1. Introduzione

Questo manuale descrive le procedure fondamentali per l'utilizzo dell'ambiente SIMP/STEP. Vengono spiegati il modo per lanciare esperimenti già esistenti e le funzioni fondamentali del SIMP, così da costruirne di nuovi. Il tutto viene svolto servendosi di alcuni esempi presenti nella distribuzione dell'ambiente attualmente presente nel laboratorio di calcolo.

2. Caratteristiche dell'ambiente SIMP/STEP

L'ambiente di programmazione SIMP/STEP è strutturato in tre livelli : il livello STEP-engine, l'ambiente SIMP, e il livello esperimenti.

Il livello più basso, l'engine, implementa le specifiche STEP (Space Time Event Processor), cioè descrive una macchina multiprocessore astratta scritta in C, ma che potrebbe anche essere costruita effettivamente con hardware dedicato (ad esempio, la CAM8 è una macchina STEP). Nell'engine sono implementati i motori di calcolo e di visualizzazione, che permettono l'esecuzione di esperimenti su PC.

Il secondo livello implementa l'ambiente SIMP vero e proprio, cioè una serie di moduli ,di funzioni e di strutture dati, scritte in linguaggio Python (di cui si descriveranno le caratteristiche principali in appendice), le quali forniscono strumenti adatti alla stesura di esperimenti di materia programmabile. La comunicazione tra i primi due livelli avviene tramite le STEP-API (application programming interface), cioè una serie di funzioni che, quando sono richiamate dal programma, si interfacciano direttamente con il primo livello ed utilizzano l'engine nel modo appropriato.

Il terzo livello è quello degli esperimenti, cioè delle applicazioni specifiche, scritte o eseguite dall'utente usando tutti gli strumenti, già pronti, che il secondo livello (l'ambiente) mette a disposizione. Nella distribuzione attuale del SIMP sono presenti vari esempi di esperimenti già pronti, per la maggior parte "tradotti" dai corrispondenti presenti sulla CAM8.

La struttura a livelli rende più semplice l'utilizzo di simulazioni di materia programmabile poiché non è necessario, ad esempio, conoscere i

dettagli del funzionamento dell'engine per scrivere un esperimento. Basta solo saper utilizzare i moduli già pronti forniti dal SIMP.

Per programmare degli esperimenti in ambiente SIMP/STEP, è necessario possedere alcuni requisiti fondamentali:

- La conoscenza del sistema operativo Linux, in una delle sue più recenti distribuzioni, poiché SIMP è momentaneamente disponibile solo per esso. In un futuro prossimo è previsto il porting verso altri sistemi operativi.
- La conoscenza del linguaggio di programmazione Python, sul quale si basa l'interfaccia utente del SIMP. Python è un linguaggio interpretato, moderno e molto semplice da apprendere ma, allo stesso tempo, di notevole potenza e rispettoso delle regole di programmazione object-oriented.
- Non è necessario, ma non guasta, possedere una conoscenza di base del funzionamento della CAM-8, visto che molta della terminologia del SIMP è ereditata da questa.

3. Usare SIMP

La sintassi per l'esecuzione di SIMP è la seguente:

SIMP [opzioni] nomefile[.x]

Le opzione disponibili da linea di comando sono :

-h	visualizza l'aiuto ed esce
-v	visualizza la versione ed esce
--demo	esegue una dimostrazione
-d	debug mode: ricompila le tavole di LookUp e le ColorMap invece di usare quelle eventualmente già pronte.
-D	debug di sistema

Ad esempio, supponiamo di voler lanciare l'esperimento *simp/examples/hpp/hpp.x*

Una volta posizionati nella directory corretta, basta lanciare il comando

simp *hpp.x*

e compare la finestra grafica con la configurazione iniziale. Osserviamo che l'estensione ".x" può essere omessa, poiché viene aggiunta

automaticamente dall'interprete. Se la procedura è corretta ,ciò che si vede corrisponde alla fig. 2.1.

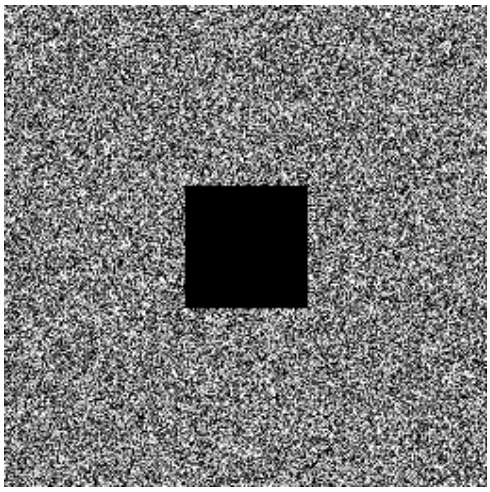

Fig. 2.1

A questo punto, si hanno a disposizione diversi comandi per operare sull'esperimento. Il tasto da premere viene riportato in grassetto tra parentesi quadre. Il parametro ARG presente in alcuni comandi indica un valore numerico da usare come argomento del comando. Il numero relativo ad ARG va digitato prima dell'istruzione corrispondente, ad esempio:

ARG **[comando]**

In generale ARG è un valore decimale; se si vuole che sia un esadecimale si deve anteporre il carattere "#" all'inserimento del valore numerico.

[INVIO] Esecuzione continua: l'esperimento viene eseguito ininterrottamente fino alla pressione di **[SPAZIO]** o del tasto **[q]** che rappresenta quit.

[SPAZIO] Passo singolo: l'esperimento viene eseguito un passo alla volta ad ogni pressione della barra spaziatrice. Se prima della barra viene inserito un valore numerico, vengono eseguiti ARG passi.

