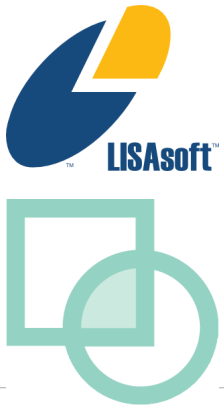
Jiffle

Una breve introduzione



Indice

Credits

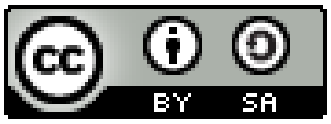
Da pagina 2 a pagina 15, questo manuale è frutto di una traduzione parziale della documentazione online di Jiffle, reperibile qui. Copyright e diritti intelettuali appartengono quindi a *Michael Bedward*.

La restante parte del manuale é stata scritta da:

- Riccardo Rigon (Università di Trento)
- Silvia Franceschi (Hydrologis and ISPRA)
- Leonardo Perathoner (Università di Trento)
- Andrea Antonello (Hydrologis)
- Giuseppe Formetta (Università di Trento)
- Aaron Iemma (Università di Trento)

La citazione corretta è: Rigon et al., Quaderni di Dipartimento, Dipartimento di Ingegneria Civile ed Ambientale, Università di Trento, 2012.

Per quanto possibile, questo manuale è stato redatto con l'uso di software libero, tra i quali LaTeX per La scrittura, grazie al quale Leonardo Perathoner ha approntato un apposito stile.



1 Introduzione

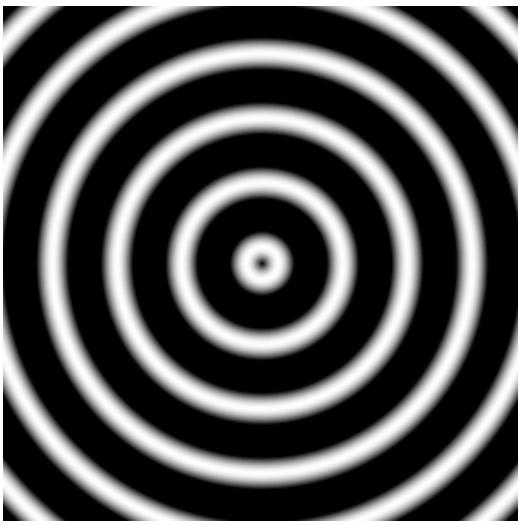
1.1. COSA E PERCHÈ

Jiffle è un semplice linguaggio di scripting in grado di lavorare con le immagini raster, ed è fatto per lasciarti scrivere più codice... con meno codice!

Per illustrare quel che si intende, proviamo a mettere a confronto un metodo Java per disegnare una semplice funzione matematica 3-D e uno script Jiffle equivalente. La funzione che useremo è:

$$z_{xy} = sin(8\pi D_{xy})$$

Dove D_{xy} è la distanza tra la posizione (x,y) di un pixel e il centro dell'immagine. La funzione produce un set di onde concentriche, emananti dal centro dell'immagine, o più poeticamente, delle *increspature sulla superficie illuminata di un immobile lago...*



..Di seguito il codice Java per disegnare la funzione: utilizza un iteratore JAI (Java Advanced Imaging) per settare i valori dei pixel. Ometteremo qui le dichiarazioni di importazione delle librerie JAI, concentrandoci esclusivamente sul metodo:

```
public void createRipplesImage(WritableRenderedImage destImg) {

        // dimensioni dell'immagine
        final int width = destImg.getWidth();
        final int height = destImg.getHeight();

        // coordinata del primo pixel
        int x = destImg.getMinX();
        int y = destImg.getMinY();

        // coordinata del pixel centrale
        final int xc = x + destImg.getWidth() / 2;
        final int yc = y + destImg.getHeight() / 2;

        // termini costanti
        double C = Math.PI * 8;

        WritableRectIter iter = RectIterFactory.
        createWritable(destImg, null);
```

```
        do {
            double dy = ((double) (y - yc)) / yc;
            do {
                double dx = ((double) (x - xc)) / xc;
                double d = Math.sqrt(dx * dx + dy * dy);
                iter.setSample(Math.sin(d * C));
                x++ ;
            } while (!iter.nextPixelDone());

            x = destImg.getMinX();
            y++;
            iter.startPixels();

        } while (!iter.nextLineDone());
    }
```

...Ed ora, lo script Jiffle equivalente:

```
init {
  // coordinate del centro dell'immagine
  xc = width() / 2;
  yc = height() / 2;

  // termini costanti
  C = M_PI * 8;
}

dx = (x() - xc) / xc;
dy = (y() - yc) / yc;
d = sqrt(dx*dx + dy*dy);

destImg = sin(C * d);
```

Confrontato con il metodo Java, lo script Jiffle:

- E' molto più corto;
- E' di più facile lettura, dato che l'algoritmo non è oscurato da robaccia fine a sè stessa;
- **Non** usa cicli!

