List of Equations in
Causal Cognitive Architecture 3: A Solution to the Binding Problem

Cognitive Systems Research, in press

Howard Schneider
Sheppard Clinic North

The variables and parameters below are discussed in the above referenced article. This list is simply intended as a supplementary resource while reading the paper. For reasons of brevity, dot notation is used in algorithmic portions of the equations—this is discussed in the paper. As well, the full algorithmic expansion of the dot notation is provided by the supplementary Python code, and included at the end of this equation list.

self.all_maps = np.empty((6, 6, 6, 30 * SECONDS_TO_RUN, 1000000, 50),dtype=object) (1)

self.all_maps = np.empty((6, 6, 6, 1000000, 50),dtype=object) (2)

$$\mathbf{S_1} \in R^{m\_1 x n\_1 x o\_1} \quad (3)$$

$$\mathbf{S_{1,t}} := \text{visual inputs(t)} \quad (4)$$

$$\mathbf{S_2} \in R^{m\_2 x n\_2 x o\_2} \quad (5)$$

$$\mathbf{S_{2,t}} := \text{auditory inputs(t)} \quad (6)$$

$$\mathbf{S_3} \in R^{m\_3 x n\_3 x o\_3} \quad (7)$$

$$\mathbf{S_{3,t}} := \text{olfactory inputs(t)} \quad (8)$$

$$\boldsymbol{\sigma} := \text{sensory system identification code} \in N \quad (9)$$

$$\boldsymbol{n\_\sigma} := \text{total number of sensory systems} \in N \quad (10)$$

$$\boldsymbol{s(t)} = [\ \mathbf{S_{1,t}}, \mathbf{S_{2,t}}, \mathbf{S_{3,t}}, \ldots, \mathbf{S_{n\_\sigma,t}}] \quad (11)$$

$$\boldsymbol{s'(t)} = \text{Input\_Sensory\_Vectors\_Shaping\_Modules}(\ \boldsymbol{s(t)}\ ) = [\mathbf{S'_{1,t}}, \mathbf{S'_{2,t}}, \mathbf{S'_{3,t}}, \ldots, \mathbf{S'_{n\_\sigma,t}}] \quad (12)$$

$$\mathbf{S'_{\sigma,t}} \in R^{mxnxo} \quad (13)$$

$$\mathbf{mapno} := \text{map identification code} \in N \quad (14)$$

$$\boldsymbol{\Theta} := \text{total number of local navigation maps in a sensory system } \boldsymbol{\sigma} \in N \quad (15)$$

$$\mathbf{LNM_{(\sigma,mapno)}} \in R^{mxnxo} \quad (16)$$

$$\boldsymbol{all\_maps_{\sigma,t}} = [\mathbf{LNM}_{(\sigma,1,t)}, \mathbf{LNM}_{(\sigma,2,t)}, \mathbf{LNM}_{(\sigma,3,t)}, \ldots, \mathbf{LNM}_{(\sigma,\Theta,t)}] \quad (17)$$

$$\Upsilon := \textbf{mapno} \text{ of best matching map in a given set of navigation maps} \in \textbf{mapno} \quad (18)$$

$$\mathbf{LNM}_{(\sigma,\Upsilon,t)} = \text{Input\_Sensory\_Vectors\_Associations\_Module}_{\sigma}.\text{match\_best\_local\_navigation\_map}(\mathbf{S'}_{\sigma,t}) \quad (19)$$

$$\mathbf{h} = \text{number of differences allowed to be copied onto existing map} \in R \quad (20a)$$

$$\textbf{new\_map} := \textbf{mapno} \text{ of new local navigation map added to current sensory system } \boldsymbol{\sigma} \in \textbf{mapno} \quad (20b)$$

$$|\text{ differences } (\mathbf{S'}_{\sigma,t}, \mathbf{LNM}_{(\sigma,\Upsilon,t)})| \leq \mathbf{h}, \Rightarrow \mathbf{LNM'}_{(\sigma,\Upsilon,t)} = \mathbf{LNM}_{(\sigma,\Upsilon,t)} \cup \mathbf{S'}_{\sigma,t} \quad (21)$$

