In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

Assignment 2

Welcome to Assignment 2! In this assignment you are allowed to work ***individually or in pairs***. It is worth 30 points in total. Exercises 1 is worth 5 points, exercise 2 is worth 10 points, and 3 can give you 15 points.

There is a 5 point minimum for passing this assignment (you need to pass all four assignments to be able to pass the course).

Submission details: Your submission should contain two pdf's.

1. A pdf version of your filled out colaboratory on Canvas. You can do this by pressing `cmd/ctrl+p` (you know the drill from there).
2. For exercise 1, you need to hand in your hand-written solutions in a LaTeX pdf. We only accept solutions written in LaTeX, i.e. not Word or any other text editor. We recommend Overleaf, if you do not already have a favourite LaTeX editor (which is also provided by KTH).

Contents

In this assignment we will experiment with the following topics (not necessarily in this order):

- Conjugate priors
- Gibbs sampling
- Metropolis-Hastings
- Convergence diagnostics
- Probabilistic modelling
- Bayesian decision theory
- The Predictive posterior distribution

1. Conjugate Priors

Conjugate distributions are very important and widely used distributions in Bayesian statistics. Having a closed-form expression for the posterior distribution provides great convenience.

It is important to feel confident with derivations and being able to find closed-form expressions. For this task, we give you well-known likelihood functions & prior distributions and ask you to *derive* the closed form expressions of the posterior distributions.

For this exercise, you need to hand in your hand-written solutions in a LaTeX pdf. We only accept solutions written in LaTeX.

Q1.1 There are N i.i.d. data points sampled from a Normal distribution; $\mathcal{N}(x_i|\mu,\sigma^2)$ for $i \in [N]$ where μ is the mean and σ^2 is the variance. Assume σ^2 is known and the mean has the Normal prior distribution $\mathcal{N}(\mu|\mu_0,\sigma_0^2)$. Show the posterior distribution of mean is

$$\mathcal{N}\left(\mu \mid \frac{\sigma_0^2 N}{\sigma^2+N\sigma_0^2}\bar{x}+\frac{\sigma^2}{\sigma^2+N\sigma_0^2}\mu_0,\ \left(\frac{1}{\sigma_0^2}+\frac{N}{\sigma^2}\right)^{-1}\right)$$

where $\bar{x}$ is the sample mean.

Q1.2 There are N i.i.d. data points sampled from a Normal distribution; $\mathcal{N}(x_i|\mu,\sigma^2)$ for $i \in [N]$ where μ is the mean and σ^2 is the variance. Assume μ is known and the variance has the Inverse-Gamma prior distribution $\mathcal{IG}(\sigma^2|\alpha,\beta)$. Show the posterior distribution of variance is

$$\mathcal{IG}\left(\sigma^2 \mid \alpha+\frac{N}{2},\ \beta+\frac{1}{2}\sum_{i=1}^{N}(x_i-\mu)^2\right)$$

Q1.3 There are N i.i.d. data points sampled from a Normal distribution; $\mathcal{N}(x_i|\mu,\tau^{-1})$ for $i \in [N]$ where μ is the mean and $\tau = 1/\sigma^2$ is the precision. Assume μ has the Normal prior $\mathcal{N}(\mu|\mu_0,(n_0\tau)^{-1})$ and the precision has the Gamma prior distribution $Ga(\tau|\alpha,\beta)$. Show the posterior distributions of mean and precision are

$$\mathcal{N}\left(\mu \mid \frac{N}{N+n_0}\bar{x}+\frac{n_0}{N+n_0}\mu_0,\ (N\tau+n_0\tau)^{-1}\right)$$
$$Ga\left(\tau \mid \alpha+\frac{N}{2},\ \beta+\frac{1}{2}\sum_{i=1}^{N}(x_i-\bar{x})^2+\frac{Nn_0}{2(N+n_0)}(\bar{x}-\mu_0)^2\right)$$

where $\bar{x}$ is the sample mean.

2. The Two-Dimensional Gaussian

Consider the following posterior,

$$p(\theta|x) = \mathcal{N}\left(\theta|x,\Sigma\right),$$

where $\theta = (\mu_1,\mu_2)$ are the unknown means of the Gaussian distribution that generated $x = (x_1,x_2)$, i.e.

$$x \sim \mathcal{N}(x|\theta,\Sigma).$$

Furthermore, the covariance matrix is known,

$$\Sigma = \begin{pmatrix} 1 & 0.8 \\ 0.8 & 1 \end{pmatrix},$$

and the prior on θ is uniform.

Q2.1 Use Bayes' rule to derive the considered posterior, given what you know above. That is, clearly specify the joint distribution, the likelihood function, the prior distribution and the marginal likelihood, and then show that $p(\theta|x) = \mathcal{N}(\theta|x, \Sigma)$.

Bayes' Theorem: $p(\theta|x) \propto p(x|\theta)p(\theta)$

Likelihood Function:

$$p(x|\theta) = \mathcal{N}(x|\theta, \Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\theta)^T\Sigma^{-1}(x-\theta)\right)$$

Our prior $p(\theta) \propto C$ is considered non-informative since it does not favor any values

The marginal likelihood $p(x) = \int p(x|\theta)p(\theta)d\theta$ which is a normalization constant that ensures the gaussian integrates to 1

Therefor our posterior distribution:

$$p(\theta|x) \propto p(x|\theta) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\theta)^T\Sigma^{-1}(x-\theta)\right) = \mathcal{N}(\theta|x, \Sigma)$$

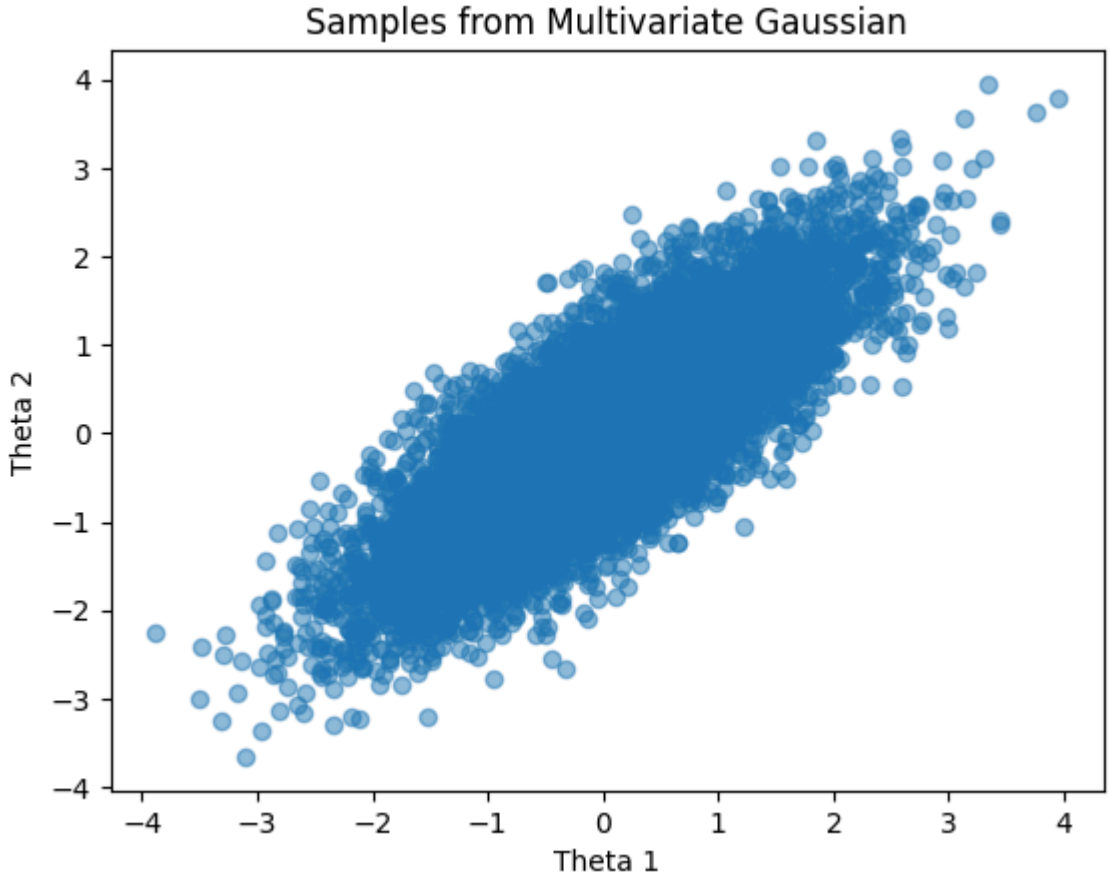
Q.E.D

Although this problem is not intractable, we are now going to use MCMC to sample from the posterior. We will assume that $x = (0, 0)$.

Before we start using MCMC, let's utilize that we know the posterior distribution by sampling from it directly.

Q2.2 Sample 10000 values of θ from $p(\theta|x)$. Visualize the samples in a scatter plot.

