Journal of Sedimentary Research, v. 84

APPENDIX 1

GRAIN-SIZE COMPONENTS AS MARKERS OF ORIGIN AND DEPOSITIONAL PROCESSES IN THE COASTAL ZONE OF THE GOLFE DU LION (MEDITERRANEAN SEA, FRANCE)

J. PAUL BARUSSEAU[1] AND RAPHAËL BRAUD[2]

[1] CEFREM, Université de Perpignan, 52 avenue Paul Alduy, Perpignan cedex, 66860, France
[2] NIDUS, 1 rue Claude Debussy, 37255 Montbazon, 37255, France
e-mail: brs@univ-perp.fr
DOI: http://dx.doi.org/10.2110/jsr.2014.48

```
# -*- coding: utf-8 -*-


"""

This program is free software: you can redistribute it and/or modify it
under the terms of the GNU Lesser General Public License as published by
the Free Software Foundation, either version 3 of the License, or(at your
option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU Lesser
General Public License for more details.

You should have received a copy of the GNU Lesser General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.#


This script helps to compute gaussian parameters that "fit" real data


@authors :Jean-Paul Barusseau (brs@univ-perp.fr) (Scientific authoring)
Raphaël Braud (rbraud@nidus.fr) (Script programming)


To use it, you must have a working installation of :


Python: http://www.python.org/ (Python Programming Language)


The following modules:
```

```
numpy: http://numpy.scipy.org/ (Scientific Computing Tools for Python )

matplotlib: http://matplotlib.org/ (2D plotting library)

"""


from __future__ import division # As strange as it seems, Python
default's behavior is to do euclidian division, let's switch to
"classical" division


from scipy import array, arange, sqrt, exp, sign

from scipy.optimize import leastsq

from scipy.special import erf

from numpy import interp

from math import pi


import matplotlib.pyplot as plt


#==================#

# HELPER FUNCTIONS #

#==================#


def differentiate(lst1, lst2):

    # This function computes a discrete diffentiation

    pairs1 = zip(lst1[:-1], lst1[1:])

    pairs2 = zip(lst2[:-1], lst2[1:])

    tmp1 = [(item[1]-item[0]) for item in pairs1]

    tmp2 = [(item[1]-item[0]) for item in pairs2]

    new2 = [item[1]/item[0] for item in zip(tmp1, tmp2)]

    tmp2 = [0]+new2

    tmp22 = [(a+b)/2 for a,b in zip(tmp2, tmp2[1:]+[0])]
```

```
    return lst1, tmp22


# ----


def np(x,y,z):

    # Normalisation

    return 100*abs(x)/(abs(x)+abs(y)+abs(z))


def phi(x, mu, sigma):

    # Gaussian Cumulative Distribution Function (cdf)

    return (1+erf((x-mu)/abs(sigma)/sqrt(2)))/2


def triple_phi(x, params):

    # Weighted gaussian CDF

    return np(params[2], params[5], params[8])*phi(x, params[0],
params[1])+np(params[5], params[2], params[8])*phi(x, params[3],
params[4])+np(params[8], params[2], params[5])*phi(x, params[6],
params[7])


def gauss(x,mu, sigma):

    # Gauss function

    return exp((-1*(x-mu)**2)/(2*abs(sigma)**2))/sqrt(2*pi*abs(sigma)**2)


def triple_gauss(x, params):

    # Weighted gaussian

    return np(params[2], params[5], params[8])*gauss(x, params[0],
params[1])+np(params[5], params[2], params[8])*gauss(x, params[3],
params[4])+np(params[8], params[2], params[5])*gauss(x, params[6],
params[7])


# ----
```

```
def sup(a):

    # Sup constraint function generator

    def foo(x):

        return x-a

    return foo


def inf(a):

    # Inf constraint function generator

    def foo(x):

        return a-x

    return foo


def supinf(a,b):

    # "Between" constraint function generator

    def foo(x):

        return sup(a)(x)*inf(b)(x)

    return foo


def c(x, func):

    # This function creates "some constraints" to avoid getting "stupid"
values

    return (1-sign(max(0, func(x))))*1000000


def errors(p,y,x):

    # Error estimation

    err = (y - triple_phi(x,p))+\

          c(p[0], supinf(-6,6))+\

          c(p[3], supinf(-6,6))+\
```

```
          c(p[6], supinf(-6,6))
    return err

#==================#
#   DATA LOADING   #
#==================#

lines="""-1,584962501;0,00