[q] Termina il programma.

[h] Visualizza l'aiuto in linea e dà una breve descrizione dei comandi disponibili.

[=]	Ingrandimento: ingrandisce nella stessa finestra grafica.
[-]	Ritorna alle dimensioni normali.
[i]	Zoom in: raddoppia la dimensione della finestra.
[o]	Zoom out: dimezza la dimensione della finestra.
[Ctrl-p],[↑]	Scroll up: sposta il contenuto della finestra verso l'alto. Se viene specificato un intero ARG, sposta la finestra di ARG pixel.
[Ctrl-f],[→]	Pan right: sposta il contenuto della finestra verso destra. Se viene specificato un intero ARG, sposta la finestra di ARG pixel.
[Ctrl-n],[↓]	Scroll down: sposta il contenuto della finestra verso il basso. Se viene specificato un intero ARG, sposta la finestra di ARG pixel.
[Ctrl-b],[←]	Pan left: sposta il contenuto della finestra verso sinistra. Se viene specificato un intero ARG, sposta la finestra di ARG pixel.
[Ctrl-r]	Ridisegna la finestra grafica (refresh).
[Ctrl-l]	Ripristina il livello di zoom originale della finestra grafica.
[g]	Get pattern: carica una nuova configurazione iniziale dal file poi specificato.
[p]	Put pattern: salva la configurazione attuale nel file poi specificato.
[b]	Esegue un test di valutazione della velocità (benchmark) di ARG passi.

3. Scrivere esperimenti in SIMP

Gli esperimenti SIMP/STEP si basano sul concetto di gas reticolare piuttosto che su quello di automa cellulare classico. Non si parla, quindi, di celle, bensì di *segnali* ed *eventi*.

Un esperimento è un semplice file di testo, con estensione ".x". Per scriverlo può essere utilizzato qualunque editor di testo.

L'esperimento è strutturato in sezioni che vengono trattate separatamente:

Intestazione : In questa sezione, la macchina virtuale viene costruita definendo le dimensioni dello spazio di lavoro e dichiarando i segnali che saranno usati nell'esperimento.

Legge di movimento (trasporto dei dati) : Viene descritto il trasporto dei segnali in maniera molto simile alla procedura di *kicking* della CAM8.

Definizione degli eventi : Le funzioni di transizione vengono descritte in Python e verranno compilate nelle tavole di LookUp (da qui in avanti chiamate EventTable).

Definizione delle ColorMap : Le ColorMap sono anch'esse funzioni scritte in Python e servono a visualizzare sullo schermo i segnali con i colori stabiliti.

Inizializzazione : L'engine e lo spazio di simulazione vengono opportunamente preparati al lavoro successivo con una configurazione iniziale particolare.

Segue una descrizione delle più importanti funzioni SIMP.

3.1 Intestazione

Ogni esperimento da eseguire in ambiente SIMP/STEP inizia con alcune dichiarazioni fondamentali, di cui alcune implicite e altre necessarie. La prima istruzione, che può essere omessa perché implicitamente aggiunta dal SIMP, è:

NewExperiment ()

Essa crea un nuovo oggetto che ingloba l'esperimento corrente.
Dopo di che vanno dichiarate le dimensioni dello spazio di simulazione, mediante l'uso di una variabile speciale **dimen**, con la seguente sintassi:

dimen = *lista di interi*

dove *lista di interi* è un elenco contenente le singole dimensioni spaziali. Ad esempio, se volessimo definire come spazio un quadrato di dimensioni 256x256, la sintassi corretta sarebbe:

dimen = [256,256]

mentre per un parallelepipedo 10x100x100 dovremmo semplicemente scrivere:

dimen = [10,100,100]

avendo cura di rispettare l'ordine delle dimensioni. Per compiti particolari potrebbero servire strutture più complesse; il SIMP offre la possibilità di definire delle *liste di liste di interi*, come nel seguente esempio:

dimen = [[100,1,1],[1,100,1],[1,1,100]]

Dopo aver definito lo spazio, si devono dichiarare i segnali da usare nel corso dell'esperimento. Per ciò si usa la funzione:

NewSig (*numero di stati*)

Essa crea un oggetto di tipo *segnale* con un numero di stati specificato da un numero intero. Ad esempio, se si volesse creare un segnale **SIG** a quattro stati, chiamati 0,1,2,3, si dovrebbe scrivere:

SIG = **NewSig**(4)

Per chiarire meglio, si supponga di voler costruire un esperimento che utilizzi un vicinato di Von Neumann. I segnali sono quelli relativi al centro **c,** al nord **n**, all'est **e**, al sud **s** e all'ovest **w**, tutti in grado di assumere uno stato dall'insieme {0,1}. La sintassi per la creazione dei necessari segnali è:

c = **NewSig**(2)

n= **NewSig**(2)
s = **NewSig**(2)
w = **NewSig**(2)
e = **NewSig**(2)

I segnali rappresentano un passaggio di informazioni tra celle vicine. Essi hanno un valore di input e un valore di output. L'input rappresenta lo stato nel quale si trova la cella al passo precedente, l'output il nuovo valore dello stato. Per accedere al valore di output di un segnale occorre posporre ad esso il carattere di underscore (_).

Ad esempio, si considerino i segnali **c** (cento) e **w** (ovest). Se si ha come scopo quello di copiare il vecchio stato di **w** come nuovo stato di **c** si deve assegnare al valore di output di **c** quello di input di **w**. L'istruzione adatta allo scopo è:

c._ = w

È anche possibile raggruppare alcuni segnali dello stesso tipo in un unico insieme ordinato. Ciò può essere fatto mediante la classe **bundle** che può assumere un numero di stati pari al prodotto di quelli contenuti nei singoli segnali che lo compongono.