$$|\text{ differences } (\mathbf{S'}_{\sigma,t}, \mathbf{LNM}_{(\sigma,\Upsilon,t)})| > \mathbf{h}, \Rightarrow \mathbf{LNM'}_{(\sigma,\Upsilon,t)} = \mathbf{LNM}_{(\sigma,\,new\_map,t)} \cup \mathbf{S'}_{\sigma,t} \quad (22)$$

$$\boldsymbol{lnm_t} = [\mathbf{LNM'}_{(1,\Upsilon,t)}, \mathbf{LNM'}_{(2,\Upsilon,t)}, \mathbf{LNM'}_{(3,\Upsilon,t)}, \ldots, \mathbf{LNM'}_{(n\_\sigma,\Upsilon,t)}] \quad (23)$$

$$\mathbf{NM}_{\mathbf{mapno}} \in R^{mxnxo}, \mathbf{IPM}_{\mathbf{mapno}} \in R^{mxnxo}, \mathbf{LPM}_{\mathbf{mapno}} \in R^{mxnxo} \quad (24)$$

$$\mathbf{\Theta\_NM} := \text{total NM's} \in N, \ \mathbf{\Theta\_IPM} := \text{total IPM's} \in N, \ \mathbf{\Theta\_LPM} := \text{total LPM's} \in N \quad (25)$$

$$\boldsymbol{all\_LNMs_t} := [\boldsymbol{all\_maps_{1,t}}, \boldsymbol{all\_maps_{2,t}}, \boldsymbol{all\_maps_{3,t}}, \ldots, \boldsymbol{all\_maps_{n\_\sigma,t}}] \quad (26)$$

$$\boldsymbol{all\_NMs_t} := [\mathbf{NM}_{1,t}, \mathbf{NM}_{2,t}, \mathbf{NM}_{3,t}, \ldots, \mathbf{NM}_{\Theta\_NM,t}] \quad (27)$$

$$\boldsymbol{all\_IPMs_t} := [\mathbf{IPM}_{1,t}, \mathbf{IPM}_{2,t}, \mathbf{IPM}_{3,t}, \ldots, \mathbf{IPM}_{\Theta\_IPM,t}] \quad (28)$$

$$\boldsymbol{all\_LPMs_t} := [\mathbf{LPM}_{1,t}, \mathbf{LPM}_{2,t}, \mathbf{LPM}_{3,t}, \ldots, \mathbf{LPM}_{\Theta\_LPM,t}] \quad (29)$$

$$\boldsymbol{all\_navmaps_t} := [\boldsymbol{all\_LNMs_t}, \boldsymbol{all\_NMs_t}, \boldsymbol{all\_IPMs_t}, \boldsymbol{all\_LPMs_t}] \quad (30)$$

$$\boldsymbol{modcode} := \text{module identification code} \in N \quad (31)$$

$$\boldsymbol{mapcode} := [\boldsymbol{modcode}, \boldsymbol{mapno}] \quad (32)$$

$$\boldsymbol{\chi} := [\boldsymbol{mapcode}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}] \quad (33)$$

$$\boldsymbol{feature} \in R, \ \boldsymbol{action} \in R \quad (34)$$

Φ_feature := last **feature** contained by a cube, **Φ_action** := last **action** contained by a cube, **Φ_χ** := last χ (i.e., address) contained by a cube (35)

$\boldsymbol{cubefeatures_{\chi,t}} := [\boldsymbol{feature_{1,t}},\ \boldsymbol{feature_{2,t}},\ \boldsymbol{feature_{3,t}},\ \ldots,\ \boldsymbol{feature_{\Phi\_feature,t}}]$ (36)

$\boldsymbol{cubeactions_{\chi,t}} := [\boldsymbol{action_{1,t}},\ \boldsymbol{action_{2,t}},\ \boldsymbol{action_{3,t}},\ \ldots,\ \boldsymbol{action_{\Phi\_action,t}}]$ (37)

$\boldsymbol{linkaddresses_{\chi,t}} := [\chi_{1,t}, \chi_{2,t}, \chi_{3,t}, \ldots, \chi_{\Phi\_\chi,t}]$ (38)

$\boldsymbol{cubevalues_{\chi,t}} := [\boldsymbol{cubefeatures_{\chi,t}}, \boldsymbol{cubeactions_{\chi,t}}, \boldsymbol{linkaddresses_{\chi,t}}]$ (39)

$\boldsymbol{cubevalues_{\chi,t}} = \boldsymbol{all\_navmaps_{\chi,t}}$ (40)

