LQR Forward pass

Notation

We use the following quadratic expansion of the costs, value-function and Q-function:

$$c(\mathbf{x},\mathbf{u}) = \frac{1}{2}\begin{pmatrix}\mathbf{x}\\ \mathbf{u}\end{pmatrix}^T \mathbf{C} \begin{pmatrix}\mathbf{x}\\ \mathbf{u}\end{pmatrix} + \begin{pmatrix}\mathbf{x}\\ \mathbf{u}\end{pmatrix}^T \mathbf{c} + const.$$

$$V(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{V}\mathbf{x} + \mathbf{x}^T\mathbf{v} + const.$$

$$Q(\mathbf{x},\mathbf{u}) = \frac{1}{2}\begin{pmatrix}\mathbf{x}\\ \mathbf{u}\end{pmatrix}^T \mathbf{Q} \begin{pmatrix}\mathbf{x}\\ \mathbf{u}\end{pmatrix} + \begin{pmatrix}\mathbf{x}\\ \mathbf{u}\end{pmatrix}^T \mathbf{q} + const.$$

where

$$\mathbf{Q} = \begin{pmatrix}\mathbf{Q}_{\mathbf{x},\mathbf{x}} & \mathbf{Q}_{\mathbf{x},\mathbf{u}}\\ \mathbf{Q}_{\mathbf{u},\mathbf{x}} & \mathbf{Q}_{\mathbf{u},\mathbf{u}}\end{pmatrix}$$

$$\mathbf{q} = \begin{pmatrix}\mathbf{q}_{\mathbf{x}}\\ \mathbf{q}_{\mathbf{u}}\end{pmatrix}$$

The `m` and `v` in `Vm` and `vv` etc. stand for 'matrix' and 'vector'.

Code

The backward recursion function takes as arguments

- `prev_traj_distr` : The policy object of the previous iteration, used get the dimensions of the new policy object
- `traj_info` : This object contains the dynamics
- `eta` : Lagrange dual variable; needed to compute the extended cost function

```
def backward(self, prev_traj_distr, traj_info, eta):
```

We get the number of timesteps and the dimensions of the states and the actions from the previous policy object:

```
T = prev_traj_distr.T
dimU = prev_traj_distr.dU
dimX = prev_traj_distr.dX
```

We get the quadratic expansion of the cost function and the dynamics:

```
Cm_ext, cv_ext = self.compute_extended_costs(eta, traj_info, prev_traj_distr

Fm = traj_info.dynamics.Fm
fv = traj_info.dynamics.fv
```

We use slice syntax so that `Qm[index_x, index_u]` means $\mathbf{Q}_{\mathbf{x},\mathbf{u}}$ etc.

```
index_x = slice(dimX)
index_u = slice(dimX, dimX + dimU)
```

We allocate space for the Value-function and initialize the new policy object:

```
Vm = np.zeros((T, dimX, dimX))
vv = np.zeros((T, dimX))

traj_distr = prev_traj_distr.nans_like()
```

We iterate over t, starting at $T-1$ (because the last index of an array with T entries is $T-1$) and going backward:

```
for t in range(T - 1, -1, -1):
```

At each timestep, we compute the quadratic and the linear coefficients of the Q-Function:

$$\mathbf{Q}_t = \mathbf{C}_t + \mathbf{F}_t^T \mathbf{V}_{t+1} \mathbf{F}_t$$
$$\mathbf{q}_t = \mathbf{c}_t + \mathbf{F}_t^T (\mathbf{V}_{t+1} \mathbf{f}_t + \mathbf{v}_{t+1})$$

At $t = T$ we have no Value Function available, so $\mathbf{Q}_t = \mathbf{C}_t$ and $\mathbf{q}_t = \mathbf{c}_t$.

```
Qm = Cm_ext[t, :, :]
qv = cv_ext[t, :]

if t < T - 1:
```

```
    Qm += Fm[t, :, :].T.dot(Vm[t + 1, :, :]).dot(Fm[t, :, :])
    qv += Fm[t, :, :].T.dot(Vm[t + 1, :, :].dot(fv[t, :]) + vv[t + 1, :])

Qm = 0.5 * (Qm + Qm.T)
```

$\mathbf{Q}_t$ is a symmetric matrix, but numerical errors lead to `Qm` being not quite symmetric. To counter these numerical errors we symmetrize `Qm` .

Instead of directly computing $\mathbf{Q}_{t,\mathbf{u},\mathbf{u}}^{-1}$ we use Cholesky decomposition:

```
U = sp.linalg.cholesky(Qm[index_u, index_u])
L = U.T
```

We calculate $\mathbf{K}_t = -\mathbf{Q}_{t,\mathbf{u},\mathbf{u}}^{-1}\mathbf{Q}_{t,\mathbf{u},\mathbf{x}}$ and $\mathbf{k}_t = -\mathbf{Q}_{t,\mathbf{u},\mathbf{u}}^{-1}\mathbf{q}_{t,\mathbf{u}}$ and store them in the new `traj_distr` object:

```
traj_distr.K[t, :, :] = - sp.linalg.solve_triangular(
    U, sp.linalg.solve_triangular(L, Qm[index_u, index_x], lower = True)
)
traj_distr.k[t, :] = - sp.linalg.solve_triangular(
    U, sp.linalg.solve_triangular(L, qv[index_u], lower=True)
)
```

We store the covariance $\mathbf{\Sigma} = \mathbf{Q}_{t,\mathbf{u},\mathbf{u}}^{-1}$ and also its Cholesky decomposition and its inverse $\mathbf{\Sigma}^{-1} = \mathbf{Q}_{t,\mathbf{u},\mathbf{u}}$ in the `traj_distr` object:

```
traj_distr.pol_covar[t, :, :] = sp.linalg.solve_triangular(
    U, sp.linalg.solve_triangular(L, np.eye(dimU), lower=True)
)
traj_distr.chol_pol_covar[t, :, :] = sp.linalg.cholesky(
    traj_distr.pol_covar[t, :, :]
)
traj_distr.inv_pol_covar[t, :, :] = Qm[index_u, index_u]
```

We calculate the quadratic and the linear coefficients of the Value-function, which are used in the next iteration:

$$\mathbf{V}_t = \mathbf{Q}_{t,\mathbf{x},\mathbf{x}} + \mathbf{Q}_{t,\mathbf{x},\mathbf{u}}\mathbf{K}_t$$
$$\mathbf{v}_t = \mathbf{q}_{t,\mathbf{x}} + \mathbf{Q}_{t,\mathbf{x},\mathbf{u}}\mathbf{k}_t$$

```
Vm[t, :, :] = Qm[index_x, index_x] + \
```

```
            Qm[index_x, index_u].dot(traj_distr.K[t, :, :])
Vm[t, :, :] = 0.5 * (Vm[t, :, :] + Vm[t, :, :].T)
vv[t, :] = qv[index_x] + Qm[index_x, index_u].dot(traj_distr.k[t, :])
```

After that, the loop ends.

Finally, we return the new `traj_distr` object:

```
return traj_distr
```