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

mean = [0, 0]
cov = [[1, 0.8], [0.8, 1]]
samples = np.random.multivariate_normal(mean, cov, 10000)

plt.scatter(samples[:, 0], samples[:, 1], alpha=0.5)
plt.xlabel('Theta 1')
plt.ylabel('Theta 2')
plt.title('Samples from Multivariate Gaussian')
plt.show()
```



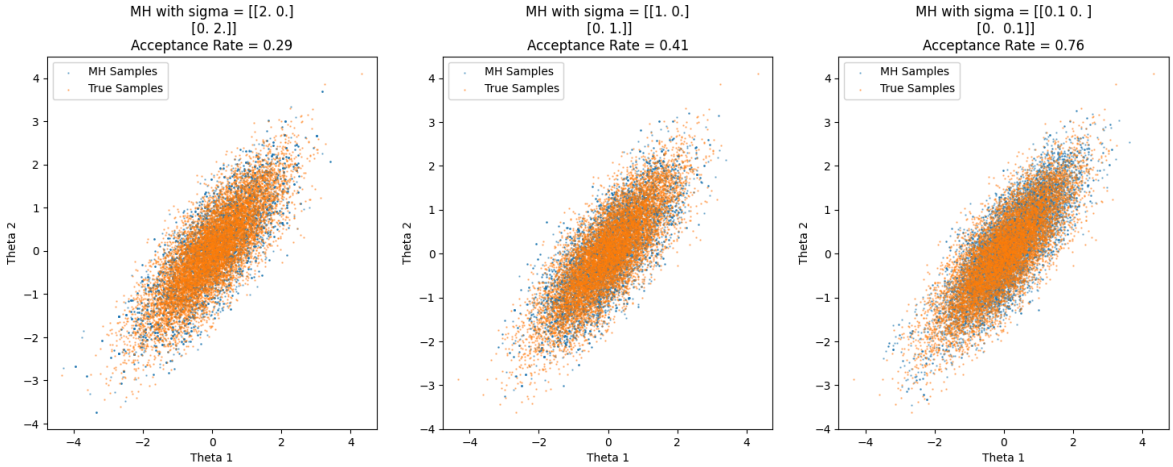
Now we shall generate and compare samples from two MCMC algorithms, specifically the Metropolis-Hastings (MH) algorithm and Gibbs sampling. Recall that in MCMC algorithms we are evaluating our Markov chain samples using the joint distribution, and not the posterior.

Q2.3 Let the proposal distribution $q(\theta'|\theta^t)$ be a Gaussian with σ^2 variance. Write an MH algorithm that samples 10k points from $p(\theta|x)$, and scatter plot the samples.

Tune σ^2 so that the MH samples are similar to those generated in Q2.1, and report σ^2.

In [3]:
```
from scipy.stats import multivariate_normal, norm, uniform, beta

np.random.seed(0)

# if we reject x+1 = x
nsamples = 10000
burn_in = 1000

def metropolis_hastings_step(x_0, sigma):
    x_star = np.random.multivariate_normal(x_0, sigma)
    acceptance_prob = min(1, target_distribution(x_star) / target_distrib
    if uniform.rvs() < acceptance_prob:
        return x_star, True
    else:
        return x_0, False

def target_distribution(x):
```

```
    return multivariate_normal.pdf(x, mean, cov)

def metropolis_hastings(nsamples, x_0, sigma, burn_in):
    x = np.zeros((nsamples + burn_in, 2))
    x[0] = np.random.multivariate_normal(x_0, sigma)
    acceptances = 0
    for i in range(1, nsamples + burn_in):
        x[i], accepted = metropolis_hastings_step(x[i-1], sigma)
        if accepted:
            acceptances += 1
    acceptance_rate = acceptances / (nsamples + burn_in)
    return x[burn_in:], acceptance_rate


sigmas = [ 2*np.eye(2), np.eye(2), 0.1*np.eye(2)]
true_samples = np.random.multivariate_normal([0,0], [[1,0.8],[0.8,1]], ns

fig, axes = plt.subplots(1, 3, figsize=(18, 6))

for idx, sigma in enumerate(sigmas):
    mh_samples, acceptance_rate = metropolis_hastings(nsamples, mean, sig
    axes[idx].scatter(mh_samples[:, 0], mh_samples[:, 1], alpha=0.5, labe
    axes[idx].scatter(true_samples[:, 0], true_samples[:, 1], alpha=0.5,
    axes[idx].set_title(f'MH with sigma = {sigma}\nAcceptance Rate = {acc
    axes[idx].set_xlabel('Theta 1')
    axes[idx].set_ylabel('Theta 2')
    axes[idx].legend()

plt.show()
```



Q2.4 Derive the conditionals $p(\theta_1|\theta_2^t, x)$ and $p(\theta_2|\theta_1^t, x)$. Partitioning of variables and matrices:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}$$

Formula for Conditional Mean of a Multivariate Normal Distribution:

Given the joint distribution:

$$p(\theta|X) \sim \mathcal{N}\left(\theta| \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}\right)$$

$p(\theta_1|\theta_2^t, x)$

$$\theta_1|\theta_2 = E[\theta_1|\theta_2 = \theta_2^t] = x_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \theta_2^t)$$

$$\Sigma_{1|2} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}$$

$p(\theta_2|\theta_1^t, x)$

$$E[\theta_2|\theta_1 = \theta_1^t] = x_2 + \Sigma_{21}\Sigma_{11}^{-1}(x_1 - \theta_1^t)$$

$$\Sigma_{2|1} = \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}$$

Full conditionals with our known Σ:

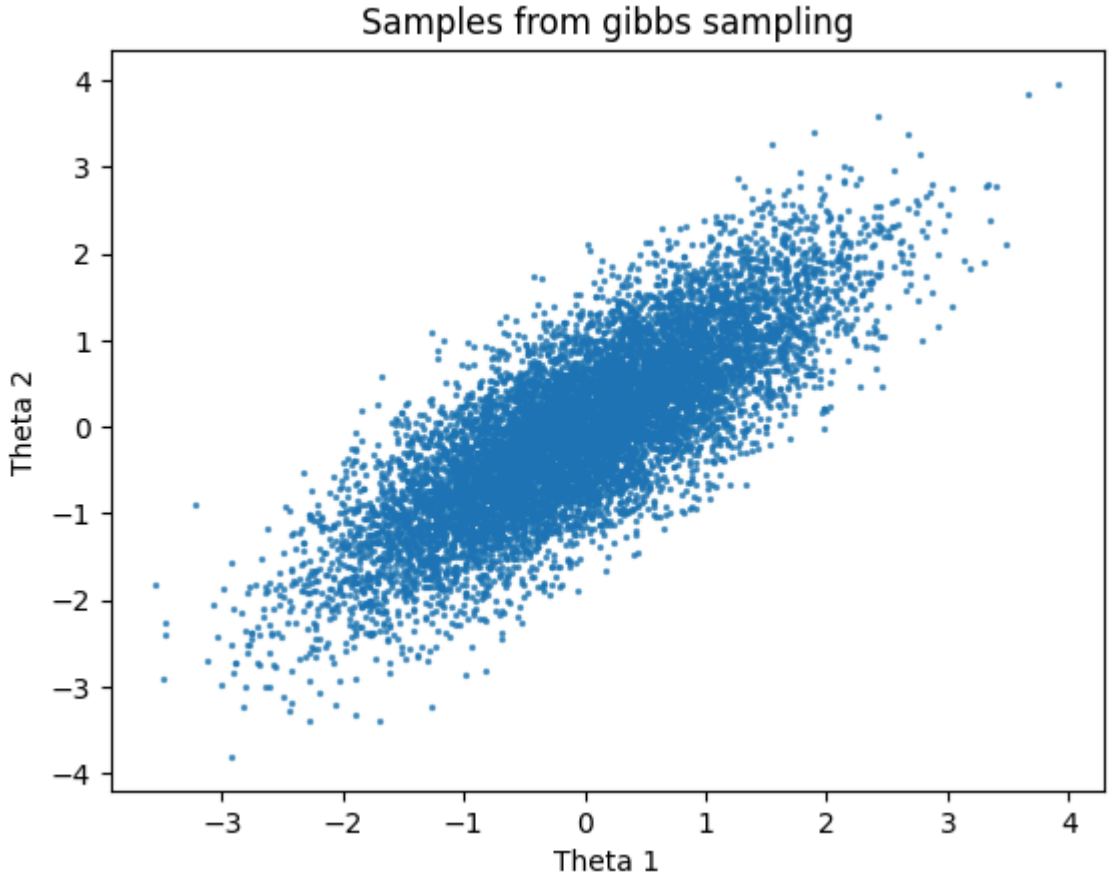
$$\rho = 0.8$$

$$p(\theta_1|\theta_2^t, x) \sim N(x_1 + \rho(x_2 - \theta_2^t), 1 - \rho^2)$$

$$p(\theta_2|\theta_1^t, x) \sim N(x_2 + \rho(x_1 - \theta_1^t), 1 - \rho^2)$$

Write a Gibbs sampler which you then use in order to generate 10k samples from $p(\theta|x)$. Scatter plot the results.

In [4]:
```
np.random.seed(0)

sigma = [[1, 0.8], [0.8, 1]]
num_samples = 10000
thetas = np.zeros((num_samples, 2))
rho = sigma[0][1] / np.sqrt(sigma[0][0] * sigma[1][1])

def theta1_given_2(rho, theta_2):
    return np.random.normal(rho*theta_2, np.sqrt(1 - rho**2))

def theta2_given_1(rho, theta_1):
    return np.random.normal(rho*theta_1, np.sqrt(1 - rho**2))

for i in range(1, num_samples):
    theta_1 = theta1_given_2(rho, thetas[i-1, 1])
    theta_2 = theta2_given_1(rho, theta_1)
    thetas[i] = [theta_1, theta_2]

