Performance-Verbesserung in Python

Das Scipy Modul

Weave

Mirko.Heger@informatik.uni-hamburg.de

Übersicht

- Einführung
- weave.inline()
- weave.blitz()

Einführung

- Einbindung von C oder C++ Code
  - 1,5 bis 30 mal schneller
- Kann auch langsamer sein
  - Viele Funktionsaufrufe

Einführung

- Weave.inline()
  - Führt C-Code in Python aus
- Weave.blitz()
  - Übersetzt Numpy-Ausdrücke in C++
- ext_tools
  - Bietet Klassen für Erweiterungs-Module

weave.inline()

weave.inline()

- Kompiliert und führt C/C++-Code aus
- `Inline(c_code, var_list)`
- return_val

```
>>>a = 1
>>>weave.inline('printf("%d\\n",a);', ['a'])
1
```

Erster Aufruf

- .cpp wird kompiliert und geladen,

  katalogisiert und ausgeführt
- Kompilierte Funktionen werden in Cache gespeichert
- Im neuen Interpreter
  - Modul laden
  - ausführen

Typprüfung

```
>>>weave.inline('printf(“%d\\n“,a);', ['a'])
>>>weave.inline(r'printf(“%d\n“,a);', ['a'])

>>>a = 'string'
>>>weave.inline(r'printf(“%d\n“,a);', ['a'])
32956972
```

- Weave kompiliert neue Funktion
  - `assert(type(a) == type(1))`

Argumente

- ```
  inline(code, arg_names, local_dict = None
         global_dict = None,
         force = 0,
         compiler = '',
         verbose = 0,
         support_code = None,
         type_factories = None,
         ***kw)
  ```

Distutil keywords können auch angegeben werden
  ```

Rückgabewerte

Dies funktioniert nicht

```
>>> a = 1
>>> weave.inline("a++;",['a'])
>>> a
2
```

Stattdessen

```
>>> a = 1
>>> weave.inline("a++;",['a'])
>>> a
1
```

Rückgabewerte

```
>>> a= [1,2]
>>> weave.inline("PyList_SetItem(a.ptr(),0,
                        PyInt_FromLong(3));",['a'])
>>> print a
[3, 2]
```

„Immutable types: tuples, string and number do not alter the Python variables“

return_val = Rückgabewert

Rückgabewert

```
def my_sum(a):
    n=int(len(a))
    code="""    int i;
    long int counter;
    counter =0;
    for(i=0;i<n;i++) {
        counter=counter+a(i); }
    return_val=counter;
    """
    err=weave.inline(code, ['a','n'],
      type_converters=converters.blitz,compiler='gcc')
    return err
```

weave.blitz()

weave.blitz()

- Funktioniert nur mit Numpy-Arrays und Python Skalaren

- Kompiliert Numpy-Ausdrücke
- 2-10 mal schneller, Erster Aufruf sehr langsam
- Neu Kompilieren für jeden Array-Typ
- Alle werden gespeichert

5 Punkte mitteln

```
a = np.random.random_integers(0,255,(512,512))
b = np.random.random_integers(0,255,(512,512))

a[1:-1,1:-1] =(b[1:-1,1:-1] + b[2:,1:-1] + b[:-2,1:-1]
            + b[1:-1,2:] +b[1:-1,:-2]) / 5.

expr =
"a[1:-1,1:-1] =(b[1:-1,1:-1] + b[2:,1:-1] + b[:-2,1:-1]
            + b[1:-1,2:] +b[1:-1,:-2]) / 5."
Weave.blitz(expr)
```

Geschwindigkeit

Numpy	0,46349 s	0,00853 s	0,19809
Blitz	0,05843 s	0,00552 s	0,07389
	78,95526 s	5,12341 s	
	(400 MHz Celeron Red Hat)	(rzmac21)	(2048x2048)

- Mit Numpy entwickeln
- Fertige Funktionen kompilieren

Einschränkungen

- ** power operator
- array broadcasting
- start:stop:step
- `>>> result = weave.blitz_eval("b + c + d")`
- Viele if / then
- Es funktionieren
    - Alle Numpy-Arrays
    - slice()

Einschränkungen

```
# 4 point average.

>>> expr = "u[1:-1, 1:-1] = (u[0:-2, 1:-1] +
  u[2:, 1:-1] + u[1:-1,0:-2] + u[1:-1,2:])*0.25 "

>>> u = zeros((5, 5), 'd'); u[0,:] = 100

>>> exec (expr)

>>> u

array([[ 100.,  100.,  100.,  100.,  100.],
       [   0.,   25.,   25.,   25.,    0.],
       [   0.,    0.,    0.,    0.,    0.],
       [   0.,    0.,    0.,    0.,    0.],
       [   0.,    0.,    0.,    0.,    0.]])
```

Einschränkungen

```
# 4 point average.
>>> expr = "u[1:-1, 1:-1] = (u[0:-2, 1:-1] +
  u[2:, 1:-1] + u[1:-1,0:-2] + u[1:-1,2:])*0.25 "
>>> u = zeros((5, 5), 'd'); u[0,:] = 100
>>> weave.blitz(expr)
>>> u
Array([[100.,100.    ,100.     , 100.     ,100.],
       [ 0., 25.    ,31.25    , 32.8125   ,  0.],
       [ 0.,  6.25  , 9.375   , 10.546875 ,  0.],
       [ 0.,  1.5625, 2.734375,  3.3203125,  0.],
       [ 0.,  0.    , 0.      ,  0.       ,  0.]])
```

Optimierung

- Nur einfache Operationen werden in Numpy schnell berechnet

```
a = 1.2 * b + c * d
```

- In Numpy

```
temp1 = 1.2 * b
temp2 = c * d
a = temp1 + temp2
```

- Allocation ist langsam
- Es werden drei Schleifen berechnet

Optimierung

```
a = 1.2 * b + c * d
```

```
for(int i = 0; i < M; i++)
    for(int j = 0; j < N; j++)
        a[i,j] = 1.2 * b[i,j] + c[i,j] * d[i,j]
```

- Keine temporären Arrays
- Nur 1 statt 3 Schleifen

Fazit

- Geschwindigkeitsvorteile

- Blitz ohne weitere Sprachkenntnisse nutzbar
- Leicht anzuwenden

Ende

Danke für die Aufmerksamkeit