Isia Roma - Sede di Pordenone - Corsi di Matematica per il Design

Introduzione a Python

Luciano Battaia

23 gennaio 2020

In questo fascicolo proponiamo una elementare e sintetica introduzione ad alcune caratteristiche di Python, in particolare a Python 2.7, in quanto gli script saranno fatti girare utilizzando il compilatore interno a Rhinoceros 6. Quasi tutto quello che diremo funziona anche in Python 3. Quando necessario evidenzieremo le differenze tra le due versioni.

Queste note non sono un manuale di Python, ma contengono solo alcune nozioni di base e non possono essere utilizzate senza il supporto delle lezioni in aula, di cui costituiscono solo lo schema.

1 Variabili e tipi

Il concetto di *variabile* è fondamentale in tutti i linguaggi di programmazione. Una variabile è un contenitore di dati (una "scatola", o "cassetto") situato in una porzione di memoria del computer destinato a contenere valori che possono essere utilizzati ed eventualmente modificati nel corso dell'esecuzione di un programma.

In qualche modo quando si introduce una variabile si deve indicare al software il nome che vogliamo assegnarle e che cosa ci vogliamo mettere dentro: numeri interi, numeri reali, parole (stringhe), liste di numeri, immagini, o quant'altro. In Python l'indicazione del tipo di variabile è di tipo dinamico: in sostanza a seconda di come definisco la variabile Python capisce che tipo di variabile è. Una volta dato un nome ad una variabile potremo richiamarla semplicemente con quel nome: in sostanza il nome è un'etichetta che viene piazzata davanti al cassetto in cui è inserita la variabile.

È molto importante conoscere il tipo di variabile, in quanto certe manipolazioni sono consentite solo tra erte variabili. Vediamo un esempio. Supponiamo di voler introdurre le variabili `a, b, c, d`, di cui le prime due contengono parole e le ultime due numeri interi. Nella finestra di editing scriveremo il seguente codice:

```
a="Buongiorno"
b=" mondo."
c=2
d=10
```

Se chiediamo (vedremo fra un momento come) al software di scrivere di seguito il contenuto di a e di b otterremo `Buongiorno mondo`. Se chiediamo di scrivere c^d otterremo 1024. Se chiediamo di scrivere $a + c$ otterremo errore, in quanto non si può sommare una stringa con un numero. Se vogliamo cambiare il contenuto di una variabile (cioè di un cassetto), basta ridefinirla. Consideriamo il codice seguente.

```
a="Buongiorno"
b=" mondo."
```

```
c=2
d=10
a="Buonasera"
c=3
```

Se ora chiediamo di scrivere di seguito a e b di seguito otterremo `Buonasera mondo`, se chiediamo di scrivere c^d otterremo 59049.

2 L'istruzione/funzione print

Il comando `print` è il comando di base per ottenere un output visibile da un programma.

In Python 2.7, quello implementato in Rhino 6, il comando `print` è una "dichiarazione" ("statement") che richiede una serie di argomenti, separati da una virgola, e li stampa in output separati da uno spazio. Gli argomenti possono essere stringhe (parole racchiuse tra apici, spesso commenti), calcoli matematici, variabili già definite, ecc. Vediamo un esempio (** è il simbolo di potenza in Python). Il codice

```
a="Buongiorno mondo"
b=2
c=10
print "I valori inseriti sono:",a,",",b,",",c,"."
print "Invece b elevato a c vale:",b**c
```

produce l'output

```
I valori inseriti sono: Buongiorno mondo , 2 , 10 .
Invece b elevato a c vale: 1024
```

In Python 3 `print` diventa una funzione e ha la sintassi tipica delle funzioni, ovvero racchiude i suoi argomenti tra parentesi tonde, separandoli con una virgola. Vedremo esempi su altre funzioni. Per quanto ci riguarda, questa, assieme a quella relativa a `input` che vedremo fra poco, è la principale differenza tra Python 2.7 e Python 3.

3 Principali tipi di dati

I principali tipi di dati, cioè di variabili che ci interessano sono:

- Valori booleani: `True` e `False`.
- Numeri interi: `int`.
- Numeri reali, o in virgola mobile: `float`.
- Stringhe: `str`. Le stringhe sono sequenze di caratteri, ovvero parole o successioni di parole.
- Liste: `list`. Le liste sono elenchi di oggetti eterogenei, ovvero gli elementi possono essere di qualunque tipo, anche altre liste.
- Tuple: `tuple`. La tuple sono come le liste, ma non possono essere modificate. Evitare di introdurre caratteri non ascii (per esempio le lettere accentate), anche se si potrebbe fare.
- Insiemi: `set`. Gli insiemi sono quello che normalmente si intende per insieme in matematica.

Per recuperare il tipo di una variabile introdotta basta usare la funzione `type()`. Per esempio

```
a=9.7
print type(a)
```

produce l'output

```
<type 'float'>
```

Per maggiore chiarezza di lettura dell'output si può far scrivere a `print` un commento.

```
a=9.7
print "Tipo della variabile a:",type(a)
```

produce l'output

```
Tipo della variabile a: <type 'float'>
```

Per ognuno dei tipi sopra considerati esiste una particolare sintassi per introdurre i valori, in modo che Python riconosca automaticamente il tipo. Proponiamo un esempio complessivo, autoesplicativo, introducendo anche dei commenti utili alla comprensione: i commenti devono essere preceduti da #.

```
a=8 #numero intero
print "Valore di a:",a,", tipo:",type(a)
b= 3.4 #numero in virgola mobile
print "Valore di b:",b,", tipo:",type(b)
c="Buongiorno mondo!" #stringa
print "Valore di c:",c,", tipo:",type(c)
d=[1,2.3,"Ciao"] #lista
print "Valore di d:",d,", tipo:",type(d)
e=(1,2.3,"Ciao") #tupla
print "Valore di e:",e,", tipo:",type(e)
f={1,3.5,"pippo"} #insieme
print "Valore di f:",f,", tipo:",type(f)
g=set([9,17.4,"luciano"]) #alternativo per insieme
print "Valore di g:",g,", tipo:",type(g)
```

il cui output è

```
Valore di a: 8 , tipo: <type 'int'>
Valore di b: 3.4 , tipo: <type 'float'>
Valore di c: Buongiorno mondo! , tipo: <type 'str'>
Valore di d: [1, 2.3, 'Ciao'] , tipo: <type 'list'>
Valore di e: (1, 2.3, 'Ciao') , tipo: <type 'tuple'>
Valore di f: set([3.5, 1, 'pippo']) , tipo: <type 'set'>
Valore di g: set([17.4, 9, 'luciano']) , tipo: <type 'set'>
```

In realtà in alcuni casi i numeri in virgola mobile vengono scritti in output con diverse cifre dopo la virgola, ma per il momento non ci preoccupiamo di questo fatto.

Con questi oggetti si possono fare varie cose. Non entreremo nei dettagli, riservandoci di approfondire in seguito. Proponiamo solo qualche esempio.

Esempio 1. Unione di stringhe.

```
stringa1="Buongiorno "
stringa2="mondo."
print stringa1+stringa2
```

con output

```
Buongiorno mondo!
```

Esempio 2. Lunghezza di una stringa.

```
stringa1="Buongiorno "
print len(stringa1)
```

con output: 11.

Esempio 3. Unione di liste, lunghezza di una lista, individuazione di un elemento di dato posto in una lista, inserimento di un oggetto nuovo in un posto di una lista.

```
lista1=[1,2,5,7]
lista2=[11,13,15]
lista3=lista1+lista2
print "Lista unione:",lista3
print "Lunghezza lista3:", len(lista3)
print "Terzo elemento della lista3:",lista3[2]
lista3.insert(4,"luciano")#inserimento dopo il posto 4
print "Nuova lista 3:",lista3
```

con output

```
Lista unione: [1, 2, 5, 7, 11, 13, 15]
Lunghezza lista3: 7
Terzo elemento della lista3: 5
Nuova lista 3: [1, 2, 5, 7, 'luciano', 11, 13, 15]
```

Esempio 4. Unione, intersezione, differenza di insiemi. Elementi ripetuti non contano.

```
insieme1={1,2,3,4,2,3,5,"a","b","c","a","b"}
print "Insieme1:",insieme1
insieme2={2,4,6,8,"b","c"}
print "Unione:",insieme1 | insieme2
print "Intersezione:",insieme1 & insieme2
print "insieme1-insieme2:", insieme1-insieme2
print "insieme2-insieme1:", insieme2-insieme1
```

con output

```
Insieme1: set(['b', 1, 2, 3, 4, 5, 'a', 'c'])
Unione: set(['b', 1, 2, 3, 4, 5, 6, 8, 'a', 'c'])
Intersezione: set(['b', 2, 4, 'c'])
insieme1-insieme2: set([1, 3, 5, 'a'])
insieme2-insieme1: set([6, 8])
```

4 Le funzioni raw_input e input

In Python 2.7 esistono due funzioni per l'immissione di dati da tastiera e precisamente `raw_input` e `input`. La prima accetta in ingresso qualunque tipo di dati, ma lo memorizza sempre in una stringa, la seconda accetta solo numeri interi o float. Il seguente script fa capuire la differenza.

```
numero1=raw_input("Inserisci un numero")
print "numero 1 di tipo:",type(numero1)
print 2*numero1
numero2=input("Inserisci un numero")
print "numero 2 di tipo:",type(numero2)
print 2*numero2
```

Se si inserisce, ad ognuna delle due richieste, il numero 5 si ottiene l'output

```
numero1 di tipo: <type 'str'>
55
```

```
numero2 tipo: <type 'int'>
10
```

Tutto è corretto, in quanto nel primo caso si raddoppia semplicemente la stringa 5, nel secondo caso la stringa 5 viene trasformata prima in un intero e quindi ne viene calcolato il doppio.

In Python 3 la funzione `raw_input` è stata soppressa e la funzione `input` si comporta esattamente come la `raw_input` di Python 2.7.

Per motivi che qui non possiamo spiegare, in Python 2.7 conviene usare *solo* la funzione `raw_input` e poi fare manualmente la conversione da un tipo ad un altro, come è obbligatorio in Python 3. Lo vedremo nel primo esempio del prossimo paragrafo.

5 Un po' di sintassi

5.1 L'istruzione if

È l'istruzione più comune a tutti i linguaggi di programmazione, e non solo; ne abbiamo già visto l'uso in Geogebra. Vediamone l'uso in un esempio significativo di per sè, e che non ha bisogno di molti commenti. In questo esempio facciamo anche uso della funzione `raw_input()` che permette di inserire dati da tastiera

Il programmino (un po' cretino a dire il vero,ma che ha lo scopo di spiegare la logica dell'istruzione in esame), che in Python si chiama *Script*, aspetta che l'utente inserisca un numero intero dalla tastiera; successivamente converte la stringa fornita da `raw_input` in un intero; di seguito se il numero è tra 0 e 9 manda un messaggio che indica di quante cifre è composto il numero, analogamente se è tra 10 e 99, oppure tra 100 e 999; con altri numeri manda il messaggio "Numero negativo o maggiore di 1000".

```
n=raw_input("Inserisci un numero intero ")
nint=int(n)
if 0<=nint<10:
    print "Il numero introdotto ha una cifra"
elif 10<=nint<99:
    print "Il numero ha due cifre"
elif 100<=nint<999:
    print "Il numero ha tre cifre"
else:
    print "Numero negativo o maggiore di 1000"
```

In ogni programma in cui è previsto un input da parte dell'utente ci deve un controllo sulla congruità dell'input per evitare il *crash* del programma. In questo programmino se uno inserisce una stringa Python segnala un errore. Ci sono molti modi per effettuare un controllo ed evitare questo tipo di problemi. Ne proponiamo uno qui, senza commenti, che consente di fare anche più errori.

Il succo di questo codice è il seguente.