plt.scatter(thetas[:, 0], thetas[:, 1], alpha=0.7, s=2)
plt.xlabel('Theta 1')
plt.ylabel('Theta 2')
plt.title('Samples from gibbs sampling')
plt.show()
```



3. The Bayesian Burglars

In this assignment we will act as burglars who decide whether to break into a house or not using Bayesian decision theory.

Let $C(x, h)$ be the cost of decision x given the state, h. $x \in \{0, 1\}$ is the decision to break in or not, and $h \in \{0, 1\}$ tells us if the house owners are home or not. The break-in would last during the time interval Δt and we can not know h during Δt before taking a decision. In order to be data driven decision makers, we have collected N data points, $\mathcal{D} = \{y_i, \tau_i\}_{i=1}^{N}$, where y_i is a binary variable indicating if the home owners are home, and τ_i is the time point of the observation.

Based on our data, we seek to take a Bayesian approach to the problem. Namely, we want to compute the *Bayes risk* associated with breaking and entering *or not* breaking and entering during a given time slot

$$\mathcal{R}(\Delta t) = \sum_{x=0}^{1} \mathbb{E}_{p(h|\Delta t, \mathcal{D})} \left[C(x, h) \right],$$

where

$$p(h|\Delta t, \mathcal{D}) = \int_0^1 p(h, \theta|\Delta t, \mathcal{D}) d\theta = \int_0^1 p(h|\theta) p(\theta|\Delta t, \mathcal{D}) d\theta,$$

is the predictive posterior distribution, and θ is the probability that the house owners are home.

Take a moment to appreciate how powerful the Bayes risk quantity is. Yes, here we are using it in a silly setting, but it is widely applicable to all sorts of advanced and simple problems where it makes sense to quantify the risk or expected gain.

Let's rewrite the risk expression in terms of the predictive posterior

$$\mathcal{R}(\Delta t) = \sum_{x=0}^{1} \mathbb{E}_{\int p(h|\theta)p(\theta|\Delta t,\mathcal{D})d\theta}\left[C(x,h)\right]$$

and write out the posterior over θ using Bayes' rule

$$p(\theta|\Delta t,\mathcal{D}) = \frac{p(\mathcal{D}|\theta,\Delta t)p(\theta)p(\Delta t)}{p(\mathcal{D},\Delta t)}.$$

Unfortunately, $p(\theta|\Delta t,\mathcal{D})$ is intractable as it involves the computation of a nasty marginal likelihood. However, we *do* know how to compute the joint distribution (the numerator), so we decide to solve the problem using a Metropolis-Hastings algorithm!

To summarize the objective of the exercise, we want to compute the risk function above. To do this, we need to compute an expectation with respect to the intractable posterior predictive distribution. Instead, we can approximate the posterior predictive by sampling from the posterior over θ via MH. Then, using the approximation of $p(h|\Delta t,\mathcal{D})$, the risk function can be computed for both choices of x. The exercise is divided into a series of subproblems.

The distributions and functions in the generative model

The likelihood function is a conditional Bernoulli likelihood

$$p(\mathcal{D}|\theta,\Delta t) = \prod_{i=1}^{N} \theta^{1\{(y_i=1)\wedge(\tau_i\in\Delta t)\}}(1-\theta)^{1\{(y_i=0)\wedge(\tau_i\in\Delta t)\}}(1/2)^{1\{\tau_i\notin\Delta t\}},$$

where $\wedge$ is the logical "AND" character.

The θ prior is a Beta distribution

$$p(\theta) = \text{Beta}(\theta|\alpha,\beta)$$

with hyperparams α, β.

The Δt prior, i.e. the prior belief of during which time slot it is appropriate to perform the break in, is unnormalized and factorisable

$$p(\Delta t) = p(t_u|t_l)p(t_l)$$

with

$$p(t_l) = 1\{t_l \in [0, 3600 \cdot 24]\}$$

and

$$p(t_u|t_l) = 1\{t_u \in [t_l + 3600, 3600 \cdot 24]\},$$

where t_u denotes upper bound of the break-in time slot, and t_l the lower bound. In other words, when the break in ends and starts. At shortest, the break in has to last one hour, 3600 seconds, which is reflected in the trunctation using the lower bound in $p(t_u|t_l)$.

The proposal distributions in the MH algorithm

The proposal for θ is a mirrored uniform distribution with step size ϵ, conditioned on the previous state, $\theta^{(k-1)}$.

The proposal for t_l is a normal distribution with step size σ^2, conditioned on the previous state value for t_l, $t_l^{(k-1)}$.

The proposal for t_u is a truncated exponential distribution with CDF

$$F(t_u|t_l, \lambda) = 1 - e^{-\lambda(t_u - U)},$$

and rate parameter λ. Here $U = 3600 + t_l$ is the size of the truncation, which ensures that proposed ending times for the break in occur at least 3600 seconds after the proposed start of the break in, t_l.

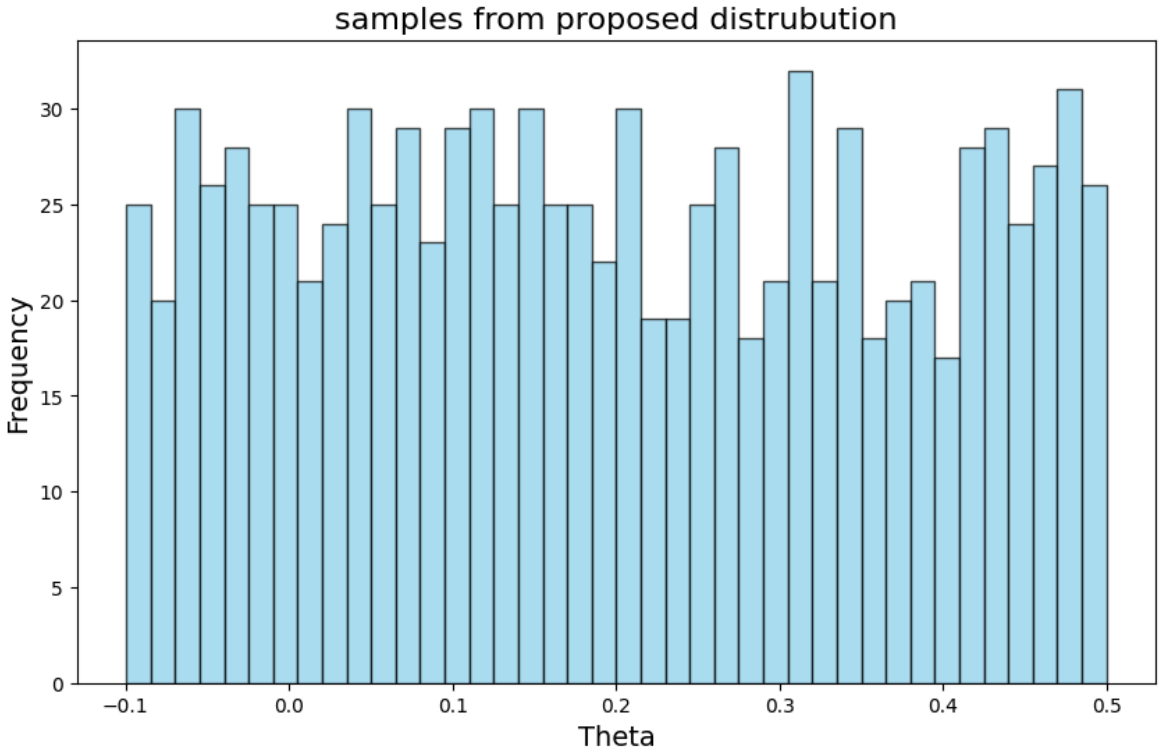
Q3.1 In the following three subproblems, you will define and visualize the proposals

Q3.1.1. A mirrored (one-dimensional) uniform distribution only admits density in a specified interval, *mirroring back* whatever part of its support falls outside the interval back on to the interval. The density that falls outside the interval is superpositioned on top of the non-mirrored density.

Using indicator functions, formulate the pdf $q(\theta|\theta^{(k-1)})$. Implement the distribution in numpy (you should be able to sample from it, and evaluate its likelihood). Show are you are able to sample from it by sampling 1000 samples and plotting them in a histogram. Use $\theta^{(k-1)} = 0.2$ and $\epsilon = 0.3$ when sampling.

In [5]:
```
epsilon = 0.3
theta_k_minus_1 = 0.2

np.random.seed(0)

def get_sample_q_theta_given_theta_k_minus_1(theta_k_minus_1, epsilon):
```

```
    lower_limit = theta_k_minus_1 - epsilon
    upper_limit = theta_k_minus_1 + epsilon
    proposal = np.random.uniform(lower_limit, upper_limit)
    return proposal


