LuaTEX + pdfliteral

Roberto Giacomelli

Sommario

\pdfliteral è un comando ben conosciuto del motore di composizione pdftex con quasi lo stesso significato del comando \special di TEX. Quello che si ottiene con queste primitive è l'inclusione nel documento finale di materiale grezzo che poi il driver potrà interpretare.

In questo articolo verrà esaminato da un punto di vista pratico come lo sviluppatore LuaTEX programmando esclusivamente in Lua la libreria node, può efficacemente inserire materiale pdfliteral per produrre semplice grafica.

Abstract

\pdfliteral is a well known command of the typesetting engine pdftex with nearly the same meaning of TEX's \special. What these commands do is to insert raw material in the final document that then the device drivers can later interpret.

In this paper it will be examined with a practice point of view how the LuaTEX developer can insert pdfliteral material programming exclusively in Lua by means of the node library to produce simple graphic figures.

1 Motivazione

In queste settimane sto lavorando a un nuovo pacchetto che per il momento terrò segreto. Una delle necessità funzionali di questo lavoro mi ha portato a interessarmi delle primitive del formato PDF per produrre con la massima efficienza possibile semplice grafica come linee o rettangoli.

Oltre a questo, per alcune semplici considerazioni, ho deciso di sviluppare il pacchetto in LuaTEX e perciò è nata la seguente domanda: programmando solamente in Lua esiste un modo di disegnare usando direttamente i codici nativi PDF?

La risposta è si e nelle prossime sezioni ne troverete una conferma. Nella prima parte dell'articolo illustrerò il modo in cui è possibile inserire grafica nei documenti TEX nonostante il compositore non ne sia in grado, sfruttando funzionalità esterne a esso. Nella seconda parte esaminerò da vicino il tipo di nodo pdf_literal di LuaTEX per creare dei semplici disegni con le specifiche del formato PDF. Nella terza e ultima parte applicativa creerò disegni al tratto programmando solamente in Lua all'interno di LuaTEX.

2 Risorse e riferimenti

In questa sezione indicherò i principali riferimenti per la comprensione del testo e alcune notizie per la compilazione degli esempi di codice riportati nel testo.

2.1 Requisiti per la comprensione

Per la comprensione del testo occorre un minimo di conoscenza del linguaggio Lua, che comprende la sintassi dei costrutti di base e i concetti principali per adoperare le tabelle, l'unico tipo di dato strutturato predefinito. Questa conoscenza si acquisisce in poche ore ma si avvisa il lettore che per essere produttivi, anche con Lua occorre apprendere il linguaggio in profondità, capirne le diverse componenti semantiche come le closure, le funzioni di prima classe, gli iteratori e le metatabelle, nonché conoscere almeno nei fondamenti il funzionamento interno dell'interprete — che è scritto in C — e le conseguenze della presenza del Garbage Collector.

Per fortuna la conoscenza si può acquisire per gradi e a seconda delle necessità. La migliore guida per Lua è il libro dell'Autore principale stesso del linguaggio, Roberto Ierusalimschy, denominato PIL acronimo di *Programming in Lua* (Ierusalimschy, 2013). Ormai questo testo sta diventando obbligatorio nella libreria di un buon sviluppatore TEX accanto al TEX book (Knuth, 1984) o al LATEX Companion (Mittelbach *et al.*, 2004). In effetti, visto che parleremo di grafica è più opportuno consultare il LATEX Graphics Companion (Goossens *et al.*, 2007).

Un'altra risorsa per documentarsi su Lua è il mio articolo che presentai al GUIT meeting 2012 a Napoli (Giacomelli, 2012). Nella prima parte di questo lavoro sono spiegati il tipo tabella e i princìpi della programmazione a oggetti in Lua.

Dal lato TEX, è necessario conoscere la differenza tra motore di composizione e formato, ovvero la differenza per esempio tra LuaTEX e LuaLATEX essendo gli esempi scritti in plain. Molto utile è poi conoscere i fondamenti dei registri box. Per questi importanti elementi è utile studiare gli Appunti di programmazione in LATEX e TEX di Enrico Gregorio (Gregorio, 2009) scaricando il documento dalla rete, oppure consultare il libro TEX by Topic (Eijkhout, 2014) che qualche tempo fa il GUIT inviò a tutti i propri soci.

2.2 Esecuzione del codice

Fondamentale, e quindi altamente consigliato, è compilare gli esempi riportati con la propria distri-

buzione TEX — preferibilmente riscrivendo i sorgenti per rendersi meglio conto dei concetti espressi dalla semantica del codice — e sperimentare gli effetti di modifiche o estensioni.

Tutti gli esempi di codice sono stati compilati con le versioni dei programmi e dei pacchetti inclusi nella TeX Live 2016 con il sistema operativo Ubuntu 14.04. Quest'ultimo dato dovrebbe essere ininfluente perché il codice TEX è multi-piattaforma.

Nei sorgenti ho inserito la *riga magica* per facilitare la compilazione con gli shell editor che offrono questa comoda funzionalità. Per esempio, nei listati per predisporre l'editor alla compilazione con LuaTEX si potrà trovare specificato:

```
% !TeX program = LuaTeX
```

Durante la compilazione è possibile incorrere in errori dovuti alle recenti modifiche dei nomi delle primitive in LuaTEX introdotte dalla versione 0.85 per rendere più razionale la separazione dei moduli interni. Il pacchetto luatex85 risolve queste incompatibilità almeno fino a quando non saranno aggiornate le varie librerie, come per esempio il pacchetto TikZ che evidentemente ricorre al suo interno alle primitive rinominate.

2.3 Gestione della memoria

Questo lavoro è dedicato alla tecnologia dei nodi di LuaTEX per realizzare grafica vettoriale in modo diretto con il linguaggio PDF. Trascurerò aspetti essenziali per le applicazioni reali come la gestione della memoria.

In Lua gli oggetti in memoria vengono automaticamente distrutti dal Garbage Collector quando non più necessari. Si tratta di un componente software che a runtime, ovvero durante l'esecuzione del programma, interviene sull'informazione in memoria non più indirizzabile.

Non tutti i tipi di dato sono soggetti alla gestione automatica della memoria. Tra questi ci sono proprio i nodi di LuaTEX, implementati come oggetti *userdata*. Se si perde il riferimento ai nodi la memoria che essi occupano diventa non raggiungibile causando un memory leak (mancata deallocazione di registri di memoria). Se si eliminano i nodi prematuramente si causerà la perdita di validità dei puntatori causando un segmentation fault (accesso alla memoria non valido).

In generale questi effetti disastrosi si evitano verificando molto attentamente il codice. D'altra parte si tratta di un argomento tutt'altro che secondario di cui soffre la programmazione in linguaggi di sistema come il C. Aggiungo per completezza che è possibile effettuare questi controlli automaticamente, per esempio il linguaggio Rust introduce alcuni concetti semantici che portano il compilatore a intercettare le situazioni di errore sulla memoria che possono condurre anche a problemi di sicurezza.

3 La grafica in TEX

In questa sezione presenterò diverse modalità con cui è possibile creare grafica — la possibilità di creare disegni specificando forme e coordinate attraverso un linguaggio — in TEX. Le entità geometriche vettoriali non possono essere elaborate da TEX perché esso non conosce alcuna primitiva grafica ma dà la possibilità di inserire nel documento qualsiasi codice si voglia con il comando `\special`.

All'apertura del documento composto il programma di lettura, che chiamerò *driver*, interpreterà queste porzioni di codice e ne produrrà il risultato a video o in stampa. Nel caso in cui il codice *speciale* non sia corretto potremo ricevere in apertura un errore di corruzione del file oppure effetti imprevedibili che renderebbero inutilizzabile il documento.

Per realizzare un disegno occorre perciò studiare attentamente il formato del documento in uscita, normalmente corrispondente al PDF, per iniettare il codice speciale conforme senza che TEX possa avvertirci di eventuali errori.

Ma è necessario fare in modo che il disegno non si sovrapponga agli altri elementi della pagina perché TEX della figura non conosce nulla perciò nemmeno le sue dimensioni. Queste informazioni, note come *bounding box*, devono essere calcolate e poi comunicate al motore di composizione perché lasci lo spazio corrispondente, per esempio creando una scatola orizzontale `hbox`.

Da notare che poiché si è legati alle funzionalità grafiche del driver esterno, il disegno ne avrà le stesse limitazioni. Per esempio, le specifiche PDF non comprendono il disegno di curve di secondo grado come i cerchi, così è necessario approssimare queste funzioni con curve di Bézier del terzo ordine, come spiega Claudio Beccari nel suo articolo uscito per il precedente GUITmeeting di Trento (Beccari, 2015), dal titolo "I calcoli matematici in `pdftex`".

3.1 Le primitive grafiche del formato PDF

Ma qual è il codice speciale secondo il formato PDF per disegnare una linea? Studiando il PDF Reference ufficiale (Adobe Systems Incorporated, 2006) che si può scaricare dal sito di Adobe, si scopre che la descrizione di una linea si basa sul concetto di *percorso*, una sequenza di punti nel piano definiti attraverso il sistema di coordinate (x, y).

