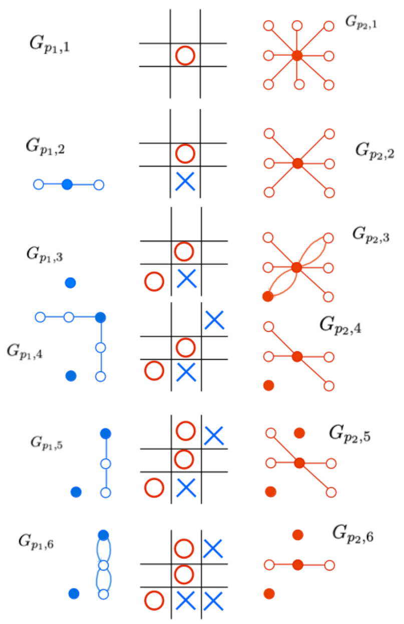
Stochastic AI in a N,N,N Game

Alonso Vega

August 3, 2020

A $N \times N \times N$ adversarial binary board game is a generalization of the classical $3 \times 3 \times 3$ game known as Tic-Tac-Toe. We implement this abstract game using a verity of data types and mathematical manipulation. Computational efficiency and generalization was paramount upon implementation. It is our hope that this software is used by those who have the desire to further its beauty and shared for further improvement.

We are currently working on developing a EVALUATION function for the α-$\beta Search()$ subroutine. From a given state of the board x_t, this function should robustly be able to estimate the expected utility, $\hat{u} : X \to [-1, 1], \boldsymbol{x}_t \mapsto \hat{u}(\boldsymbol{x}_t)$. An idea is to model the current state of the discrete board as two Cartesian graph, (G_{p_1}, G_{p_2}), and identify the features that may imply an optimal control. Below is an example in a game of Tic-Tac-Toe.



Consider the state $\boldsymbol{x} = [\boldsymbol{x}_1 \ \boldsymbol{x}_2 \ ... \ \boldsymbol{x}_N] = [\tilde{\boldsymbol{x}}_1 \ \tilde{\boldsymbol{x}}_2 \ ... \ \tilde{\boldsymbol{x}}_N]^T \in \{-1, 0, 1\}^{N \times N}$ can be decomposed as rows or columns vectors, where the tilde corresponds to row vectors. Thus, if there is only actions of one player in those vectors, then we form a winning path. Same goes for the diagonal and the anti-diagonal $(diag_+(\boldsymbol{x}), diag_-(\boldsymbol{x}))$. The parallel edges between nodes $\{(i_0, j_0), (i_1, j_1), ..., (i_N, j_N)\} \subset$

$V(G_{p_k,t})$ imply a reinforced possible winning path. Each node of each player's graph can be either filled ($filled((i_1, j_2)) = 1_2 \Leftrightarrow \nexists \mathbf{e}^{i_1}_{i_2} \in U_t$), denoted a past action input, or empty, implying a possible future action($\exists \mathbf{e}^{i_1}_{i_2} \in U_t$). These unfilled nodes can be destroyed by the other players moves at a current time.

We also wish to further generalize this N, N, N game into a M, N, K game. That is, to a board of size $M \times N$ where either player must obtain K input actions in physical neighboring sequence to win.

The associations between each class and their methods are shown below.

Below we illustrate a short summary for each interaction among our subroutines shown above.

Methods	**Inputs**	**Summary**
ee	i, N	Called upon by F_ss and $F2\_ss$ to pass $\boldsymbol{e}_i$.
ee_big	i, N, l	Creates and passes $\boldsymbol{E}_i$ using the *ee* function.
Jac_num	$\boldsymbol{\theta}, i, N, l$	Uses *ee_big* function to construct $\boldsymbol{J}_i(\boldsymbol{\theta})$.
$play\_1rob$	$-$	Running the for_kin_func function in a loop, this function captures multiple frames to construct robot configuration animation.
$play\_2rob$	$-$	Running the $for\_kin\_func\_2mod$ function in a loop, this function captures multiple frames to construct the 2 module robot configuration animation.
solve_robot	$-$	Solves equations of motion from F_ss by passing it the magnitudes of the SMA forces. Here we obtain the solution $\mathbf{x}(t)$, so we can pass $\boldsymbol{\theta}(t)$ and $\boldsymbol{p}(t)$ to for_kin_func.
F_ss	$t, \mathbf{x}, N, \boldsymbol{u}, l$	Formulates equations of motion using *ee*, *ee_big*, and Jac_num to be solved using the *solve_robot* subroutine.
for_kin_func	$\boldsymbol{\theta}, \boldsymbol{p}, l$	Draws out the robot's configuration from the given $\boldsymbol{\theta}$ and $\boldsymbol{p}$.
solve_robot_2mod	$-$	Solves equations of motion for the 2 module robot from $F2\_ss$ by passing it the magnitudes of the SMA forces. Here we obtain the solution $\mathbf{x}(t)$, so we can pass $\boldsymbol{\theta}(t)$ and $\boldsymbol{p}(t)$ to $for\_kin\_func\_2mod$.
$F2\_ss$	$t, \mathbf{x}, N, \boldsymbol{u}, l$	Formulates equations of motion using *ee*, *ee_big*, and Jac_num to be solved using the *solve_robot_2mod* subroutine.
$for\_kin\_func\_2mod$	$\boldsymbol{\theta}, \boldsymbol{p}, l$	Draws out the 2 module robot's configuration from the given $\boldsymbol{\theta}$ and $\boldsymbol{p}$.