samples = [get_sample_q_theta_given_theta_k_minus_1(theta_k_minus_1, epsi

plt.figure(figsize=(10, 6))
plt.hist(samples, bins=40, alpha=0.7, color='skyblue', edgecolor='black')
plt.xlabel('Theta', fontsize=14)
plt.ylabel('Frequency', fontsize=14)
plt.title('samples from proposed distrubution', fontsize=16)
plt.show()
```



Q3.1.2 Formulate the pdf of $q(t_l|t_l^{(k-1)})$ with a variable σ^2. You will need to choose a step size, σ^2, before starting the experiment.

$$q(t_l|t_l^{(k-1)}) \sim Gaussian(t_l^{k-1}, \sigma^2)$$

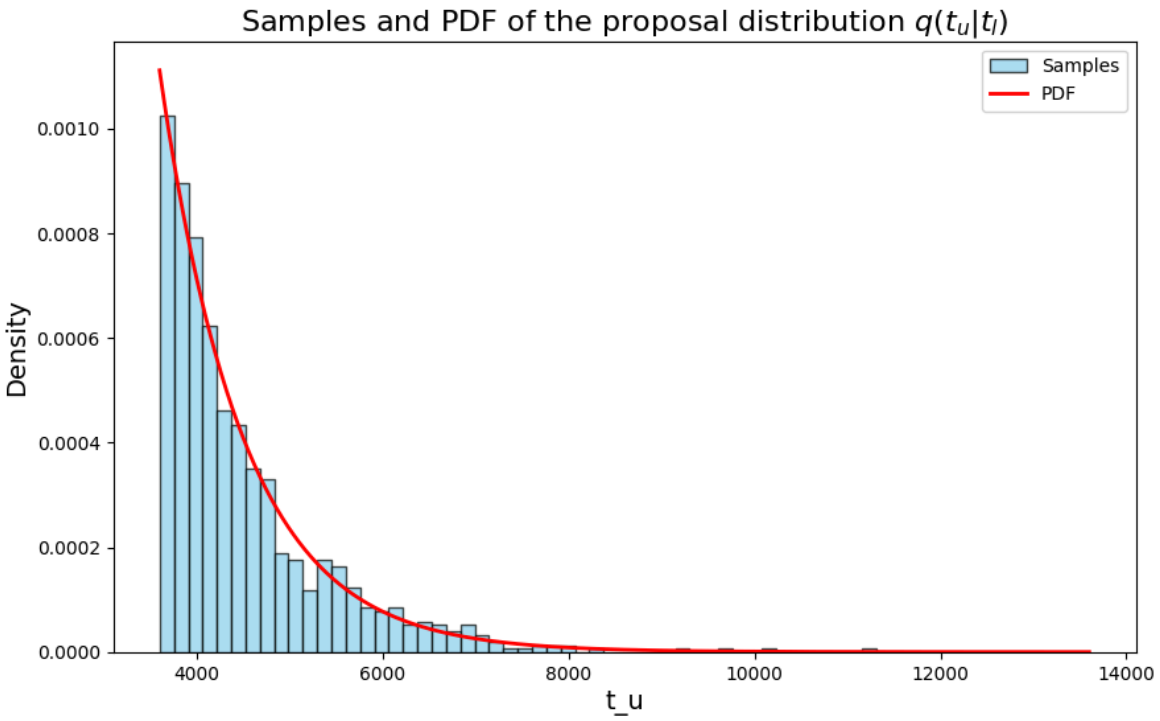
Q3.1.3 Use the given CDF to formulate the pdf of the proposal distribution, $q(t_u|t_l)$. Sampling from this pdf is not directly supported in numpy. Use the inverse-sampling trick to sample 1000 samples from it as you let $t_l = 0$. Visualize the histrogram of the samples with a plot.

$$y = 1 - \exp(-\lambda(tu - U)) \Leftrightarrow ln(1-y) = -\lambda(tu - U) \Leftrightarrow tu = U - \frac{ln(1-y)}{\lambda}$$

Inverse CDF $= U - \frac{ln(1-y)}{\lambda}$

In [6]:
```
np.random.seed(0)
lambda_param = 1/900
U = 3600
```

```
num_samples = 1000

y = np.random.uniform(0, 1, num_samples)

t_u_samples = U - np.log(1 - y) / lambda_param

def q_t_u_given_t_l(sample, lambda_param, U):
    return lambda_param * np.exp(-lambda_param * (sample - U))

t_u_values = np.linspace(U, U + 10000, 1000)
pdf_values = q_t_u_given_t_l(t_u_values, lambda_param, U)

plt.figure(figsize=(10, 6))
plt.hist(t_u_samples, bins=50, alpha=0.7, color='skyblue', edgecolor='bla
plt.plot(t_u_values, pdf_values, 'r-', lw=2, label='PDF')
plt.xlabel('t_u', fontsize=14)
plt.ylabel('Density', fontsize=14)
plt.title('Samples and PDF of the proposal distribution $q(t_u|t_l)$', fo
plt.legend()
plt.show()
```

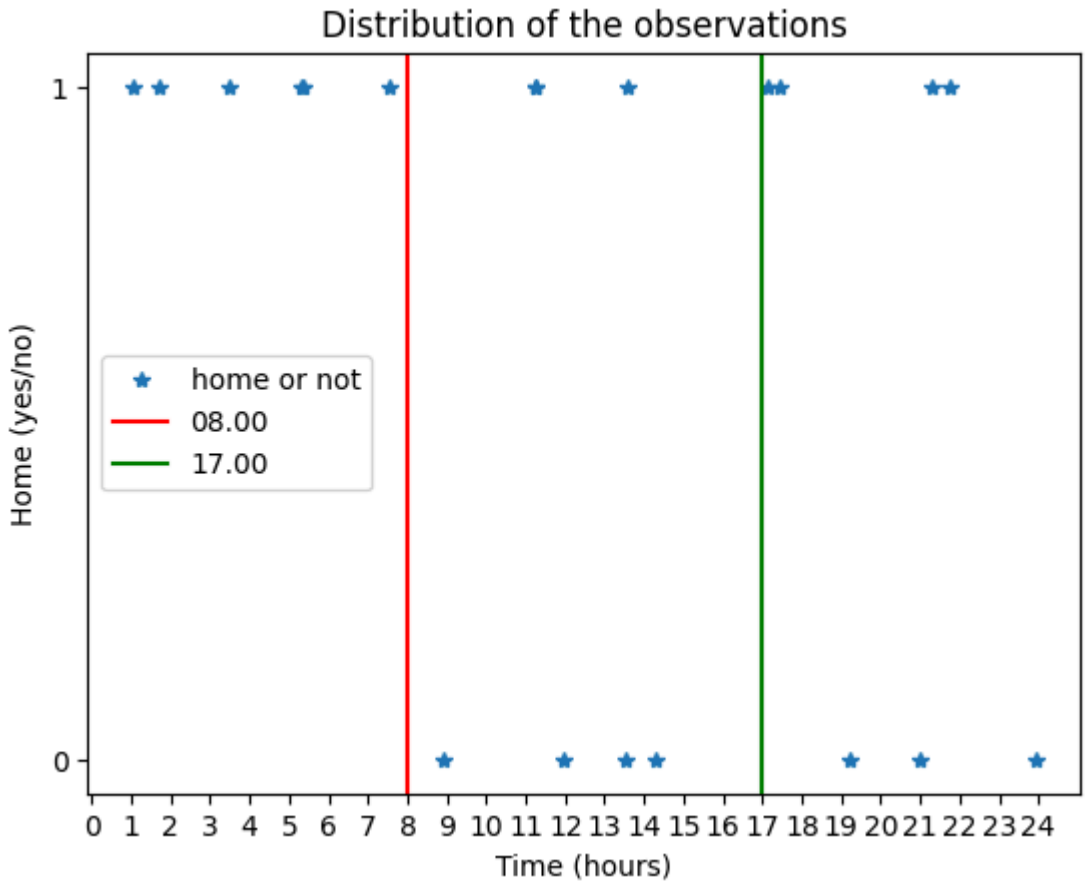


Generating data and implementing the algorithm

Next we need data. Use the following cell to generate the data and take a moment to reflect about the data-generating process. Beyond this cell, we will forget about the underlying data-generating process.

In [7]:
```
def generate_data(N=30):
    # we more frequently go to collect data during the middle of the day
    part_of_day = np.argmax(
        np.random.multinomial(1, [1/4, 1/2, 1/4], N), axis=-1
```

```
        )
    time_intervals_low = np.array([0.0, 8 * 3600, 17 * 3600])
    time_intervals_high = np.array([8 * 3600, 17 * 3600, 24 * 3600])

    # after deciding which part of the day to go collect data,
    # the exact time point is uniformly chosen
    tau_i = np.random.uniform(low=time_intervals_low[part_of_day],
                      high=time_intervals_high[part_of_day])

    # probabilities that the owners are home or not depend on the part of
    home_prob = np.array([0.9, 0.2, 0.7])

    # randomly sample the event that the owners are home
    u = np.random.uniform(0, 1, N)
    y_i = (u < home_prob[part_of_day]).astype(float)

    # the data is composed of N combinations of home-or-not events and ti
    return y_i, tau_i


# do not change the seed or the number of observations, N
np.random.seed(0)
D = generate_data(N=20)

idx = np.argsort(D[1])
plt.plot(D[1][idx] / 3600., D[0][idx], '*', label='home or not')
plt.axvline(8, 0, 1, c='r', label='08.00')
plt.axvline(17, 0, 1, c='g', label='17.00')
plt.xlabel('Time (hours)')
plt.ylabel('Home (yes/no)')
plt.yticks([0, 1])
plt.xticks(np.arange(25))
plt.legend(loc='center left')
plt.title('Distribution of the observations')
plt.show()
```



Q3.2 Implement the MH algorithm. Run it until convergence, where you decide when the chains have convergenced. Motivate your convergence statement by using the convergence diagnostics discussed in the lectures. Convey your arguments with plots and numbers. A mandatory convergence diagnostic is to run and compare multiple chains.

Before you are ready to implement the algorithm, you need to select the two step sizes, ϵ and σ^2. Specify clearly your choices, and motivate them. There are no right or wrong choices, but better or worse alternatives. It is OK to revise your choices after evaluating the acceptance ratio of the MH algorithm, but here you should share your a priori beliefs.

Finally, let $\lambda = 900$ and $\alpha = \beta = 1$

Answer question regarding choice of ϵ here

Epsilon defines range from which we sample from our uniform distrubution for θ centered around the state θ^{k-1} i.e it controls how big the space we sample our proposal state. Should be less than 1. We start of with 0.1

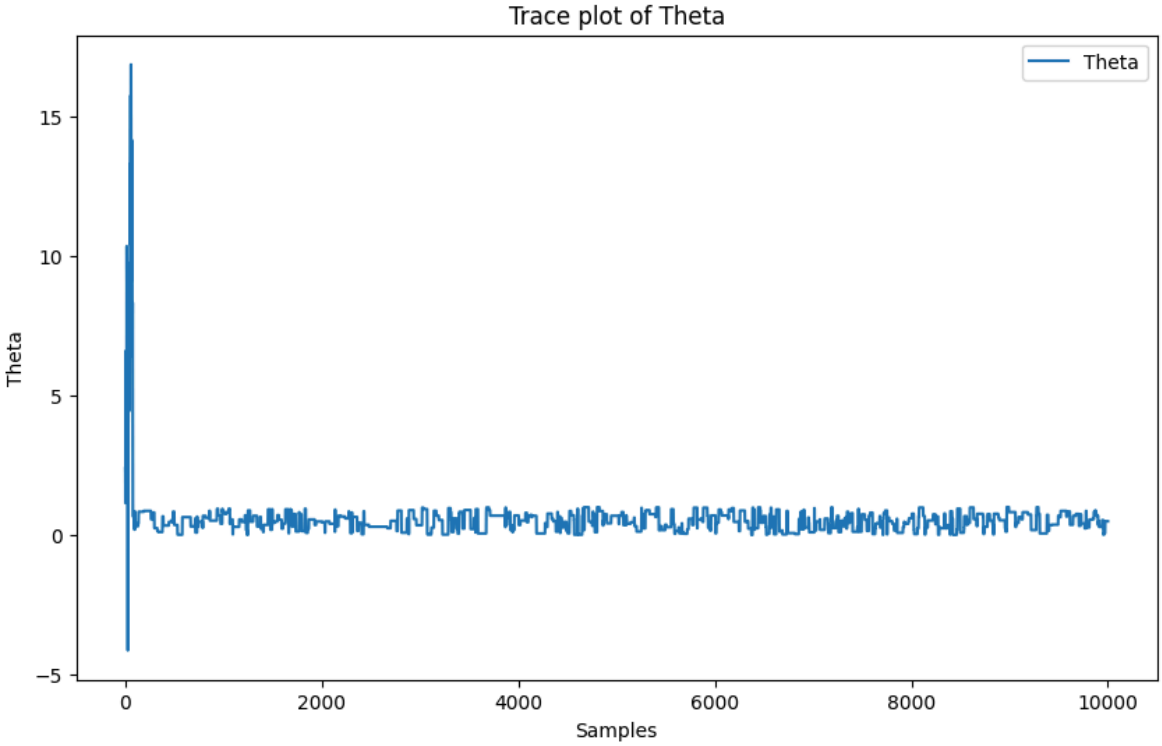
Answer question regarding choice of σ^2 here Lets set to 0.1 aswell

In [8]:
```
from scipy.stats import beta, norm
import numpy as np

