HW 30

In [1]:
```
# import jax
# import jax.numpy as jnp
import numpy as np
import pandas as pd
# import PIL
# import scipy
import sympy as sp
from matplotlib import pyplot as plt
from scipy.optimize import minimize
```

When solving learning problems, there can be many types of things in one's "bucket of models." We've discussed various functions, such as lines or other polynomials, ellipses, etc. Another set of objects one may consider are dynamic systems. Dynamic systems are useful for characterizing many phenomena.

Consider the following dynamic system characterized by a simple *difference equation*:

$$x_{k+1} = ax_k + bu_k \tag{1}$$

where $k = 0, 1, 2, \ldots$ represents instances in time, $x_k \in \mathbb{R}$ is a real-valued sequence indexed by k, and $u_k \in \mathbb{R}$ is another real-valued sequence indexed by k that represents an input to the system. The parameters, $a, b \in \mathbb{R}$, are real-valued and characterize the system. That is to say, each element of a "bucket of models" of these types of models is characterized by its parameters a and b.

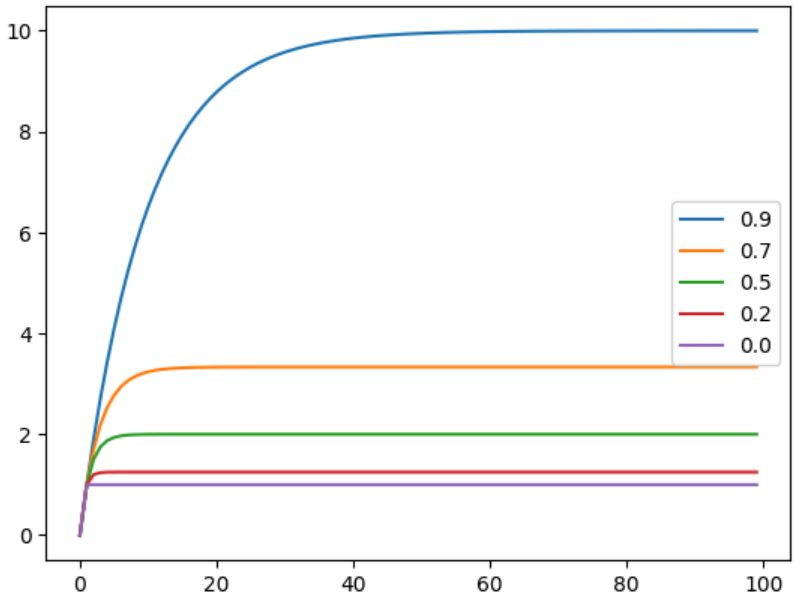
In [2]:
```
def f(a, xk, b, uk):
    return a*xk+b*uk
```

a. Let $x_0 = 0$, $u_0 = 0$, $b = 1$, and $u_k = 1$ for $k = 1, 2, \ldots$. On the same plot, plot x_k for the five different systems given by $a = .9$, $a = .7$, $a = .5$, $a = .2$, and $a = 0$.

In [3]:
```
x0 = 0
# u0 = 0
# uk = 1
b = 1
k_multi = np.linspace(0, 99, 100)
a_multi = np.array([0.9, 0.7, 0.5, 0.2, 0])

for a in a_multi:
    x_multi = [x0]
    for k in k_multi:
        xk = x_multi[-1]
        x_multi.append(f(a, xk, b, 1))
    x_multi = np.array(x_multi[:-1])
    plt.plot(k_multi, x_multi, label=str(a))
plt.legend()
```

Out[3]:
```
<matplotlib.legend.Legend at 0x1185e0c20>
```



b. Let $x_0 = 0$. Define u^N to be the vector in $\mathbb{R}^{N+1}$ given by $[\ u_0 \quad u_1 \quad \dots \quad u_N\ ]'$ and x^N to be the vector in $\mathbb{R}^N$ given by $[\ x_1 \quad \dots \quad x_N\ ]'$ (notice the first term for x^N is not x_0). Write a matrix, H, in terms of a and b in (1) that characterize the mapping from u^N to x^N, so that $x^N = Hu^{N-1}$.