Nonostante il SIMP si occupi della gestione della memoria necessaria a contenere il numero di segnali da noi creati, è comunque necessario che tale numero sia limitato per non incorrere nei problemi derivanti dal rallentamento della velocità di calcolo e dall'eccessivo dimensionamento delle *EventTable*. Un buon suggerimento potrebbe essere di non usare più di 16 segnali a due stati (corrispondenti a 16 bit) oppure , corrispondentemente, 8 segnali a quattro stati.

Segue la descrizione di due istruzioni opzionali che possono essere inserite nella fase di intestazione.

DEBUG = *int*

dove *int* corrisponde a 0 o 1. Se **DEBUG** viene settato a 1 le EventTable vengono ricompilate anziché essere lette da disco.

VIEWER = *nome del visualizzatore*

permette di specificare quali routine grafiche utilizzare per la visualizzazione dell'esperimento. In realtà, attualmente è disponibile il solo Xviewer per Linux.

L'intestazione si chiude con due istruzioni fondamentali:

MakeEngine ()

che crea l'appropriata macchina virtuale per l'esecuzione dell'esperimento in base ai segnali definiti, e

MakeRenderer (*s1,s2,…)*

che seleziona i segnali da visualizzare nella finestra grafica.

3.2 Legge di movimento (trasporto dei dati)

Il trasporto dei dati, in ambiente SIMP/STEP, avviene in maniera molto simile all'ambiente CAM/STEP. Viene anche mantenuta una terminologia simile. L'oggetto che descrive i movimenti dei singoli segnali, infatti, è chiamato **kick**. Si suppone di disporre su strati sovrapposti *(tiers)* i segnali da noi definiti nella prima fase. La procedure di kicking può essere descritta come lo "scivolamento" degli strati uno sull'altro, per ottenere delle pile di bit sulle quali applicare le EventTable che saranno definite in 3.3.

Un oggetto **kick** viene creato dalla funzione **NewKick**() con la seguente sintassi:

kick1 = **NewKick**()

dove **kick1** è l'oggetto creato.

A questo punto si possono definire i movimenti dei segnali lungo gli assi. Si consideriuno spazio bidimensionale. Se si volesse, ad esempio, muovere il segnale **s1** di una posizione in basso, e **s2** di una posizione a sinistra, si avrebbe:

k = **NewKick**()
k[**s1**] = [0,+1] una posizione verso il basso
k[**s2**] = [-1,0] una posizione a sinistra

Da questo esempio si vede anche come, in ambiente SIMP, il verso positivo degli assi sia rivolto verso destra per l'asse X e verso il basso per l'asse Y. Per chiarire meglio si consideri l'esempio del vicinato di Von Neumann. Siano **c**,**w**,**e**,**n**,**s** i segnali necessari e si voglia definire un movimento che li incolonni sullo stesso sito (quello di **c**). Allora le corrispondenti istruzioni sono:

k = **NewKick**()
k[**n**] = [0,+1]
k[**s**] = [0,-1]
k[**w**] = [+1,0]
k[**e**] = [-1,0]
k[**c**] = [0,0]

l'ultima riga può essere omessa poichè i movimenti sono impostati a zero per default.

3.3 Definizione degli eventi

Un evento è una funzione, scritta in Python, che definisce una regola di transizione. Non ci sono particolari restrizioni nella definizione della funzione. È chiaro, però, che maggiore è la sua complessità, minore è la velocità di esecuzione dell'esperimento. Al momento dell'esecuzione della simulazione, ogni funzione definita in questo modo viene compilata in una specifica EventTable.

Rifacendosi al modello di gas reticolare, i segnali hanno degli stati di input e degli stati di output (sostanzialmente uno stato di input è lo stato al passo attuale, quello di output è lo stato in cui si troverà il segnale al passo successivo). Le EventTable associano ad un particolare stato di input il corrispondente stato di output similmente a quanto accade nella CAM8 con le LookUp Table.

Si supponga di avere un segnale **s** a quattro stati, cioè definito da:

s = NewSig(4)

Gli stati di input possono essere usati in espressioni logiche o aritmetiche con numeri interi (ad esempio nell'istruzione "if") e per poter accedere ad essi basta usare il nome del segnale, senza alcun attributo. Per poter scrivere sullo stato di output, invece, occorre usare l'attributo "_" (underscore), altrimenti non verrebbe modificato nulla. Per esempio se si volesse porre **s** nello stato 2, quando fosse nello stato 1, dovremmo scrivere:

if **s** == 1: **s**._ = 2

Se si tenta di assegnare a un segnale un valore al di fuori del range consentito (numero stati –1) il programma si blocca e si ottiene un messaggio di errore.

