PART III

Python codes

This appendix contains programs, functions or code fragments written in Python. Each code is referred to in the text; the page where the reference is made is given in the header.

First some general instructions are given on how to work with these codes. Python is a general-purpose interpretative language, for which interpreters are available for most platforms, including Windows. Python is in the public domain and interpreters are freely available.[1] Most applications in this book use a powerful numerical array extension *NumPy*, which also provides basic tools in linear algebra, Fourier transforms and random numbers.[2] Although Python version 3 is available, at the time of writing NumPy requires Python version 2, the latest being 2.6. In addition, applications may require the scientific tools library *SciPy*, which relies on *NumPy*.[3] Importing *SciPy* automatically implies the import of *NumPy*.

Users are advised first to download Python 2.6, then the most recent stable version of *NumPy*, and then *SciPy*. Further instructions for Windows users can be found at www.hjcb.nl/python.

There are several options to produce plots, for example *Gnuplot.py*,[4] based on the *gnuplot* package[5] or *rpy*[6] based on the statistical package "R."[7] But there are many more.[8] Since the user may find it difficult to make a choice, we have added yet another, but very simple to use, plotting module called `plotsvg.py`. It can be downloaded from the author's website.[9] Its plotting routines produce SVG output files (Scalable Vector Graphics, a W3C standard) that can be viewed by an SVG-enabled browser. Among others, the Firefox, Opera and Google Chrome browsers (but not Internet Explorer) have native SVG support. While customized plots are possible, automatic plots of functions, points and cumulative distributions can be simply made. For example, the following code produces automatic display of the cumulative distribution of 200 normally distributed random numbers on a probability scale (on which normal distributions should give a straight line):

Python code 0.1 *Demo plotsvg*

[1] www.python.org.
[2] www.scipy.org/numpy.
[3] www.scipy.org/SciPy.
[4] http://gnuplot-py.sourceforge.net/.
[5] www.gnuplot.info/.
[6] http://rpy.sourceforge.net/.
[7] www.r-project.org/.
[8] See http://wiki.python.org/moin/NumericAndScientific/Plotting.
[9] www.hjcb.nl/python/.

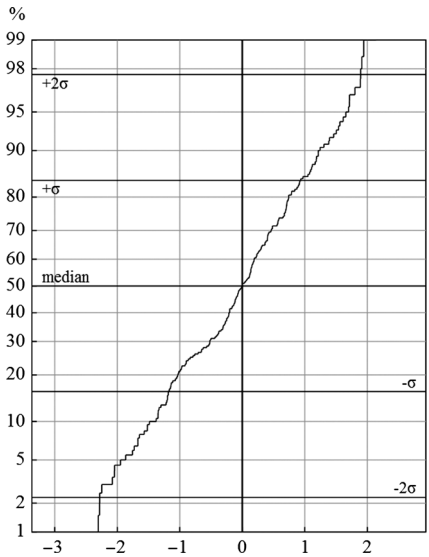

Figure P.1 Plotting demo: a cumulative plot on a probability scale of 200 random samples from a normal distribution.

```
from scipy import *
from plotsvg import *
r=randn(200)
autoplotc(r,yscale='prob')
```

with Fig. P.1 as a result.

Comments:
The module `plotsvg.py` defines a class `Figure()` with methods `frame()` to define a frame with titles allowing for logarithmic and probability scales, `plotp()` to plot a series of points with or without connecting lines and error bars, `plotc()` to plot cumulative distributions, `plotf()` to plot functions and a number of utilities like `addtext()` and `addobject()`. There are also stand-alone programs like `autoplotp()` for quick graphs.

Another module that can be downloaded from the author's website is *physcon.py*. This module contains most of the fundamental physical constants as SI values in the form of a dictionary. In addition the following symbols are defined as the SI value (float): `alpha, a_0, c, e, eps_0, F, G, g_e, g_p, gamma_p, h, hbar, k_B m_d, m_e, m_n, m_p, mu_B, mu_e, mu_N, mu_p, mu_0, N_A, R, sigma, u`.