1. Introduce una variabile ausiliaria, di tipo booleano assegnandole il valore `True`: questa variabile serve a far eseguire il successivo ciclo `while` (di cui parleremo fra poco) fin quando non viene posta a `False`.
2. Successione esegue un tentativo chiede l'inserimento di un intero ed esegue un tentativo di trasformazione dell'input inserito dall'utente in un tipo `int`. Se la cosa funziona, modifica il valore di `test` in `False` e il ciclo `while` termina; il programma passa all'istruzione successiva.
3. Se il tentativo di trasformazione in intero non è riuscito il programma prosegue con la parte `except` del costrutto `try/except`, manda due messaggi all'utente e il ciclo `while` ricomincia.
4. Quando finalmente l'utente ha inserito un valore accettabile il programma prosegue.

```
test = True
while test:
    try:
```

```
        n = int(raw_input("Inserisci un numero intero: "))
        test=False
    except ValueError:
        print "Hai inserito una stringa."
        print "Ora comportati bene e inserisci un numero intero: "
print "Valore inserito correttamente!"
if 0<= n <10:
    print "Il numero introdotto ha una cifra "
elif 10 <= n <99:
    print "Il numero ha due cifre."
elif 100 <= n <999:
    print "Il numero ha tre cifre."
else :
    print " Numero negativo o maggiore di 1000."
```

Questa volta se si inserisce una stringa Python si lamenta, ma ci propone un nuovo inserimento, fino a quando non ci decidiamo a inserire un numero.

L'indentazione del codice, che risulta evidente in questo programmino, è una delle caratteristiche peculiari di Python (e di pochissimi altri linguaggi): è appositamente studiata per rendere il più leggibile possibile un codice: essa rende evidente dove termina un blocco di istruzioni e dove comincia il successivo. Ogni editor studiato per Python implementa questa caratteristica di default, anche RhinoPython lo fa.

5.2 L'istruzione for

Come in tutti i linguaggi di programmazione, l'istruzione `for` permette di ripetere certe operazioni per una determinato numero di volte. Vediamo alcuni esempi.

```
for x in "abc":
    print x
```

produce l'output

```
a
b
c
```

```
for i in range(10):
    print i
```

produce l'output

```
0
1
2
3
4
5
6
7
8
9
```

```
for i in range(5,8):
    print i
```

produce l'output

```
5
6
7
```

```
for i in range(1,10):
    print i**2
```

produce l'output

```
1
4
9
16
25
36
49
64
81
```

```
for i in range(1,10,2):
    print i**2
```

produce l'output

```
1
9
25
49
81
```

5.3 L'istruzione while

L'istruzione `while` differisce dall'istruzione `for` in quanto non ripete un ciclo un determinato numero di volte, ma finché la condizione seguente è vera. Anche qui mostriamo alcuni esempi.

```
x=1
while x<5:
    print x
    x += 1
```

produce l'output

```
1
2
3
4
```

```
x=1
while x<10:
    print "Il quadrato di ",x,"vale ",x**2
    x += 2
```

produce l'output

```
Il quadrato di  1 vale  1
Il quadrato di  3 vale  9
Il quadrato di  5 vale  25
Il quadrato di  7 vale  49
Il quadrato di  9 vale  81
```

6 Funzioni

Le *funzioni* in Pyhton, ma anche in tutti i linguaggi di programmazione, sono esattamente la stessa cosa che abbiamo definito in matematica: sono scatolette con ingranaggi che ricevono in input uno o più dati e restituiscono in output un dato. Nella matematica oggetto del nostro corso gli oggetti in input sono solitamente numeri, coppie di numeri, terne di numeri, quelli in output numeri. Per esempio se scrivo

$$f(x,y) = x^2 + 2xy - 1,$$

"passando alla funzione" i numeri 2 e 3 mi viene restituito il numero 15: si scrive $f(2,3) = 15$.

Se voglio fare la stessa cosa in Python devo prima definire la funzione e poi richiamarla, passandole i valori.

```
def funzionemia(x,y):
    return x**2+2*x*y-1
print funzionemia(2,3)
```

produce l'output 15.

L'esempio è volutamente elementare (anzi forse anche stupido), ma fa capire il senso. Potrei poi modificarlo consentendo all'utente di inserire i valori di x e y.

```
def funzionemia(x,y):
    return x**2+2*x*y-1
x=int(raw_input("Inserisci la x"))
y=int(raw_input("Inserisci la y"))
print "Valore calcolato:",funzionemia(x,y)
```

È importante ricordare che il nome delle variabili usate nella definizione della funzione non centrano nulla con i nomi delle variabili che poi costruisco: sono nomi solo locali alla funzione. Il seguente codice è identico al precedente.

```
def funzionemia(x,y):
    return x**2+2*x*y-1
p=int(raw_input("Inserisci la x"))
q=int(raw_input("Inserisci la y"))
print "Valore calcolato:",funzionemia(p,q)
```

Proponiamo un secondo esempio: la funzione che calcola il minimo di tre numeri inseriti dall'utente.

```
def minimo(a,b,c):
    if a<b and a<c:
        return a
    if b<c:
        return b
    return c
x=int(raw_input("Introduci il primo numero:"))
y=int(raw_input("Introduci il secondo numero:"))
z=int(raw_input("Introduci il terzo numero:"))
print "Minimo dei tre numeri",x,y,z,":",minimo(x,y,z)
```

7 Un primo sguardo alle interazioni con la finestra grafica di Rhino

Utilizzando Python si possono avere interessanti interazioni con la finestra grafica di Rhino. In generale per fare occorre istruire lo script Python ad utilizzare la sintassi di Rhino. Per fare questo, all'inizio dello script, bisogna inserire la seguente istruzione:

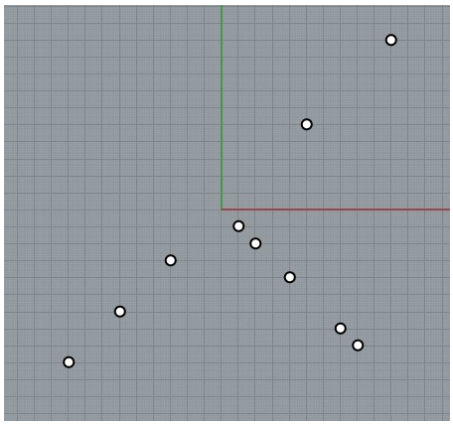
```
import rhinoscriptsyntax as rs
```

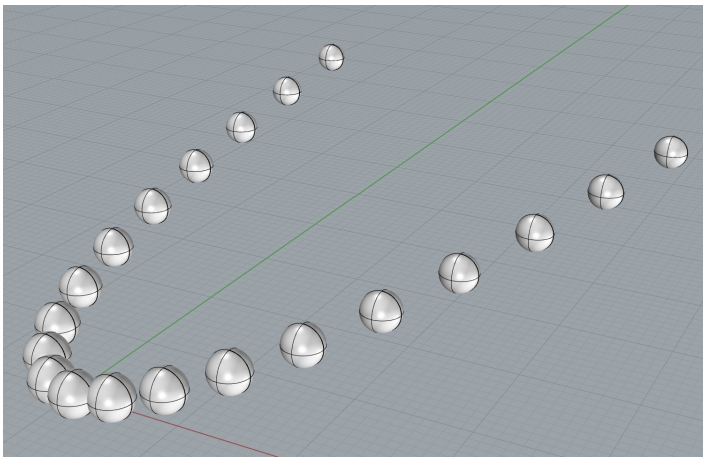
Questa "importazione" consente di utilizzare le capacità grafiche di Rhino direttamente da Python. Vediamo due esempi, banali ed autoesplicativi, solo per iniziare. Nel paragrafo successivo faremo qualche ulteriore considerazione.

```
import rhinoscriptsyntax as rs

numeri =[1,2,3,4,5,6,7,8,9,10]
for n in numeri:
    if n % 5 == 0:
        rs.AddPoint([10*n,10*n,0])
    elif n % 3 == 0:
        rs.AddPoint([-10*n,-10*n,0])
    else:
        rs.AddPoint([10*n,-10*n,0])
```

```
import rhinoscriptsyntax as rs

for n in range(-10,10):
    rs.AddSphere([3*n,n**2,0],2)
```

L'output nella finestra di Rhino è, rispettivamente, il seguente.



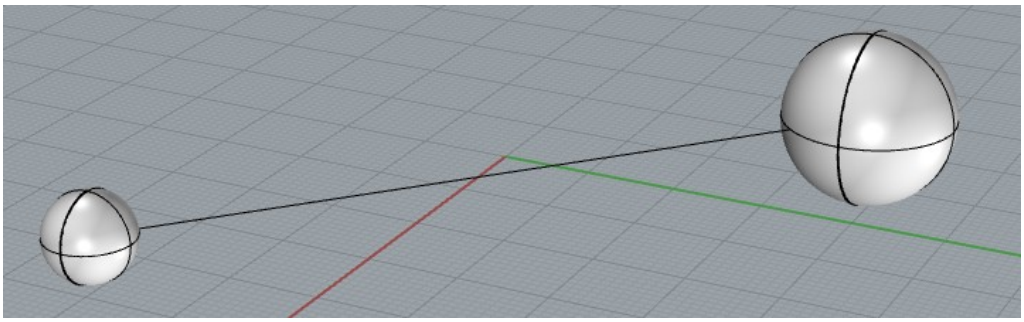


Si possono inserire in input anche dati prelevati direttamente dalla finestra grafica di Rhino. Vediamo un esempio, molto semplice, che disegna una linea tra due punti indicati con il mouse sulla finestra grafica e poi aggiunge due sfere all'inizio e alla fine.

```
import rhinoscriptsyntax as rs

start=rs.GetPoint("Inizio linea")
if start:
    end=rs.GetPoint("Fine linea")
    if end:
```

```
        rs.AddLine(start,end)
rs.AddSphere(start,5)
rs.AddSphere(end,10)
```

L'output della finestra grafica di Rhino, dopo aver scelto due punti con il mouse, è il seguente.



8 Qualche ulteriore dettaglio sullo scripting in Rhino

Come tutti i programmi di grafica che si rispettino, anche Rhino ha la possibilità di inserire oggetti, oltreché utilizzando i comandi delle finestre grafiche, anche utilizzando appositi *script*, ovvero opportuni codici che eseguono operazioni che, almeno in parte, sono non eseguibili solo "con l'uso del mouse".

Rhino ha un suo linguaggio per fare questo, chiamato *RhinoScript*. Si tratta di un linguaggio abbastanza semplice e intuitivo. Tuttavia è basato, per affermazione stessa degli sviluppatori di Rhino, su una tecnologia vecchia e abbastanza superata. Inoltre, ed è forse la cosa più importante, RhinoScript non è ben supportato dalla comunità degli utenti allo stesso modo di Python. Dunque la scelta migliore per chi voglia usare Rhino per applicazioni elevate è quella di usare lo scripting in Python, linguaggio moderno, molto flessibile, ben supportato dalla comunità degli utenti e ormai adottato come standard per innumerevoli applicazioni.

La cosa importante è che tutti i comandi disponibili in RhinoScript possono essere importati in Python utilizzando il già descritto comando di importazione[(1)]:

```
import rhinoscriptsyntax as rs
```

Per chi vuole conoscere tutti i dettagli dello scripting Pyhton in Rhino e della sintassi di base di Pyhton, tutto il necessario si troba al link https://developer.rhino3d.com/guides/rhinopython/.

Quali sono i motivi per usare lo scripting in Rhino? Ne elenchiamo alcuni, presi dallo stesso sito ufficiale degli sviluppatori.

- Automatizzare azioni ripetitive in maniera molto rapida ed elegante rispetto all'inserimento manuale.
- Eseguire azioni che non sono previste nel set standard di comandi di Rhino.
- Generare geometrie usando algoritmi o formule (vedremo alcuni esempi di generazione di superfici o curve).
- Molto, molto altro. Dopotutto Python è un linguaggio di programmazione, dunque può fare un'infinità di cose, molte più di quelle che qualunque progettista di un software può pensare di inserire di default.

9 Ulteriori esempi

Abbiamo già visto le funzioni `AddPoint()`, `GetPoint()`, `AddLine()`, `AddSphere()`, che, lo ricordiamo, devono essere usate in Python solo dopo averle importate con il descritto comando e precedute da "`rs.`". Tutte queste funzioni sono in sostanza le stesse usate dal software (anche se noi non le vediamo direttamente)

[1] `rs` è usato per abbreviare le scritture. Si potrebbe scrivere direttamente `rhinoscriptsyntax` o qualunque altra abbreviazione si desideri; tuttavia `rs` è di gran lunga la più comune, anche perché usata negli esempi proposti dagli sviluppatori di Rhino.

quando usiamo i pulsanti e il mouse. Esaminiamo ora qualche ulteriore esempio, un po' più complesso e coinvolgente altre funzioni.