Spesso è conveniente definire una funzione **fanout**() che, al termine di una serie di calcoli, copi lo stato del segnale osservato in quello dei segnali vicini. Per evitare che venga copiato il vecchio stato si usa la funzione speciale

Latch(*Lista di segnali*)

che permette di collegare l'output di un segnale all'input di un altro. Si veda un esempio con il solito vicinato di Von Neumann: Siano **c,e,w,s,n** i segnali già definiti e si supponga di aver definito una funzione

```
def fanout() :
    n._ = s._ = e._ = w._ = c
```

che copia lo stato di **c** nei segnali vicini. Se prima di essa non si usasse la funzione **Latch()** si copierebbe nei vicini lo stato che **c** aveva all'inizio del passo attuale (input) e non quello calcolato con una opportuna funzione di transizione. Quindi lo schema della funzione di transizione che verrà poi compilata nell'EventTable è:

```
def  trans() :
    # calcoli della funzione
    Latch()
    fanout()
```

3.4 Definizione dello Step

Una volta definite tutte le funzioni che si vogliano compilate nelle EventTable si può definire lo "step" vero e proprio con la funzione **STEP**(). All'interno di essa si devono usare, nell'ordine appropriato alla particolare simulazione, una serie di istruzioni:

DoTravel(***exkick***)

dove ***exkick*** è il nome dell'oggetto **kick** creato in precedenza

DoInteract(***extrans***)

dove ***extrans*** è la funzione di transizione da compilare nell'EventTable

Per esempio, avendo definito un'unica funzione **trans**(), e un unico kick **K**, la funzione **STEP**() diventerebbe:

```
def STEP():
    DoTravel(K)
    DoInteract(trans)
```

Per esperimenti più complessi, è necessario aver definito più oggetti **kick** ed EventTable. Si potrebbe anche dover fare delle scelte su quali usare in un determinato momento. Dato che la funzione **STEP**() è una funzione Python come tutte le altre può essere definita in maniera più complessa, ad esempio:

```
def STEP():
    if condizione1 :
        DoTravel(K1)
        DoInteract(trans1)
    elif condizione2 :
        DoTravel(K2)
        DoInteract(trans2)
        DoInteract(trans3)
```

3.5 Definizione delle ColorMap

L'ambiente SIMP possiede degli *pseudosegnali* definiti appositamente per la costruzione delle ColorMap. Essi corrispondono ai colori visualizzabili nella finestra grafica e sono: **Red**, **Grn**, **Blu**, **Grey**. Possono assumere valori in virgola mobile da 0.0 (assenza di colore) a 1.0 (saturazione massima) relativi a diverse intensità del colore associato. Poiché si comportano come i segnali normali, per assegnare ad essi un valore è necessario usare l'attributo "_".

La definizione di una ColorMap è molto simile a quella di una EventTable, con la differenza che le assegnazioni vengono fatte solo agli output degli pseudosegnali. Si può dire, quindi, che le ColorMap sono delle tavole che a un particolare stato di input di un segnale fanno corrispondere un colore ben determinato sulla finestra di visualizzazione.

Per fare un esempio, supponiamo di voler visualizzare in rosso pieno un punto in cui il segnale **s1** si trovi allo stato **1** e il segnale **s2** allo stato **0**. La funzione sarebbe cosi definita:

def **MiaMappa**():
    if (**s1** == 1 **and s2 ==** 0) : **Red**._ = 1.0

Nella versione corrente del SIMP le mappe sono compilate e selezionate con l'unica istruzione:

UseCmap(*nome mappa*)

3.6 Inizializzazione

L'ultima sezione di un esperimento è quella in cui è inizializzazioto lo spazio. Nelle versioni preliminari del SIMP era necessario definire alcune funzioni per settare il passo iniziale, il passo di aggiornamento e un eventuale passo finale (**update_scan**(), **starting_scan**(), **stopping_scan**()). Nell'attuale versione non ce n'è bisogno, in quanto la funzione **STEP**(), definita in 3.4, viene automaticamente usata come funzione di aggiornamento. Se si avesse bisogno di un passo iniziale diverso, si potrebbe scrivere una funzione *ad hoc* da richiamare all'interno della funzione di inizializzazione.

Alcune utili istruzioni che possono essere incluse nella funzione di inizializzazione :

SetSpace(*lista di segnali*, [*bot*,[*top*]])
    Pone gli stati di output dei segnali definiti nella lista uguali a un determinato valore.

Vengono coinvolti solo i segnali del sottospazio compreso tra "bot" e "top"; se "top" non è definito allora il sottospazio è composto solo dal punto specificato da "bot".

Esempio: Se si vuol porre **s1 =** 1 nel sottospazio compreso tra le coordinate (50,50) e (100,100) allora l'istruzione è:

SetSpace(**s1** = 1, [50,50],[100,100])

Un'altra utilissima funzione è:

Randomize([*bot,top*],*segnale=proporzione*)
    Pone gli stati dei segnali compresi tra "bot" e "top" in modo casuale secondo la distribuzione definita da "proporzione".

"Proporzione" è una lista di interi che definisce il rapporto relativo tra gli stati del segnale.

Per esempio, volendo porre un segnale **s1** a tre stati (0,1,2) in modo tale che il valore degli stati sia distribuito secondo la proporzione 3/4/1 allora la funzione diventerebbe:

Randomize(**s1** = [3,4,1])

Sarà compito del SIMP normalizzare il rapporto definito in **Randomize**(). L'istruzione **Randomize**(**s1**=[12,16,4]) sarebbe, infatti, perfettamente identica alla precedente.

Altre funzioni di controllo sono quelle di gestione della finestra grafica.

SetMag(ingrandimento)
Decide il valore di ingrandimento della finestra

SetZoom(zoom)
Pone il valore di zoom

SetOrigin(origine)
Decide il punto di origine del sottospazio da visualizzare

SetWindowsSize(X,Y)
Decide le dimensioni della finestra

PanLeft(*n*)
Sposta lo spazio di *n* punti a sinistra

PanRight(*n*)
Sposta lo spazio di *n* punti a destra

Per una lista completa delle funzioni di visualizzazione del SIMP si rimanda all'appendice B.