In [4]:
```
vec_len = 4
x0 = 0
symbol_string = " ".join([f'u{str(n)}' for n in range(vec_len+1)])
u_n = sp.symbols(symbol_string) # make it the same length as x_n
symbol_string = " ".join([f'x{str(n)}' for n in range(1, vec_len+1)])
x_n = sp.symbols(symbol_string)
a, b = sp.symbols('a b')

eqs = []
H = []

for i in range(vec_len+1):
    # update eqs
    xk = x0 if not i else x_n[i-1]
    uk = u_n[i]
    eq = sp.Eq(xk, f(a, (eqs[-1].rhs if i else xk), b, uk))
    eqs.append(eq)
    # update m
    if H:
        H = [[*line, 0] for line in H]
        H.append([a*el or b for el in H[-1]])
    else:
        H.append([b])

H = sp.Matrix(H)
H
```

Out[4]:
$$\begin{bmatrix} b & 0 & 0 & 0 & 0 \\ ab & b & 0 & 0 & 0 \\ a^2b & ab & b & 0 & 0 \\ a^3b & a^2b & ab & b & 0 \\ a^4b & a^3b & a^2b & ab & b \end{bmatrix}$$

In [5]:
```
# convert matricies to same dims
X = sp.Matrix(x_n)
U = sp.Matrix(u_n[:-1])
H = H[:-1, :-1] # type: ignore
# now we can write as X=HU
```

c. Define a vector in terms of the parameters a and b from (1) given by:

$$\Theta^N = \begin{bmatrix} b \\ ab \\ a^2b \\ \vdots \\ a^{N-1}b \end{bmatrix}$$

These are known as the first N *Markov parameters* of the system. Rewrite the expression $x^N = Hu^{N-1}$ in terms of a square matrix G so that $x^N = G\Theta^N$. Note that if data (u_k, x_k) were available for $k = 0, 1, 2, ..., N$, and $x_0 = 0$, then x^N and G would both be known and one could solve for the parameters in Θ provided G is invertible. Solve for Θ^4 if $u^3 = \begin{bmatrix} 5 & 2 & 0 & 1 \end{bmatrix}$ and $x^4 = \begin{bmatrix} 10 & 9 & 4.5 & 4.25 \end{bmatrix}$.

In [6]:
```
O_N = H[:, 0] # type: ignore
G = sp.Matrix.zeros(*H.shape) # type: ignore
for i in range(G.shape[0]):
    for j in range(G.shape[1]):
        if j>i:
            break
        G[i, j] = U[i-j]
eq_b = sp.Eq(X, H*U)
eq_c = sp.Eq(X, G*O_N)
assert eq_b == eq_c # these are the same equations
```

In [7]:
```
U_c = sp.Matrix([[5], [2], [0], [1]])
G_c = sp.Matrix.zeros(*H.shape) # type: ignore
for i in range(G.shape[0]):
    for j in range(G.shape[1]):
        if j>i:
            break
        G_c[i, j] = U_c[i-j]
X_c = sp.Matrix([[10], [9], [4.5], [4.25]])
solution = sp.solve(sp.Eq(X_c, G_c*O_N))
solution
```

Out[7]:
```
[{a: 0.500000000000000, b: 2.00000000000000}]
```

In [8]:
```
# O^N can be expressed based on a and b as:
np.array([
    [2],
    [2*0.5],
    [2*0.5**2],
    [2*0.5**3]
])
```

Out[8]:
```
array([[2.  ],
       [1.  ],
       [0.5 ],
       [0.25]])
```

d. Suppose we can not observe x_k directly, but instead obtain noisy measurements $y_k = x_k + e_k$, where $e_k \sim \mathcal{N}(0, \sigma)$. Let y^N be the vector in $\mathbb{R}^N$ given by $\begin{bmatrix} y_1 & \dots & y_N \end{bmatrix}'$. Formulate a least squares problem to estimate a and b from u^N and y^N. Hint: recall that $\log(a^k b) = k \log a + \log b$.