Esempio 5. Divisione di curve in punti e tracciamento di segmenti.

```
import rhinoscriptsyntax as rs

def disegnaLineaTraCurve(Curva1,Curva2,numDivs):
    Listapti1=rs.DivideCurve(Curva1,numDivs,True,True)
    Listapti2=rs.DivideCurve(Curva2,numDivs,True,True)

    count=0
    for i in Listapti1:
        rs.AddLine(Listapti1[count],Listapti2[count])
        count+=1

PrimaCurva=rs.GetObject("Scegli la prima curva",4)
SecondaCurva=rs.GetObject("Scegli la seconda curva",4)
divs=int(raw_input("Quante divisioni?"))

ogg1=disegnaLineaTraCurve(PrimaCurva,SecondaCurva,divs)
```

Ricordare che le indentazioni sono fondamentali, mentre le linee vuote inserite sono solo per maggiore chiarezza.

Vediamo di commentare brevemente.

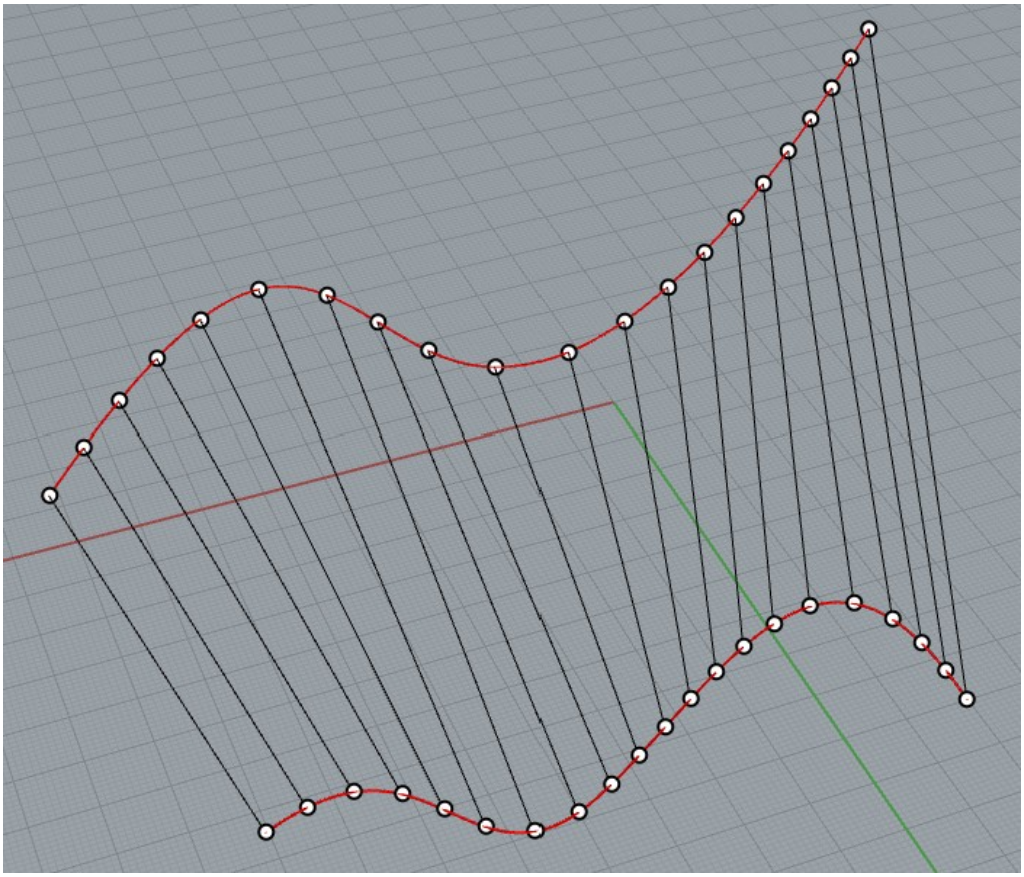
- La funzione `disegnaLineaTraCurve()` ha bisogno di tre dati in ingresso: una prima curva, una seconda curva, in quante parti devono essere divise.
- Una volta ricevuti i dati in ingresso, la funzione produce due liste di punti, ciascuna costituita dai punti in cui le due curve vengono divise, e per fare questo utilizza la funzione `DivideCurve()`, la quale a sua volta deve ricevere quattro dati in ingresso:la curva da dividere, il numero di divisioni e due informazioni booleane, di cui la prima dice se costruire o no i punti, la seconda se restituire o no la lista di punti.
- Il successivo ciclo `for` costruisce un segmento tra due punti corrispondenti, uno per ciascuna linea.
- La funzione `GetObject()` manda un messaggio e richiede un numero per sapere che tipo di oggetto; il numero 4 è per le curve. Altri numeri per altre cose. Quindi perché il programma funzioni devo avere prima costruito, in qualche modo, due curve e poi le selezionerò con il mouse.
- Successivamente devo immettere da tastiera il numero di divisioni (attenzione: non viene fatto un controllo dell'input, quindi non scherzare con la tastiera!).
- Infine viene richiamata la funzione di disegno, passandole i parametri necessari.

Una variante interessante, soprattutto quando si hanno lunghi cicli da eseguire. Rhino di default esegue un redraw dello schermo mentre lo script viene eseguito: nell'esempio precedente si vede materialmente il disegno successivo dei vari segmenti. Per impedire questo e avere un risultato più rapido si può bloccare il redraw, ricordandosi di riattivarlo alla fine. Di seguito la variante completa. Provatele entrambe con 100 suddivisioni.

```
import rhinoscriptsyntax as rs

rs.EnableRedraw(False)

def disegnaLineaTraCurve(Curva1,Curva2,numDivs):
    Listapti1=rs.DivideCurve(Curva1,numDivs,True,True)
```

```
    Listapti2=rs.DivideCurve(Curva2,numDivs,True,True)

    count=0
    for i in Listapti1:
        rs.AddLine(Listapti1[count],Listapti2[count])
        count+=1

PrimaCurva=rs.GetObject("Scegli la prima curva",4)
SecondaCurva=rs.GetObject("Scegli la seconda curva",4)
divs=int(raw_input("Quante divisioni?"))

ogg1=disegnaLineaTraCurve(PrimaCurva,SecondaCurva,divs)

rs.EnableRedraw(True)
```

Provate anche a mettere a `False` il primo booleano di `DivideCurve()`. Di seguito il risultato della divisione di due curve con 20 punti e la costruzione di 20 segmenti.

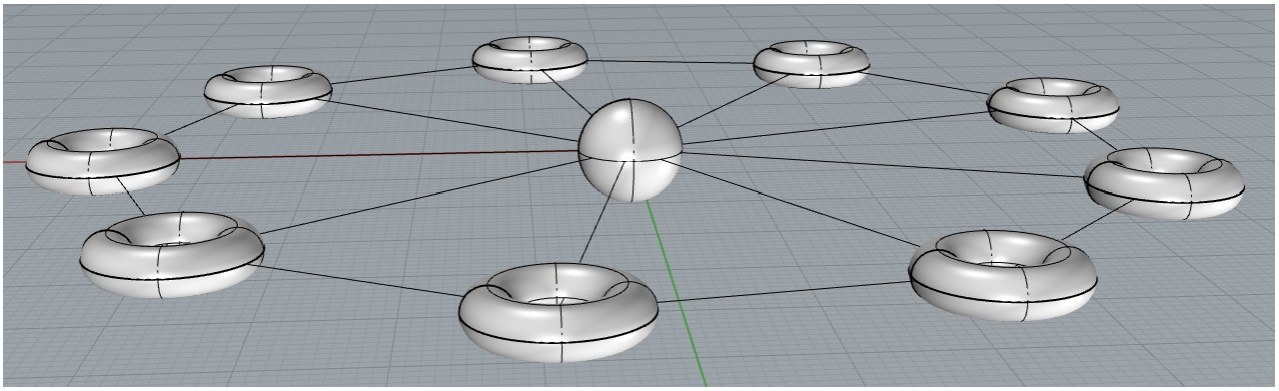


Esempio 6. Una successione di 9 ciambelle disposte sui vertici di un ennagono regolare, congiunti tra di loro e con una sfera posta al centro. Leggendo il codice si dovrebbe capire la logica. Le novità rispetto agli esempi precedenti sono le funzioni:

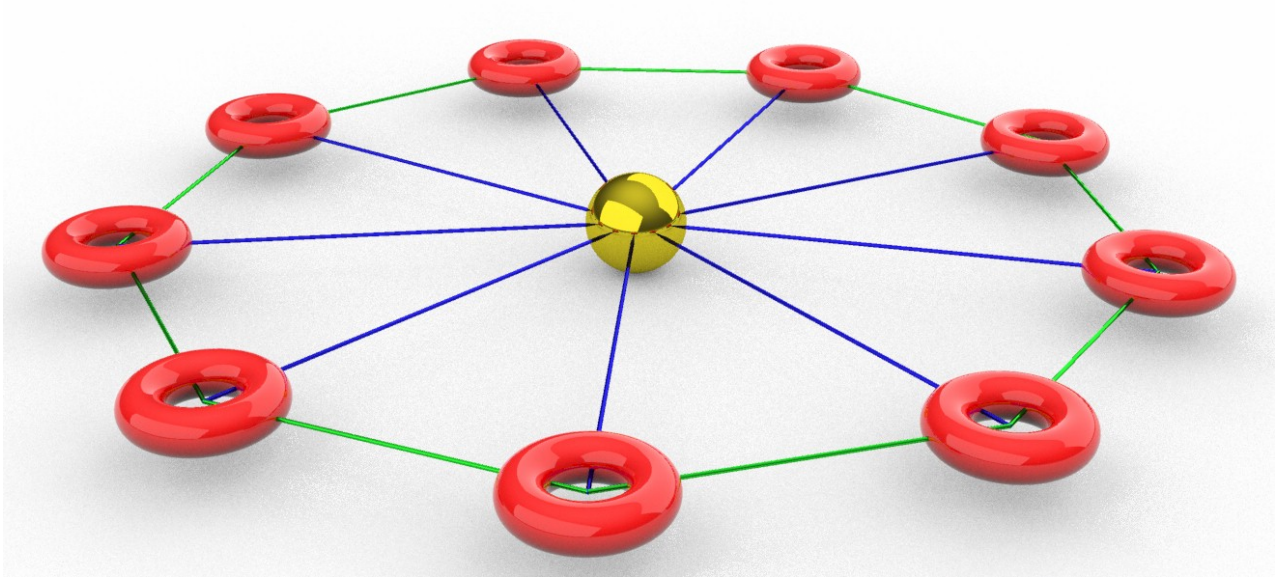
- `AddCircle(<centro>, <raggio>)`
- `DeleteObject(<Oggetto da cancellare>)` (in questo caso il cerchio che è servito solo per costruire i vertici dell'ennagono).
- `AddTorus(<centro>,<raggio maggiore>,<raggio minore>)`

```
import rhinoscriptsyntax as rs

circolo=rs.AddCircle([0,0,0],50)
ListaPti=rs.DivideCurve(circolo,9,False,True)

rs.DeleteObject(circolo)

rs.AddSphere([0,0,0],5)
count=-1
for i in ListaPti:
    rs.AddLine(ListaPti[count],ListaPti[count+1])
    rs.AddLine(ListaPti[count],[0,0,0])
    rs.AddTorus(ListaPti[count],5,2)
    count += 1
```

La figura che segue mostra l'output.



La figure che segue mostra l'output con una veloce renderizzazione senza pretese.



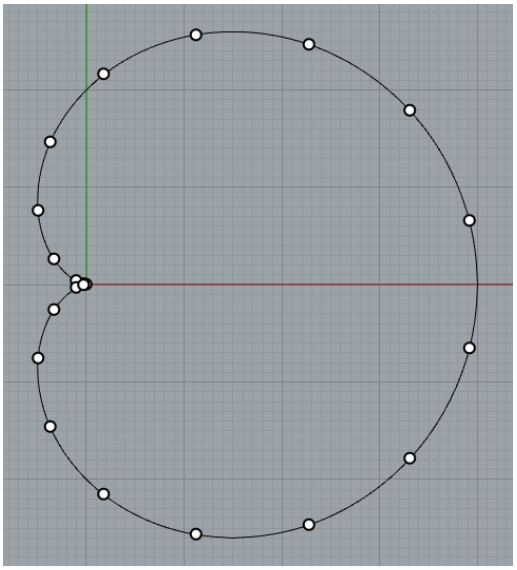
Esempio 7. Un esempio di curva parametrica, precisamente una cardiode. Le equazioni parametriche di questa curva e della seguente si possono trovare nel file *Curve di Bézier e spline*.