Per disegnare una linea si da inizio a un percorso *muovendoci* nel primo punto con l'operazione `moveto`, proseguendo nel secondo punto con l'operazione `lineto` e infine disegnando il percorso appena costruito con l'operatore `stroke`.

La sintassi di questi operatori ricorda la notazione inversa polacca: prima si inseriscono gli argomenti e poi il nome dell'operatore tanto che

Tabella 1: Sintassi in notazione postfissa di alcuni operatori per la costruzione di grafica vettoriale nei documenti nel formato PDF.

Sintassi	Categoria/Descrizione
	Categoria stato grafico speciale:
`q`	Salva lo stato grafico corrente nello stack
`Q`	Ripristina lo stato grafico precedente nello stack
⟨*a*⟩ ⟨*b*⟩ ⟨*c*⟩ ⟨*d*⟩ ⟨*e*⟩ ⟨*f*⟩ `cm`	Modifica la matrice di trasformazione corrente concatenando quella specificata dai sei numeri degli operandi
	Categoria stato grafico generale:
⟨*dim*⟩ `w`	Imposta lo spessore di lineea nello stato grafico
⟨*code*⟩ `j`	Imposta lo stile con cui unire le linee del percorso nello stato grafico
⟨*code*⟩ `J`	Imposta lo stile con cui disegnare gli estremi delle linee di un percorso nello stato grafico
	Categoria costuzione percorsi:
⟨*x*⟩ ⟨*y*⟩ `m`	Inizia un nuovo tratto di percorso (un percorso è l'unione di più percorsi continui detti *sub-path*)
⟨*x*⟩ ⟨*y*⟩ `l`	Aggiunge al percorso una linea dal punto corrente al punto specificato che diviene poi il nuovo punto corrente del percorso
⟨*x*⟩ ⟨*y*⟩ ⟨*width*⟩ ⟨*height*⟩ `re`	Aggiunge al percorso un rettangolo come completo sub-path
`h`	Chiude il percorso corrente aggiungendo un segmento dal punto corrente al punto iniziale del sub-path
	Categoria tracciamento percorsi:
`S`	Disegna il percorso corrente

per essa il termine usato nel reference di Adobe è *notazione postfissa*. Il disegno di una linea inclinata dall'origine del sistema di riferimento al punto di coordinate $(32, 16)$ ha come codice speciale il seguente:

```
0   0 m
32 16 l
S
```

Non una singola istruzione ma ben tre in cui tutte le misure sono in punti grandi `bp`[1], e le operazioni descritte sono identificate da un codice letterale, i cui significati sono riportati nella tabella 1.

Queste codifiche derivano dal formato Postscript, un vero e proprio linguaggio basato sulla struttura dati dello stack in notazione postfissa. Un bell'articolo per chi volesse approfondire questo linguaggio si trova nel numero 7 di ArsTEXnica (Nisi, 2009) a opera di Riccardo Nisi.

3.2 Un esempio in `pdftex`

Prima di realizzare il disegno della linea appena descritta con il classico compositore `pdftex`, è utile precisare che in questo motore è disponibile il comando `\pdfliteral` da preferire rispetto a `\special` per la sintassi leggermente più semplice e per il nome più significativo. Possiamo leggerne la definizione nel manuale di `pdftex` (Thành, Hàn Thế *et al.*, 2016). Il comando accetta un argomento tra graffe che prima verrà espanso e poi inserito nel documento PDF inalterato.

Ecco il codice dell'esempio:

```
% !TeX program = pdftex
\pdfcompresslevel=0
\nopagenumbers
\pdfliteral {
0 0 m
32 16 l
S}
\bye
```

Poiché abbiamo richiesto che il file PDF non venga compresso dando un opportuno valore al comando `\pdfcompresslevel`, possiamo leggerlo con un editor di testi per rintracciare il nostro codice speciale per il disegno della linea, ecco il frammento d'interesse:

```
stream
1 0 0 1 72 769.89 cm
 0 0 m 32 16 l S

endstream
```

La riga speciale compare immutata al di sotto di una riga inserita dal programma che definisce la *matrice di trasformazione*, come evidenzia il codice `cm` a cui accennerò nella prossima sezione.

L'insieme delle proprietà come spessore di linea e colore fanno parte dello stato grafico. Non solo è possibile modificarne i valori ma se ne può creare una copia salvandola nello *stack* per poi ripristinarla successivamente.

Nel momento del ripristino, quei valori tornano al valore precedente. Si attiva così un ambiente locale all'interno del quale le modifiche ai parametri permangono convenientemente solo fino al suo termine, esattamente come succede per i gruppi in TEX.

1. Il bp è l'unità di misura *punto grande* del formato Postscript e vale 1/72 di pollice perciò circa 0,3528 mm mentre il pt punto tipografico vale 1/72,27 pollici.

Lo stato grafico si copia nello stack corrente con l'operatore `q` e può essere ripristinato con l'operatore `Q`. Modifichiamo il codice per il disegno della linea per lavorare in locale. Aggiungiamo anche l'impostazione dello spessore del tratto a 4 bp con l'operatore `w` e lo stile di chiusura sulla forma circolare con l'operatore `J` (proprietà definita *round cap* individuata con il valore 1):

```
% !TeX program = pdftex
\pdfcompresslevel=0
\nopagenumbers
\pdfliteral {
q
4 w
1 J
0 0 m
32 16 l
S
Q
}
\bye
```

3.3 La matrice di trasformazione

Per completezza di esposizione, accenno agli effetti della *matrice di trasformazione*. Si tratta di un operatore geometrico che applicato alle entità che seguono la definizione della matrice, ne modifica scala, posizione e rotazione.

La chiave dell'operatore di modifica della matrice è `cm`. Esso ha l'effetto di concatenare alla matrice di trasformazione dello stato grafico quella nuova definita da ben 6 parametri numerici. L'ordine di applicazione influisce sulla geometria del disegno finale.

I sei parametri indipendenti dati in ordine producono quattro diversi tipi di trasformazioni elementari:

- *traslazione* del sistema di riferimento specificata dai parametri $(1 \quad 0 \quad 0 \quad 1 \quad t_x \quad t_y)$, dove t_x e t_y sono le coordinate di traslazione dell'origine;
- *scalamento* del sistema di riferimento con i parametri $(s_x \quad 0 \quad 0 \quad s_x \quad 0 \quad 0)$, dove s_x e s_y sono i fattori di scala secondo i rispettivi assi;
- *rotazione* del sistema di riferimento con i parametri $(\cos\theta \quad \sin\theta \quad -\sin\theta \quad \cos\theta \quad 0 \quad 0)$ dove l'angolo θ è l'angolo di rotazione;
- *distorsione* del sistema di riferimento con i parametri $(1 \quad \tan\alpha \quad \tan\beta \quad 1 \quad 0 \quad 0)$ dove l'angolo α è la distorsione rispetto all'asse x e l'angolo β è la distorsione rispetto all'asse y.

Se osserviamo la riga dello stream mostrato nell'esempio precedente, si riconosce facilmente che il comando `\pdfliteral` ha inserito l'operatore `cm` con una matrice di trasformazione che trasla il disegno di 1 pollice in direzione x e di 271,60 mm in direzione dell'asse y, che è la distanza che si ottiene sottraendo dall'altezza della pagina di 297 mm il valore di 1 pollice.

Il disegno è stato inserito sulla pagina nel punto attivo in quel momento che, essendo il documento vuoto se non per il codice speciale, è il punto più in alto a sinistra della gabbia del testo, definita per default con margini di 1 pollice. Come già ricordato, il motore nella composizione dell'oggetto speciale non tiene in alcuna considerazione le dimensioni della figura trattandola come un'entità puntiforme.

Da notare che la traslazione è controllata anche dall'opzione del comando `\pdfliteral` che può essere `direct` o `page`. Nel caso venga specificata una di queste due opzioni infatti, non verrà inserita la matrice di trasformazione. Per i dettagli, si confronti ancora il manuale del compositore `pdftex` compilando qualche esempio di codice per valutare il comportamento del comando.

3.4 La libreria `mplib` di LuaTEX

La grafica non è nativa di TEX e nemmeno in LuaTEX, ma in quest'ultimo motore di composizione è disponibile la libreria `mplib` che altro non è che un interprete Metapost.

Con Metapost si ha a disposizione un potente motore per la grafica in grado di risolvere nativamente equazioni lineari, eseguire funzioni ricorsive, memorizzare figure, e molto altro ancora. Per un'introduzione si veda l'articolo di Klaus Höppner apparso su questa stessa rivista (Höppner, 2008) qualche anno fa.

Il funzionamento della libreria è semplice: si chiama il metodo `execute()` su un'istanza di `mplib`, fornendogli il codice Metapost. Se non ci sono errori otterremo una tabella Lua contenente la descrizione geometrica degli oggetti grafici (si consulti il manuale di LuaTEX per i dettagli (The LuaTEX development team, 2016)) e alcuni metodi per ricavarne la rappresentazione nei formati Postscript o SVG. L'output PDF invece può essere estratto per esempio con l'ausilio del pacchetto luamplib (Hagen *et al.*, 2016) che scorre le entità grafiche e le traduce in primitive PDF.