In [9]:
```
# an example Y
E = sp.Matrix(np.random.randn(vec_len).T)
Y = X + E
Y
```

Out[9]:
$$\begin{bmatrix} x_1 + 0.494534657178151 \\ x_2 - 0.931087453653116 \\ x_3 + 0.169175375830879 \\ x_4 + 1.04686109873668 \end{bmatrix}$$

In [10]:
```
def solve_d(U_d, Y_d):
    G_d = sp.Matrix.zeros(U.shape[0], U.shape[0]) # type: ignore
    for i in range(G_d.shape[0]):
        for j in range(G_d.shape[1]):
            if j>i:
                break
            G_d[i, j] = U_d[i-j]
    best_fit_theta_vector = G_d.pinv_solve(Y_d)
    log_best_fit_theta_vector = np.log(np.array(best_fit_theta_vector, dtype=np.float64))

    # because of log(a^k*b)=klog(a)+log(b) we can make a linear model as follows:
    log_a_vec = np.linspace(0, best_fit_theta_vector.shape[0]-1, best_fit_theta_vector.shape[0])
    log_b_vec = np.ones(best_fit_theta_vector.shape[0])
    logA = np.stack([log_a_vec, log_b_vec]).T
    logab, *_ = np.linalg.lstsq(logA, log_best_fit_theta_vector)
    loga, logb = logab
    return np.exp(loga), np.exp(logb)
```

e. Given $u^3 = \begin{bmatrix} 5 & 2 & 0 & 1 \end{bmatrix}$ and $y^4 = \begin{bmatrix} 10.5 & 9.5 & 4.51 & 4.3 \end{bmatrix}$, solve the least squares problem you formulate above to estimate a and b.

In [11]:
```
U_e = sp.Matrix([[5], [2], [0], [1]])
Y_e = sp.Matrix([[10.5], [9.5], [4.51], [4.3]])
a_e, b_e = solve_d(U_e, Y_e)
a_e, b_e
```

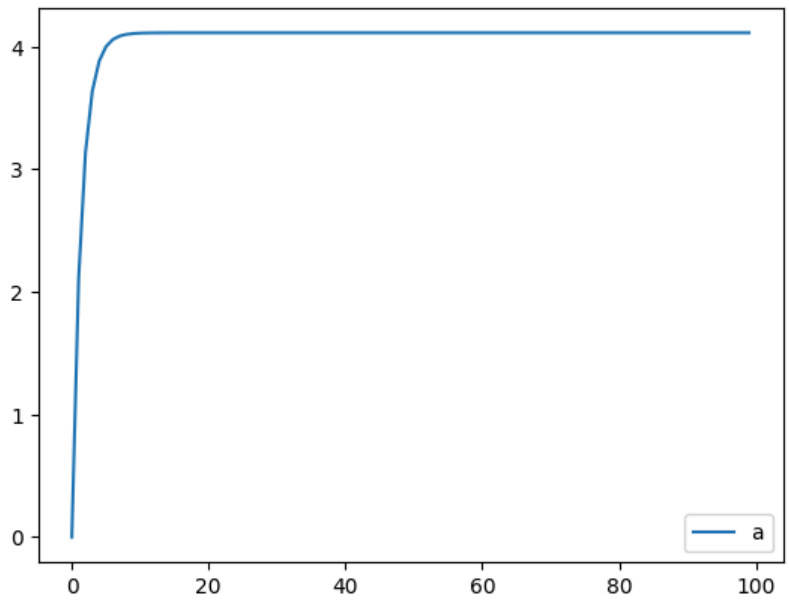
Out[11]:
```
(array([0.48696924]), array([2.1108072]))
```

In [12]:
```
# let's plot as in (a) with our new values
k_multi = np.linspace(0, 99, 100)

x_multi = np.zeros(101)
for i, k in enumerate(k_multi):
    xk = x_multi[i]
    x_multi[i+1] = f(a_e, xk, b_e, 1)
x_multi = np.array([x for x in x_multi[:-1]])
plt.plot(k_multi, x_multi, label=str(a))
plt.legend()
```

```
/var/folders/ss/nzbq79b506g04rrfrmmd37kh0000gn/T/ipykernel_71364/3034000250.py:7: DeprecationWarning: Conve
rsion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a s
ingle element from your array before performing this operation. (Deprecated NumPy 1.25.)
  x_multi[i+1] = f(a_e, xk, b_e, 1)
```

Out[12]:
```
<matplotlib.legend.Legend at 0x118a0bb10>
```



Acknowledgment

Work in this repository and with associated assignments and projects may be adapted or copied from similar files used in my prior academic and industry work (e.g., using a LaTeX file or Dockerfile as a starting point). Those files and any other work in this repository may have been developed with the help of LLM's like ChatGPT. For example, to provide context, answer questions, refine writing, understand function call syntax, and assist with repetitive tasks. In these cases, deliverables and associated work reflect my best efforts to optimize my learning and demonstrate my capacity, while using available resources and LLM's to facilitate the process.

ChatGPT Conversation