1 Helping Programs

The helping programs were small constituents of the main programs in the other categories.

The smallest of the bunch is the $ee(i, N)$ function. This function returns $\boldsymbol{e}_i$, the i-th elementary ordered basis in the $\mathbb{R}^N$ vector space. We obviously need to send it the dimension of the vector space and indicate which order basis element we desire. This column vector is excessively useful to simplify the dynamic and kinematic equations to a more compact matrix form, thus it is mainly used in $F\_ss(t,x,N,u,l)$ and $F2\_ss(t,x,N,u,l)$; discussed briefly. The $ee(i,N)$ subroutine is principally used to implement the $ee\_big(i,N)$ function.

Next up is $ee\_big(i,N)$, which implements the following matrix:

$$\boldsymbol{E}_i = \begin{bmatrix} \boldsymbol{e}_i & \mathbf{0}_{N\times 1} \\ \mathbf{0}_{N\times 1} & \boldsymbol{e}_i \end{bmatrix} \in \mathbb{R}^{2N\times 2}$$

Just as the $ee(i,N)$ function, this subroutine is mainly used in $F\_ss(t,x,N,u,l)$ and $F2\_ss(t,x,N,u,l)$ to represent the equations of motion in a matrix form for simpler implementation.

In our endeavor we found ourselves using the Jacobian occasionally, so we decided to ostracize it. The function $Jac\_num(theta,i,N,l)$ constructs the Jacobian,

$$\boldsymbol{J}_i(\boldsymbol{\theta}) \ = l\boldsymbol{E}_i^T \begin{bmatrix} \boldsymbol{K}^T\boldsymbol{S_\theta} \\ -\boldsymbol{K}^T\boldsymbol{C_\theta} \end{bmatrix} \in \mathbb{R}^{2\times(N)}$$

Physically; the input $theta$ is simply the $\boldsymbol{\theta}(t)$ function, i represents the $\boldsymbol{p}_i$ position from where the Jacobian is constructed $(\dot{\boldsymbol{p}}_i = \boldsymbol{J}_i(\boldsymbol{\theta})\dot{\boldsymbol{\theta}})$, N is the number of links, and l is the geometric half-length of each link. The transpose of this function or its matrix output is primarily used to map the input forces at different locations to the generalized joint space; thus it was mainly used in the $F\_ss(t,x,N,u,l)$ and $F2\_ss(t,x,N,u,l)$ code.

Lastly, we made a sub-category for this helping function group; movie(). These functions take no inputs, they repeatedly call the forward kinematics function, discussed shortly, which graphically constructs the configuration of our system at a given time, and captures that graphical frame; overall to build the entire motion of our system over an interval of time. This function may only be manually initiated after we solve the equations of motion using $solve\_robot()$, considered shortly, due to the use of some local variables from said function.

2 1 Module Functions

The 1 module functions group consist of 3 members: $solve\_robot()$, $F\_ss(t,x,N,u,l)$, and $for\_kin\_func(theta,p,l)$.

The $solve\_robot()$ function is where we solve the system of equations using the $ode45(\cdot)$ solver. Here we initialize the initial conditions $\mathbf{x}_0$ and input actuating forces $\boldsymbol{u}(t)$ to pass them to the $ode45(\cdot)$ solver. Finally, this program also plots all the joint angles $(\boldsymbol{\theta}(t))$ and center of mass position $(\boldsymbol{p}(t))$ with respect to time within the interval specified by the local variable t_span.

The ODE solver requires the anonymous function that represents the system of equations in the form $\dot{\mathbf{x}} = F(\mathbf{x},\boldsymbol{u})$. This $F(\cdot)$ function is implemented as $F\_ss(t,x,N,u,l)$, which constructs the needed system of equations. The t input is not used, nevertheless needed, x represents the vector of states, N is the number of links needed to build the $N+2$ equations, u is the input vector of forces, and l is passed because of its local use in the $solve\_robot$ function. In this program we can set all physical parameters, like the damping coefficients and link mass.

As mentioned earlier the $for\_kin\_func(theta,p,l)$ creates graphically the arrangement of our system at time t, such that $theta = \boldsymbol{\theta}(t)$ and $p = \boldsymbol{p}(t)$. It also, for demonstration purposes, draws out the SMAs on our elastica.

3 2 Module Functions

The last group of functions is the 2 Module functions: $solve\_robot\_2mod()$, $F2\_ss(t, x, N, u, l)$, and $for\_kin\_func\_2mod(theta, p, l)$. They do the same as the 1 Module functions with the alterations that they work for the 2 module robot, utilizing 4 different SMAs at 4 different locations in our system.

The function $solve\_robot\_2mod()$'s main difference compared to its 1 module counterpart is the dimension of the input force vector $u(t)$, which is an input to $F2\_ss(t, x, N, u, l)$ in its place of $F\_ss(t, x, N, u, l)$; now it's a four-element vector instead of a two-element due to the extra two SMAs.

Next, the $for\_kin\_func_2mod(theta, p, l)$ function now has to graphically illustrate the two additional SMAs.

Finally, the $F2\_ss(t, x, N, u, l)$ subroutine is implemented with the modified equations of motion described in the Locomotion: 2 Module Robot section.