Quest'ultima caratteristica è quella che più di tutte permette a Jiffle di essere tanto conciso. In Jiffle, non si scrive codice da iterare sull'immagine sorgente: piuttosto, si specifica come calcolare il valore di un `pixel individuale`, e il sistema di runtime da applicare a tutta l'immagine.

Ora, qualche lettore potrebbe gridare all' eresia dopo il confronto di cui sopra, dato che nonostante si sia presentato il codice Jiffle, non abbiamo mostrato il necessario codice Java per implementarlo al meglio. Ad essere onesti, non abbiamo mostrato le chiamate di codice e le dichiarazioni importanti nemmeno per il codice Java, ma nell'interesse dell'onestà, ecco un modo per far partire lo script Jiffle in un programma Java:

```
JiffleBuilder builder = new JiffleBuilder();

        // These chained methods read the script from a file,
        // create a new image for the output, and run the script
        builder.script(scriptFile).dest("destImg", 500, 500).run();

        RenderedImage result = builder.getImage("destImg");
```

1.2. CHE COSA NON SI PUÒ FARE CON JIFFLE

Jiffle è ancora un linguaggio piuttosto nuovo, e qualcuna delle capacità che speriamo di includere in futuro non sono ancora state inserite. Vi sono anche alcune limitazioni, imposte dalla natura del sistema di runtime di Jiffle.

- Le immagini di destinazione devono essere, come tipo di dato, dei `DOUBLE`

  (Java DataBuffer.TYPE_DOUBLE). Jiffle effettua tutti i calcoli usando valori a doppia precisione, incipendentemente dai tipi dell'immagine.

- Le immagini di destinazione possono includere una sola banda.

- Gli oggetti di runtime di Jiffle iterano attraverso il sorgente e le imamgini di destinazione tramite ordinate (`X`,colonne) e coordinate (`Y`, righe). Algoritmi che richiedono un differente ordine di iterazione, come il metodo ad angolo di diamante per generare superficie frattali

  (`http://www.gameprogrammer.com/fractal.html#diamond` saranno praticamente impossibili da implementare in Jiffle, o semplicemente richiederanno molto meno lavoro se eseguiti direttamente in Java.

2 Sintesi del linguaggio

2.1. STRUTTURA DI UNO SCRIPT JIFFLE

Uno script consta in un corpo, volendo preceduto da uno o più `blocchi speciali` che sono utilizzati per dichiarare variabili ed opzioni di controllo del runtime. Ritorneremo a questi blocchi più tardi, prima diamo un'occhiata alle proprietà generali del linguaggio.

2.2. COMMENTI

```
/*
 * blocchi di commenti
 * in stile C sono
 * pienamente supportati in Jiffle
 */

// anche come commenti in linea
```

2.3. VARIABILI

2.3.1. tipi e dichiarazioni delle variabili

Jiffle supporta i seguenti tipi di variabili:

Scalari	Singoli valori. In Jiffle, tutti i valori scalari corrispondono al tipo Java `double`
Array	Un array di valori scalari ridimensionato dinamicamente
Immagine	Un'immagine che rappresenta l'immagine sorgente o destinataria di uno script

Un supporto per degli array multidimensionali deve ancora essere aggiunto

Jiffle usa una dichiarazione pigra per le variabili scalari... Ovvero, è possibile iniziare subito ad utilizzare un nome di variabile nel corpo dello script. In questo snippet:

```
// I nomi di variabile non richiedono dichiarazioni precedenti
val = max(0, image1 - image2);
```

Tuttavia, variabili di tipo array devono esse dichiarate prima dell'uso, in modo che Jiffle possa distinguerle dagli scalari:

```
// dichiara un array vuoto
foo = [];

// dichiara un array con valori iniziali
bar = [1, 2, 42];
```