Python code 0.2 *Demo physcon*

```
import physcon as pc
pc.help()
```

This will list available functions, variables and keys

```
pc.descr('avogadro')
```

This will describe avogadro: name, symbol, value, standard error, relative s.d., unit, data source

```
N=pc.N_A
```

This will assign the value $6.02214179e+023$ *to N*

Python code 2.1 (page 6) *Generate and plot Fig. 2.2*

```
from scipy import *
x = 8.5 + randn(30)
xr = x.sort().round(2)
from plotsvg import *
autoplotc(xr,title='Cumulative distribution')
autoplotc(xr,title='Cumulative distribution',\
          yscale='prob')
```

Python code 2.2 (page 8) *Generate histogram of Fig. 2.3*

```
from plotsvg import *
hisx = [6.5,7.5,8.5,9.5,10.5,11.5]
hisy = [1,7,8,10,2,2]
f = Figure()
f.frame([6,12],title='Histogram')
f.plotp([hisx,hisy],symbol='halfbar',\
    symbolfill=Darkgrey,symbolstroke=Black)
f.show()
```

Python code 2.3 (page 8) *Some array methods and functions*

```
from scipy import *
n=alen(x)      # assigns length of array x to n
m=x.mean()     # assigns mean of x to m
msd=x.var()    # assigns mean squared deviation
               # of x to msd
```

```
rmsd=x.std()  # assigns root mean squared deviation
              # of x to rmsd
```

Python code 2.4 (page 8) *Generate percentiles of a given data set*

```
from scipy import *
from scipy import stats
def percentiles(x, per=[1,5,10,25,50,75,90,95,99]):
# x = 1D-array
# per = list of percentages
    scores=zeros(len(per),dtype=float)
    i=0
    for p in per:
        scores[i]=stats.scoreatpercentile(x,p)
        i++
    return scores
```

Comments:
The scipy.stats function `scoreatpercentile(x,p)` gives the p-th percentile, i.e. the value $\geq p\%$ of the data and $\leq (100-p)\%$ of the data. If that is not a single value, linear interpolation is used.

Python code 2.5 (page 16) *Plot on a logarithmic scale*

```
from plotsvg import *
time=[20.,40.,60.,80.,100.,120.,140.,160.,180.]
conc=[75.,43.,26.,16.,10.,5.,3.5,1.8,1.6]
err=[4.,3.,3.,2.,2.,1.,1.,1.]
f=Figure()
f.frame([[0,200],[1,100]],xlabel='time <i>t</>/s',\
    ylabel='concentration <i>c</i>/mmol L<sup>-1\
    </sup>', yscale='log')
f.plotp([time,conc],ybars=err)
f.show()
```

Comments:
This code produces Fig. 2.7 with the module `plotsvg` available from the author's web site www.hjcb.nl/python/. An SVG file is produced and displayed by a suitable browser (Firefox, Opera, Google Chrome, but not Internet Explorer).

Python code 3.1 (page 25) *Monte Carlo generation of equilibrium constant*

```
from scipy import *
from plotsvg import *
def Keq(a,b,V1,V2,x):              # define equilibrium
                                   # constant
    V=V1+V2
    K=x/((a/V-x)*(b/V-x))*1000.  # convert to L/mol
    return K
n=1000                             # set number of
                                   # samples
a0=5.0; a=a0+randn(n)*0.2          # mmol
b0=10.0; b=b0+randn(n)*0.2         # mmol
V10=0.1; V1=V10+randn(n)*0.001     # L
V20=0.1; V2=V20+randn(n)*0.001     # L
x0=5.0; x=x0+randn(n)*0.35         # mmol/L
K=Keq(a,b,V1,V2,x)                 # L/mol (array of
                                   # K-values)
K0=Keq(a0,b0,V10,V20,x0)           # L/mol (K at
                                   # central values)
print 'K from values without noise = %g' % (K0)
print 'number of samples = %d' % (n)
print 'average and std of K = %g +/- %g' %\
      (K.mean(), K.std())
```

Generate Figure 3.1:

```
f=Figure()
f.size=[5500,6400]
f.frame([[4.,7.5],[0,100]],title='Equilibrium\
        constant', yscale='prob',\
        xlabel='<i>K</><sub>eq</sub>/L mol<sup>-1\
        </sup>', ylabel='cumulative probability\
        distribution')
f.plotc(K)
f.show()
```