In SIMP esiste la possibilità di creare nuovi comandi per l'interprete da associare a un tasto durante l'esecuzione. Questi vengono creati come normali funzioni Python mentre il collegamento con il giusto tasto avviene con l'istruzione:

BindKey("*tasto*", *funzione*)

Se, per esempio, si volesse associare al tasto "U" una funzione che mostri uno stato particolare si può definire una funzione :

def **Show_sig**()
    # istruzioni varie
e assiociarla al tasto "U" con l'istruzione

BindKey("U", **Show_sig**)

3.7 Altre istruzioni del SIMP/STEP

Una lista completa di tutte le istruzioni, i costrutti e le funzioni speciali dell'ambiente SIMP si trova nell'appendice B. È interessante, però, presentarne alcune particolarmente utili per compiti particolari.

Per effettuare dei *benchmark*, cioè delle prove di velocità, sull'esperimento, esiste la funzione:

Benchmark (*numero di passi*)

che esegue uno specifico numero di passi e riporta il tempo impiegato per la computazione. Con l'istruzione:

ResetBench()

si riazzerano le variabili di banchmarking.
Per aprire un esperimento dall'interno di quello in esecuzione si usa la funzione SIMP:

RunExperimentFile(*nome file*).

4. Un esempio pratico: HPP.x

Il modello HPP simula il movimento di particelle gassose.
Si consideri un reticolo bidimensionale quadrato e si voglia descrivere il movimento di particelle gassose su di esso con moto rettilineo uniforme. Ad ogni passo una particella si muove lungo una delle quattro direzioni possibili e, se incontra una particella che proviene dalla direzione opposta, collide con essa e cambia direzione di 90 gradi.

Il file corrispondente a questo esperimento si trova nella directory "simp/demo/hpp/" e il listato completo è riportato in appendice.

L'implementazione di questo modello in SIMP inizia, come spiegato nei paragrafi precedenti, con la dichiarazione dello spazio di lavoro :

```
DEBUG  = 1          #ricompia le EventTable
X      = 0x100
Y      = 0x100
dimen  = [X,Y]
```

Viene definito uno spazio di 256x256 celle (0x100 x 0x100 in esadecimale) e, essendo la variabile debug impostata a 1, le EventTable saranno ricompilate ogni volta che si lancerà l'esperimento. Alla variabile **dimen** viene assegnata la dimensione dello spazio definita con le variabili **X** e **Y**.

```
p0  = NewSig(2)
p1  = NewSig(2)
p2  = NewSig(2)
p3  = NewSig(2)
pp  = Bundle(p0,p1,p2,p3)
```

Nel bundle pp vengono riuniti i 4 segnali p0,p1,p2,p3. Poiché devono essere visualizzati tutti, viene dichiarato l'intero bundle come segnale su cui effettuare il rendering.

```
MakeRenderer(pp)
```

L'engine, creato con queste caratteristiche, simulerà quattro *bitplane* (strati) in quanto sono stati dichiarati 4 segnali da 1 bit ciascuno, quindi ciascuno con due stati possibili.

```
K    = NewKick()
K[p0] = [0,0]
K[p1] = [-1,0]
K[p2] = [0,-1]
K[p0] = [-1,-1]
```

Questa è la definizione del *calcio* **K.**
Analizzandola, si vede che vengono eseguite le seguenti operazioni:

lo strato 0 rimane fermo

lo strato 1 viene calciato verso ovest
lo strato 2 viene calciato verso nord
lo strato 3 viene calciato verso nord-ovest

o con altra terminologia:

il segnale **p0** rimane fermo
il segnale **p1** viene inviato a ovest
il segnale **p2** viene inviato a nord
il segnale **p3** viene inviato a nord-ovest

```
def hpp():
    if ((p0==p3) and (p1==p2) and (p0!=p1)):
        p0._ = p1;   p1._ = p0
        p3._ = p2;   p2._ = p3
        Latch()
    swap()
```

La funzione **hpp**() è la funzione di transizione e rappresenta le regole dell'automa HPP. Da notare che la funzione **swap**() richamata all'interno di **hpp**() esegue un semplice scambio diagonale tra i segnali:

```
def swap():
    p0._ = p3;   p1._ = p2
    p3._ = p0;   p2._ = p1
```

La funzione **STEP**() riassume l'evoluzione del sistema, richiamando il trasporto dei dati e la tavola costruita a partire dalla funzione **hpp**():

```
def STEP():
    DoTravel(K)
    DoInteract(hpp)

def bw4():
    Gray._ = (pp)/4.0
```

La ColorMap **bw4**() assegna 4 livelli di grigio a ogni punto a seconda di quanti tra i 4 bit **p0,p1,p2,p3** sono accesi.

Infine il comando **UseCmap(bw4)** compila la mappa **bw4** in una CMap.

```
def initialize():
    RandomizeSpace(1,2,pp)
    pp._ = 0
    SetSpace([pp],[X*3/8,Y*3/8],[X*5/8,Y*5/8])
```

La funzione di inizializzazione assegna a tutti i bit un valore di 1 con una probabilità di ½ e, infine, crea uno spazio vuoto al centro della mappa con la funzione **SetSpace.**

BindKey("I",initialize)

In conclusione si assegna al tasto "**I**" la funzione di inizializzazione così che, ogni volta che questo viene premuto, lo spazio si riporta alla configurazione iniziale, senza dover riavviare il programma.

5. Un altro esempio: SIMPLIFE.x

Per comprendere meglio alcune caratteristiche dell'ambiente SIMP, è opportuno considerare un altro esempio: Il file LIFE.X, di cui si riporta il listato in appendice (disponibile anche nella sottodirectory /simp/demo/life o /simp/examples/life a seconda della distribuzione).