Al contrario di linguaggi come Ruby, non è possibile qui cambiare il tipo di una variabile all'interno di uno script:

```
// crea una variabile array
foo = [1, 2, 3];

// tentando di utilizzarla come scalare si otterra' un errore di compilazione
foo = 42;

// ..creare una variabile scalare bar e utilizzarla come
// fosse un tipo array fara' ugualmente arrabbiare il compilatore
// (<< e' l'operatore d'aggiunzione)
bar = 42;
bar << 43; // error
```

2.3.2. nomi

I nomi di variabili devono iniziare con una lettera, opzionalmente seguita da una combinazione di lettere, numeri, underscore e punti. Le lettere possono essere maiuscole o minuscole, considerando che i nomi di variabile sono `case-sensitive`.

2.3.3. ambito di visibilitá

Tutte le variabili scalari che appaiono per la prima volta nel corpo dello script hanno una `visibilità a pixel`: il loro valore è ignorato dopodichè ogni pixel di destinazione è processato. Variabili dichiarate nel blocco `init`, se presenti, hanno una `visibilità ad imamgine`: il loro valore persiste tra pixel e pixel.

```
init {
    // una variabile con visibilita' ad imamgine con valore iniziale
    foo = 0;
}

// Una variabile che appare per la prima volta nel corpo dello script
// ha visibilita' a pixel
bar = 0;
```

3 Operatori

3.1. OPERATORI ARITMETICI

Simbolo	Descrizione
	Elevamento a potenza
*	Moltiplicazione
/	Divisione
%	Resto della divisione
+	Somma
-	Sottrazione
=	Assegnamento
+=	Assegnamento additivo
-=	Assegnamento sottrattivo
=	Assegnamento moltiplicativo
/=	Assegnamento divisionale
%=	Assegnamento in modulo

3.2. OPERATORI LOGICI

Simbolo	Descrizione
&&	AND logico
||	OR logico
^	XOR logico
==	Test d'eguaglianza
!=	Test di disuguaglianza
>	Più grande di
>=	Più grande di o uguale a
<=	Minore di od uguale a
<	Minore di
!	Complemento logico

3.3. ESPRESSIONI TERNARIE

Un esempio:

```
// setta "foo" a 1 se "bar" >10; altrimenti a zero
foo = bar > 10 ? 1 : 0;
```

4 Controllo del flusso

4.1. CONTROLLI `IF-ELSE`

E' possibile utilizzare la familiare dichiarazione `if-else` all'interno di uno script Jiffle:

```
if (foo > 0) n++ ;

if (bar == 42) {
  result = 1;
} else {
  result = 0;
}
```

4.2. CICLI

Una delle particolarità di Jiffle che lo rendono conciso per creare script è la capacità di scrivere codice che non abbia bisogno di ciclare tra l'immagine sorgente e di destinazione, visto che è il sistema di runtime a farlo in automatico. Un sacco di vostri script non richiederanno alcuna dichiarazione di cicli. In ogni caso, Jiffle mette a disposizione i costrutti di ciclo, utili quando ad esempio si lavora con le vicinanze di qualche pixel, o per eseguire calcoli iterativi.

4.2.1. ciclo foreach

Probabilmente la maggior parte delle volte in cui si dovrà utilizzare un ciclo in uno script Jiffle, questo sarà un ciclo foreach. La forma generica è:

`foreach` (**var** in **elementi** `obiettivo`)

dove:

- `var` è una variabile scalare che sarà settata a turno per ogni valore di `elementi`;
- `elementi` è un array, o una sequenza (vedi sotto);
- `target` è una singola dichiarazione, o un blocco di codice, delimitato da parentesi graffe;

Il seguente esempio itera un ciclo sulle vicinanze di un pixel 3x3, e conta il numero di valori che sono più grandi di un certo valore di soglia. Utilizza una notazione a **sequenza**, che ha la forma **Valore_basso:Valore_alto**. Ogni variabile del ciclo è settata a turno a -1 o 1. Quindi, le variabili del ciclo sono utilizzate per accedere ad una `posizione relativa del pixel` nell'immagine sorgente.

```
n = 0;
foreach (dy in -1:1) {
    foreach (dx in -1:1) {
        n += srcimage[dx, dy] > someValue;
    }
}
```