Il codice che segue può essere utilizzato per tracciare una curva parametrica, conoscendone l'equazione. La curva viene tracciata come curva di interpolazione di punti ottenuti dall'equazione parametrica. Il codice è ottimizzato per i migliori risultati e non entriamo nei dettagli. Può tranquillamente essere usato per

tracciare diverse curve, sostituendo le equazioni qui usate con altre equazioni. Si segnala solo che questo codice usa funzioni matematiche non di base, e quindi occorre introdurre una speciale libreria di Python che contiene queste funzioni: si tratta della libreria `math` e per questo le funzioni vengono richiamate con il codice `math.`, tipico delle funzioni di librerie importate. Si noti che due linee di codice sono spezzate per esigenze tipografiche. Questo può essere fatto o usando un backslash o andando a capo dopo una virgola. Bisogna stare attenti ad andare a capo in Python, perché andando a capo di norma si determina la fine di una riga di codice.

```
import rhinoscriptsyntax as rs
import math

def draw_parametric_curve(function, param_range, num_points = 20):
    points = []
    for i in range(num_points):
        t = param_range[0] + (param_range[1] - \
        param_range[0]) * float(i)/float(num_points - 1)
        points.append(function(t))

    rs.AddInterpCurve(points)

def miaCurva(t):
    return [10*(1-2*math.cos(t)+math.cos(2*t)),
    10*(2*math.sin(t)-math.sin(2*t)),0]

draw_parametric_curve(miaCurva, (0, 2*math.pi))
```

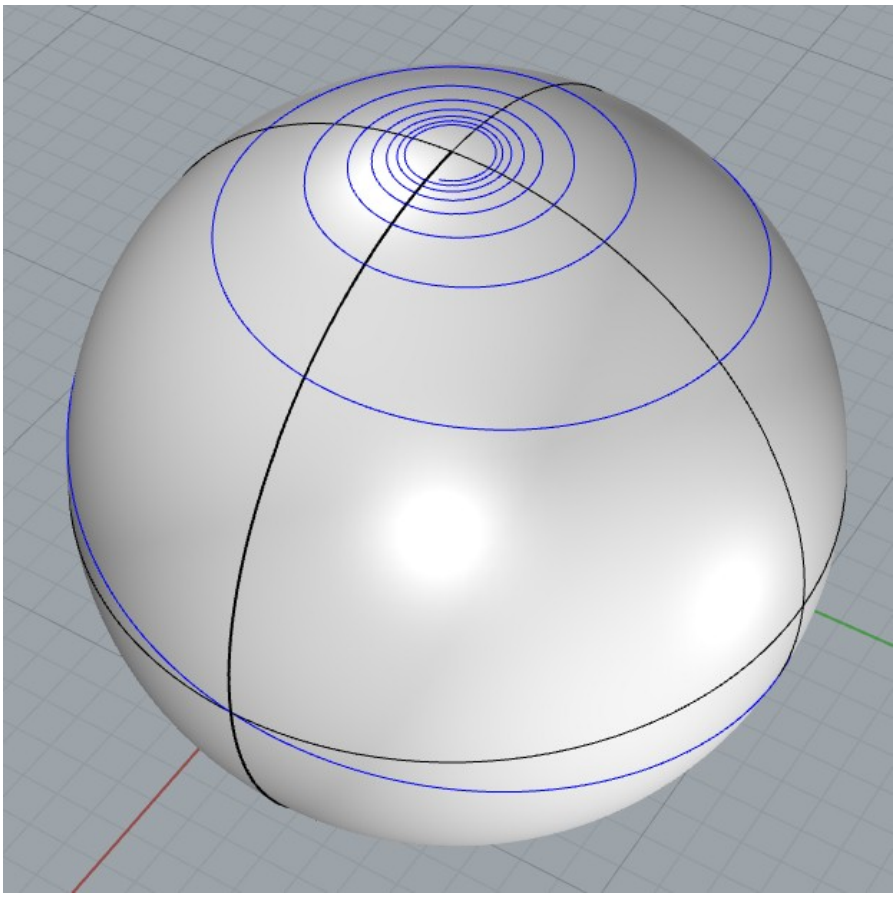
Il numero di punti presenti nella definizione di `draw_parametric_curve` deve essere tanto più alto quanto più la curva è complessa. Nella figura che segue mostriamo l'output del codice precedente in cui abbiamo anche evidenziato i punti di interpolazione: l'abbiamo fatto a posteriori dopo aver ottenuto la curva tracciata con lo script.



Esempio 8. Una curva parametrica 3D, precisamente una spirale su una sfera. Il codice per la curva parametrica è lo stesso dell'esempio precedente. naturalmente le equazioni sono diverse. Abbiamo inoltre disegnato anche la sfera su cui la curva è tracciata. Si noti che il numero di punti è stato portato a 1000 per la complessità di questa curva.

```
import rhinoscriptsyntax as rs
import math

r=10
p=0.2
rs.AddSphere((0,0,0),r)

def draw_parametric_curve(function, param_range, num_points = 1000):
    points = []
    for i in range(num_points):
        t = param_range[0] + (param_range[1] - param_range[0]) * \
        float(i)/float(num_points - 1)
        points.append(function(t))

    rs.AddInterpCurve(points)

def spirale_su_sfera(t):
    return [r*math.cos(t)/math.sqrt(1+p*p*t*t),
    r*math.sin(t)/math.sqrt(1+p*p*t*t),-r*p*t/math.sqrt(1+p*p*t*t)]

draw_parametric_curve(spirale_su_sfera, (-50,50))
```

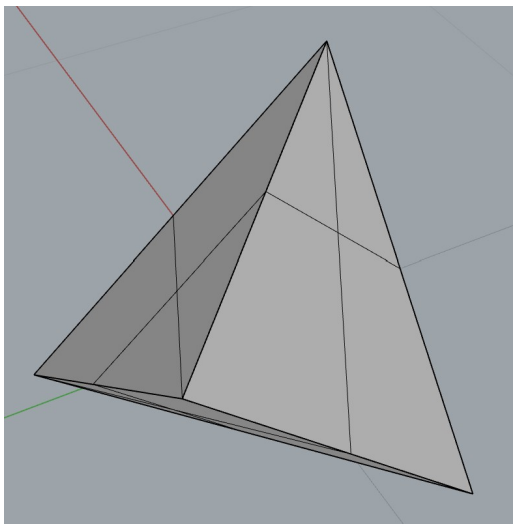
La figura presente mostra l'output di questo codice.

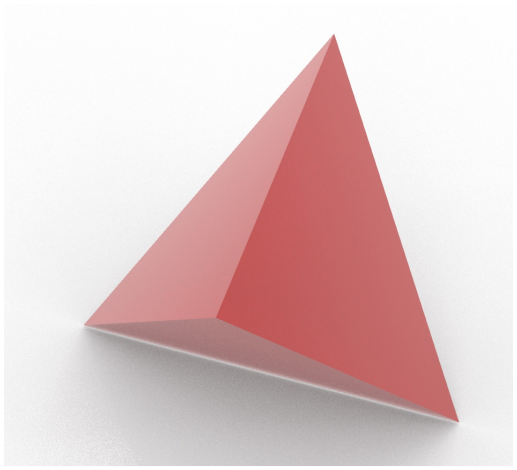


La figura seguente mostra lo stesso output con una renderizzazione elementare.

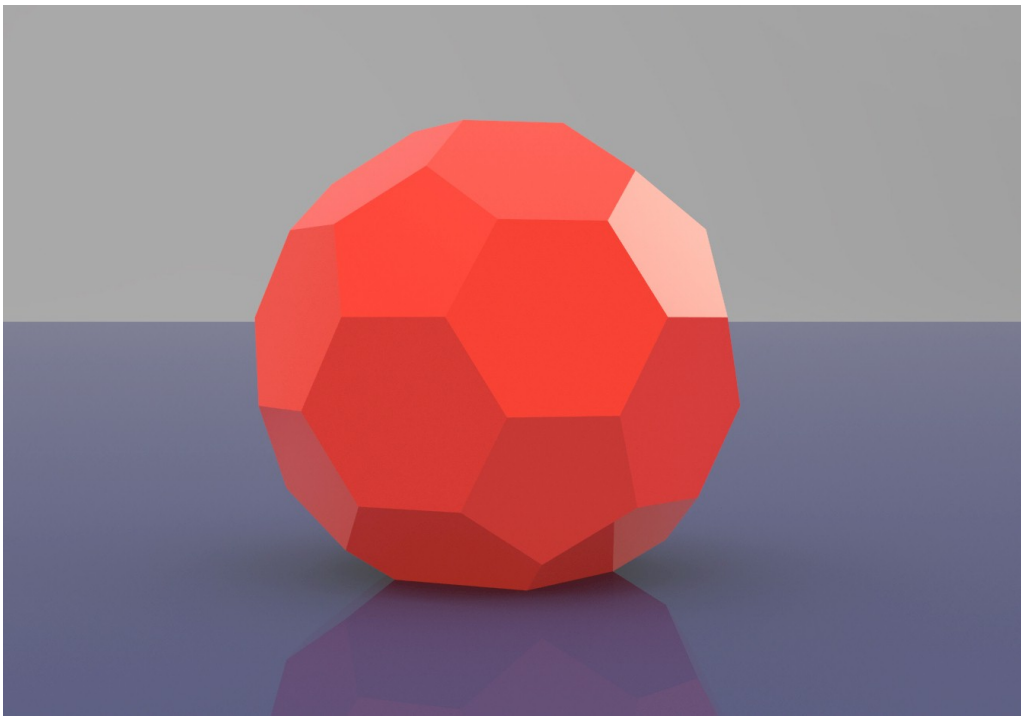
Esempio 9. Costruzione di poliedri. RhinoPython può essere utilizzato efficacemente per la costruzione in maniera rapida di poliedri, se si conoscono le coordinate dei vertici. L'esempio che segue mostra il caso del tetraedro, con l'output normale e poi velocemente renderizzato.

```
import rhinoscriptsyntax as rs
import math

C0=math.sqrt(2)/4

V0 = ( C0, -C0,  C0)
V1 = ( C0,  C0, -C0)
V2 = (-C0,  C0,  C0)
V3 = (-C0, -C0, -C0)

rs.AddPolyline([V0,V1,V2,V0])
rs.AddPolyline([V1,V0,V3,V1])
rs.AddPolyline([V2,V3,V0,V2])
rs.AddPolyline([V3,V2,V1,V3])
```





I poliedri a facce regolari (in particolare i solidi Platonici e Archimedei) possono essere costruiti anche direttamente in Rhino senza fare ricorso a RhinoPython: ne forniamo un esempio con l'icosaedro troncato costruito incollando e ruotando opportunamente pentagoni ed esagoni nella finestra di Rhino.



Molto più difficile ottenere poliedri più complessi, come i poliedri regolari stellati o quelli non Platonici e non Archimedei senza l'uso di RhinoPython.

Esempio 10. Questo esempio mostra il caso del *Grande dodecaedro*, ovvero uno dei 4 solidi di Keplero Poinsot.

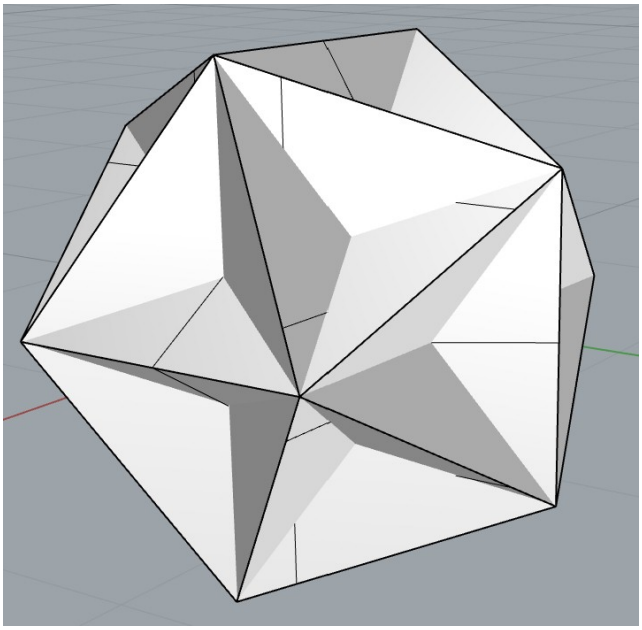
Come si può vedere dal codice non ci sono differenze di rilievo rispetto al caso del tetraedro; naturalmente ci saranno 12 vertici anziché 4. Le coordinate dei vertici si possono ottenere da numerosi programmi o da siti internet.

