].
- The observed measurements suggest that our presumed precision model is probably fine, and that a value close to 4 is most likely the true precision value, but due to random measurement errors we cannot be certain. The true value of tau almost certainly lies in the range [2, 6].

Summary

The mechanical slider model and marker measurements discussed in this Appendix are rudimentary but highlight a variety of important differences between the least-squares (LS) and Bayesian approaches to inverse kinematics (IK). While the LS approach minimizes an explicit function of the measured data, the Bayesian approach instead maximizes the maximim a posteriori probabilities (MAP) of all stochastic variables. Consequenty, the Bayesian approach propogates measurement uncertainty to the IK solution, but the LS solution does not. The Bayesian and LS squares approaches yield equivalent IK solutions under an assumption of infinite measurement precision. Last, results above show that Bayesian IK can be sensitive to modeling assumptions, including especially to the prior distributions.