def proposal_for_theta(theta_k_minus_1, epsilon): #90%
    lower_limit = theta_k_minus_1 - epsilon
```

```
    upper_limit = theta_k_minus_1 + epsilon
    proposal = np.random.uniform(lower_limit, upper_limit)
    return proposal

def proposal_for_tl(t_l_k_minus_1, sigma):
    t_l = np.random.normal(t_l_k_minus_1, sigma)
    return t_l

def proposal_for_tu(y, lambda_param, U):
    y = np.random.uniform(0, 1)
    t_u = U - np.log(1 - y) / lambda_param
    return t_u

def target_likelihood(theta, t_l, t_u, D):
    y, tau = D

    for i in range(len(y)):
        first_term = 1
        second_term = 1

        #print(f"y: {y[i]}, tau: {tau[i]}, theta: {theta}, t_l: {t_l}, t_

        if y[i] == 1 and t_l < tau[i] < t_u:
            first_term = theta
        elif y[i] == 0 and t_l < tau[i] < t_u:
            second_term = second_term = 1 - theta
        else:
            third_term = 1/2

    return first_term * second_term * third_term

def target_prior_theta(theta, alpha, beta_param): #100%
    return beta.pdf(theta, alpha, beta_param)

def target_prior_delta_t(t_l, t_u): #100%
    p_tl = 1 if 0 <= t_l <= 24 * 3600 else 0
    p_tu_given_tl = 1 if t_l <= t_u <= 24 * 3600 else 0
    prior_delta_t = p_tu_given_tl * p_tl
    return prior_delta_t

def target_distribution(theta, t_l, t_u, D, alpha, beta_param):
    prior_theta = target_prior_theta(theta, alpha, beta_param)
    prior_delta_t = target_prior_delta_t(t_l, t_u)
    likelihood = target_likelihood(prior_theta, t_l,t_u, D)
    p_z = likelihood * prior_theta * prior_delta_t
    #print(f"Likelihood: {likelihood}, Prior theta: {prior_theta}, Prior
    return p_z

def run_simulation(theta_init, t_l_init, t_u_init, n_iterations=10000):
    theta_current = theta_init
    t_l_current = t_l_init
    t_u_current = t_u_init
    samples = []
    acceptance_count = 0

    for i in range(n_iterations):
        theta_proposed = proposal_for_theta(theta_current, epsilon)
        p_z = target_distribution(theta_current, t_l_current, t_u_current
        p_z_prim = target_distribution(theta_proposed, t_l_current, t_u_c
        q_z = proposal_for_theta(theta_current, epsilon)
```

```
        q_z_prim = proposal_for_theta(theta_proposed, epsilon)
        acceptance_prob = min(1, p_z_prim * q_z / (p_z * q_z_prim))

        if uniform.rvs() < acceptance_prob:
            theta_current = theta_proposed
            t_l_current = proposal_for_tl(t_l_current, sigma)
            t_u_current = t_l_current + proposal_for_tu(t_l_current, lamb
            acceptance_count += 1

        samples.append(theta_current)

    acceptance_rate = acceptance_count / n_iterations
    return samples, acceptance_rate
np.random.seed(0)
epsilon = 4
alpha, beta_param = 1, 1
lambda_param = 900
U = 3600
sigma = 100
theta_init = np.random.uniform(1 - epsilon, 1 + epsilon)
t_l_init = proposal_for_tl(np.random.randint(0, 3600*24), sigma)
t_u_init = t_l_init + proposal_for_tu(t_l_init, lambda_param, U)

samples, acceptance_rate = run_simulation(theta_init, t_l_init, t_u_init)

plt.figure(figsize=(10, 6))
plt.plot(samples, label='Theta')
plt.xlabel('Samples')
plt.ylabel('Theta')
plt.title('Trace plot of Theta')
plt.legend()
plt.show()

print(f'Acceptance Rate: {acceptance_rate}')
```

```
/var/folders/_j/hc6yz5ys04z66vnnmxnn2x780000gn/T/ipykernel_21239/28107261
8.py:67: RuntimeWarning: invalid value encountered in scalar divide
  acceptance_prob = min(1, p_z_prim * q_z / (p_z * q_z_prim))
/var/folders/_j/hc6yz5ys04z66vnnmxnn2x780000gn/T/ipykernel_21239/28107261
8.py:67: RuntimeWarning: divide by zero encountered in scalar divide
  acceptance_prob = min(1, p_z_prim * q_z / (p_z * q_z_prim))
```

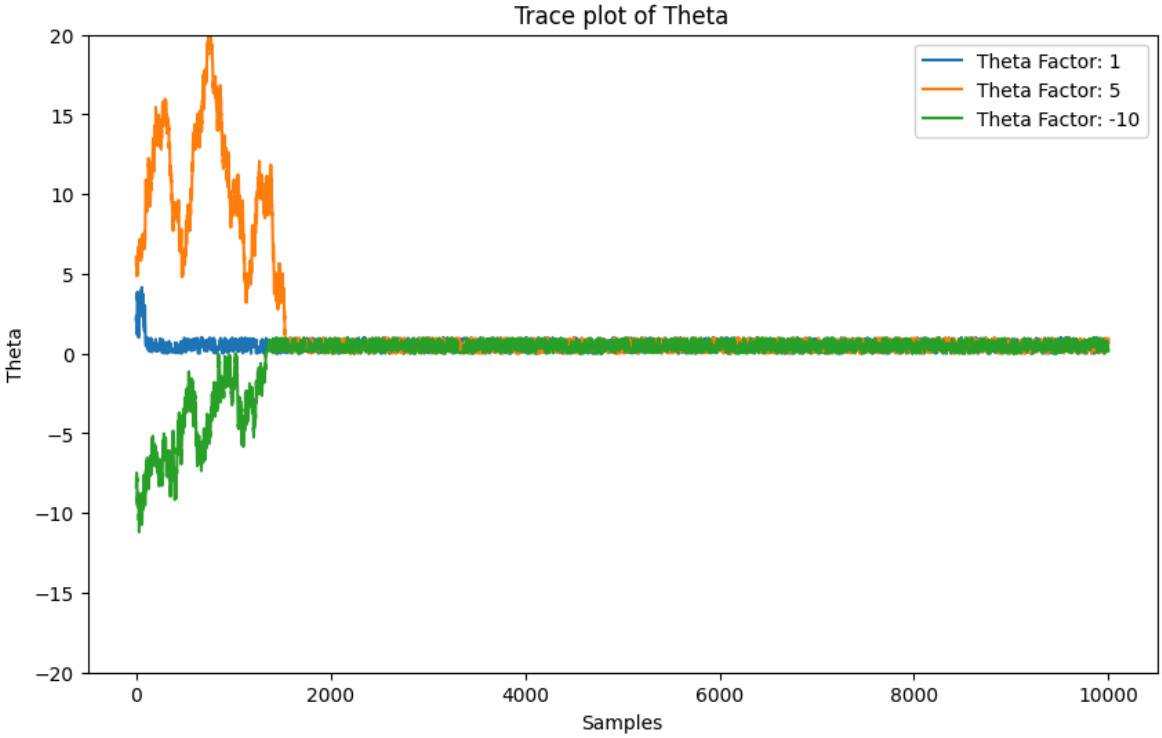