-1,321928095;0,60
-1;1,59
-0,678071905;2,95
-0,321928095;6,07
0;10,34
0,321928095;18,47
0,64385619;30,20
1,043943348;54,60
1,395928676;72,27
1,713118852;83,20
1,943416472;90,43
2,286304185;96,80
2,582079992;99,59
3;99,97
3,365871442;99,98
3,590744853;99,99
3,921390165;100,00""".split('\n')

datax=[]
datay=[]
```

```
for line in lines:
    item0,item1=line.split(';')
    item0=item0.replace(',', '.') # Fix separator issue
    item1=item1.replace(',', '.') # Fix separator issue
    datax.append(float(item0))
    datay.append(float(item1))

print datax
print datay

datax = array(datax)
datay = array(datay)

minx = min(datax)
maxx = max(datax)

distribx, distriby = differentiate(datax, datay)

interp_datax = arange(minx, maxx, (maxx-minx)/500)
interp_datay = interp(interp_datax, datax, datay)
interp_distriby = interp(interp_datax, datax, distriby)

# Arbitrary start value
middle_mu = interp(50, datay, datax)
min_mu = interp(20, datay, datax)
max_mu = interp(80, datay, datax)
bs = (maxx-minx)/6
```

```
params0 = array([middle_mu, bs, 75, min_mu, bs-0.2, 15, max_mu, bs+0.2,
10])



#===================#

#   OPTIMIZATION   #

#===================#


# Optimization using leastsq

pbest = leastsq(errors, params0, args=(interp_datay,interp_datax),
full_output=1, maxfev=100000)


bestparams = pbest[0]

cov_x = pbest[1]

print 'Best parameters :'

print '======================'

print 'mu1    :', bestparams[0]

print 'sigma1 :', abs(bestparams[1])

print 'weight1 :', np(bestparams[2], bestparams[5], bestparams[8])

print 'mu2    :', bestparams[3]

print 'sigma2 :', abs(bestparams[4])

print 'weight2 :', np(bestparams[5], bestparams[2], bestparams[8])

print 'mu3    :', bestparams[6]

print 'sigma3 :', abs(bestparams[7])

print 'weight3 :', np(bestparams[8], bestparams[2], bestparams[5])

print '-------------------------------------'


#===================#
```

```
# DRAWING RESULTS  #

#==================#


# Feel free to modify the following four lines to suit your graphical
needs :

LINEWIDTH = 2

MARKERSIZE = 3

MARKERWIDTH = 5

LEGENDLOCATION = 'upper left'


f = plt.figure()


graphe1 = f.add_subplot(121)

graphe2 = f.add_subplot(122)


datafit = triple_phi(interp_datax, bestparams)

fun1 = 100*phi(interp_datax, bestparams[0], bestparams[1])

fun2 = 100*phi(interp_datax, bestparams[3], bestparams[4])

fun3 = 100*phi(interp_datax, bestparams[6], bestparams[7])


graphe1.plot(datax, datay, 'x', markersize=MARKERSIZE,
markeredgewidth=MARKERWIDTH, label='data')

graphe1.plot(interp_datax, datafit, 'r', linewidth=LINEWIDTH,
label='fit')

graphe1.plot(interp_datax, fun1, 'b', linewidth=LINEWIDTH, label='g1
(%.2f/%.2f/%.2f)'%(bestparams[0], abs(bestparams[1]), np(bestparams[2],
bestparams[5], bestparams[8])))

graphe1.plot(interp_datax, fun2, 'g', linewidth=LINEWIDTH, label='g2
(%.2f/%.2f/%.2f)'%(bestparams[3], abs(bestparams[4]), np(bestparams[5],
bestparams[2], bestparams[8])))
```

```
graphe1.plot(interp_datax, fun3, 'Y', linewidth=LINEWIDTH, label='g3
(%.2f/%.2f/%.2f)'%(bestparams[6], abs(bestparams[7]), np(bestparams[8],
bestparams[2], bestparams[5])))

graphe1.legend(loc=LEGENDLOCATION)

graphe1.set_title(u'Cumulative functions')

graphe1.grid(True)


fitx, fity = differentiate(interp_datax, datafit)

x1, y1 = differentiate(interp_datax, fun1)

x2, y2 = differentiate(interp_datax, fun2)

x3, y3 = differentiate(interp_datax, fun3)


graphe2.plot(distribx, distriby, 'x', markersize=MARKERSIZE,
markeredgewidth=MARKERWIDTH, label='data')

graphe2.plot(fitx, fity, 'r', linewidth=LINEWIDTH, label='fit')

graphe2.plot(x1, 100*gauss(x1, bestparams[0], bestparams[1]), 'b',
linewidth=LINEWIDTH, label='g1')

graphe2.plot(x1, 100*gauss(x1, bestparams[3], bestparams[4]), 'g',
linewidth=LINEWIDTH, label='g2')

graphe2.plot(x1, 100*gauss(x1, bestparams[6], bestparams[7]), 'y',
linewidth=LINEWIDTH, label='g3')

graphe2.legend(loc=LEGENDLOCATION)

graphe2.set_title(u'Distributions')

graphe2.grid(True)


plt.show()
```