Di seguito si presenta lo stesso esempio, ma stavolta usando un **array** per ogni ciclo:

```
delta = [-1, 0, 1];
n = 0;
foreach (dy in delta) {
    foreach (dx in delta) {
        n += srcimage[dx, dy] > someValue;
    }
}
```

4.2.2. ciclo while

Un ciclo condizionale che esegue la dichiarazione target o il blocco di codice, finchè la sua espressione condizionale è non-zero.

```
ynbr = y() - 500;
total = 0;
while (ynbr <= y() + 500) {
    xnbr = x() - 500;
    while (xnbr <= x() + 500) {
        total += srcimage[$xnbr, $ynbr];
        xnbr += 100;
    }
    ynbr += 100;
}
```

4.2.3. ciclo until

Un ciclo condizionale che esegue la dichiarazione target o il blocco di codice, finchè la sua espressione condizionale è non-zero.

```
ynbr = y() - 500;
total = 0;
until (ynbr > y() + 500) {
    xnbr = x() - 500;
    until (xnbr > x() + 500) {
        total += srcimage[$xnbr, $ynbr];
        xnbr += 100;
    }
    ynbr += 100;
}
```

4.3. ISTRUZIONI BREAK E BREAKIF

Jiffle mette a disposizione l'istruzoone **break** per uscire incondizionatamente da un ciclo:

```
n = 0;
foreach (i in 1:10) {
    if (foo[i] != null) {
        n++ ;
    } else {
        break;
    }
}
```

..Ed esiste pure l'istruzione **breakif**:

```
n = 0;
n = 0;
foreach (i in 1:10) {
    breakif(foo[i] == null);
    n++ ;
}
```

5 Funzioni

5.1. FUNZIONI NUMERICHE GENERICHE

Nome	Descrizione	Argomenti	Valori di ritorno	Note
`abs(x)`	Valore assoluto	valore double	valore assoluto di x	
`acos(x)`	Arcocoseno	valore nel range [-1,1]	angolo in radianti	
`asin(x)`	Arcoseno	valore nel range [-1,1]	angolo in radianti	
`atan(x)`	Arcotangente	valore nel range [-1,1]	angolo in radianti	
`cos(x)`	Coseno	angolo in radianti	coseno [-1,1]	
`degToRad(x)`	Gradi in radianti	angoli in radianti	angoli in gradi	
`exp(x)`	Esponenziale	Valore double	e^x	
`floor(x)`	Parte intera	Valore double	Parte intera di x come double	
`isinf(x)`	Test valore infinito	Valore double	1 se $x = \pm\inf$, 0 altrimenti	
`isnull(x)`	Test valore nullo	Valore double	1 se x è `null`, 0 altrimenti	
`log(x)`	Logaritmo naturale	Valore positivo	$log_e x$	
`log(x, b)`	Logaritmo	x: valore positivo; b: base	$log_b x$	
`radToDeg(x)`	Radianti in gradi	Angoli in radianti	Angoli in gradi	
`rand(x)`	Numeri pseudocasuali	Valore double	Valori in [0,x]	Funzione volatile
`randInt(x)`	Numeri pseudocasuali	Valore double	Parte intera di un valore nel range 0,x	= `floor(rand(x))`
`round(x)`	Arrotondamento	Valore double	valore arrotondato	
`round(x, n)`	Arrotonda ad un multiplo di n	x:double; n:numero intero	Valore arrotondato $\frac{x}{n}$	Ex: `round(44.5, 10)= 40`
`sin(x)`	Seno	Angolo in radianti	seno [-1,1]	
`sqrt(x)`	Radice quadrata	Valori non-negativi	$\sqrt{x}$	
`tan(x)`	Tangente	Angolo in radianti	valore double	

5.2. FUNZIONI LOGICHE

Nome	Descrizione	Argomenti	Valori di ritorno
`con(x)`	Condizionale	Valore double	1 se x è non-zero, zero altrimenti
`con(x, a)`	Condizionale	Valori double	a se x è non-zero, zero altrimenti
`con(x, a, b)`	Condizionale	Valori double	a se x è non-zero, b altrimenti
`con(x, a, b, c)`	Condizionale	Valori double	a se x è positivo, b se è zero, c se è negativo