```
Acceptance Rate: 0.0556
```

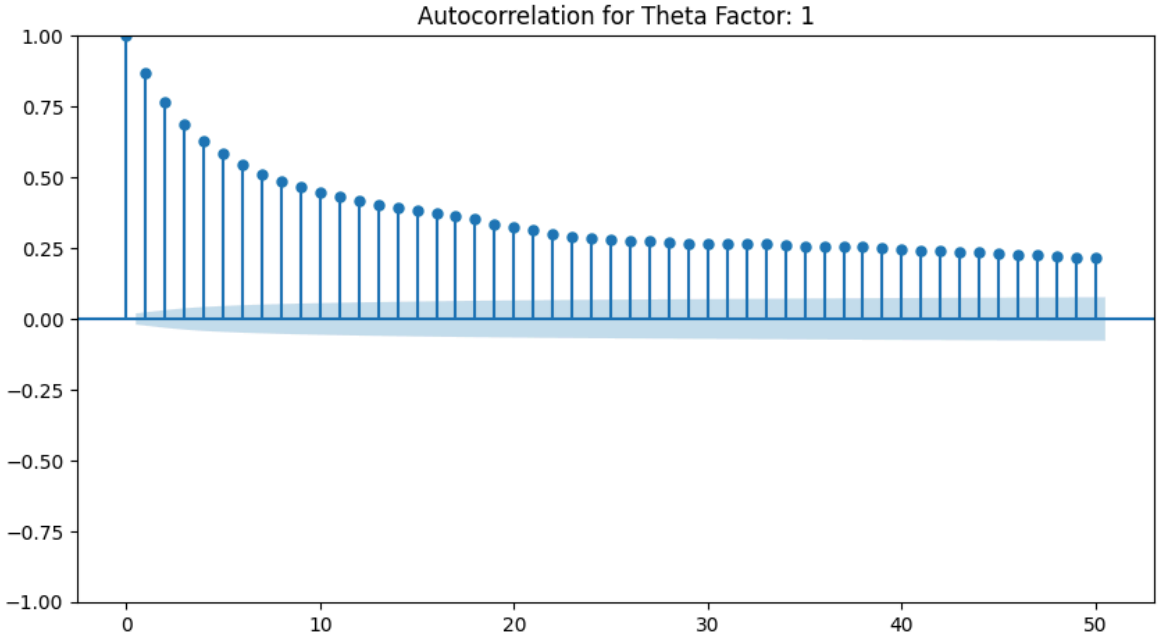
In [9]:
```
np.random.seed(0)
theta_factors = [1, 5, -10]
epsilon = 0.6
alpha, beta_param = 1, 1
U = 3600
sigma = 1

samples = np.zeros((len(theta_factors), 10000))
acceptance_rates = np.zeros(len(theta_factors))

for idx, factor in enumerate(theta_factors):
    theta_init = np.random.uniform(1 - epsilon, 1 + epsilon) + factor
    t_l_init = proposal_for_tl(np.random.randint(0, 3600*24), sigma)
    t_u_init = t_l_init + proposal_for_tu(t_l_init, lambda_param, U)
    samples[idx], acceptance_rates[idx] = run_simulation(theta_init, t_l_

plt.figure(figsize=(10, 6))
plt.ylim(-20, 20)
for idx, factor in enumerate(theta_factors):
    plt.plot(samples[idx], label=f'Theta Factor: {factor}')
plt.xlabel('Samples')
plt.ylabel('Theta')
plt.title('Trace plot of Theta')
plt.legend()
plt.show()

print(f'Acceptance Rates: {acceptance_rates}')
import statsmodels.api as sm

def plot_autocorrelation(samples, lags=50):
    fig, axes = plt.subplots(len(samples), 1, figsize=(10, 6 * len(sample
    if len(samples) == 1:
        axes = [axes]
    for idx, sample in enumerate(samples):
        sm.graphics.tsa.plot_acf(sample, lags=lags, ax=axes[idx])
```

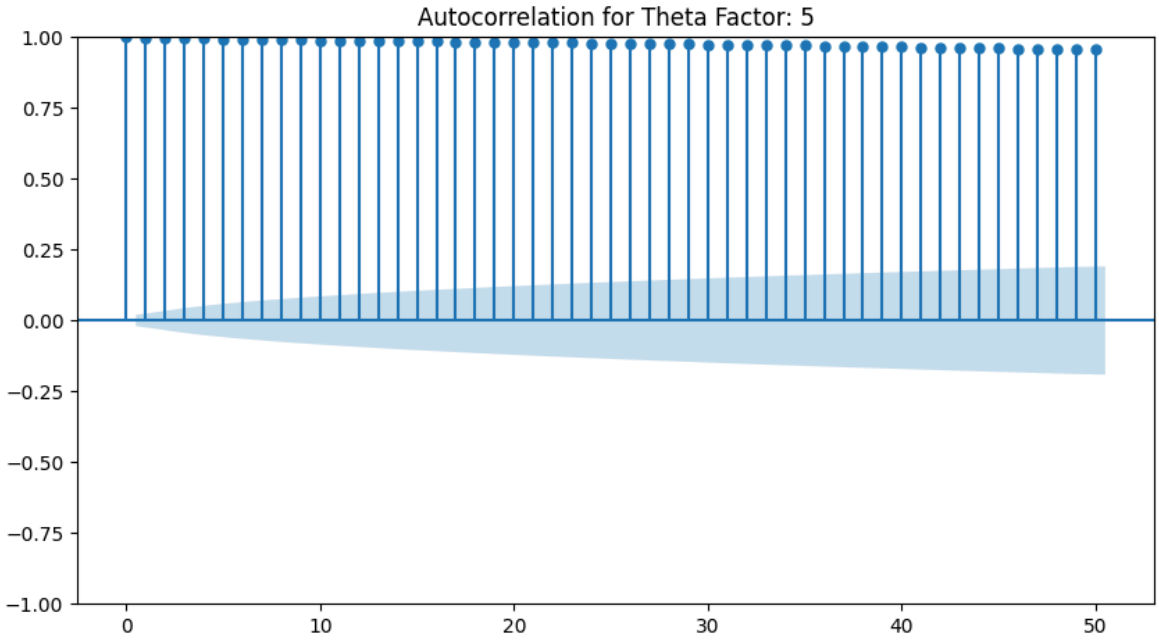
```
        axes[idx].set_title(f'Autocorrelation for Theta Factor: {theta_fa
    plt.show()

plot_autocorrelation(samples)


# We see that there is some autocorrelation for the samples with theta fa
```

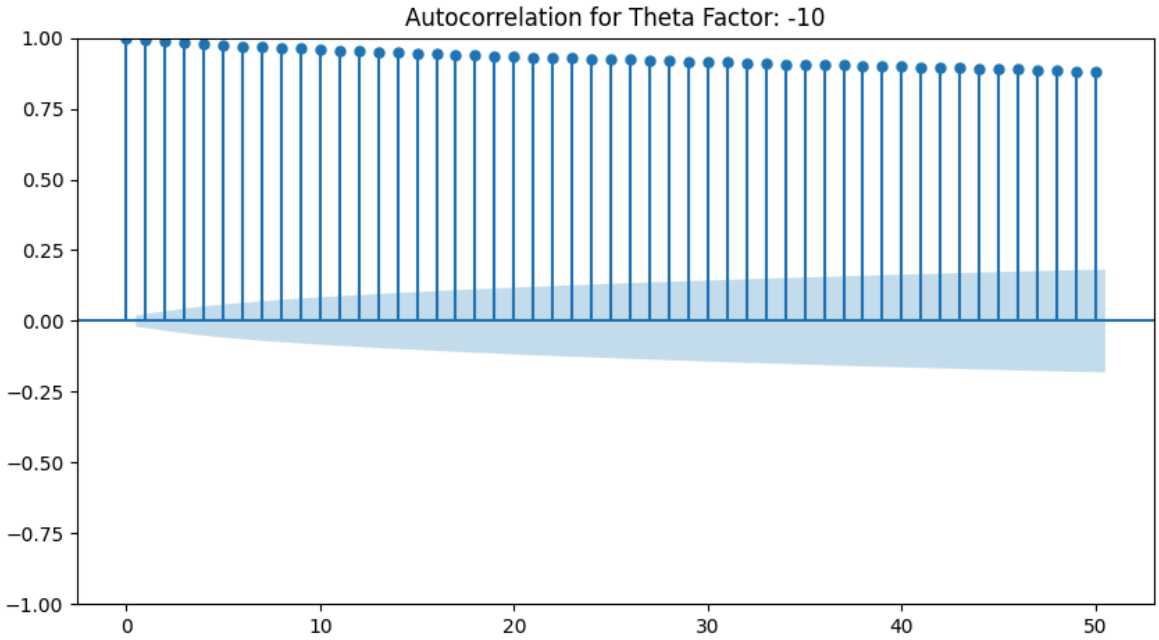
```
/var/folders/_j/hc6yz5ys04z66vnnmxnn2x780000gn/T/ipykernel_21239/28107261
8.py:67: RuntimeWarning: invalid value encountered in scalar divide
  acceptance_prob = min(1, p_z_prim * q_z / (p_z * q_z_prim))
/var/folders/_j/hc6yz5ys04z66vnnmxnn2x780000gn/T/ipykernel_21239/28107261
8.py:67: RuntimeWarning: divide by zero encountered in scalar divide
  acceptance_prob = min(1, p_z_prim * q_z / (p_z * q_z_prim))
```