$\boldsymbol{linkaddresses_{\chi,t}} = \boldsymbol{link(\chi,t)}$ (41)

$\boldsymbol{grounded\_feature} := \forall_{feature}: \boldsymbol{feature} \in \boldsymbol{all\_LNMs_{\chi}}$ (42)

$\forall_{\chi,t}: \boldsymbol{all\_navmaps_{\chi,t}} = \boldsymbol{grounded\_feature}$ OR $\boldsymbol{link(all\_navmaps_{\chi,t})} \neq [\ ]$ OR $\boldsymbol{all\_navmaps_{\chi,t}} = [\ ]$ (43)

$\boldsymbol{s'\_series(t)} = [\boldsymbol{s'(t-3)}, \boldsymbol{s'(t-2)}, \boldsymbol{s'(t-1)}, \boldsymbol{s'(t)}]$ (44)

visual_series(t) = Sequential/Error_Correcting_Module**.**visual_inputs(***s'_series(t)***) (45)

auditory_series(t) = Sequential/Error_Correcting_Module**.**auditory_inputs(***s'_series(t)***) (46)

visual_motion(t) = Sequential/Error_Correcting_Module**.**visual_match(***visual_series(t)***) (47)

auditory_motion(t) = Sequential/Error_Correcting_Module**.**auditory_match(***auditory_series(t)***) (48)

$\mathbf{VNM} \in R^{mxnxo},\ \mathbf{AVNM} \in R^{mxnxo}$ (49)

$\mathbf{VNM'_t} = \mathbf{VNM_t} \cup \boldsymbol{visual\_motion(t)}$ (50a)

$\mathbf{VNM''_t} = \mathbf{VNM'_t} \cup \boldsymbol{auditory\_motion(t)}$ (50b)

$\mathbf{AVNM_t}$ = Sequential/Error_Correcting_Module**.**auditory_match_process(***auditory_series(t)***) (51)

$\mathbf{VSNM} \in R^{mxnxo}$ (52)

$\boldsymbol{visual\_segmented\_series(t)} = [\mathbf{VSNM_{t-3}}, \mathbf{VSNM_{t-2}}, \mathbf{VSNM_{t-1}},$ and $\mathbf{VSNM_t}]$ (53)

visseg_motion(t) = Sequential/Error_Correcting_Module**.**visual_match(***visual_segmented_series(t)***) (54)

$\mathbf{VSNM'_t = VSNM_t} \cup \boldsymbol{visseg\_motion(t)}$ (55)

$\mathbf{LNM'_{(1,\Upsilon,t)}} = \boldsymbol{lnm_t}[0]$ (56)

$\mathbf{CONTEXT} := \in R^{mxnxo}$ (57)

$\mathbf{WNM} := \in R^{mxnxo}$ (58)

$\mathbf{CONTEXT_t = WNM_t}$ (59)

$\mathbf{VSNM_t}$ = Object_Segmentation_Gateway_Module**.**visualsegment($\mathbf{LNM'_{(1,\Upsilon,t)}}$, $\mathbf{VNM''_t}$, $\mathbf{CONTEXT_t}$) (60)

$\mathbf{WNM_t}$ = Causal_Memory_Module**.**match_best_multisensory_navigation_map($\mathbf{VSNM'_t}$, $\mathbf{AVNM_t}$, $\mathbf{LNM'_{(3,\Upsilon,t)}}$**,** $\mathbf{LNM'_{(4,\Upsilon,t)}}$**,** …, $\mathbf{LNM'_{(\mathit{n\_\sigma},\Upsilon,t)}}$) (61)

h' = number of differences allowed to be copied onto existing navigation map $\in R$ (62)

$\boldsymbol{actual_t} = [\mathbf{VSNM'_t}, \mathbf{AVNM_t}, \mathbf{LNM'_{(3,\Upsilon,t)}}, \mathbf{LNM'_{(4,\Upsilon,t)}}, \ldots, \mathbf{LNM'_{(\mathit{n\_\sigma},\Upsilon,t}}]$ (63)

$\mathbf{NewNM} \in R^{mxnxo}$ (64)

$|\ \mathbf{differences(}\boldsymbol{actual_t}, \mathbf{WNM_t)}\ | \leq \mathbf{h'}, \Rightarrow \mathbf{WNM'_t = WNM_t} \cup \boldsymbol{actual_t}$ (65)