Con questo pacchetto disegnamo la solita linea inclinata di spessore 4 bp compilando con LuaTEX il seguente sorgente:

```
% !TeX program = LuaTeX
% LuaTeX >= 0.85
\pdfvariable compresslevel=0
\input luamplib.sty

\mplibcode
beginfig(0);
draw (0,0) -- (32,16)
     withpen pencircle scaled 4;
endfig;
\endmplibcode
\bye
```

Il pacchetto luamplib definisce l'ambiente in stile plain `\mplibcode` e `\endmplibcode`, e non solo traduce le entità grafiche nel formato PDF, ma nasconde tutti i dettagli delle chiamate in Lua e cura l'inserimento della figura nel documento con una scatola orizzontale dalle opportune dimensioni.

Con la stessa tecnica precedente, andiamo ora a leggere il frammento di stream contenuto nel file PDF non compresso con un editor di testo. La parte d'interesse è la seguente:

```
stream
1 0 0 1 74 751.89 cm
q
0.000 0.000 0.000 rg 0.000 0.000 0.000 RG
10.000000 M
1 j
1 J
4.000000 w
0.000000 0.000000 m
32.000000 16.000000 l
S
Q

endstream
```

Rispetto al codice manuale presentato nella sezione precedente abbiamo delle nuove linee di codice che agiscono localmente: le chiavi `rg` e `RG` impostano lo spazio colore e il colore stesso, la chiave `M` imposta la dimensione massima della larghezza di giunzione delle linee potenzialmente infinita quando l'angolo tra i segmenti vale 180°, e infine la chiave `j` seleziona il codice 1 che corrisponde allo stile di unione delle linee al modo *round join*. Nel nostro esempio, essendoci una sola linea lo stile di join non influisce sul disegno.

La matrice di trasformazione opera ancora una traslazione semplice nel punto corrente della pagina ma con uno spostamento orizzontale verso destra di 2 bp e uno spostamento verso il basso si 4 bp, dovuti allo stile circolare del disegno degli estremi e allo spessore della linea di 4 bp. Il punto d'inserimento dell'oggetto speciale puntiforme è quindi l'origine del suo sistema di riferimento.

La cosa importante è che anche l'uso del motore grafico interno `mplib` si traduce nelle stesse righe di stream secondo le specifiche PDF. Diverso è solamente il modo in cui queste linee di codice vengono prodotte, cioè attraverso il potente linguaggio grafico di Metapost.

3.5 Il pacchetto TikZ

Nel formato plain la stessa linea si può disegnare anche con il pacchetto PGF/TikZ con il seguente codice:

```
% !TeX program = pdftex
\pdfcompresslevel=0
\nopagenumbers
\input tikz.tex

\noindent\tikzpicture
    \draw[line width=4bp, line cap=round]
        (0,0) -- (32bp,16bp);
\endtikzpicture
\bye
```

Esaminando ancora una volta la sequenza di streaming contenuta nel file PDF ottenuto compilando con `pdftex` il sorgente precedente, si ottiene un risultato equivalente:

```
stream
1 0 0 1 74 751.89 cm
q
0 0 0 RG
0 0 0 rg
0.3985 w
q
q
4.00005 w
1 J
0.0 0.0 m
32.0004 16.0002 l
S
Q
Q
n
Q

endstream
```

Si noti come la precisione delle coordinate numeriche di `pgf` è leggermente inferiore rispetto a quella della libreria interna `mplib` accoppiata con il pacchetto luamplib per la conversione della geometria nello stream PDF.

Tuttavia le linee di codice speciale sono praticamente le stesse. In più compare solo l'operatore `n` che prescrive che i percorsi siano terminati senza che siano disegnati, ma il punto in cui compare si trova al di fuori del disegno, come mostrano gli operatori di copia/ripristino dello stato grafico. Interessante notare come `q` e `Q` siano annidati tre volte, probabilmente perché si tratta di uno schema fisso effettivamente utile solo nel momento in cui l'utente richiede funzionalità di disegno particolari.

4 LuaTEX e i nodi `pdfliteral`

I *nodi* programmati in Lua sono elementi tipografici completi che possono essere immessi nelle liste principali orizzontale, verticale o matematica durante la composizione del documento per essere posizionati sulla pagina.

Essi possono essere concatenati in liste a rappresentare un intero capoverso oppure possono essere impacchettati in scatole orizzontali o verticali.

Si ottiene lo stesso risultato tipografico che con le usuali primitive del motore di composizione, ma la differenza sostanziale la fa il fatto che i nodi sono programmati in Lua e ciò amplia notevolmente le possibilità applicative del sistema.

4.1 Un esempio di codice

Per mostrare l'equivalenza della tecnologia dei nodi con i comandi TEX classici, e come utile riferimento, scegliamo un primo esempio semplice di una scatola orizzontale di larghezza 64 pt contenente tre lettere distanziate in modo uguale. Con TEX si realizza questa struttura con il codice:

```
\hbox to 64pt{A\hfil B\hfil C}
```

con il risultato seguente:

A B C

La stessa costruzione può essere ottenuta programmando esclusivamente in Lua. Una volta attivato il *modo Lua* con la primitiva `\directlua`, si procederà in ordine con il:

- creare i tre nodi di tipo `glyph` per le lettere A, B e C;
- creare i due nodi di tipo `glue` per gli spazi allungabili;
- concatenare tutti i nodi in una lista nella corretta sequenza;
- impacchettare la lista in una scatola di tipo orizzontale dall'esatta larghezza;
- inviare la scatola alla lista principale del compositore.

I passi da compiere sono implementati nel seguente listato completo e pronto per essere compilato da LuaTEX, che riprodurrà l'esempio delle tre lettere:

```
% !TeX program = LuaTeX
\directlua{
function glyph(c)
    local n = node.new("glyph")
    n.char = string.byte(c)
    n.font = font.current()
    return n
end

function glue()
    local n = node.new("glue")
    n.stretch = 32*65536
    return n
end
}

\leavevmode\directlua{
local a = glyph "A" --[=[ creaz nodi --]=]
local b = glyph "B"
local c = glyph "C"
local g1, g2 = glue(), glue()

--[=[ concatenazione lista --]=]
local head = node.insert_after(a, a, g1)
head = node.insert_after(head, g1, b)
head = node.insert_after(head, b, g2)
head = node.insert_after(head, g2, c)

--[=[ impacchettamento lista nel box --]=]
local hbox = node.hpack(
    head, tex.sp "64pt", "exactly"
)

--[=[ invio box alla main list --]=]
node.write(hbox)
}
\bye
```

In Lua con la tecnologia dei nodi, i processi di lettura dei token, d'espansione, di esecuzione, non ci sono. Si passa subito all'ultima fase del processo visuale — si tratta della modalità in quattro tempi con cui opera TEX — perché i nodi elementari sono assemblati in strutture tipografiche già pronte per il posizionamento sulla pagina.

Con piccole modifiche al codice, è possibile memorizzare la scatola in un registro che poi verrà inserito nel documento con gli usuali comandi `\box` per usare e svuotare il registro oppure `\copy` per usare ma mantenere inalterato il contenuto del registro:

```
% !TeX program = LuaTeX

% a new box register
\newbox\mybox

\directlua{
--[==[ as before ... --]==]

--[==[ packing the list in a box --]==]
local hbox = node.hpack(
    head, tex.sp "64pt", "exactly"
)

--[==[ saving list in a box register --]==]
tex.box["mybox"] = hbox
}

\box\mybox
\bye
```

Nei listati precedenti vi sono molti dettagli che non spiegerò. Rimando il lettore al mio precedente articolo introduttivo sulla tecnologia dei nodi (Giacomelli, 2016).

4.2 Nodi grafici `pdf_literal`

Il nodo `pdf_literal` è un sottotipo del nodo `whatsits` che raggruppa molte funzionalità dirette verso il formato PDF. Esso accetta solamente tre parametri: una lista di attributi, il numero che imposta il `mode` che corrisponde alle opzioni possibili per la primitiva `\pdfliteral` e il campo stringa `data` a cui assegnare lo stream.

Questo è tutto quello che è necessario sapere per disegnare la linea inclinata presa come semplice prototipo:

```
% !TeX program = LuaTeX
% LuaTeX >= 0.85
\pdfvariable compresslevel=0
```

FIGURA 1: Risultato del codice LuaTEX riportato nel testo in cui il codice Lua per l'inserimento diretto di un nodo di tipo `pdf_literal` è inserito tra due parole. Per maggiore chiarezza sono state aggiunte due linee tratteggiate sugli assi coordinati e l'ingrandimento rispetto alle dimensioni originali.