Esso è la trasposizione per SIMP/STEP del famoso "gioco della vita" di John Conway. Le regole sono molto semplici: ogni cella ha due stati, vivo o morto (1 o 0).

Si consideri il vicinato di Moore di cui è disponibile una rappresentazione grafica in Fig 1.3.

Ad ogni passo si sommano i valori delle celle del vicinato. Se la cella era morta e la somma dei valori dei vicini è 3, la cella diventa viva, altrimenti rimane morta. Se, invece, la cella era viva e la somma dei valori dei vicini è minore di 2 o maggiore di 3 la cella muore.

In questo esempio viene implementato anche l'*echo* (nel segnale **p**), che contiene il valore della cella al passo precedente ovvero, usando la terminologia del SIMP, il valore di input del segnale **q** viene copiato nel segnale di output del segnale **p**, e la traccia, che serve a mantenere memoria delle celle che in qualche momento sono state vive.
In fase di debug sarebbe opportuno cominciare con l'istruzione:

DEBUG = 1

che forza la ricompilazione delle EventTable e delle ColorMap ad ogni lancio dell'esperimento.

L'esperimento vero e proprio comincia con la definizione delle dimensioni dello spazio. Life utilizza una mappa bidimensionale di 256x256 celle (0x100 x 0x100 in esadecimale):

X = 0x100
Y = 0x100
dimen = [**X**,**Y**]

Si sarebbe anche potuto scrivere direttamente

dimen = [0x100,0x100]

ma in questo modo la leggibilità del programma è migliore.

Il secondo passo è la definizione dei segnali. Sarà necessario uno per ogni vicino e per la cella, quindi, essendo usato il vicinato di Moore, si devono definire i segnali relativi ai nove *vicini*. Inoltre serve un segnale per l'*echo*. Tutti i segnali avranno solo due stati: 0 = cella morta, 1 = cella viva.

```
q  = NewSig(2)      # il sito considerato
p  = NewSig(2)      # il segnale di echo
N  = NewSig(2)      # vicinato di Moore
S  = NewSig(2)
W  = NewSig(2)
E  = NewSig(2)
NW = NewSig(2)
NE = NewSig(2)
SW = NewSig(2)
SE = NewSig(2)
```

Il segnale **q** rappresenta la cella che deve essere visualizzata nella finestra grafica e **p** rappresenta l'eco, cioè lo stato della cella al passo precedente.

Successivamente viene dato il via alla costruzione dell'engine con l'istruzione

MakeEngine()

Gli unici segnali che si vogliono visualizzare nella finestra sono **q** e **p**, quindi si utilizza l'istruzione:

MakeRenderer(**q**,**p**)

Gli altri segnali non saranno mai visualizzati poichè rappresentano solo il vicinato e quindi sono utilizzati solo dalla funzione di transizione.

Il trasporto dei dati che, come è già stato detto, è un'implementazione del vicinato di Moore :

```
kick = NewKick()
kick[N]  = [0,+1]
kick[S]  = [0,-1]
kick[W]  = [+1,0]
kick[E]  = [-1,0]
kick[NW] = [+1,+1]
kick[NE] = [-1,+1]
kick[SW] = [+1,-1]
kick[SE] = [-1,-1]
```

Si consideri, ad esempio, l'istruzione :

```
kick[NW] = [+1,+1]
```

Essa definisce un movimento del segnale **NW** di una posizione verso il basso e di una verso destra (nel linguaggio della CAM8 è un *calcio* verso sud-est).

Terminata la definizione del trasporto dei dati, comincia l'implementazione della funzione di transizione e la compilazione dell'EventTable associata. In questo caso è utile costruire prima una funzione di fanout che copi lo stato del segnale **q** in tutti i sui vicini al termine di ogni passo. Essa è definita in modo molto semplice:

```
def fanout():
    N._=S._=W._=E._=NW._=NE._=SW._=SE._ = q
```

Poiché si sta copiando il valore dello stato di input di **q** nello stato di output dei vicini, occorre fare attenzione all'uso del carattere "_" (underscore). Quando viene usata la funzione **fanout**() così definita, si deve sempre usare anche la funzione speciale **Latch**(), come spiegato in 2.4.3.

Una possibile implementazione della funzione di transizione di LIFE potrebbe essere scritta con una serie di costrutti **if**..**then,** ad esempio:

```
SUM = q + N + S + W + E + NE + NW+ SW + SE
if (q==0 and SUM==3) :
    q._=1
```

```
elif (q==1 and (SUM<2 or SUM>3)) :
    q._=0
```

Grazie alle potenzialità del Python, però, si può ottenere la stessa cosa con una lista indicizzata (*tuple*) che rappresenta direttamente una tavola di LookUp:

```
SUM = q + N + S + W + E + NE + NW+ SW + SE
q._ = [0,0,q,1,0,0,0,0,0][SUM]
```

In questo caso, **SUM**, rappresenta l'indice della lista e si vede subito che alla posizione 0 (**SUM** = 0) corrisponde un valore di **q** di 0, alla posizione 1 corrisponde un valore di **q** uguale a 0 e la stessa cosa accade alle posizioni 3…8.

Quello che interessa osservare è che alla posizione 2 (**SUM**=2) corrisponde un valore di **q** pari a quello del passo precedente, mentre alla posizione 3 (**SUM**=3), **q** assume lo stato 1.