5.3. FUNZIONI D'ANALISI STATISTICA

Nome	Descrizione	Argomenti	Valori di ritorno
`max(x, y)`	Massimo	Valori double	Massimo tra x e y
`max(ar)`	Massimo	Array	Massimo tra i valori dell'array
`mean(ar)`	Media	Array	Media dei valori dell'array
`min(x, y)`	Minimo	Valori double	Minimo tra x e y
`min(ar)`	Minimo	Valori double	Minimo tra i valori dell'array
`median(ar)`	Mediana	Array	Mediana dei valori dell'array
`mode(ar)`	Moda	Array	Moda dei valori dell'array
`range(ar)`	Range	Array	Range dei valori dell'array
`sdev(ar)`	Deviazione standard	Array	Deviazione standard campionaria dei valori dell'array
`sum(ar)`	Somma	Array	Somma dei valori dell'arrat
`variance(ar)`	Varianza	Array	Varianza campionaria dei valori dell'array

5.4. FUNZIONI DI PROCESSAMENTO DELLE AREE

Nome	Valori di ritorno
`height()`	Altezza dell'area processante
`width()`	Larghezza dell'area processante
`xmin()`	Minima ordinata X dell'area processante
`ymin()`	Minima coordinata Y dell'area processante
`xmax()`	Massima ordinata X dell'area processante
`ymax()`	Massima coordinata Y dell'area processante
`x()`	Ordinata X della regione processante corrente
`y()`	Ordinata Y della regione processante corrente
`xres()`	Altezza del pixel
`yres()`	Lunghezza del pixel

6 Blocchi speciali

6.1. IL BLOCCO OPZIONI

Questo blocco è usato per coordinare il ocmportamento della runtime di Jiffle. Fino ad ora, solo le opzioni *esterne* sono supportate.

Ad esempio, il seguente codice dirà a Jiffle di ritornare un valore zero per ogni richiesta dei valori di pixel che ricadono all'esterno dei contorni dell'immagine sorgente:

```
options {
    outside = 0;
}
```

Il seguente script ricava il massimo valore in un kernel 3x3, centrato su ogni immagine sorgente, e lo scrive sull'immagine di destinazione. Usa l'opzione esterna per trattare le locazioni del kernel oltre i contorni dell'immagine sorgente come valori `null` che saranno ignorati dalla funzione `max`.

```
// questo script implementa un filtro di massimo
// su un kernel di vicinanza 3x3

// setta le opzioni per trattare la location fuori
// dall'immagine sorgente come valori null
options { outside = null; }

foreach (dy in -1:1) {
  foreach (dx in -1:1) {
      values << src[dx, dy];
  }
}

dest = max(values);
```

Se l'opzione *outside* non venisse settata, qualsiasi richiesta di un valore all'esterno dei contorni dell'immagine causerà un errore di runtime nella compilazione di Jiffle.

6.2. IL BLOCCO IMMAGINE

E' utilizzato per associare variabili con una sorgente (sola lettura) e una destinazione (sola scrittura) in forma di immagine. Ad esempio:

```
images {
    foo = read;
    bar = read;
    result = write;
}
```

Come mostrato nel codice di cui sopra, il blocco contiene dichiarazioni nella forma `nome = (leggi | scrivi)`. Se questo blocco è assegnato, il compilatore Jiffle si aspetterà che contenga le dichiarazioni per ogni variabile immagine usata nello script. Se non è assegnato, i nomi di variabili possono essere definiti come rappresentanti sorgenti o destinazioni usando i metodi messi a disposizione dalle classi `Jiffle` e `JiffleBuilder`.

6.3. IL BLOCCO INIT

Questo blocco dichiara le variabili che avranno *visibilità all'interno dell'immagine* durante l'elaboraizone.

Ad ogni variabile potrà opzionalmente essere assegnato un valore iniziale, come nel caso di `foo` nell'esempio seguente:

```
init {
    foo = 42;
    bar;
}
```

Se un valore iniziale non è assegnato, almeno no deve essere *iniettato* durante il runtime.

7 Specificare la posizione dell'immagine sorgente

La posizione dei pixel e la banda dell'immagine sono specificate usando la notazione a parentesi quadra.