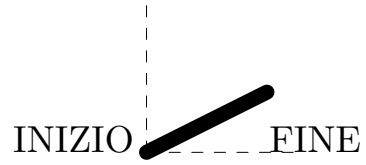


FIGURA 2: Risultato del codice LuaTEX riportato nel testo in cui la figura è inserita all'interno di una scatola orizzontale delle dimensioni corrispondenti a quelle del bounding box del disegno. Per maggiore chiarezza sono state aggiunte due linee tratteggiate sugli assi coordinati e un ingrandimento rispetto alle dimensioni originali.

```
\nopagenumbers

\directlua{
local n = node.new("whatsit", "pdf_literal")
n.mode = 0
n.data = [[q 4.0 w 1 J 0 0 m 32 16 l S Q]]
node.write(n)
}
\bye
```

Se si compila con LuaTEX l'esempio minimo, tutto funziona correttamente: la scrittura del codice non ha riservato sorprese: si ottiene una pagina vuota tranne che per la linea posizionata in alto a sinistra. Caricando il file PDF in un editor di testi, possiamo verificare che effettivamente lo stream compare immutato nel documento finale e che al solito è stata inserita automaticamente la matrice di trasformazione per traslare la figura nel punto d'inserimento più alto e a sinistra della gabbia del testo.

Inserendo del testo prima e dopo il comando `\directlua` nel codice precedente, per esempio le parole INIZIO e FINE, il risultato è quello riportato nella figura 1. L'oggetto grafico è stato posizionato nel punto d'inserimento sulla pagina subito dopo la lettera O della parola INIZIO sulla linea di base e con dimensioni nulle. La linea appare sovrapposta al testo che non si accorge minimamente della presenza dell'oggetto assumendo la stessa esatta posizione che avrebbe assunto in assenza del nodo `pdfliteral`.

4.3 Creare spazio per la figura

Una possibile soluzione per far si che il compositore mantenga sulla pagina lo spazio necessario alla figura, consiste nell'inserire una scatola orizzontale vuota delle dimensioni uguali a quelle della figura con il codice:

```
\raise <ydim> \hbox to <xdim> {}
```

Nel caso della linea d'esempio, considerando la chiusura di forma tonda delle estremità di raggio 2 bp, le dimensioni della figura sono $x_{\mathrm{dim}} = 36\,\mathrm{bp}$ e $y_{\mathrm{dim}} = 20\,\mathrm{bp}$, dunque:

```
% as before ...
INIZIO%
\raise 20bp \hbox to 36bp {}%
\directlua{
local n = node.new("whatsit", "pdf_literal")
n.mode = 0
n.data = [[q 4.0 w 1 J 0 0 m 32 16 l S Q]]
node.write(n)
}FINE
\bye
```

Ovviamente questa soluzione non funziona. La scatola non si sovrappone alla figura, semplicemente perché il punto d'inserimento dopo l'`hbox` si sarà spostato a destra sulla linea base di 36 bp ed è in quel punto che viene inserito il nodo `n`. Utile esercizio per il lettore è verificare l'effetto dell'aggiunta di uno spazio negativo con `\hskip -36bp`.

Non ci rimane che inserire la figura all'interno della scatola nell'angolo in basso a sinistra con il seguente codice:

```
% as before ...
INIZIO%
\hbox to 36bp {%
\vbox to 20 bp {}
\directlua{
local n = node.new("whatsit", "pdf_literal")
n.mode = 0
n.data = [[q 4.0 w 1 J 0 0 m 32 16 l S Q]]
node.write(n)
}%
\hfill}FINE
```

Il risultato di questo secondo tentativo riportato nella figura 2 sembra sia corretto ma osservando attentamente con l'aiuto dei tratteggi aggiunti al disegno, si nota che la linea risulta di poco più bassa rispetto alla linea base. Il motivo è che il nodo viene inserito rispetto all'origine del sistema di riferimento della figura. Nel nostro caso la linea ha uno spessore di 4 bp con estremità arrotondate e perciò *riempie* un'area al di sotto dell'asse delle ascisse con un punto più basso che scende di 2 bp.

La soluzione adottatata dal pacchetto `luamplib` è quella di creare una scatola orizzontale in quattro successivi passaggi in TEX. Semplificando il codice del pacchetto si legge:

```
\newbox\mpbox
\hbox{
  % set some dimensions
  ...
  \setbox\mpbox\vbox{% passaggio 1
```

```
  \noindent
  % insert here graphic material
  ...
  }
  \setbox\mpbox\hbox % passaggio 2
    {\hskip-\MPllx bp%
     \raise-\MPlly bp%
     \box\mpbox
  }%
  % passaggio 3
  \setbox\mpbox\vbox to \MPheight {
     \vfill
     \hsize\MPwidth
     \wd\mpbox0pt%
     \ht\mpbox0pt%
     \dp\mpbox0pt%
     \box\mpbox
  }%
  % passaggio 4
  \wd\mpbox\MPwidth
  \ht\mpbox\MPheight
  \box\mpbox
}
```

Nel codice riportato, al passaggio 2, si riconosce facilmente la correzione per tener conto di un bounding box che può essere al di sopra o al di sotto dell'asse x oppure a sinistra o a destra dell'asse y: le dimensioni `\MPllx` e `\MPlly` sono infatti le coordinate con segno dell'angolo in basso a sinistra del bounding box. Le distanze `\MPwidth` e `\MPheight` sono invece le dimensioni assolute del rettangolo che circoscrive la figura.

La costruzione di una scatola orizzontale per il corretto posizionamento della figura non è così semplice essendo necessario spostare l'origine del sistema di riferimento del disegno per far coincidere il bounding box ai bordi della scatola stessa.

4.4 Bounding box con i nodi

Quello che nella sezione precedente ho individuato come passaggio 2 si può realizzare anche con i nodi. Illustrerò tra un momento due diverse soluzioni che prevedono entrambe una scatola dentro l'altra, diversamente dal codice di `luamplib` dove a ogni passaggio la scatola viene per così dire *espansa* in se stessa.

In generale, se (P_0, P_1) sono i punti nel piano che definiscono i vertici rispettivamente inferiore/sinistro e superiore/destro del rettangolo del bounding box della figura, la traslazione rispetto all'angolo della scatola orizzontale dovrà essere di $-x_0$ e $-y_0$, mentre la sua larghezza dovrà essere $w = x_1 - x_0$ e la sua altezza $h = y_1 - y_0$.

4.4.1 Prima soluzione

L'idea è di effettuare prima lo spostamento verticale dell'oggetto speciale puntiforme all'interno di una scatola `vbox` usando una lunghezza rigida, e poi fare la stessa cosa ma in una scatola orizzontale a cui infine sono imposte le dimensioni uguali a quelle del bounding box.

Per mostrare solamente il codice Lua, che compie queste operazioni, scriverò una funzione che chiamerò `pack()`. Gli argomenti necessari sono il nodo `n` con il materiale speciale, e le quattro coordinate del bounding box rispetto al sistema di riferimento della figura:

```
function pack(n, x0, y0, x1, y1)
    local vskip = node.new("glue")
    vskip.width = -y0
    local vhead = node.insert_after(
        n, n, vskip
    )
    local vbox = node.vpack(vhead)

    local hskip = node.new("glue")
    hskip.width = -x0
    local hhead = node.insert_after(
        hskip, hskip, vbox
    )
    local hbox = node.hpack(hhead)

    hbox.width = x1 - x0
    hbox.heigth = y1 - y0
    return hbox
end
```

4.4.2 Seconda soluzione

La seconda soluzione si basa sul campo `shift` dei nodi `hlist`. Si effettua per prima cosa lo spostamento orizzontale con la solita lunghezza rigida e poi si impongono alla scatola orizzontale lo shift verticale e le dimensioni:

```
function pack(n, x0, y0, x1, y1)
    local hskip = node.new("glue")
    hskip.width = -x0
    local hhead = node.insert_after(
        hskip, hskip, n
    )
    local hbox = node.hpack(hhead)
        hbox.shift = y0

    local hbox2 = node.hpack(hbox)
    hbox2.width = x1 - x0
    hbox2.height = y1 - y0
    return hbox2
end
```

Rispetto alla precedente, questa soluzione ha il vantaggio di creare una sola grandezza elastica. Da notare che il campo `shift` non avrebbe effetto se la scatola non fosse inserita in una ulteriore scatola. Questo comportamento non è proprio del tutto coerente, e le future versioni di LuaTEX potrebbero apportare modifiche.

4.4.3 Test finale

Il risultato della prova finale è mostrato in figura 3. È prodotto con il seguente sorgente in cui va aggiunto il corpo della funzione `pack()` con una delle due precedenti soluzioni:

```
% !TeX program = LuaTeX
% LuaTeX >= 0.85
```

FIGURA 3: Il risultato del test finale del metodo per impacchettare all'interno di una scatola orizzontale dalle corrette dimensioni, il semplice disegno di una linea inclinata a cui è stato aggiunto per utile riferimento un rettangolo dal bordo sottile e tratteggiato. La figura è ingrandita con la stessa scala di quelle anologhe precedenti.

```
\pdfvariable compresslevel=0
\nopagenumbers
\newbox\mybox

\directlua{
function pack(n, x0, y0, x1, y1)
--[==[ as before --]==]
end
}

\directlua{
local n = node.new("whatsit", "pdf_literal")
n.mode = 0
n.data = [[
q
4.0 w 1 J 0 0 m 32 16 l S
0.125 w
[3] 0 d
-2 -2 36 20 re
S
Q
]]

local x0, y0 = tex.sp "-2bp", tex.sp "-2bp"
local x1, y1 = tex.sp "34bp", tex.sp "18bp"
tex.box["mybox"] = pack(n, x0, y0, x1, y1)
}
INIZIO\box\mybox FINE
\bye
```

Il fatto sorprendente è che i due file PDF ottenuti con le due diverse soluzioni della funzione `pack()` risultano essere *identici*.