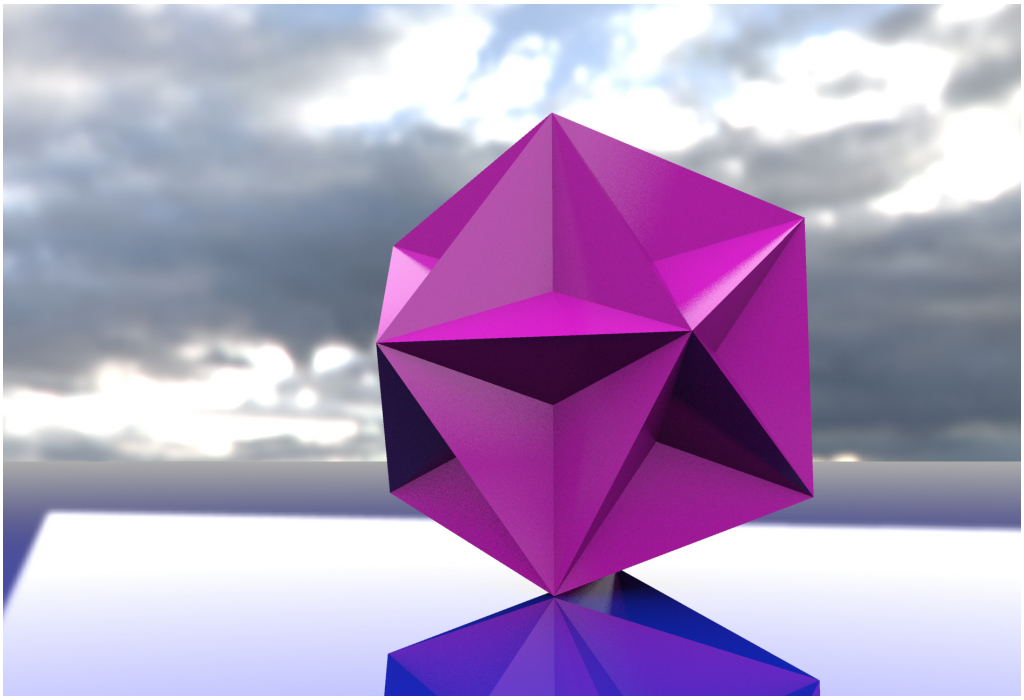
In questo codice abbiamo inserito un commento su più linee. Abbiamo già visto che in Python i commenti su una linea devono essere preceduti da #; i commenti su più linee vanno preceduti e seguiti da tre doppie virgolette """. Se si vogliono costruire i vertici del poliedro basta decommentare (cioè togliere i tre doppi apici prima e dopo) tutto il gruppo di righe che contiene gli `rs.AddPoint()`. Si noti, sia qui che nel codice relativo al tetraedro, che per ottenere un poligono abbiamo usato la funzione `rs.AddPolilyne()`, fornendo come argomento la lista dei vertici, e ripetendo il primo vertice in fondo in modo da avere un poligono chiuso.

```
import rhinoscriptsyntax as rs
import math
C0=(1+math.sqrt(5))/4
V0  = ( 0.5,  0.0,   C0)
V1  = ( 0.5,  0.0,  -C0)
V2  = (-0.5,  0.0,   C0)
V3  = (-0.5,  0.0,  -C0)
V4  = (  C0,  0.5,  0.0)
V5  = (  C0, -0.5,  0.0)
V6  = ( -C0,  0.5,  0.0)
V7  = ( -C0, -0.5,  0.0)
```

```
V8  = ( 0.0,   C0,  0.5)
V9  = ( 0.0,   C0, -0.5)
V10 = ( 0.0,  -C0,  0.5)
V11 = ( 0.0,  -C0, -0.5)
"""
rs.AddPoint(V0)
rs.AddPoint(V1)
rs.AddPoint(V2)
rs.AddPoint(V3)
rs.AddPoint(V4)
rs.AddPoint(V5)
rs.AddPoint(V6)
rs.AddPoint(V7)
rs.AddPoint(V8)
rs.AddPoint(V9)
rs.AddPoint(V10)
rs.AddPoint(V11)
"""
rs.AddPolyline([V0,V2,V7,V11,V5,V0])
rs.AddPolyline([V0,V5,V1,V9,V8,V0])
rs.AddPolyline([V0,V8,V6,V7,V10,V0])
rs.AddPolyline([V1,V3,V6,V8,V4,V1])
rs.AddPolyline([V1,V4,V0,V10,V11,V1])
rs.AddPolyline([V1,V11,V7,V6,V9,V1])
rs.AddPolyline([V2,V0,V4,V9,V6,V2])
rs.AddPolyline([V2,V6,V3,V11,V10,V2])
rs.AddPolyline([V2,V10,V5,V4,V8,V2])
rs.AddPolyline([V3,V1,V5,V10,V7,V3])
rs.AddPolyline([V3,V7,V2,V8,V9,V3])
rs.AddPolyline([V3,V9,V4,V5,V11,V3])
```

La figura seguente mostra l'output normale, la successiva un veloce rendering.

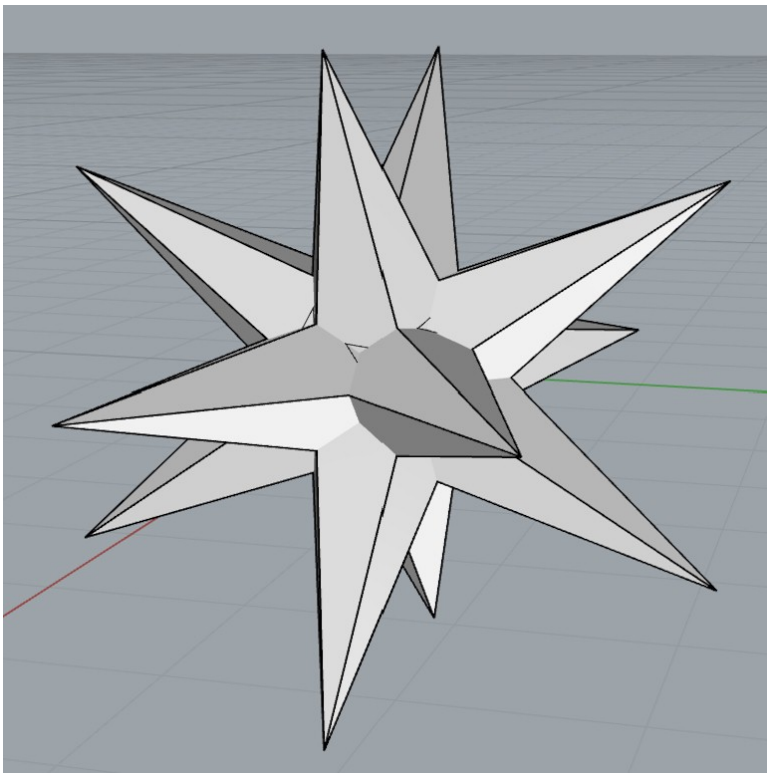


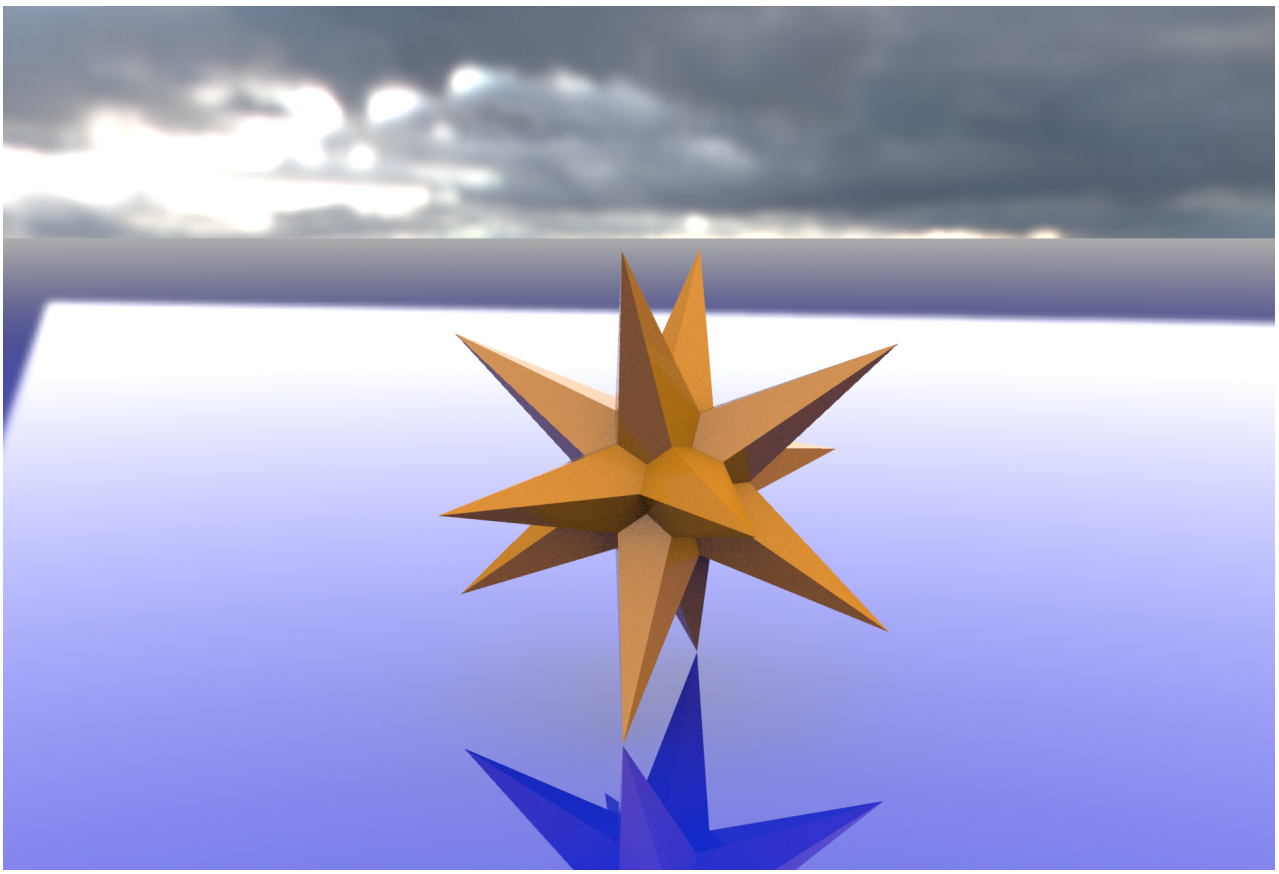


Esempio 11. Come ulteriore esempio proponiamo, senza alcun commento, un poliedro del gruppo dei duali dei poliedri di Badoureau-Coxeter, precisamente il duale di $U41$, ovvero il *Medio Icosaedro Triambico*.

```
import rhinoscriptsyntax as rs
import math

C0 = (3 - math.sqrt(5)) / 2
C1 = (math.sqrt(5) - 1) / 2
C2 = (1 + math.sqrt(5)) / 2
C3 = (3 + math.sqrt(5)) / 2

V0  = ( C2, 0.0,  C3)
V1  = ( C2, 0.0, -C3)
V2  = (-C2, 0.0,  C3)
V3  = (-C2, 0.0, -C3)
V4  = ( C3,  C2, 0.0)
V5  = ( C3, -C2, 0.0)
V6  = (-C3,  C2, 0.0)
V7  = (-C3, -C2, 0.0)
V8  = (0.0,  C3,  C2)
V9  = (0.0,  C3, -C2)
V10 = (0.0, -C3,  C2)
V11 = (0.0, -C3, -C2)
V12 = ( C0, 0.0,  C1)
V13 = ( C0, 0.0, -C1)
V14 = (-C0, 0.0,  C1)
V15 = (-C0, 0.0, -C1)
V16 = ( C1,  C0, 0.0)
V17 = ( C1, -C0, 0.0)
V18 = (-C1,  C0, 0.0)
```

```
V19 = (-C1, -C0, 0.0)
V20 = (0.0,  C1,  C0)
V21 = (0.0,  C1, -C0)
V22 = (0.0, -C1,  C0)
V23 = (0.0, -C1, -C0)

rs.AddPolyline([V0,V14,V6,V19,V11,V22,V0])
rs.AddPolyline([V0,V22,V7,V23,V1,V17,V0])
rs.AddPolyline([V0,V17,V11,V13,V9,V16,V0])
rs.AddPolyline([V0,V16,V1,V21,V6,V20,V0])
rs.AddPolyline([V0,V20,V9,V18,V7,V14,V0])
rs.AddPolyline([V3,V13,V4,V17,V10,V23,V3])
rs.AddPolyline([V3,V23,V5,V22,V2,V19,V3])
rs.AddPolyline([V3,V19,V10,V14,V8,V18,V3])
rs.AddPolyline([V3,V18,V2,V20,V4,V21,V3])
rs.AddPolyline([V3,V21,V8,V16,V5,V13,V3])
rs.AddPolyline([V12,V2,V22,V11,V17,V4,V12])
rs.AddPolyline([V12,V4,V20,V6,V14,V10,V12])
rs.AddPolyline([V12,V10,V17,V1,V16,V8,V12])
rs.AddPolyline([V12,V8,V14,V7,V22,V5,V12])
rs.AddPolyline([V12,V5,V16,V9,V20,V2,V12])
rs.AddPolyline([V15,V1,V23,V10,V19,V6,V15])
rs.AddPolyline([V15,V6,V21,V4,V13,V11,V15])
rs.AddPolyline([V15,V11,V19,V2,V18,V9,V15])
rs.AddPolyline([V15,V9,V13,V5,V23,V7,V15])
rs.AddPolyline([V15,V7,V18,V8,V21,V1,V15])
```





10 Qualche esempio di superficie parametrica

Le superfici dello spazio possono essere descritte matematicamente mediante equazioni parametriche, esattamente come le linee, solo che, in questo caso, servono due parametri, ovvero bisogna dare la x, la y e la z, mediante espressioni matematiche contenenti due parametri.