Con questo sistema, tutta la funzione si transizione si riduce a :

```
def bare():                 # nome della funzione
    SUM= q + N + S + W + E + NE + NW+ SW + SE
    q._ = [0,0,q,1,0,0,0,0,0][SUM]
    Latch()      # si richiama Latch() prima di fanout()
    fanout()
```

Per implementare la traccia e l'eco si usano due piccole funzioni aggiuntive che aggiungono delle semplici istruzioni prima della funzione **bare**().

Per l'eco :

```
def echo():
    p._ = q
    bare()
```

che copia lo stato di **q** in **p** e richiama **bare**(), mentre per la traccia :

```
def trace():
    p._ = p|q
    bare()
```

che assegna a **p** lo stato 1 se, al passo precedente **p** o **q** si trovavano nello stato 1 (cella viva).

Per definire il passo di calcolo, conviene introdurre una variabile, chiamata **event**, che assuma, secondo i comandi richiamati da testiera durante l'esecuzione, il valore **echo** o **trace**. Con questi accorgimenti si può finalmente definire lo step:

```
def STEP():
    DoTravel(kick)
    DoInteract(event)
```

In questo modo verranno eseguiti in successione, ad ogni passo, il movimento (calcio) **kick** e la EventTable **event** (che in realtà, come si vedrà poi, rappresenta **echo** o **trace**).

Terminata la fase di definizione della funzione di transizione, occorre caratterizzare, in modo molto simile la ColorMap. Con l'istruzione **MakeRenderer**(**q**,**p**), è già stato stabilito che i soli segnali ad essere visualizzati sono **q** e **p**, quindi la mappa **krgb**() si può definire in questa maniera:

```
def krgb():
    if q and p :        Red._ = 1
    elif q and not p :  Grn._ = 1
    elif not q and p :  Blu._ = 1
```

Questa mappa visualizzerà in verde le celle appena nate (in altri termini, i siti in cui il segnale **q** sia appena passato allo stato 1), in rosso le celle vive che lo erano anche al passo precedente e in blu la traccia, cioè le celle morte il cui segnale di echo sia allo stato 1.

Per compilare ed attivare la mappa appena definita si usa il comando :

```
UseCmap(krgb)
```

Per inizializzare l'esperimento, si costruisce la funzione **init**():

```
def init():
    global event
    event = echo
    Randomize(q=[1,1])
    DoInteract(fanout)
```

La prima istruzione che compare all'interno di **init**() è la definizione di una variabile globale chiamata **event**, che viene impostata al valore **echo**. Questo significa che, dal primo passo, la funzione di transizione che SIMP usa è **echo**() (traccia disattivata).

L'istruzione **Randomize**(**q**=[1,1]) distribuisce casualmente, con un rapporto di uno a uno, le celle vive e morte su tutto lo spazio. L'inizializzazione termina con un'esecuzione di **fanout**() per copiare lo stato di **q** sui vicini.

Per poter accedere alla funzione **init**() anche durante l'esecuzione dell'esperimento (cioè per poterlo re-inizializzare in ogni momento) si assegna a un tasto ("I") il comando corrispondente :

BindKey("I",**init**)

Infine, la funzione che permette di attivare la traccia (di passare alla EventTable **trace**()) e di assegnarla a un tasto è:

```
def toggle_trace():
    global event
    if event == echo : event = trace
    else event = echo

    BindKey("I",toggle_trace)
```

6. Elenco delle funzioni SIMP

Benchmark `([updates][steps])`	Esegue un test di velocità.
BindKey `(key,func[,name])`	Associa una funzione ad un tasto..
Bundle `class`	Classe per la gestione di insiemi di segnali.
CaptureWindow `([fname])`	Cattura la finestra corrente e la salva in un file.
Cmap `class`	Classe che indica la ColorMap
DEBUG `< Int >`	Flag che specifica se ricompilare le tavole o no.
Demagnify `()`	Riduce l'ingrandimento.
DoBurst `([NUM])`	Esegue NUM passi.
DoInteract `(func)`	Esegue la funzione *func*.
DoKick `(kick)`	Sinonimo di DoTravel.
DoTransport `(kick)`	Sinonimo di DoTravel.
DoTravel `(kick)`	Esegue il calcio *kick*.
ENGINE `< String >`	Nome dell'Engine da usare per l'esperimento.
Flush `()`	Flush the current instructions to the engine.
GetPattern `([fname])`	`Carica il file di configurazione` *fname*.
Kick `class`	Classe che rappresenta i calci.

Latch `([sigs])` Copia gli output negli input di un segnale.

LoadLookup `(fname)` Carica il file di configurazione *fname*.

MAG `< Int >` Fattore di ingrandimento con cui iniziare.

Magnify `()` Aumenta l'ingrandimento.

MakeCmap `(func)` Compila una ColorMap.

MakeEngine `()` Crea l'Engine per l'esperimento.

MakeLookup `(func)` Compila una EventTable dalla funzione *func*.

MakeRenderer `(signal,[signal]*)` Definisce i segnali da visualizzare.

NUMSTEPS `< Int >` Numero di passi da eseguire prima della visualizzazione.

NewExperiment `(fname)` Crea un nuovo esperimento.

NewKick `([frame_kick])` Crea un nuovo oggetto calcio.

NewSig `(nstates)` Crea un nuovo segnale di *nstates* stati.

PanLeft `([n])` Muove l'origine di *n* pixel a sinistra.

PanRight `([n])` Muove l'origine di *n* pixel a destra.

PutPattern `([fname])` Salva la configurazione corrente in un file.

RENDERER `< String >` Nome del "renderer" da usare per l'esperimento.

Randomize `([[from][,to],][signal=[w0,w1...]]*)` Crea un parallelepipedo con valori casuali

ResetBench `()` Inizializza le variabili per il test..