Un altro fatto interessante, è che se affianchiamo il disegno più di una volta come nel codice:

```
IZ\copy\mybox\copy\mybox\box\mybox FN
```

lo stream che otterremo nel file PDF sarà il seguente[2]:

```
stream
BT
/F1 9.9 Tf 1 0 0 1 91.9 749.8 Tm [(IZ)]TJ
ET
1 0 0 1 126.027 751.89 cm
 q 4.0 w 1 J 0 0 m 32 16 l S Q
1 0 0 1 36 0 cm
 q 4.0 w 1 J 0 0 m 32 16 l S Q
1 0 0 1 36 0 cm
 q 4.0 w 1 J 0 0 m 32 16 l S Q
1 0 0 1 -198.027 -751.89 cm
BT
/F1 9.9 Tf 1 0 0 1 232.0 749.8 Tm [(FN)]TJ
ET

endstream
```

dove compare dopo ogni figura la matrice di trasformazione che rappresenta la traslazione verso destra di 36 bp, la larghezza del disegno, che ogni volta si somma alla precedente.

Nella prossima sezione mi dedicherò ad alcuni esempi applicativi programmando i disegni solo in Lua per produrre finalmente qualcosa di più interessante rispetto alla linea inclinata utilizzata come esempio per tutta questa sezione.

4.5 Disegno misto con TikZ

Una volta che il disegno realizzato in Lua con lo stream nel formato PDF si trova all'interno di un registro `box`, è possibile inserirlo in una precisa posizione all'interno di una figura in TikZ?

Traducendo in codice plain, la domanda appena posta è la seguente:

```
% !TeX program = LuaTeX
\input tikz.tex

% define \mybox as before...

\tikzpicture
% draw a control grid
\draw[step=2bp,help lines]
    (0,0) grid (50bp,42bp);
% draw the control red rectangle
\draw[red, line width= 1.08pt]
    (12bp, 20bp) rectangle (48bp, 40bp);
% draw the 'external' box
\node at (30bp, 30bp) {\box\mybox};
\endtikzpicture
```

Compilando si ottiene l'esatto posizionamento all'interno del rettangolo rosso della linea inclinata che lo ricordo ha dimensioni esterne di 36 bp per 20 bp. Essa viene posizionata nel nodo con punto d'ancoraggio il centro esattamente come previsto — se lo si desidera si possono usare diversi punti d'ancoraggio usando l'opzione `anchor`.

Questa tecnica funziona e rende possibile fare grafica mista con parti di figura realizzate in TikZ e altre parti realizzate in Lua.

5 Disegni in libertà

Ora ho davanti come un foglio bianco avendo a fianco la matita. Sfoglio qualche libro e trovo quasi subito qualcosa da disegnare...

5.1 Cornici 1

Il primo disegno è riportato in figura 4. Si tratta di una serie di quadrati concentrici di spessore degradante verso l'esterno. Per realizzarla ho scritto

2. Le righe sono state accorciate solo un po' senza alterarne il significato per rimanere nella giustezza della colonna eliminando qualche decimale e il disegno del rettangolo tratteggiato di contorno della linea.

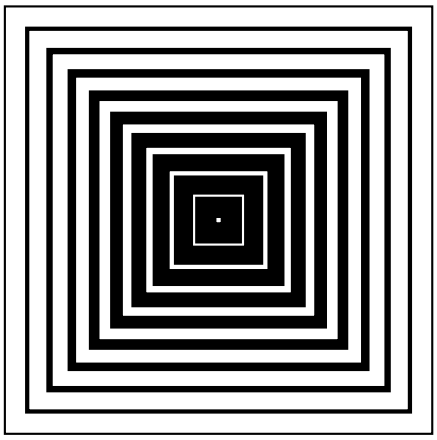


Figura 4: Una serie di quadrati concentrici di diverso spessore e misura del lato ottenuto con la tecnologia dei nodi con le specifiche grafiche del formato PDF. Lo spessore del tratto degrada dal centro verso l'esterno mentre al contrario per la distanza tra i quadrati.

alcune funzioni Lua di servizio che operano tutte su una tabella/array sempre passata come primo argomento. Ogni funzione aggiunge una stringa di testo in sequenza che contiene le istruzioni speciali `pdfliteral`. Per esempio la funzione per disegnare un rettangolo è la seguente:

```
plib.rect = function(fig, x0, y0, x1, y1)
    local t = fig.data
    local fmt = "%0.6f %0.6f %0.6f %0.6f re"
    t[#t+1] = string.format(fmt,
                x0, y0, x1-x0, y1-y0
    )
    local w = fig.width/2
    if not fig.bb then
        fig.bb = {x0-w, y0-w, x1+w, y1+w}
    else
        local xmin = fig.bb[1]
        local ymin = fig.bb[2]
        local xmax = fig.bb[3]
        local ymax = fig.bb[4]
        if x0 - w < xmin then
            fig.bb[1] = x0-w
        end
        if y0 - w < ymin then
            fig.bb[2] = y0-w
        end
        if x1 + w > xmax then
            fig.bb[3] = x1+w
        end
        if y1 + w > ymax then
            fig.bb[4] = y1+w
        end
    end
end
```

La maggior parte del codice è impegnata ad aggiornare il bounding box della figura, indispensabile per il posizionamento sulla pagina, secondo la logica seguente: ogni volta che si disegna un elemento, poiché ne conosco la geometria, posso verificarne i limiti d'ingombro. A scapito della generalità ma a vantaggio delle prestazioni, si potrebbe invece impostare il bounding box secondo la logica: se conosci la figura che stai disegnando allora ne conosci anche le dimesioni.

Questa seconda logica è quella adottata dal pacchetto pstricks (Zandt, 2003) che richiede all'utente le coordinate dei confini della figura come argomento obbligatorio dell'ambiente pspicture. Il primo approccio invece, è quello adottato dal pacchetto TikZ.

Riporto il codice di una seconda funzione, quella che restituisce la stringa da asssegnare al campo `data` del nodo, perché non utilizza la concatenazione di stringhe ma la funzione `table.concat()` della libreria standard di Lua, molto più efficiente nell'assemblare un insieme di stringhe:

```
plib.stream = function (fig)
    local t = fig.data
    return table.concat(t, "\n")
end
```

Con le funzioni ausiliarie raggruppate nella tabella chiamata `plib`, la parte più significativa del codice che produce la figura dei quadrati è la seguente:

```
\directlua{
local fig = plib.newfig()

local i = 1
local s = 10
local k = 1

while s > 0 do
    plib.penwidth(fig, s)
    local m = i + s/2
    plib.rect(fig, -m, -m, m, m)
    plib.stroke(fig)
    i = i + s + k
    s = s - 1
    k = k + 1
end
plib.closefig(fig)

local n = node.new("whatsit", "pdf_literal")
n.mode = 0
n.data = plib.stream(fig)
tex.box["mybox"] = pack(n, plib.bb(fig))
}
```

Dal lato tecnico delle specifiche PDF, è importante osservare *quando* nel codice avviene la chiamata alla funzione `plib.stroke()` — che non fa altro che aggiungere la `S`. Siccome lo spessore del tratto cambia per ogni quadrato, è necessario inserire l'operatore `stroke` dopo ogni inserimento dell'operatore `re` per il disegno del rettangolo. Se non fosse così, per esempio se inserissimo una `S` solamente alla fine, i percorsi verrebbero si disegnati ma tutti con l'ultimo spessore di linea inserito.

Questo significa che l'operatore `w` per lo spessore di linea non fa altro che scrivere quel parame-

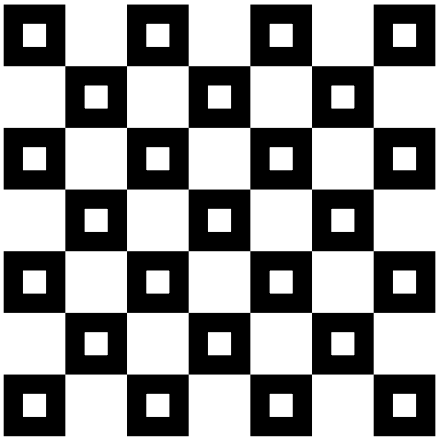


FIGURA 5: Secondo esempio applicativo, disegno di una scacchiera di rettangoli.

tro nello stack corrente sovrascrivendo il valore precedente.