$|\ \mathbf{differences(}\boldsymbol{actual_t}, \mathbf{WNM_t)}\ | > \mathbf{h'}, \Rightarrow \mathbf{WNM'_t = NewNM_t} \cup \boldsymbol{actual_t}$ (66)

$\boldsymbol{emotion} \in R$ (67)

$\mathbf{GOAL} \in R^{mxnxo}$ (68)

$\boldsymbol{autonomic} \in R$ (69)

[$\boldsymbol{emotion_t}$**,** $\mathbf{GOAL_t}$] = Goal/Emotion_Module**.**set_emotion_goal($\boldsymbol{autonomic_t}$**,** $\mathbf{WNM'_t}$) (70)

$\mathbf{WIP} \in R^{mxnxo}$ (71)

$\mathbf{WIP_t}$ = Instinctive_Primitives_Module**.**match_best_primitive($\boldsymbol{actual_t}$, $\boldsymbol{emotion_t}$**,** $\mathbf{GOAL_t}$) (72)

$\mathbf{WLP} \in R^{mxnxo}$ (73)

$\mathbf{WLP_t}$ = Learned_Primitives_Module**.**match_best_primitive($\boldsymbol{actual_t}$, $\boldsymbol{emotion_t}$**,** $\mathbf{GOAL_t}$) (74)

$\mathbf{WPR} \in R^{mxnxo}$ (75)

$\mathbf{WLP_t} = [\ ], \Rightarrow \mathbf{WPR_t = WIP_t}$ (76)

$\mathbf{WLP_t} \neq [\,],\ \Rightarrow \mathbf{WPR_t} = \mathbf{WLP_t}$ (77)

$\boldsymbol{action_t}$ = Navigation_Module.apply_primitive($\mathbf{WPR_t}$, $\mathbf{WNM'_t}$) (78)

$\boldsymbol{output\_vector} \in R^{n'}$ (79)

$\boldsymbol{action_t}$ = ["move*"], $\Rightarrow$ $\boldsymbol{output\_vector_t}$ = Output_Vector_Association_Module.
action_to_output($\boldsymbol{action_t}$, $\mathbf{WNM'_t}$) (80)

$\boldsymbol{motion\_correction} \in R^2$ (81)

$\mathbf{action_t}$ = ["move*"], $\Rightarrow$ $\boldsymbol{motion\_correction_t}$ = Sequential/Error_Correcting_Module.
motion_correction($\boldsymbol{action_t}$, $\mathbf{WNM'_t}$, $\boldsymbol{visual\_series(t)}$) (82)
$\boldsymbol{output\_vector'_t}$ = Output_Vector_Association_Module.
apply_motion_correction($\boldsymbol{output\_vector_t}$, $\boldsymbol{motion\_correction_t}$) (83)

$\boldsymbol{explanation} \in R^{n''}$ (84)

$\boldsymbol{explanation_t}$ = Navigation_Module.navmap_to_proto_lang($\mathbf{WPR_t}$, $\mathbf{WNM'_t}$, $\boldsymbol{action_t}$) (85)

$\mathbf{LNM_{(\sigma,\Upsilon,t)}}$ = Input_Sensory_Vectors_Associations_Module$_\sigma$.
match_best_local_navigation_map($\mathbf{S'_{\sigma,t}}$, $\mathbf{WNM'_{t-1}}$) (86)

$\mathbf{WPR_{t-1}}$ = ["feedback intermediate*"], $\Rightarrow \forall_\sigma : \mathbf{LNM_{(\sigma,\Upsilon,t)}}$ =
Input_Sensory_Vectors_Associations_Module$_\sigma$.extract_σ($\mathbf{WNM'_{t-1}}$) (87)

$\boldsymbol{wnm\_stack} \in R^{n'''}$ (88)

$\mathbf{WNM'_t}$ = Navigation_Module.push_stack($\boldsymbol{wnm\_stack_t}$) (89)

$\mathbf{WNM'_t}$ = Navigation_Module.pop_stack($\boldsymbol{wnm\_stack_t}$) (90)

$\boldsymbol{link(\chi,t)} \in \boldsymbol{all\_LNMs_t}$ **OR** $\boldsymbol{link(\chi,t)} \in \boldsymbol{all\_NMs_t}$,
$\Rightarrow \mathbf{WNM'_t}$ = Navigation_Module.retrieve_map($\boldsymbol{link(\chi,t)}$) (91)