Per esempio per il *Nastro di Möbius* le equazioni sono le seguenti.

$$x = \left(1+\frac{v}{2}\cos\frac{u}{2}\right)\cos u \quad , \quad y = \left(1+\frac{v}{2}\cos\frac{u}{2}\right)\sin u \quad , \quad z = \frac{v}{2}\sin\frac{u}{2} \quad , \quad 0 \le u \le 2\pi\,, -1 \le v \le 1\,.$$

Geogebra è in grado di visualizzare anche superfici complesse assegnate in equazione parametrica. Il codice generale da usare in Geogebra per questo tipo di grafici è il seguente.

`Superficie( <Espressione>, <Espressione>, <Espressione>, <Parametro variabile 1>,`
<Valore iniziale>, <Valore finale>, <Parametro variabile 2>, <Valore iniziale>, <Valore finale>)

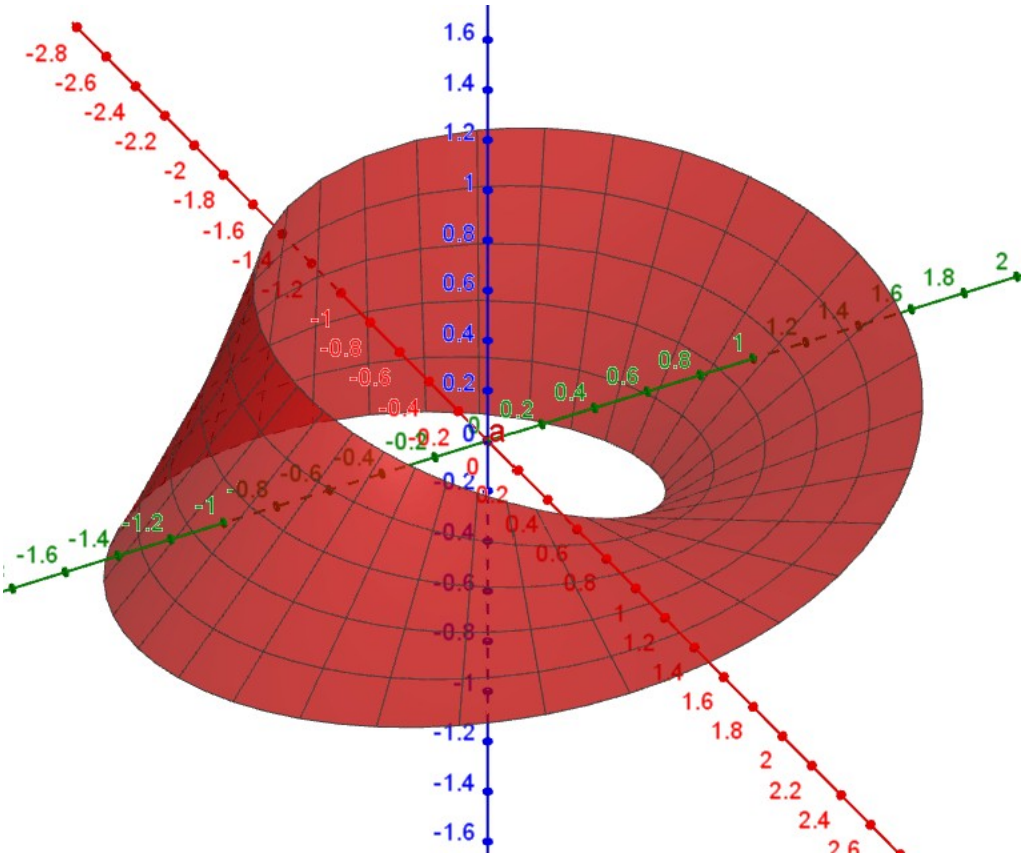
Di seguito l'adattamento di questo codice per questa superficie. Naturalmente per visualizzare l'output bisogna avere attivato la finestra "Grafici 3D".

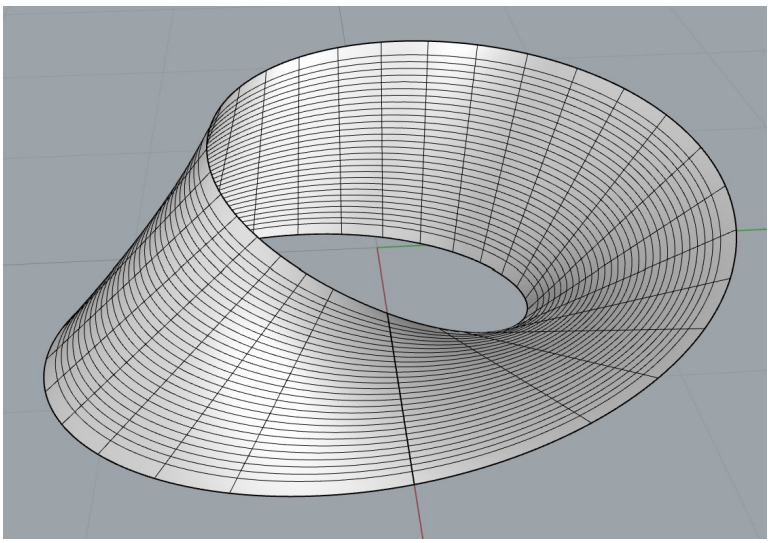
```
Superficie((1+(v/2)cos(u/2))cos(u),(1+(v/2)cos(u/2))sin(u),
                                                (v/2)sin(u/2),u,0,2pi,v,-1,1)
```

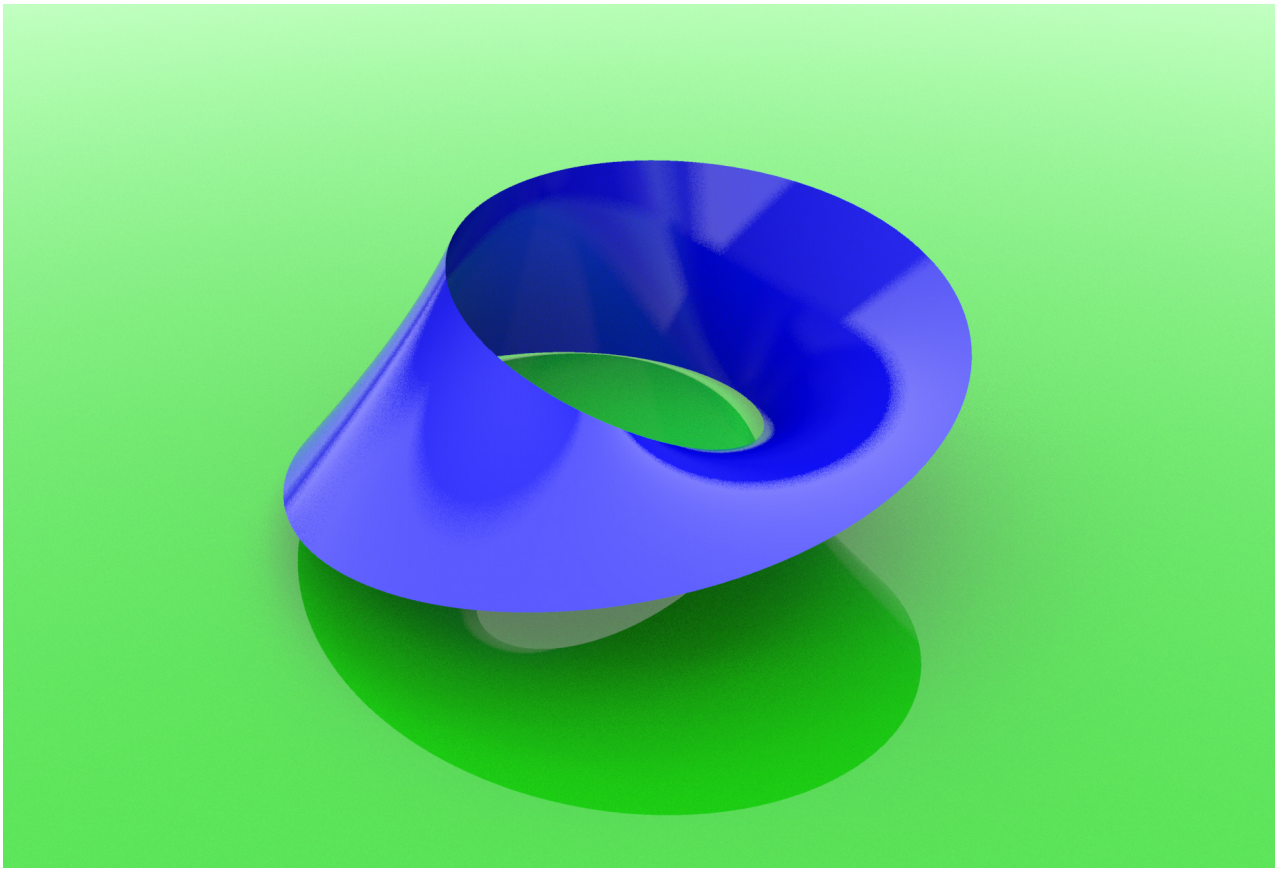
Per quanto riguarda RhinoPython occorre un piccolo script, simile a quello usato per tracciare le curve. Lo proponiamo qui di seguito, senza particolari commenti. Anche qui il numero di punti, che deve essere indicato per ciascuno dei due parametri, è tanto più grande, quanto più la superficie è complessa.

```
import rhinoscriptsyntax as rhino
import math

def draw_parametric_surface(function, param_range_u, param_range_v, \
num_points_u = 32, num_points_v = 32):
  points = []
  for i in range(num_points_u):
    u = param_range_u[0] + (param_range_u[1] - param_range_u[0]) * \
    float(i)/float(num_points_u - 1)
    for j in range(num_points_v):
      v = param_range_v[0] + (param_range_v[1] - param_range_v[0]) * \
      float(j)/float(num_points_v - 1)
      points.append( function(u,v) )
  rhino.AddSrfPtGrid((num_points_u, num_points_v), points)

def superf_Prova(u,v):
  return [(1+(v/2)*math.cos(u/2))*math.cos(u),(1+(v/2)*math.cos(u/2))*\
  math.sin(u),(v/2)*math.sin(u/2)]

draw_parametric_surface(superf_Prova, (0,2*math.pi), (-1,1))
```

Le tre immagini che seguono mostrano l'output in Geogebra, quello normale in Rhino, una semplice renderizzazione in Rhino.