5.2 Cornice 2

La parte più interessante del disegno di figura 5 è realizzare l'alternanza tra celle piene e celle vuote. Una possibilità è usare una variabile booleana che cambia di stato a ogni iterazione all'interno di un doppio ciclo `for`, uno esterno per le righe e uno interno per le colonne. Questo metodo funziona solamente se la dimensione della scacchiera è un numero dispari:

```
\directlua{
local row = 6
local isfull = true

local s = 2.5
local h = 16

local fig = plib.newfig()
plib.penwidth(fig, 2*s)

for i=0,row do
    local y0 = i * h + s
    local y1 = (i+1) * h - s
    for j=0,row do
        if isfull then
            local x0 = j * h + s
            local x1 = (j+1) * h - s
            plib.rect(fig, x0, y0, x1, y1)
        end
        isfull = not isfull
     end
end

plib.stroke(fig)
plib.closefig(fig)

local n = node.new("whatsit", "pdf_literal")
n.mode = 0
n.data = plib.stream(fig)
tex.box["mybox"] = pack(n, plib.bb(fig))
}
```

Ma le cose si possono vedere da un diverso punto di vista, quello che considera il disegno non come una scacchiera ma come la somma di griglie traslate una rispetto all'altra. Per cominciare scriverò la funzione `grid()` che disegna oggetti su una griglia regolare ricevendo tra gli argomenti una funzione che per condizione accetta le coordinate di un punto per utilizzarlo come centro di una figura.

In Lua le funzioni sono oggetti di prima classe, possono cioé essere memorizzate in variabili, una particolarità molto potente, ma non esiste il concetto d'*interfaccia* ovvero un modo per imporre che la funzione abbia una ben precisa firma[3]. Del resto Lua non è un linguaggio statico dove il codice è compilato in un file eseguibile e i tipi devono essere stabiliti in quella fase, mentre non è vietato passare a una funzione un numero e subito dopo richiamare la stessa funzione passando però una stringa.

Assumiamo che la griglia sia quadrata, ovvero che la distanza tra le righe sia la stessa che quella tra le colonne e valga `step`, che il numero delle righe sia `nx` e quello delle colonne sia `ny`, e che il primo nodo in basso a sinistra abbia coordinate `x0`, `y0`:

```
local
function grid(drawfun, nx, ny, step, x0, y0)
    for r = 0, nx-1 do
        local x = x0 + r*step
        for c = 0, ny-1 do
            local y = y0 + c*step
            drawfun(x, y)
        end
    end
end
```

Per semplicità ho eliminato il parametro `fig` intendendo che l'oggetto sarà già stato creato nel momento in cui verranno chiamate le funzioni che risolveranno il simbolo con la tecnologia delle *closure*. Chiamerò la funzione di disegno del nodo `square(x, y)`. Essa dovrà rispettare le regole d'interfaccia per le funzioni da passare a `grid()`:

```
local fig = plib.newfig()
local s = 3.65
local h = 16
local function square(x, y)
    local a = (h - s)/2
    local x0, y0 = x - a, y - a
    local x1, y1 = x + a, y + a
    plib.rect(fig, x0, y0, x1, y1)
end
```

In definitiva, il codice per il disegno della scacchiera con una struttuta del tutto generale, può anche essere il seguente:

```
\directlua{
local
```

3. La firma di una funzione non è altro che l'informazione di quali argomenti essa riceve e di quali valori essa restituisce.

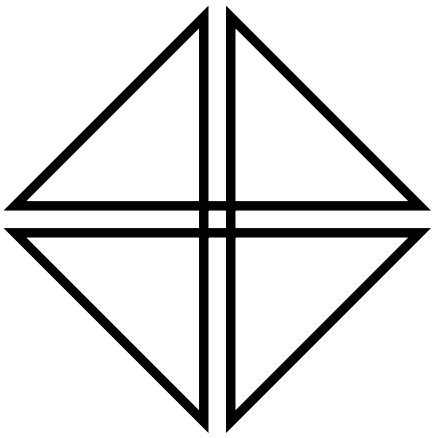


Figura 6: Disegno tracciabile senza mai staccare la penna dal foglio e quindi contenente un'unica linea che ritorna al punto di partenza è perciò chiusa, realizzato con il codice Lua riportato nel testo.

```
function grid(drawfun, nx, ny, step, x0, y0)
    --[==[ as before ]==]
end

local fig = plib.newfig()
local s = 5
local h = 16
local function square(x, y)
    --[==[ as before ]==]
end

plib.penwidth(fig, s)
grid(square, 4, 4, 2*h, 0, 0)
grid(square, 3, 3, 2*h, h, h)

plib.stroke(fig)
plib.closefig(fig)

local n = node.new("whatsit", "pdf_literal")
n.mode = 0
n.data = plib.stream(fig)
tex.box["mybox"] = pack(n, plib.bb(fig))
}
```

5.3 Linee 1

Passando ai disegni al tratto, ovvero contenenti solamente linee, ho riprodotto la spezzata di figura 6. Un buon esercizio perché per prima cosa ho imparato che non è sempre così facile determinare il bounding box. Se osserviamo la giunzione delle linee inclinate, a causa dello stile che prolunga i bordi dei tratti dotati di spessore w, i limiti si estendono oltre il punto d'intersezione degli assi di:

$$\frac{1}{2\tan\pi/8}w = 1{,}207w$$

Seconda osservazione: la figura può essere disegnata con l'idea di muovere la penna sul foglio, ovvero dando le coordinate relative rispetto al punto precedente. La funzione di servizio in Lua può essere la seguente: essa fa uso dei campi `x` e `y` della tabella che immagazzina riga per riga lo stream PDF, campi utili anche per il calcolo (semplificato) dei limiti della figura:

```
plib.lineto = function(fig, dx, dy)
    local t = fig.data
    local x, y = dx + fig.x, dy + fig.y
    fig.x, fig.y = x, y
    t[#t+1] = string.format(
        "%0.6f %0.6f l", x, y
    )
    -- bb updating
    local x0 = math.min(x, fig.x)
    local y0 = math.min(y, fig.y)
    local x1 = math.max(x, fig.x)
    local y1 = math.max(y, fig.y)

    if not fig.bb then
        fig.bb = {x0, y0, x1, y1}
    else
        if x0 < fig.bb[1] then
            fig.bb[1] = x0
        end
        if y0 < fig.bb[2] then
            fig.bb[2] = y0
        end
        if x1 > fig.bb[3] then
            fig.bb[3] = x1
        end
        if y1 > fig.bb[4] then
            fig.bb[4] = y1
        end
    end
end
```

Il frammento principale del codice che costruisce la figura inserendola in un box orizzontale è il seguente dove si comincia a disegnare dallo spigolo più alto a sinistra:

```
\directlua{
local l = 120
local d = 8
local w = 2.8

local h = (l - d)/2
local fig = plib.newfig()
plib.penwidth(fig, w)

--[==[ stile logo --]==]

plib.moveto(fig,-d/2, l/2)
plib.lineto(fig,  0, -l)
plib.lineto(fig, -h,  h)
plib.lineto(fig,  l,  0)
plib.lineto(fig, -h, -h)
plib.lineto(fig,  0,  l)
plib.lineto(fig,  h, -h)
plib.lineto(fig, -l,  0)
plib.cycle(fig)

plib.stroke(fig)
plib.closefig(fig)

local n = node.new("whatsit", "pdf_literal")
n.mode = 0
```

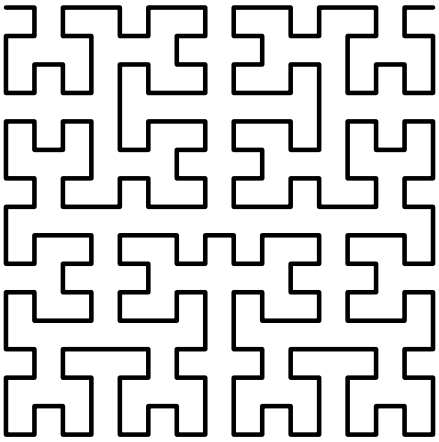


FIGURA 7: Disegno della figura ricorsiva nota come curva di Hilbert al quarto livello d'iterazione, generato con il codice Lua mostrato nel testo che fa uso dello stack delle chiamate ricorsive di funzione per memorizzare tutte le coordinate dei punti.

```
n.data = plib.stream(fig)
tex.box["mybox"] = pack(n, plib.bb(fig))
}
```

5.4 Linee 2

Il secondo e ultimo esempio di disegno a linee è la curva di Hilbert, un unico percorso ricorsivo che tende a riempire un quadrato, iterazione dopo iterazione. David Hilbert elaborò la curva nel 1891 come variante della teoria di Giuseppe Peano sulle curve che riempiono il piano, che a sua volta fu ispirato da George Cantor con i suoi primi risultati sulla cardinalità di insiemi infiniti.

La curva, rappresentata in figura 7, ha come elemento base il quadrato disegnato senza uno dei lati. Dal nostro punto di vista si tratta di un percorso che connette tutti i quadrati privi di uno dei lati disposti su una griglia regolare di un dato livello d'iterazione.