$\boldsymbol{link(\chi,t)} \in \boldsymbol{all\_IPMs_t}$ **OR** $\boldsymbol{link(\chi,t)} \in \boldsymbol{all\_LPMs_t}$,
$\Rightarrow \mathbf{WPR_t}$ = Navigation_Module.retrieve_map($\boldsymbol{link(\chi,t)}$) (92)

$\mathbf{\Gamma_{cca3}} = \boldsymbol{\eta\rho} + \sum \boldsymbol{\rho'}$ (i)

$\mathbf{e_{cca3}} = 1/\ \mathbf{\Gamma_{cca3}}$ (ii)

$\mathbf{\Gamma_{cca1}} = \boldsymbol{\tau\mu} + \sum \boldsymbol{\mu'}$ (iii)

$\mathbf{e_{cca1}} = 1/\ \mathbf{\Gamma_{cca1}}$ (iv)

$\mathbf{\Gamma_{cca3}} = 60\boldsymbol{\rho}$ (v)

cca1_sensory_vector, ***cca1_other_vectors***, ***all_cca1_sensory_vectors***, ***cca1_binding_vector***, ***cca1_working_primitive***, ***cca1_action*** $\in R^{n''''}$ (93)

cca1_sensory_vector$_{\sigma,t}$ **=** Input_Sensory_Vectors_Associations_Module**.** match_with_other_vectors ($\mathbf{S'_{\sigma,t}}$, ***cca1_other_vectors***$_t$) (94)

all_cca1_sensory_vectors$_t$:= [***cca1_sensory_vector***$_{1,t}$, ***cca1_sensory_vector***$_{2,t}$, **…**, ***cca1_sensory_vector***$_{n\_\sigma,t}$] (95)

cca1_binding_vector$_t$ **=** Sensory_Vectors_Binding_Module**.** match_with_other_vectors (***all_cca1_sensory_vectors***$_t$, ***cca1_other_vectors***$_t$) (96)

cca1_working_primitive$_t$ = (
Instinctive_Primitives_Module.match_primitive(***cca1_binding_vector***$_t$)
OR Instinctive_Primitives_Module.match_primitive(***cca1_binding_vector***$_t$)) (97)

cca1_ action$_t$:= Navigation_Module**.**all_steps_to_produce_action (***cca1_binding_vector***$_t$, ***cca1_working_primitive***$_t$) (98)

$\mathbf{\Gamma_{cca1}} = 5\boldsymbol{\mu} + 2\boldsymbol{\lambda}!$ (vi)

$\mathbf{\Gamma_{cca1}} > \mathbf{\Gamma_{cca3}}$ (vii)

$\mathbf{e_{cca3}} > \mathbf{e_{cca1}}$ (viii) □

Expansion of Dot Notation

For reasons of brevity, dot notation is used in algorithmic portions of the equations—this is discussed in the paper. Below full expansion of the dot notation is provided, either with the actual Python code, Python-style pseudocode, or mathematical-style pseudocode.

For example, consider (89, 90) above:

$$\boldsymbol{wnm\_stack} \in \mathrm{R}^{n'''} \quad (88)$$

$$\mathbf{WNM'_t} = \text{Navigation\_Module}\mathbf{.}\text{push\_stack}(\ \boldsymbol{wnm\_stack_t}\ ) \quad (89)$$

$$\mathbf{WNM'_t} = \text{Navigation\_Module}\mathbf{.}\text{pop\_stack}(\ \boldsymbol{wnm\_stack_t}\ ) \quad (90)$$

One can see the dot-notation being used—"**.push stack**" and "**.pop stack**", and one would want to know what algorithmic steps exactly occur within these equations.

With regard to (88) the actual Python code is:

```
self.gb = np.empty((self.total_maps, 6, 6, 6,self.total_objects), 
dtype=object)
self.wnm_stack = [] #holds pointers to wnm’s within self.gb
```

As noted in the text of the paper, there is the need to be able to keep multiple Working Navigation Maps readily available, and hence the use of stacks to push and pop off recently used Working Navigation Maps. **self.wnm_stack** is a vector which can store copies of Working Navigation Maps, which corresponds to (88) and the text within the paper. Rather than pass full arrays between equations, one can see that pointers to the arrays within **self.gb** (a larger array created using the Python-compatible NumPy library with support for large multi-dimensional arrays) are stored in this vector, and can be passed between equations, i.e., between Python methods.