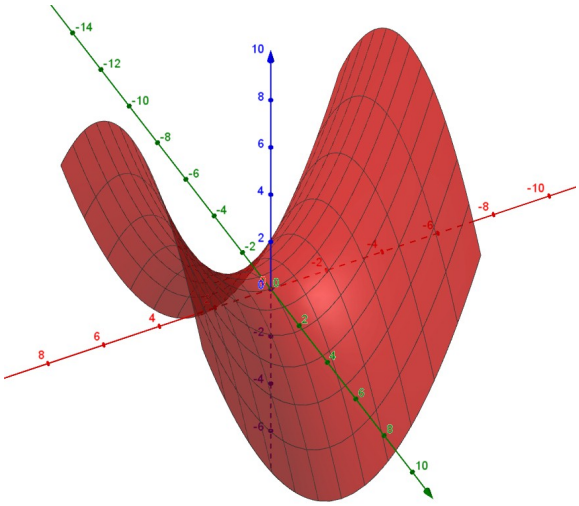
Esempio 12. Il *Paraboloide iperbolico*, di equazioni

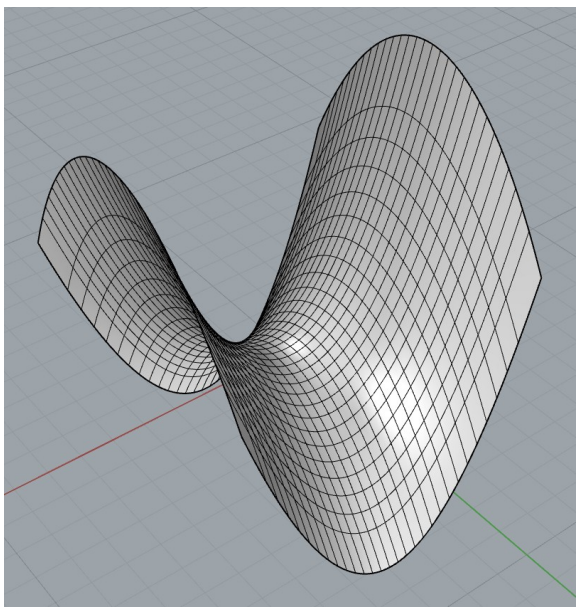
$$x = u \quad , \quad y = v \quad , \quad z = \frac{u^2}{5} - \frac{v^2}{5} \quad , \quad -5 \leq u \leq 5, -5 \leq v \leq 5.$$

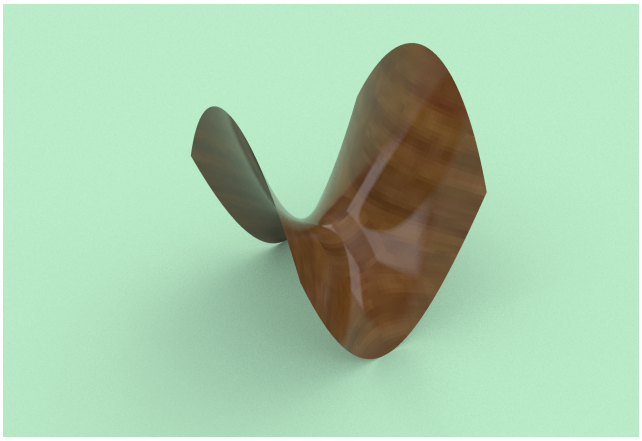
Il codice RhinoPython e le immagini ottenute in Geogebra e in Rhino sono riportati di seguito.

```
import rhinoscriptsyntax as rhino
import math

def draw_parametric_surface(function, param_range_u, param_range_v, \
```

```
num_points_u = 32, num_points_v = 32):
  points = []
  for i in range(num_points_u):
    u = param_range_u[0] + (param_range_u[1] - param_range_u[0]) * \
    float(i)/float(num_points_u - 1)
    for j in range(num_points_v):
      v = param_range_v[0] + (param_range_v[1] - param_range_v[0]) * \
      float(j)/float(num_points_v - 1)
      points.append( function(u,v) )
  rhino.AddSrfPtGrid((num_points_u, num_points_v), points)

def superf_Prova(u,v):
  return [u,v,u**2/3-v**2/5]

draw_parametric_surface(superf_Prova, (-5,5), (-5,5))
```







Esempio 13. Per chiudere questa galleria di esempi proponiamo alcune *Superfici minimali di Richmond*, precisamente quelle di grado 1, 2, 3, 4, 5. le equazioni parametriche sono le seguenti.

$$x = -\frac{\cos(t)}{r} - \frac{r^{2n+1}}{2n+1}\cos((2n+1)t) \quad , \quad y = \frac{\sin(t)}{r} + \frac{r^{2n+1}}{2n+1}\sin((2n+1)t) \quad , \quad z = \frac{2r}{n}\cos(nt),$$

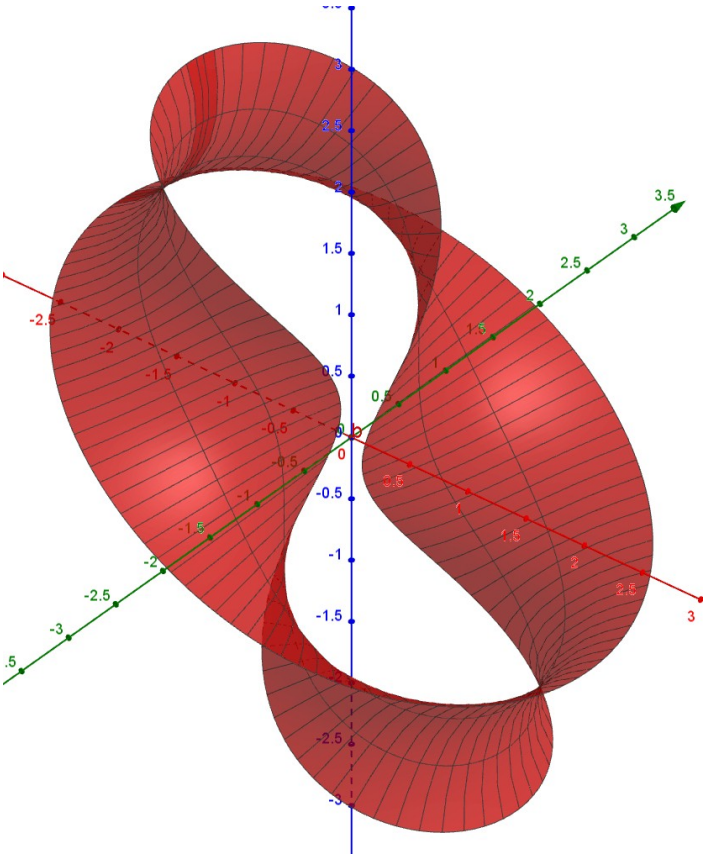
dove i due parametri r e t hanno i seguenti limiti:

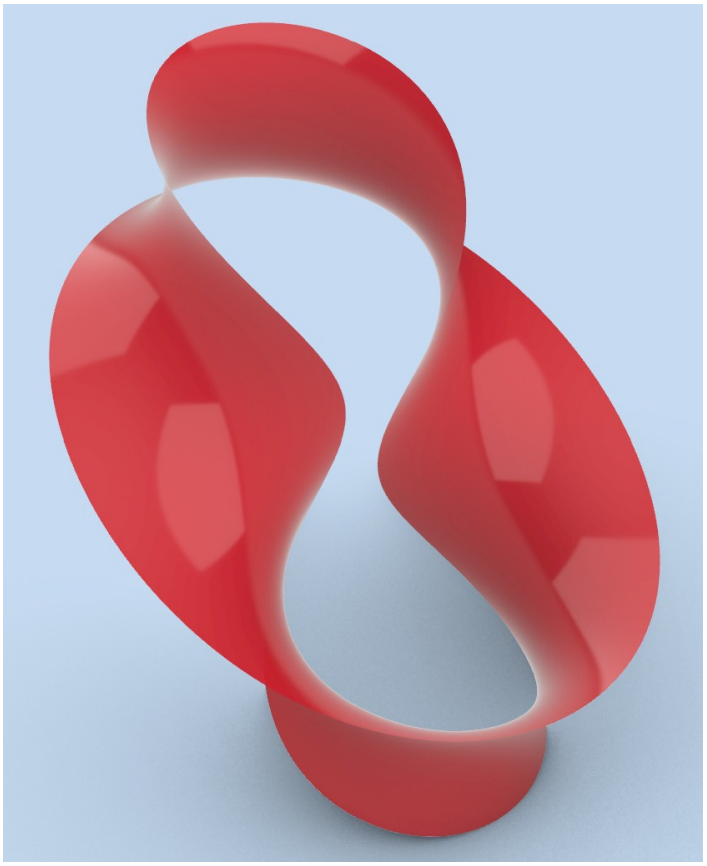
$$0.5 \le r \le 1.21 \quad , \quad 0 \le t \le 2\pi.$$

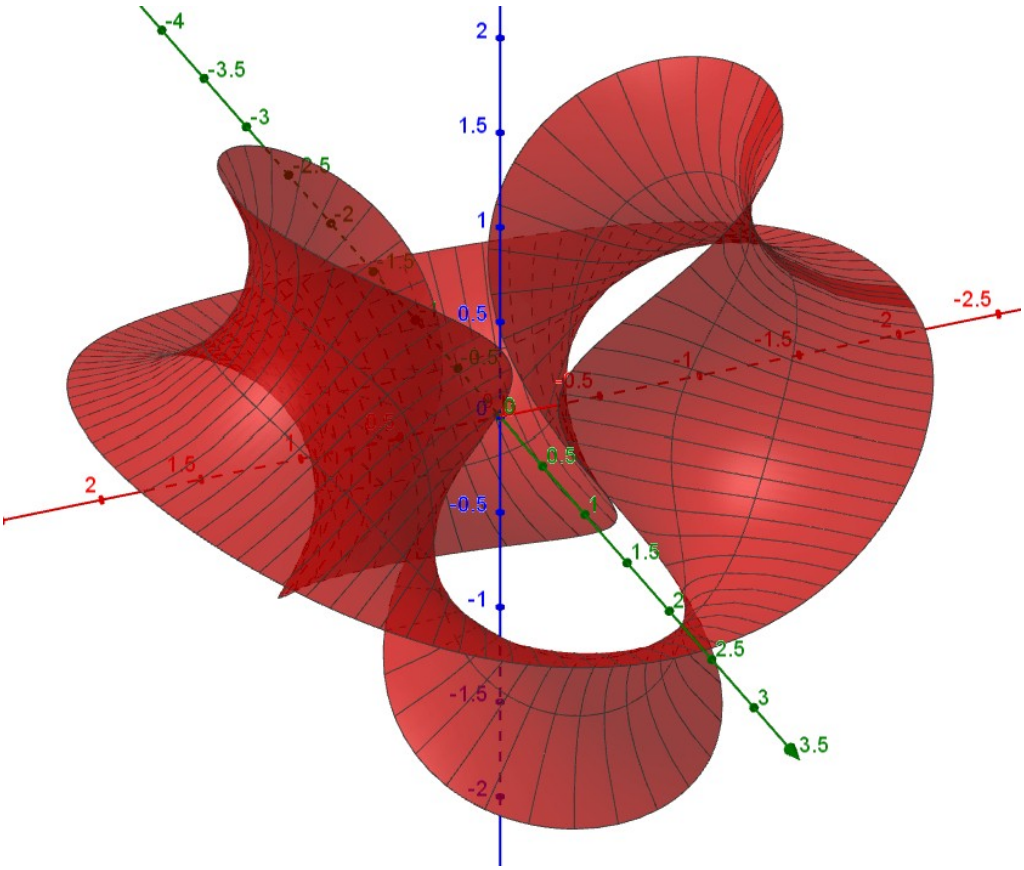
Il codice RhinoPython è il seguente, e non ha bisogno di commenti: è stato ottenuto dai precedenti semplicemente cambiando le equazioni parametriche della superficie. All'avvio viene chiesto di digitare il grado (numero intero!).