I segmenti crescono con ragione esponenziale con le iterazioni tanto che se alla prima iterazione sono 3 alla quinta sono già 832. Disegnarli con Lua e i nodi `pdf_literal` non è agevole come lo è in Metapost, dove abbiamo a disposizione le operazioni sui vettori con il tipo `pair` e quelle sul tipo `path`. Anni fa scrissi il codice in Metapost che riporto di seguito adattato alla compilazione con LuaTEX[4]:

```
% !TeX program = LuaTeX

\nopagenumbers
\pdfvariable compresslevel=0
\input luamplib.sty
\mplibcode
% Recursive Hilbert function
vardef hilbertCurve(expr c, x, level)=
   % the four basepoint
   save hilpath;
   path hilpath;
   save ax, bx, a, b;
   pair ax, bx, a, b;

   ax := c  rotatedabout(x, 90);
   bx := c  rotatedabout(x,-90);
   a  := ax rotatedabout(c ,-90);
   b  := bx rotatedabout(c , 90);

   if level = 1:
      hilpath := a--ax--bx--b;
   else:
      save atmp, btmp, axtmp, bxtmp;
      pair atmp, btmp, axtmp, bxtmp;
      atmp  := 0.25[a,b];
      btmp  := 0.25[b,a];
      axtmp := 1.25[a,ax];
      bxtmp := 1.25[b,bx];

      hilpath :=
      reverse
      hilbertCurve( a,  atmp, level-1)--
      hilbertCurve(ax, axtmp, level-1)--
      hilbertCurve(bx, bxtmp, level-1)--
      reverse
      hilbertCurve( b,  btmp, level-1);
   fi
   hilpath
enddef;

beginfig(1)
   pickup pencircle scaled 5;
   draw hilbertCurve(origin,(0,-100), 4);
endfig;
end
\endmplibcode
\bye
```

Grazie a Metapost, l'eleganza del codice è palese, e lo è anche la sequenza di streaming che riporta esattamente la definizione del percorso senza una sbavatura:

```
stream
1 0 0 1 262 579.89 cm
q
1.000 0.000 0.000 rg 1.000 0.000 0.000 RG
10.000000 M
1 j
1 J
5.000000 w
-187.500000 187.500000 m
-162.500000 187.500000 l
-162.500000 162.500000 l
-187.500000 162.500000 l
-187.500000 137.500000 l
-187.500000 112.500000 l
-162.500000 112.500000 l
-162.500000 137.500000 l
-137.500000 137.500000 l
...
endstream
```

4. La spiegazione del codice si trova nel post che pubblicai nel mio blog all'indirizzo https://robitex.wordpress.com/2009/12/31/la-curva-di-hilbert-in-metapost/.

In Lua possiamo costruirci gli stessi operatori vettoriali di Metapost così da poter semplicemente tradurre il codice ricorsivo del listato precedente. Altra via è esaminare attentamente la geometria dell'oggetto ricorsivo e ideare qualche tipo di rappresentazione. Scelgo questa seconda strada, naturalmente.

Il primo problema che si pone è questo: nella funzione ricorsiva non possiamo sapere quale sia il primo punto della curva, infatti sappiamo solo che a un certo punto avremo raggiunto il livello d'iterazione desiderato e dovremo disegnare il quadrato con il lato aperto anziché generare altri quattro vertici e proseguire la ricorsione con la misura del lato dimezzato.

Si potrebbe per questo aggiungere alla funzione ricorsiva un argomento di tipo booleano che sia vero se dobbiamo eseguire l'operazione `moveto` e falso se `lineto`, con lo svantaggio di dover controllare dal corpo della funzione ogni volta questo valore.

Potremo allora, invece di procedere subito al disegno, memorizzare la sequenza di punti in un array. Terminata la ricorsione l'array contiene tutto il percorso perciò si esegue l'operazione `moveto` per il primo punto e dal secondo punto in poi l'operazione `lineto`.

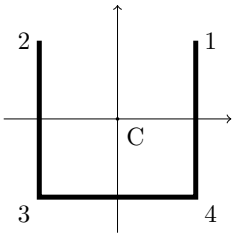
Del resto questo succede anche nella soluzione in Metapost precedente, dove una variabile di tipo `path` memorizza tutto il percorso che poi viene disegnato una volta completato dalla ricorsione. Tuttavia, con le funzioni di servizio memorizziamo già lo stream in un array perciò avremo duplicato le informazioni, prima in un array con le coppie di coordinate e poi in uno con le stringhe speciali PDF.

Penso invece che il codice si semplifichi se lasciamo che sia una funzione di servizio che chiameremo `pathto()` a scegliere l'operazione da eseguire sfruttando il fatto che nella tabella dello stream memorizziamo le coordinate del punto precedente nei campi `x` e `y` per calcolare senza considerare lo spessore delle linee, il bounding box. Se il percorso non è ancora iniziato questi campi sono nulli e quest'informazione discrimina tra il caso primo punto e il caso punto successivo:

```
plib.pathto = function(fig, x, y)
    local t = fig.data
    if fig.x then
        t[#t+1] = string.format(
            "%0.6f %0.6f l", x, y
        )
        -- bb updating
        local x0 = math.min(x, fig.x)
        local y0 = math.min(y, fig.y)
        local x1 = math.max(x, fig.x)
        local y1 = math.max(y, fig.y)
        if x0 < fig.bb[1] then
            fig.bb[1] = x0
        end
        if y0 < fig.bb[2] then
            fig.bb[2] = y0
        end
        if x1 > fig.bb[3] then
            fig.bb[3] = x1
        end
        if y1 > fig.bb[4] then
            fig.bb[4] = y1
        end
        fig.x, fig.y = x, y
    else
        t[#t+1] = string.format(
            "%0.6f %0.6f m", x, y
        )
        fig.x, fig.y = x, y
        fig.bb = {x, y, x, y}
    end
end
```

Inizio ora a illustrare la soluzione che ho elaborato per rappresentare due informazioni essenziali per il disegno ricorsivo della curva: quale lato del quadrato non deve essere disegnato e con quale orientamento si deve costruire il percorso aperto dei tre lati rimanenti: da un estremo all'altro o viceversa?

Osserviamo la seguente figura: si tratta della curva di Hilbert di livello 1, un quadrato con il lato aperto verso l'alto, una scelta che non influisce se non sulla rotazione del disegno:



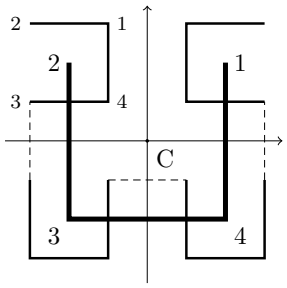
Il centro del quadrato ha coordinate $C = (x_c, y_c)$ e i suoi vertici un numero che è quello del quadrante di appartenenza nel sistema di riferimento locale. Note le coordinate del centro e la misura del lato del quadrato si ricavano immediatamente le coordinate dei vertici.

In Lua, potremo creare una tabella con i segni degli assi dei quadranti per calcolare con solo due espressioni le coordinate di un punto conoscendone il quadrante:

```
quadr = {
    { 1,  1}, -- quadrante 1
    {-1,  1}, -- quadrante 2
    {-1, -1}, -- quadrante 3
    { 1, -1}, -- quadrante 4
}

function point(xc, yc, l, q)
    local x = xc + quadr[q][1]*l/2
    local y = yc + quadr[q][2]*l/2
    return x, y
end
```

Il livello 2 si costruisce disegnando quadrati aperti secondo la posizione, sui quattro vertici del livello 1 dimezzando la misura del lato, come mostra la figura sottostante:



In questo schema ho evidenziato con il tratteggio le linee che completano la curva di secondo livello, di unione tra i quadrati. Se costruiamo il percorso inserendo i vertici nell'ordine corretto, nel disegno troveremo anche questi tratti di collegamento, ma non eviteremo di inserire due linee anche quando queste sono allineate.

I livelli successivi si ottengono continuando a costruire su tutti i vertici del livello precedente i tre lati dei quadrati di lunghezza dimezzata.

Definita la geometria si può affrontare il nocciolo del problema: trovare un modo per conoscere per ogni quadrato quale lato non deve essere disegnato e con quale verso il percorso deve essere ordinato. Questi due problemi sono risolti nella funzione precedente in Metapost con le operazioni sui vettori e applicando la funzione `reverse` sui percorsi parziali.

La soluzione che ho trovato per definire completamente il percorso di disegno per ogni singolo quadrato, si basa su un numero intero con segno: il valore assoluto del numero individua il primo vertice secondo il quadrante e il segno indica se si deve percorrere i lati in senso antiorario (segno positivo) o in senso orario (segno negativo).

Per esempio il disegno del quadrato di livello 2 in alto a sinistra della figura è definito semplicemente dal numero −2. Infatti, seguendo la regola, si parte dal nodo 2 ma per via del segno negativo si segue il senso orario trovando la sequenza di nodi 2, 1, 4, e 3. Lo stesso quadrato ma con percorso invertito si disegna a partire dal numero 3, che produce la sequenza dei quattro nodi 3, 4, 1, 2.

Empiricamente ho poi trovato il numero con segno da assegnare a ciascuno dei quattro quadrati da generare ricorsivamente per ogni vertice del livello precedente. Quest'informazione è memorizzata nella tabella che assume il ruolo di invariante per l'iteratore, alla chiave chiamata `nodes`.