7.1. POSIZIONE ASSOLUTA DEI PIXEL

Le posizioni asoslute sono specificate usando il prefisso $, molto simile alla sintassi utilizzata in alcuni fogli di calcolo:

```
// esempio: accede ai valori a x=50 y=42
value = srcimage[ $50, $42 ];
```

Variabili ed espressioni possono comunque apparire tra le parentesi:

```
value = srcimage[ $xpos, $(min(width() - 1, y() + 10)) ];
```

7.2. POSIZIONE RELATIVA DEL PIXEL

Se non sono prefissati dei valori, questi sono trattati come offset relativi alla posizione computazionale corrente:

```
// esempio: accede ai valori a x+2, y-1
value = srcimage[ 2, -1 ];
```

Così come per pixel a posizione assoluta, variabili ed espressioni pososno comunque essere utilizzate:

```
value = srcimage[ dx, dy ];
```

7.3. SPECIFICARE LA BANDA

La banda dell'imamgine è specificata da un singolo valore, variabile o espressione tra parentesi quadre. E' sempre trattata come uno specificatore assoluto:

```
// ricava un valore dalla banda 2 dell'attuale posizione computazionale
value = srcimage[ 2 ];
```

Così come per la posizione a pixel, la banda può essere specificata usando una variabile o un'espressione:

7.4. SPECIFICARE CONTEMPORANEAMENTE PIXEL E BANDA DELL'IMMAGINE

Quando si specificano banda e posizione del pixel assieme, la banda va segnalata per prima:

```
// ricava il valore dalla banda uno, in posizione x=50, y=42
value = srcimage[ 1 ][ $50, $42 ]

// ricava il valore dalla banda 1 sull'offset dx=-1, dy=3
value = srcimage[ 1 ][ -1, 3 ]
```

8 Parole riservate

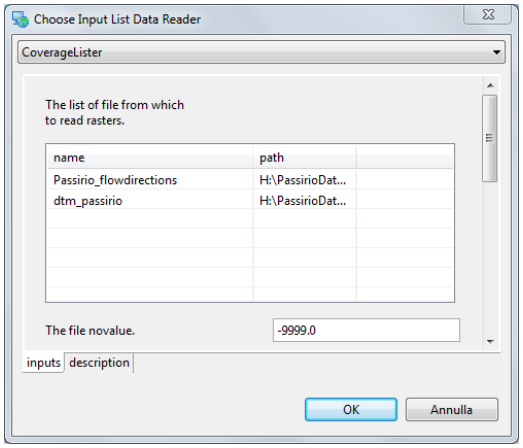
Le seguenti parole sono riservate all'interno del codice di Jiffle, e non dovrebbero mai essere usate come nomi di variabili:

- boolean
- break
- breakif
- con
- double
- else
- false
- float
- foreach
- if
- images
- in
- init
- int
- null
- options
- read
- true
- until
- while
- write

9 Esempo di utilizzo

Vediamo ora un esempio di integrazione del codice Jiffle all'interno del MapCalculator di uDig.

1. Settate a dovere le Spatial Toolbox per l'utilizzo dei moduli JGrass e OMS, preoccupiamoci prima di tutto di generare, nel caso in cui non l'avessimo ancora fatto, una mappa delle direzioni di deflusso che chiameremo "**Passirio_flowdirections**". Per farlo, utilizziamo il modulo "`Flowdirections`" dei JGrasstools. Vi si può accedere direttamente digitandone il nome nel campo di ricerca.
2. Fatto questo, spostiamoci sul modulo MapCalc dei tools.
3. Ora abilitiamo all'utilizzo le mappe "`dtm_passirio`" e "`Passirio_flowdirections`", trascinandole dalla vista piani nel campo *Coverage Lister* delle mappe di input.



4. A questo punto, inseriamo il seguente codice nel campo "Function to process" del MapCalc:

```
images{
  dtm_passirio=read;
  Passirio_flowdirections=read;
  out=write;
        }



  if(dtm_passirio>=800){
  out = Passirio_flowdirections < 5 ? null:Passirio_flowdirections;
                        }  else out=null;
```

   Miriamo con questa funzione ad estrarre in un sol colpo tutte le direzioni di deflusso che superino una certa quota (800 m) del nostro DTM, e che siano dirette solo verso Sud (direzioni 6,7 ed 8: si veda il manuale delle `Horton Machine` per ulteriori dettagli).