With regard to (89) the actual Python code is:

```
def push_stack(self, wnm: int, verbose: bool = False) -> bool:
        '''in_use_do_not_archive
        pushes a navmap, usually a working navigation map, onto 
        the stack self.wnm_stack
        -an instinctive or learned primitive assigned to the Working 
Primitive WPR'
        can push a current Working Navigation Map WNM' onto the stack 
in this method
        -in equation (89) the value returned is the WNM', i.e., this 
value does not
        change, thus, to be programmatically more useful, a bool value 
is returned
```

```
        (for example, if there is a memory full issue due to perhaps a
code or algorithmic
        error and a False will be returned)
        -WNM' could be updated with the same value plus some flag-value
that indicates
        it has been pushed onto the stack value, but this is not done
at
        present since other parts of the code do not use it
        -note: wnm is not a full map but the mapno of the working
navigation map,
        i.e., we pass pointers to the navmaps within self.gb rather than
the
        actual arrays holding the navmaps
        -small lists of pointers, thus deque not used due to dealing
with another data type
        -non-threading environment at present, thus LifoQueue not used
        -input parameters:
            wnm - mapno of a navmap, usually a working navigation map
WNM', an int
            verbose - verbose mode, useful for development
        -returns:
            True/False
        '''
        try:
            self.wnm_stack.append(wnm)
            if verbose:
                print(f'\nwnm_stack is now {self.wnm_stack}\n')
            return True
        except:
            return False
```

With regard to (89) the Python-style pseudocode is shown below. The purpose is to increase understandability without significantly reducing the precision of the code. Thus below, for example, the error methods used in the actual code are not mentioned as they are not events expected to occur routinely nor of algorithmic importance. As well, although **push_stack( )** does not explicitly return **wnm_stack**, the object-oriented code has access to the updated **wnm_stack** which is the real output of this method.

```
PUSH_STACK(WNM', wnm_stack)
INPUT: A pointer to the array holding the working navigation map WNM'
OUTPUT: An updated vector wnm_stack holding pointers to working navigation maps
push WNM' onto vector wnm_stack
```

With regard to (90) the actual Python code is:

```
def pop_stack(self, verbose: bool = False, clear: bool = False) -> int:
        '''in_use_do_not_archive
        pops a navmap, usually a working navigation map, and
        returns this value from the stack self.wnm_stack
        -note: wnm is not a full map but the mapno of the working
navigation map,
        i.e., we pass pointers to the navmaps within self.gb rather than
the
        actual arrays holding the navmaps
        -as per equation (90) this value returned corresponds to the
value that
        now will be assigned as the new Working Navigation Map WNM'
        -small lists of pointers, thus deque not used due to dealing
with another data type
        -non-threading environment at present, thus LifoQueue not used
        -input parameters:
            has access to self.wnm_stack
            clear - if set then clears whole stack, useful at a new
mission
            verbose - verbose mode, useful for development
        -returns:
            value popped off the stack, usually a mapno to a working
            navigation map (an int) WNM'
            if there is an attempt to pop a value off an empty stack
then the mapno
            of the last navmap created will be returned instead; can
consider an
            error code in the future instead or capture the exception
        '''
        if clear:
            self.wnm_stack = []
            print('\nhdata.pop_stack clear flag set, wnm_stack cleared')
            return 0
        try:
            wnm = self.wnm_stack.pop()
            if verbose:
                print(f'\nwnm popped value is {wnm} and wnm_stack is
{self.wnm_stack}\n')
            return wnm
        except:
            print('\ndebug: pop_stack unable to pop further navmaps or
other values')
            print('debug:    returning    last    navmap    created',
self.next_free_map)
            return self.next_free_map
```

With regard to (90) the Python-style pseudocode is shown below.

```
POP_STACK( wnm_stack)
INPUT:  A vector wnm_stack holding pointers to working navigation maps
OUTPUT:  A pointer to the new working navigation map WNM' popped off of wnm_stack
pop new working navigation map WNM' from vector wnm_stack
```

Expansion of Equations utilizing Dot Notation

pending