```
Acceptance Rates: [0.3878 0.4844 0.468 ]
```


Autocorrelation for Theta Factor: 1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
10
20
30
40
50


Autocorrelation for Theta Factor: 5
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
10
20
30
40
50


Autocorrelation for Theta Factor: -10
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
10
20
30
40
50

Analysis

After performing our convergence diagnostics and tuning our MH algorithm based on the convergence analysis, we are now content with our MH samples. It is time to evaluate the Bayes risk.

Let

$$C(x=1,h) = \begin{cases} 500 & : h = 0 \\ -10000 & : h = 1 \end{cases}$$

and

$$C(x=0,h) = -1000, \text{for } h = 0 \text{ and } h = 1.$$

Q3.3.1 Explain this cost function in words.

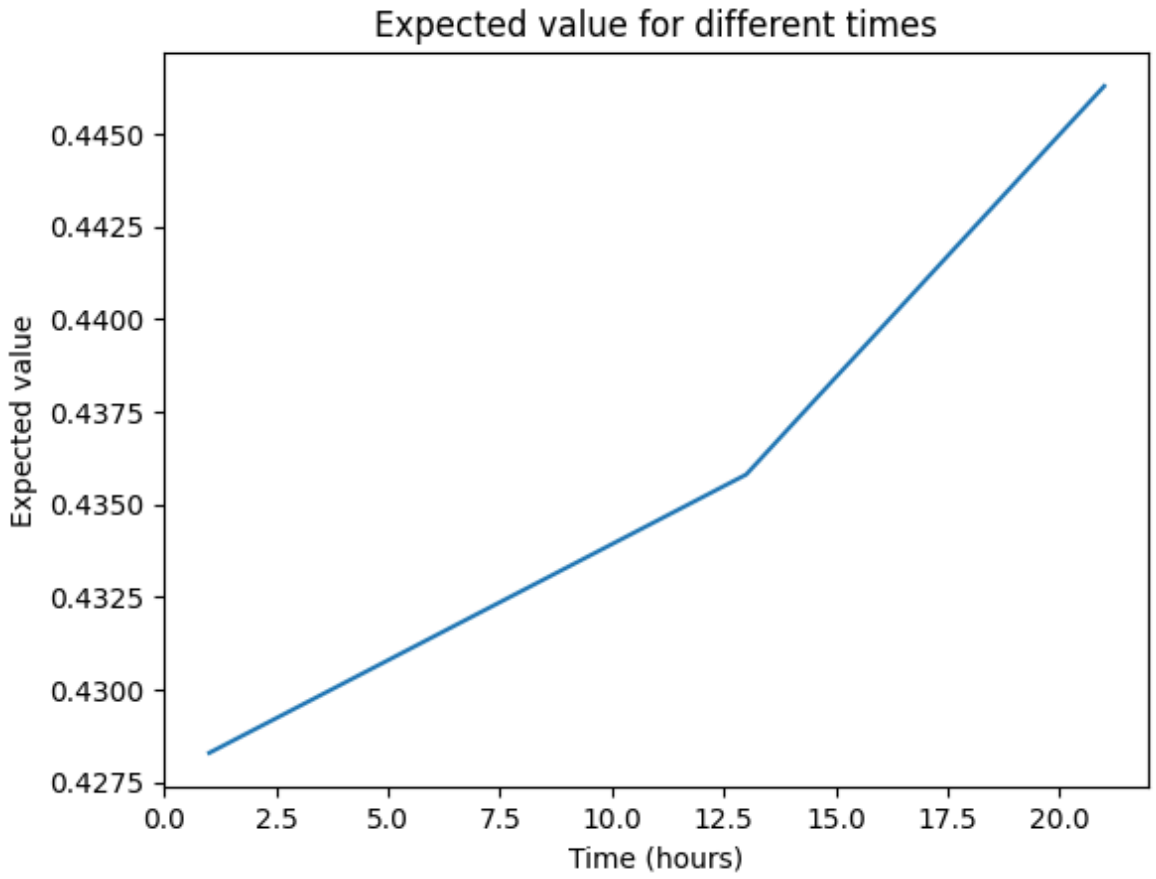
In the case we break in (x = 1) and the state h is that the house is empty we get 500, if we get caught (if the state h is that the owners are home we will get a penalty of 10000). If we don't break in (x=0) we get a penalty of 1000 no matter if the state h is that the owners are home or not home

Q3.3.2 Compute the risk of breaking in or not breaking in between 01.00-02.00, 13.00-14.00, and between 21.00-22.00. According to the results, when should we break in? Explain the results. According to the results we should break in in the morning but it is not a significant difference in risk between the different scenarios.

In [10]:
```
hour = 3600

first_attempt = hour*1
second_attempt = hour*13
third_attempt = hour*21

attempts = [first_attempt, second_attempt, third_attempt]
x_values = [0,1]

np.random.seed(40)

def risk_function(attempt):
    epsilon = 0.6
    alpha, beta_param = 1, 1
    U = 3600
    sigma = 1
    theta_init = np.random.uniform(1 - epsilon, 1 + epsilon)
    t_l_init = proposal_for_tl(attempt, sigma)
    t_u_init = t_l_init + proposal_for_tu(t_l_init, lambda_param, U)
    samples, acceptance_rate = run_simulation(theta_init, t_l_init, t_u_i
    expected_value = np.mean(samples)
    return expected_value

risks = {}
```

```
expected_values = []

for attempt in attempts:
    C = 0
    expected_value = risk_function(attempt)
    print(f"Expected value: {expected_value}")
    expected_values.append(expected_value)
    for x in x_values:
        if x == 0:
            C -= 1000
        else:
            reward = 500 - 10000
            C += reward * expected_value

    risks[f'{attempt}'] = C


plt.plot(np.array(attempts) / 3600, expected_values)
plt.xlabel('Time (hours)')
plt.ylabel('Expected value')
plt.title('Expected value for different times')
plt.show()

print(risks)
```

```
Expected value: 0.42829523790425594
/var/folders/_j/hc6yz5ys04z66vnnmxnn2x780000gn/T/ipykernel_21239/28107261
8.py:67: RuntimeWarning: invalid value encountered in scalar divide
  acceptance_prob = min(1, p_z_prim * q_z / (p_z * q_z_prim))
/var/folders/_j/hc6yz5ys04z66vnnmxnn2x780000gn/T/ipykernel_21239/28107261
8.py:67: RuntimeWarning: divide by zero encountered in scalar divide
  acceptance_prob = min(1, p_z_prim * q_z / (p_z * q_z_prim))
Expected value: 0.4358048538531697
Expected value: 0.44628212886292884
```




```
{'3600': -5068.804760090432, '46800': -5140.146111605112, '75600': -5239.6
80224197824}
```

Modifying the model

A problem with the above model is that it does not let us incorporate our prior believes of ***when*** it is more probable that the home owners are home. It is reasonable to believe that the house is empty more often during day time than night time, for instance.

Q3.4 (Worth 5 points)

Here, modify the prior on θ such that its hyperparameters depend on Δt, resulting in higher a priori probability that the house is empty between 08.00 and 17.00. Hint: use indicator functions. Clearly formulate the new generative model.

After the modification of the model, rerun the experiments above and re-evaluate the risks. ***Do not delete your previous results in the cells above! Neatly organize new code and text cells below.***

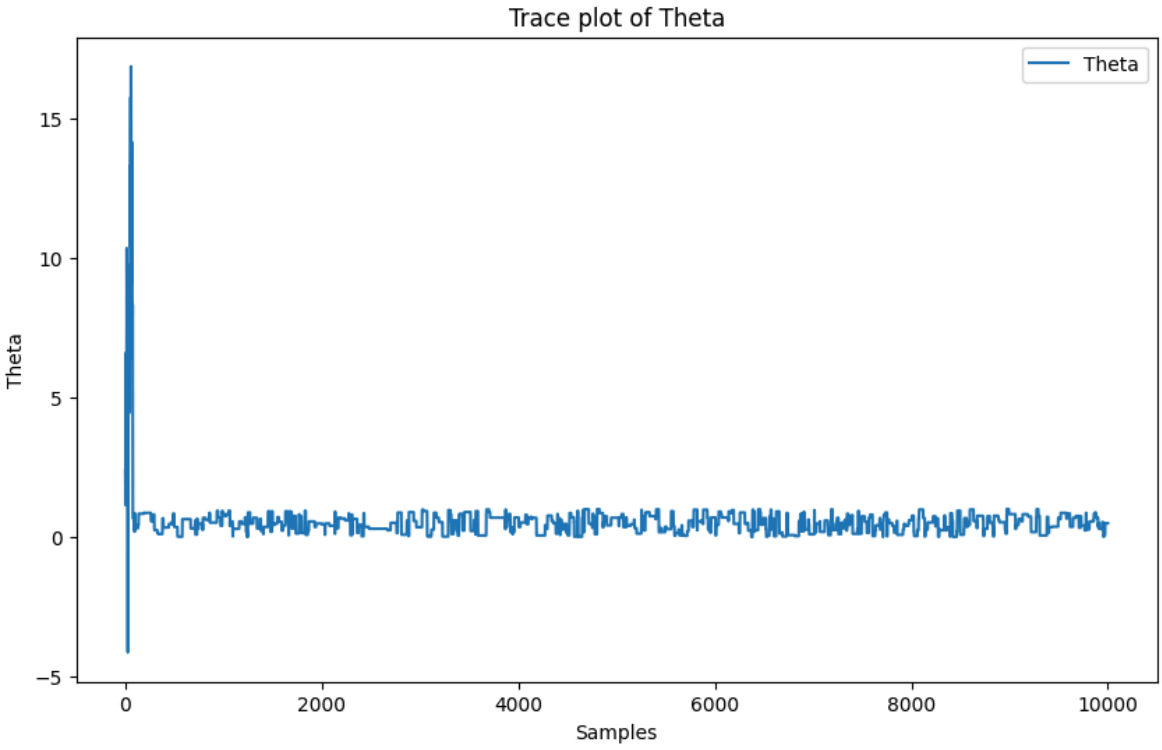
What are your reflections on the results after the modification?

In [11]:
```
from scipy.stats import beta, norm
import numpy as np

def proposal_for_theta(theta_k_minus_1, epsilon): #90%
```

```
    lower_limit = theta_k_minus_1 - epsilon
    upper_limit = theta_k_minus_1 + epsilon
    proposal = np.random.uniform(lower_limit, upper_limit)
    return proposal

def proposal_for_tl(t_l_k_minus_1, sigma):
    t_l = np.random.normal(t_l_k_minus_1, sigma)
    return t_l

def proposal_for_tu(y, lambda_param, U):
    y = np.random.uniform(0, 1)
    t_u = U - np.log(1 - y) / lambda_param
    return t_u