RestoreView `()` Centra la visuale.

RunExperimentFile `(fname)` Apre ed esegue un esperimento.

RunLoop `()` Esegue l'esperimento quando il pannello di controllo è inattivo.

SIMPCACHE `< String >` Percorso da utilizzare per la ricerca dei file di cache.

STEP `< Function >` Funzione che definisce lo STEP.

ScrollDown `([n])` Muove l'origine di *n* pixel in basso.

ScrollUp `([n])` Muove l'origine di *n* pixel in alto.

SetMag `(NUM)` Seleziona il livello di ingrandimento.

SetSpace `([[index][,offset],]signal=value,...)` Pone i segnali di un parallelepipedo ad un determinato valore.

SetZoom `(NUM)` Seleziona il livello di zoom.

Signal `class` Classe che implementa i segnali.

Sync `()` Sincronizza I segnali.

TIME_WINDOW `< Int >` Decide la lunghezza della TimeWindow

Table `class` Classe che rappresenta le funzioni di transizione.

UseCMap `(func)` Usa la ColorMap *func* per la visualizzazione.

VIEWER `< String >` Nome del visualizzatore da usare.

View `()` Disegna su schermo il contenuto dello spazio.

ZOOM `< Int >` Fattore di Zoom con cui iniziare l'esperimento.

ZoomIn `()` `Aumenta il fattore di Zoom`

Appendice

Listati

HPP.x

```
# tbach 6-7-01          -*-fundamental-*-
"""HPP Gas Rule:
Most simple gas"""

DEBUG=1

#------------------------------------------------------------------ Declarations

X       = 0x100;Y      = 0x100
DIMEN         = [X,Y]

#------------------------------------------------------------------ Signals

p0      =  NewSig(2)                    #  0 1
p1      =  NewSig(2)                    #  2 3
p2      =  NewSig(2)
p3      =  NewSig(2)
pp      =  Bundle(p0,p1,p2,p3)

MakeEngine()
MakeRenderer(pp)

#------------------------------------------------------------------ Transport
                        # Inertial transport in the Margolus "neighborhood"

K     = NewKick([.5,.5])
K[p0] = [0,0]
K[p1] = [-1,0]
K[p2] = [0,-1]
K[p0] = [-1,-1]


#------------------------------------------------------------------ Interaction

def hpp():
  if ((p0==p3) and (p1==p2) and (p0!=p1)):
    p0._ = p1;    p1._ = p0               # Horizontal Swap
    p3._ = p2;    p2._ = p3
    Latch();
  swap()
```

```
def swap():
  p0._ = p3;    p1._ = p2
  p3._ = p0;    p2._ = p1

#------------------------------------------------------------- Step

def STEP():
  DoTravel(K)
  DoInteract(hpp)

#------------------------------------------------------------- Color map

def bw4():  Gray._ = (pp)/4.0

#------------------------------------------------------------- Commands

UseCmap(bw4)
dist = [pp.states-1];
for x in range(1,pp.states): dist.append(1)
print len(dist)

print "mask %x" % pp.mask

def initialize():
  Randomize(pp=dist)
  SetSpace([X*5/16,Y*5/16],[X*11/16,Y*11/16],pp=0)

initialize();                                BindKey("I",initialize)
```

Life.x

```
# tt 6-1-01       -*-fundamental-*-
# LIFE with echo and toggleable trace


#DEBUG       = 1                              # Force Table & Cmap to update

#-------------------------------------------------------------- Geometry

X        = 0x80; Y  = 0x100
DIMEN          = [X,Y]

#-------------------------------------------------------------- Signals

q        = NewSig(2)                  # our site
p        = NewSig(2)                  # NW N NE    # echo
N        = NewSig(2)                  #
S        = NewSig(2)                  # W  q  E
W        = NewSig(2)                  #
E        = NewSig(2)                  # SW S SE
NW     = NewSig(2)
NE     = NewSig(2)
SW     = NewSig(2)
SE       = NewSig(2)

MakeEngine(); MakeRenderer(q,p)

#-------------------------------------------------------------- Travel

kick = NewKick()                      # von Neumann neighborhood
kick[N]          = [0,+1]
kick[S]= [0,-1]
kick[W]          = [+1,0]
kick[E]          = [-1,0]
kick[NW]= [+1,+1]
kick[NE]= [-1,+1]
kick[SW]= [+1,-1]
kick[SE]= [-1,-1]

#-------------------------------------------------------------- Interaction


def fanout():
  N._=S._=W._=E._=NW._=NE._=SW._=SE._ = q

def bare():
  sum = N+S+W+E+NW+NE+SW+SE
  q._ = [0,0,q,1,0,0,0,0,0][sum]
```

```
  Latch(); fanout()

def echo():
  p._ = q; bare()

def trace():
  p._ = p|q; bare()

#------------------------------------------------------------- Step

def STEP():
  DoTravel(kick)
  DoInteract(event)                          # event will be echo or trace

#------------------------------------------------------------- Color map

def krgb():
  if   q and     p: Red._ = 1
  elif q and not p: Grn._ = 1
  elif not q and p: Blu._ = 1

#------------------------------------------------------------- Initialize

UseCmap(krgb)

dist = [1,1]
def init():
  'Standard initial configuration'
  global event; event = echo
  Randomize(q=[1,1])
  DoInteract(fanout)
0;                                  BindKey("I",init)

def toggle_trace():
  global event
  if event==echo: event = trace
  else:           event = echo
0;                                  BindKey("T",toggle_trace)

init()
```