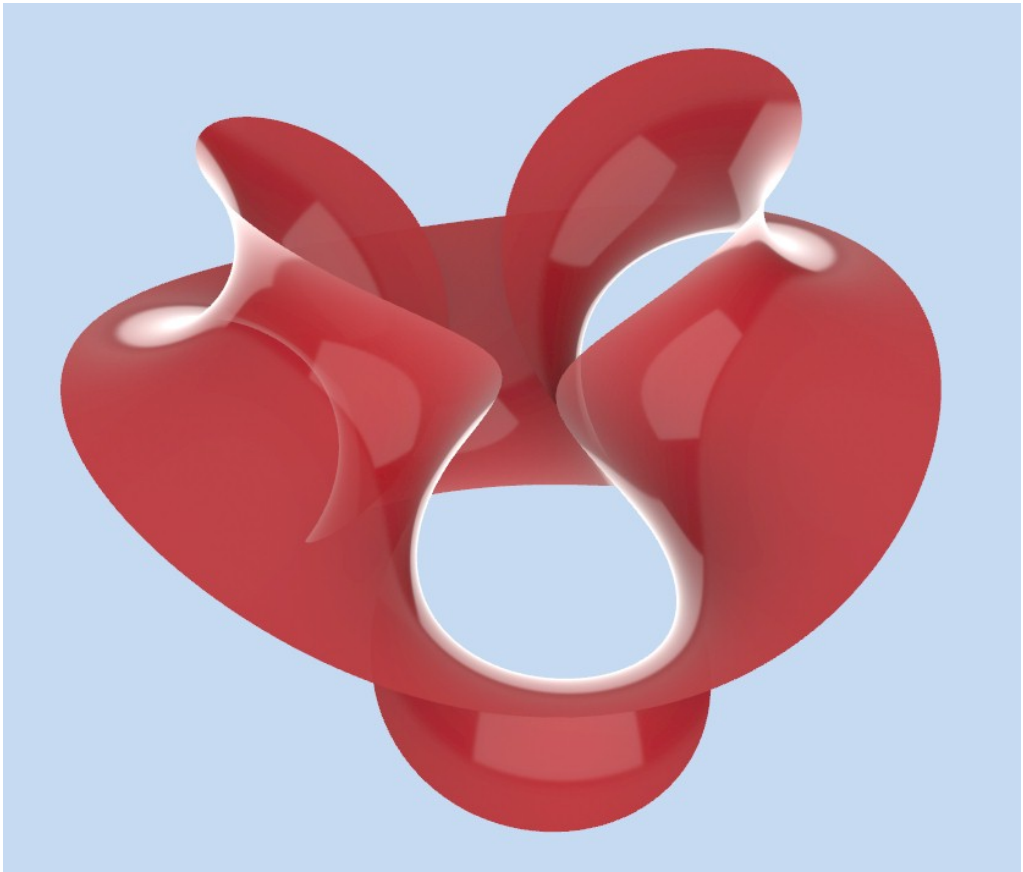
```
import rhinoscriptsyntax as rhino
import math

n=int(raw_input("Ordine (1<=n<=5, per i migliori risultati)"))

def draw_parametric_surface(function, param_range_u, param_range_v, \
num_points_u = 100, num_points_v = 100):
  points = []
  for i in range(num_points_u):
    u = param_range_u[0] + (param_range_u[1] - param_range_u[0]) * \
    float(i)/float(num_points_u - 1)
    for j in range(num_points_v):
      v = param_range_v[0] + (param_range_v[1] - param_range_v[0]) * \
      float(j)/float(num_points_v - 1)
      points.append( function(u,v) )
  rhino.AddSrfPtGrid((num_points_u, num_points_v), points)

def superf_Prova(r,t):
    return [-math.cos(t)/r-r**(2*n+1)/(2*n+1)* math.cos((2*n+1)*t),
    math.sin(t)/r+r**(2*n+1)/(2*n+1)*math.sin((2*n+1)*t),
    2*r**n/n* math.cos(n*t)]

draw_parametric_surface(superf_Prova, (0.5,1.21), (0,2*math.pi))
```

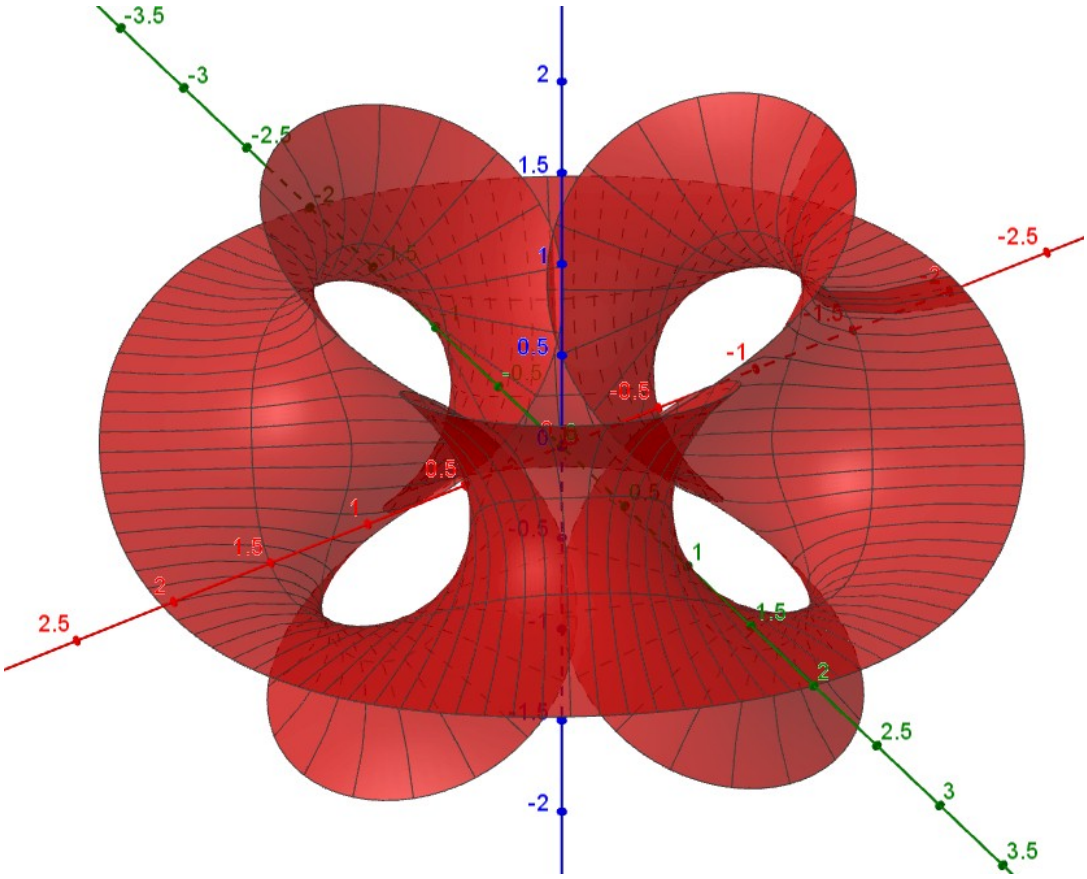
Di seguito proponiamo le immagini Geogebra e Rhino (con una renderizzazione veloce) delle superfici di Richmond di grado 1, 2, 3, 4, 5.

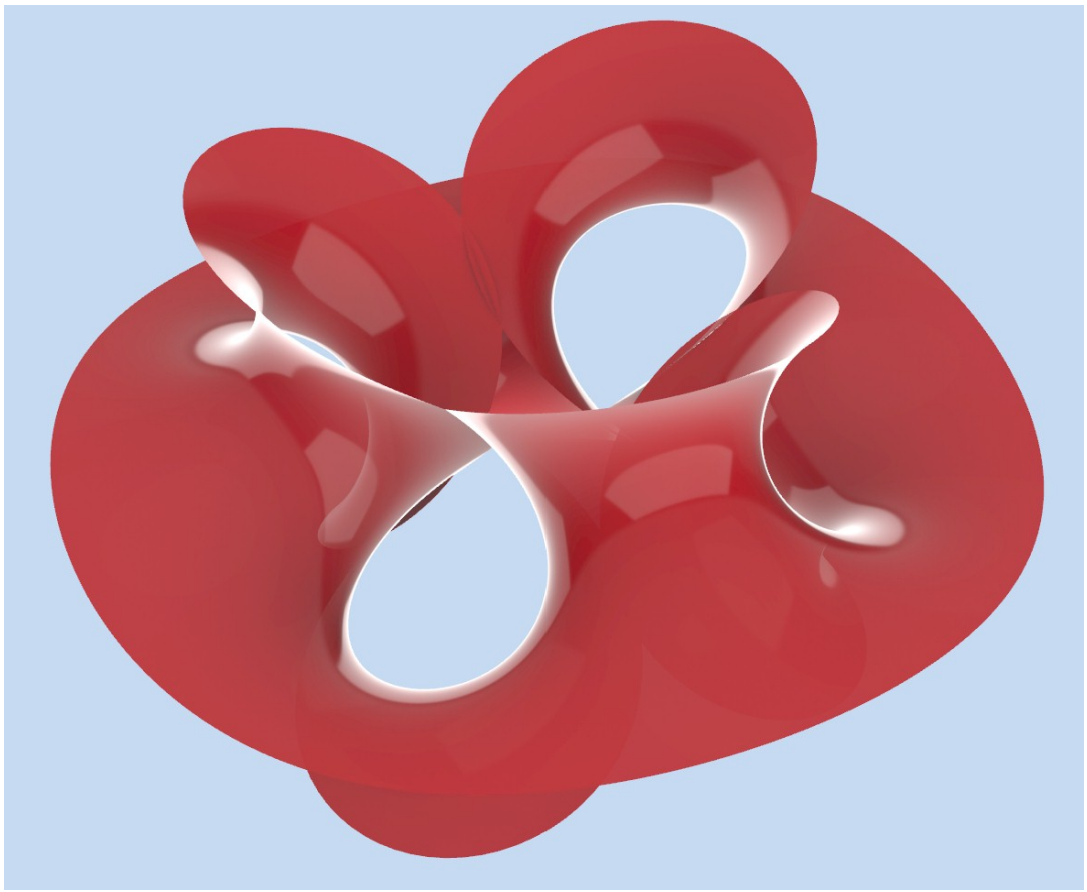


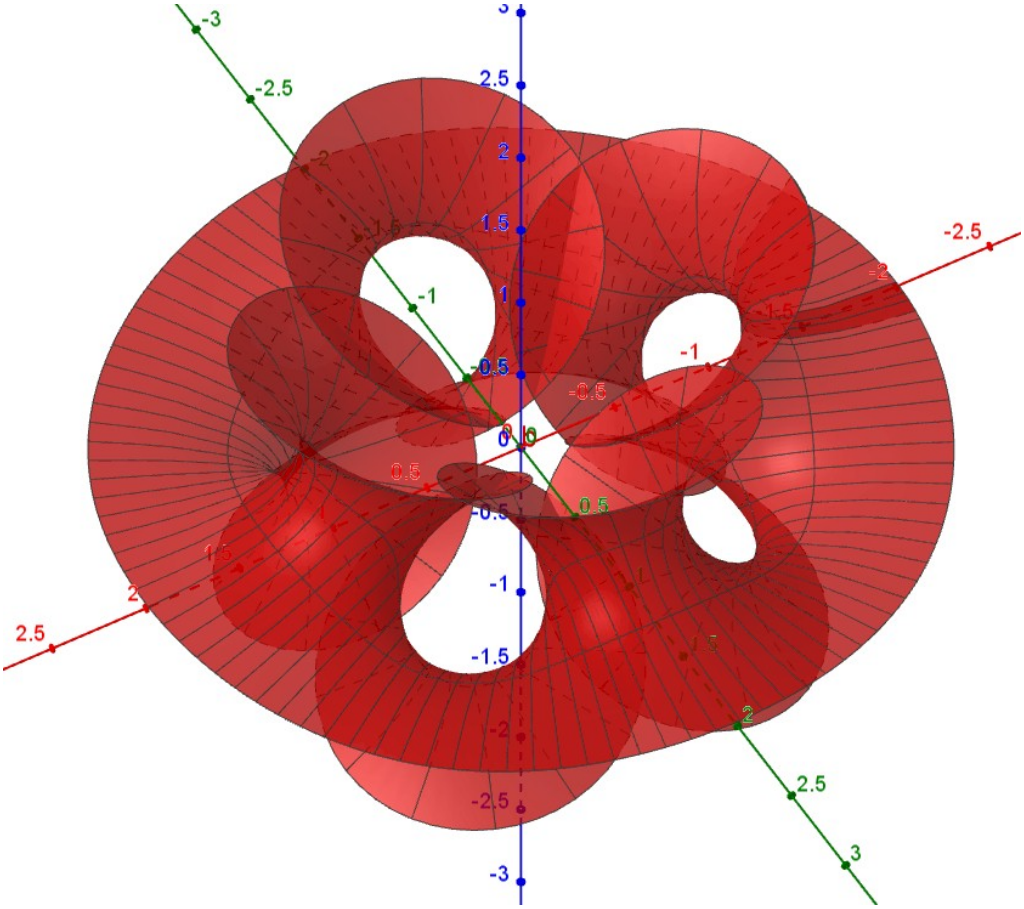



-4
-3.5
-3
-2.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
2
1.5
1
0.5
-0.5
-1
-1.5
-2
-2.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
2.5
3
3.5




-3.5
-3
-2.5
-2
2
1.5
1
0.5
-0.5
-1
-2.5
2.5
3
3.5




-3
-2.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
2.5
3