Senza entrare troppo nei dettagli del codice, l'iteratore — tecnicamente si tratta di un iteratore senza stato — con cui possono essere creati i vertici del quadrato in un nodo nella sequenza corretta dal numero del nodo di partenza è il seguente:

```
quadr = {
    { 1,  1}, -- quadrante 1
    {-1,  1}, -- quadrante 2
    {-1, -1}, -- quadrante 3
    { 1, -1}, -- quadrante 4
}

function itervertex(t, i)
    i = i + 1
    if i < 4 then
        local n = math.abs(t.n)
        if t.rev then
            n = n - i
            if n < 1 then n = n + 4 end
        else
            n = n + i
            if n > 4 then n = n - 4 end
        end
        local x = t.xc + quadr[n][1]*t.l/2
        local y = t.yc + quadr[n][2]*t.l/2
        return i, t.nodes[i+1], x, y
    end
end

function vertex(xc, yc, l, n)
    local rev = n < 0 and true or false
    if rev then
        last = -(n - 2)
        if last > 4 then last=last - 4 end
    else
        last = -(n + 2)
        if last < -4 then last=last + 4 end
    end
    local inv_state = {
        xc=xc, yc=yc, l=l, rev=rev, n=n,
        nodes = {-n, n, n, last},
     }
    return itervertex, inv_state, -1
end
```

La funzione ricorsiva controlla il livello: se uguale a 1 chiama la funzione `pathto` aggiungendo linee a partire dal punto precedente così la curva risulterà un'unica spezzata, altrimenti chiama se stessa su ognuno dei quattro vertici (da notare che l'iteratore è impiegato in entrambi i rami del costrutto `if`):

```
function hilbertCurve(fig, xc, yc, l, n, lv)
  if lv == 1 then
    for _,_, x, y in vertex(xc, yc, l, n) do
      plib.pathto(fig, x, y)
    end
  else
    for _,v, x, y in vertex(xc, yc, l, n) do
      hilbertCurve(fig, x, y, l/2, v, lv-1)
    end
  end
end
```

Il codice LuaTEX per creare il disegno non dovrà far altro che impostare lo spessore della linea, gli stili di chiusura del tratto, e poi chiamare la funzione ricorsiva con le coordinate del centro della curva, la dimensione del lato del quadrato di livello 1, il codice nodo che definisce sia il lato aperto e il

verso del disegno, e per ultimo il numero dei livelli da raggiungere.

Specificando il codice nodo +2 avremo il lato aperto uguale a quello superiore e l'ordine di tracciamento antiorario, perciò il percorso della curva comincerà in alto a sinistra e finirà in alto a destra:

```
\newbox\mybox

\directlua{
local dim = 200
local w = 5

local fig = plib.newfig()
plib.penwidth(fig, w)
plib.linecap(fig, 1)
plib.linejoin(fig, 1)

hilbertCurve(fig, 0, 0, dim, 2, 4)

plib.stroke(fig)
plib.closefig(fig)

local n = node.new("whatsit", "pdf_literal")
n.mode = 0
n.data = plib.stream(fig)
tex.box["mybox"] = pack(n, plib.bb(fig))
}
\leavevmode\box\mybox
```

La parte di codice che risolve la curva più complesso e meno leggibile è quello dell'iteratore. Lascio al lettore come esercizio, il compito di riscrivere la funzione ricorsiva generando il percorso della curva di Hilbert a partire dal primo nodo in alto a sinistra della griglia di vertici distanti $l/2^{\text{level}}$ e calcolando la posizione del nodo successivo tra le quattro possibilità conoscendo la direzione della mossa precedente.

Se agli spostamenti secondo la direzione degli assi attribuiamo i numeri $+x = 1$, $-x = 3$, $+y = 2$, $-y = 4$, cominciando dal punto più in alto a sinistra, la costruzione del livello 2 è conseguenza della successione delle 13 mosse:

```
1, 4, 3,
4,
4, 1, 2,
1,
4, 1, 2,
2,
3, 2, 1
```

Qual è la sequenza di mosse per il disegno della curva di Hilbert di livello n? Si tratta ancora di un problema ricorsivo?

6 Conclusioni

Con questo lavoro ho esplorato con pieno successo di risultati la tecnologia dei nodi di LuaTEX per creare grafica con le primitive PDF. Molte altre cose, come gli oggetti riferimento, attendono di entrare nel nostro laboratorio.

Le funzionalità a disposizione per il formato d'uscita PDF dei documenti composti con LuaTEX sono davvero molte. Abbiamo appena cominciato a sperimentarle e se si considera che possono essere programmate in Lua all'interno del motore di composizione, si giunge alla conclusione che le potenzialità applicative del nuovo motore sono notevolissime.

La stretta integrazione tra le librerie Lua e gli oggetti del compositore TEX consente di ottenere risultati tipografici probabilmente unici tra tutti i programmi utilizzati per la produzione di documenti digitali.

LuaTEX è ormai prossimo alla versione 1.0, la prima a essere stabile, un momento spero importante per il futuro del sistema TEX.

7 Ringraziamenti

Come al solito ringrazio la mia famiglia che mi concede (parecchio) tempo, specie nelle giornate estive, e in particolare mio figlio Matteo che mi ha disegnato appositamente la figura 6 a linea continua, molto interessante nella sua semplicità e perfetta per gli scopi dell'articolo, e mia moglie Silvia che ha letto la bozza con appassionata pazienza alla ricerca di errori, refusi, incoerenze. Grazie famiglia!

Luigi Scarso mi ha aiutato a capire un passaggio che riguarda il campo `shift` dei nodi di LuaTEX e non ha esitato a migliorare i sorgenti con osservazioni e precisazioni tecniche. Grazie mille.

Ringrazio infine tutta la redazione di ArsTEXnica e chi ha lavorato con impegno per la buona riuscita del meeting annuale dell'associazione.

8 L'editor Atom

Una nota riguardo alle modalità con cui è stato prodotto questo articolo, composto con il compositore `pdflatex`, riguarda l'editor utilizzato per i sorgenti. Normalmente per questo compito si utilizza uno shell editor, ovvero un programma per il testo che offre funzioni per la compilazione TEX svolgendo appunto un ruolo d'interfaccia.

Questa volta ho usato Atom, disponibile sul sito `https://atom.io/`, un editor multi-piattaforma per la scrittura di codice molto interessante anche per i sorgenti TEX.

Il cursore multiplo intelligente, la gestione automatica della tabulazione, i riquadri laterali dell'albero del progetto e della miniatura del contenuto, lo spostamento delle righe verso l'alto o verso il basso con una semplice combinazione di tasti, sono solo alcune delle sue abilità per comporre il testo.

Le funzionalità integrate di Atom per la gestione del sistema di controllo delle revisioni `git`, lo rende ancor di più adatto per produrre con il sistema TEX documenti come libri o tesi di laurea anche molto complessi e con più autori.

Riferimenti bibliografici

Adobe Systems Incorporated (2006). *The PDF Reference 1.7*. Addison-Wesley, 6ª edizione. URL http://www.adobe.com/devnet/pdf/pdf_reference_archive.html.

Beccari, C. (2015). «I calcoli matematici in pdftex». *ArsTEXnica*, (20), pp. 7–15. URL http://www.guitex.org/home/it/arstexnica.

Eijkhout, V. (2014). *TEX by Topic*. Lehmanns media.

Giacomelli, R. (2012). «Grafica ad oggetti con LuaTEX». *ArsTEXnica*, (14), pp. 53–71. URL http://www.guitex.org/home/numero-14.

— (2016). «Primi esperimenti con node di LuaTEX». *ArsTEXnica*, (21). URL http://www.guitex.org/home/numero-21.

Goossens, M., Mittelbach, F., Roegel, D. e Voss, H. (2007). *The LATEXGraphics Companion (2nd Edition)*. Addison-Wesley Professional, 2ª edizione.

Gregorio, E. (2009). «Appunti di programmazione in LATEX e TEX». https://profs.sci.univr.it/~gregorio/introtex.pdf. Giugno 2009. Seconda edizione.

Hagen, H., Hoekwater, T., Roux, E., Gesang, P. e Dohyun, K. (2016). «The luamplib package». $TEXMFDIST/doc/luatex/luamplib/luamplib.pdf. Version 2016/03/31 v2.11.3.

Höppner, K. (2008). «Introduction to MetaPost». *ArsTEXnica*, (6), pp. 5–9. URL http://www.guitex.org/home/it/arstexnica.

Ierusalimschy, R. (2013). *Programming in Lua*. Lua.org, 3ª edizione.

Knuth, D. E. (1984). *The TeXbook*. Addison-Wesley Professional, 1ª edizione.

Mittelbach, F., Goossens, M., Braams, J. e Carlisle, D. (2004). *The LATEXCompanion (Tools and Techniques for Computer Typesetting)*. Addison-Wesley Professional, 2ª edizione.

Nisi, R. (2009). «Il PostScript in LATEX». *ArsTEXnica*, (7), pp. 32–46. URL http://www.guitex.org/home/numero-7.

Thành, Hàn Thế, Rahtz, S., Hagen, H., Henkel, H., Jackowski, P. e Schröder, M. (2016). «The luamplib package». $TEXMFDIST/doc/pdftex/manual/pdftex-a.pdf. Version 2016/05/09.

The LuaTEX development team (2016). *LuaTEX Reference Manual*, 0.95 edizione.

Zandt, T. V. (2003). «User's guide». $TEXMFDIST/doc/generic/pstricks/pstricks-doc.pdf. Version 97.

▷ Roberto Giacomelli
Carrara
giaconet dot mailbox at gmail dot com