def target_likelihood(theta, t_l, t_u, D):
    y, tau = D

    for i in range(len(y)):
        first_term = 1
        second_term = 1

        #print(f"y: {y[i]}, tau: {tau[i]}, theta: {theta}, t_l: {t_l}, t_

        if y[i] == 1 and t_l < tau[i] < t_u:
            first_term = theta
        elif y[i] == 0 and t_l < tau[i] < t_u:
            second_term = second_term = 1 - theta
        else:
            third_term = 1/2

    return first_term * second_term * third_term


def target_prior_theta(theta, alpha, beta_param, t_l): #100%
    empty_lower_bound = 8 * 3600
    empty_upper_bound = 17 * 3600

    if empty_lower_bound < t_l < empty_upper_bound:
        alpha = 1
        beta_param = 1
    else:
        alpha = 10
        beta_param = 1

    return beta.pdf(theta, alpha, beta_param)

def target_prior_delta_t(t_l, t_u): #100%
    p_tl = 1 if 0 <= t_l <= 24 * 3600 else 0
    p_tu_given_tl = 1 if t_l <= t_u <= 24 * 3600 else 0
    prior_delta_t = p_tu_given_tl * p_tl
    return prior_delta_t

def target_distribution(theta, t_l, t_u, D, alpha, beta_param):
    prior_theta = target_prior_theta(theta, alpha, beta_param, t_l)
    prior_delta_t = target_prior_delta_t(t_l, t_u)
    likelihood = target_likelihood(prior_theta, t_l,t_u, D)
    p_z = likelihood * prior_theta * prior_delta_t
    #print(f"Likelihood: {likelihood}, Prior theta: {prior_theta}, Prior
    return p_z
```

```
def run_simulation(theta_init, t_l_init, t_u_init, n_iterations=10000):
    theta_current = theta_init
    t_l_current = t_l_init
    t_u_current = t_u_init
    samples = []
    acceptance_count = 0

    for i in range(n_iterations):
        theta_proposed = proposal_for_theta(theta_current, epsilon)
        p_z = target_distribution(theta_current, t_l_current, t_u_current
        p_z_prim = target_distribution(theta_proposed, t_l_current, t_u_c
        q_z = proposal_for_theta(theta_current, epsilon)
        q_z_prim = proposal_for_theta(theta_proposed, epsilon)
        acceptance_prob = min(1, p_z_prim * q_z / (p_z * q_z_prim))

        if uniform.rvs() < acceptance_prob:
            theta_current = theta_proposed
            t_l_current = proposal_for_tl(t_l_current, sigma)
            t_u_current = t_l_current + proposal_for_tu(t_l_current, lamb
            acceptance_count += 1

        samples.append(theta_current)

    acceptance_rate = acceptance_count / n_iterations
    return samples, acceptance_rate
np.random.seed(0)
epsilon = 4
alpha, beta_param = 1, 1
lambda_param = 900
U = 3600
sigma = 100
theta_init = np.random.uniform(1 - epsilon, 1 + epsilon)
t_l_init = proposal_for_tl(np.random.randint(0, 3600*24), sigma)
t_u_init = t_l_init + proposal_for_tu(t_l_init, lambda_param, U)

samples, acceptance_rate = run_simulation(theta_init, t_l_init, t_u_init)

plt.figure(figsize=(10, 6))
plt.plot(samples, label='Theta')
plt.xlabel('Samples')
plt.ylabel('Theta')
plt.title('Trace plot of Theta')
plt.legend()
plt.show()

print(f'Acceptance Rate: {acceptance_rate}')
```

```
/var/folders/_j/hc6yz5ys04z66vnnmxnn2x780000gn/T/ipykernel_21239/354942307
5.py:78: RuntimeWarning: invalid value encountered in scalar divide
  acceptance_prob = min(1, p_z_prim * q_z / (p_z * q_z_prim))
/var/folders/_j/hc6yz5ys04z66vnnmxnn2x780000gn/T/ipykernel_21239/354942307
5.py:78: RuntimeWarning: divide by zero encountered in scalar divide
  acceptance_prob = min(1, p_z_prim * q_z / (p_z * q_z_prim))
```

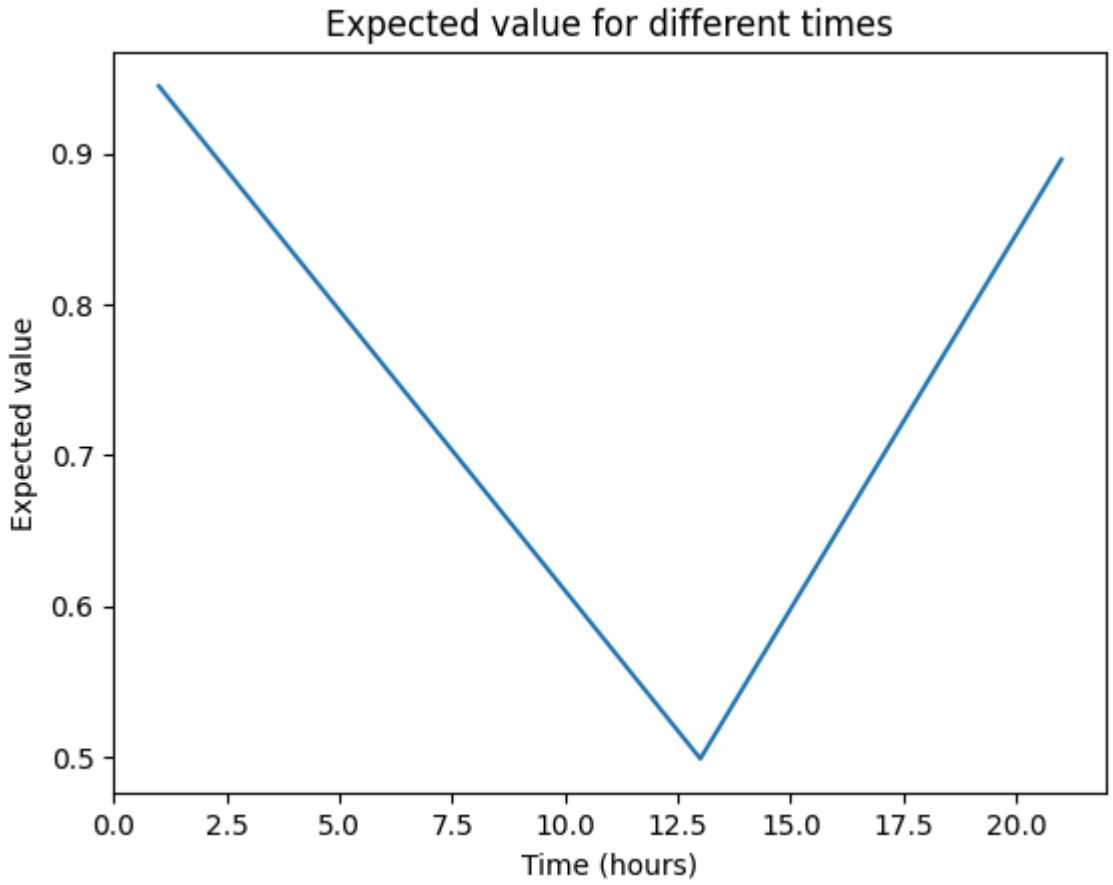


Acceptance Rate: 0.0556

In [12]:
```
hour = 3600

first_attempt = hour*1
second_attempt = hour*13
third_attempt = hour*21

attempts = [first_attempt, second_attempt, third_attempt]
x_values = [0,1]


def risk_function(attempt):
    epsilon = 0.6
    alpha, beta_param = 1, 1
    U = 3600
    sigma = 1
    theta_init = np.random.uniform(1 - epsilon, 1 + epsilon)
    t_l_init = proposal_for_tl(attempt, sigma)
    t_u_init = t_l_init + proposal_for_tu(t_l_init, lambda_param, U)
    samples, acceptance_rate = run_simulation(theta_init, t_l_init, t_u_i
    expected_value = np.mean(samples)
    return expected_value

risks = {}

expected_values = []

for attempt in attempts:
    C = 0
    expected_value = risk_function(attempt)
    expected_values.append(expected_value)

    print(f"Expected value for {attempt}: {expected_value}")
    for x in x_values:
        if x == 0:
            C -= 1000
```

```
        else:
            reward = 500 - 10000
            C += reward * expected_value

    risks[f'{attempt}'] = C


plt.plot(np.array(attempts) / 3600, expected_values)
plt.xlabel('Time (hours)')
plt.ylabel('Expected value')
plt.title('Expected value for different times')
plt.show()

print(risks)
```

```
/var/folders/_j/hc6yz5ys04z66vnnmxnn2x780000gn/T/ipykernel_21239/3549423075.py:78: RuntimeWarning: invalid value encountered in scalar divide
  acceptance_prob = min(1, p_z_prim * q_z / (p_z * q_z_prim))
/var/folders/_j/hc6yz5ys04z66vnnmxnn2x780000gn/T/ipykernel_21239/3549423075.py:78: RuntimeWarning: divide by zero encountered in scalar divide
  acceptance_prob = min(1, p_z_prim * q_z / (p_z * q_z_prim))
Expected value for 3600: 0.9448409887147995
Expected value for 46800: 0.4985479538185082
Expected value for 75600: 0.8961372104118877
```




```
{'3600': -9975.989392790596, '46800': -5736.205561275828, '75600': -9513.303498912932}
```

We shifted the beliefs so that if t_l was between $8 \cdot 3600$ and $17 \cdot 3600$ we keep alpha and beta at 1 while if it is not during the day we set alpha to 10 and beta to 1 to

shift the distrubution towards 1. The results were as expected, breaking in during the morning or in the evening is associated with a lot higher bayesian risk