5. Un copia-incolla non funzionerà: dovrete riscriverla! E nel mentre, ragionare sui singoli blocchi:

   - *images*: al suo interno si scriveranno i nomi delle immagini da utilizzare come sorgenti (`=read;`), premurandosi che abbiano lo stesso nome di quelle fornite nel Coverage Lister, e il nome dell'immagine di output (`=write;`). Una volta inizializzatele in questo blocco, saranno utilizzabili come una qualsiasi variabile di tipo immagine;
   - *if()*: la sintassi dell'espressione è:

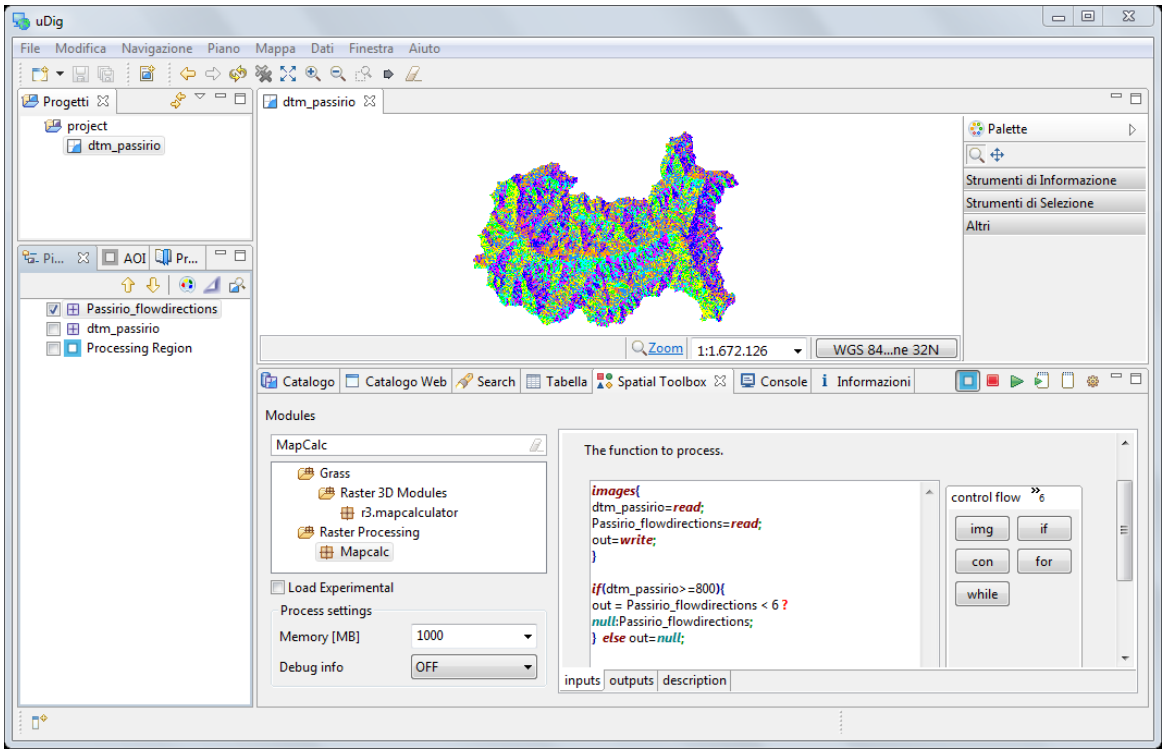
     `if`(se_vero) *operazioni1* **;** `else`(se_falso) *operazioni2* **;**

     Eventualmente, al posto del campo "*operazioni2*", è possibile annidare altri set d'espressioni, `if()` compresi, fino ad un livello arbitrario. Ricordiamo qui che l'indentazione in Jiffle (al contrario di altri derivati del Java come, ad esempio, `Python`) è indifferente... Non vale ovviamente lo stesso per la punteggiatura!

- *out = Passirio_...*: questa è un'espressione ternaria, la cui sintassi dovremmo ricordarci essere:

  verifica **?** *operazioni_se_vero* **:** *operazioni_se_falso*

6. Dovremmo quindi trovarci con un'immagine non troppo dissimile alla seguente:



7. Non scordiamoci di inserire il nome dell'immagine di output nel campo apposito ("*output*") delle Spatial Toolbox: chiamiamo la mappa "**Passirio_myflow**".

8. Lanciamo come al solito l'elaborazione, ricordandoci di imporre a MapCalc di lavorare sulla regione computazionale... E godiamoci il risultato.