Python code 4.1 (page 36) *Generate binomial functions for Figures 4.1, 4.2 and 4.3*

```
from scipy import *
from scipy import stats
```

Probability of finding k "heads" in 10 coin tossings:

```
def fun1(k): return stats.binom.pmf(k,10,0.5)
```

Probability of finding k "6"'s in 60 dice throws:

```
def fun2(k): return stats.binom.pmf(k,60,1./6.)
```

Probability of exceeding k correct guesses in 25 Zener cards:

```
def fun3(k): return stats.binom.sf(k,25,0.2)
```

Generate Fig. 4.1:

```
from plotsvg import *
x1=arange(11); y1=fun1(x1)
f=Figure()
f.frame([[-1,11],[-0.02,0.27]],\
    title="Binomial 10 coin tosses",\
    xlabel="nr of heads", ylabel="probability")
f.plotp([x1,y1], symbol='halfbar',\
    symbolstroke=Black, symbolfill=Darkgrey)
f.show()
```

Generate Fig. 4.2:

```
x2=arange(27); y2=fun2(x2)
f=Figure()
f.frame([[-1,26],[-0.01,0.15]],\
    title="Binomial 60 dice throws",\
    xlabel="nr of 6's", ylabel="probability")
f.plotp([x2,y2],symbol='halfbar',\
    symbolstroke=Black,symbolfill=Darkgrey)
f.show()
```

Generate Fig. 4.3:

```
x3=arange(16); y3=fun3(x3)
f=Figure()
f.frame([[0,12],[0,1]],\
    title="Binomial 25 Zener cards",\
    xlabel="nr correct", ylabel="survival\
    (1 - c.d.f.)")
f.plotp([x3,y3],symbol='dot',lines=Black)
f.show()
```

Python code 4.2 (page 47) *Generate Weibull distribution functions*

```
from scipy import stats
pdf=stats.weibull_min.pdf
cdf=stats.weibull_min.cdf
def f1(t):
    if (t<0.001):
        return None
    else: return pdf(t,0.5)
def g1(t): return cdf(t,0.5)
def f2(t): return pdf(t,1.)
def g2(t): return cdf(t,1.)
```

Comments:
The scipy module stats contains a large number of distribution functions. The pdf for negative c is infinity for $t = 0$, which should be excluded. The pdf's f_1, f_2 and the cdf's g_1, g_2 are suitable for plotting.

Python code 5.1 (page 67) *The bootstrap method: Generate averages from random samples*

```
def bootstrap(x,n,dof=0):
# x = 1D-array of input samples
# n = nr of averages generated
# dof = nr of degrees of freedom.
# If not specified, dof=len(x)
# returns 1D-array of averages
    from scipy import stats
    nx=len(x)
    if (dof==0): nu=nx
    else: nu=dof
    result=zeros(n,dtype=float)
    for i in range(n):
        index=stats.randint.rvs(0,nx,size=nu)
        result[i]=x[index].mean()
    return result
```

Comments:
The `randint.rvs(min,max,size=n)` function of the scipy package `stats` produces an array of n random integers >= `min` and < `max`.

`x[index]` produces an array containing the values of `x[i]`, where `i` are all values of the integer array `index`.

If `dof` is not specified, the averages are taken over as many items as there are in the input array `x`; this yields the biased bootstrap distribution. The unbiased distribution can be approximated by setting `dof` equal to the length of `x` minus 1.

Python code 6.2 (page 67) **Report**: *a program that analyzes a set of independent data.*

```
from scipy import *
from plotsvg import *
def report(data,figures=True):
    '''
Function. Reports statistics on single uncorrelated
-----------------------------------------------------
data series.
------------
arguments:
  data:         list or array [y] or [x,y] or [x,y,
                  sig] of data; if [y] then x=arange
                  (len(y))
                sig = sd of y-value; if given,
                chisq test reported, if sig not
                given, equal weights are assumed
  figures=True  if True, figures are produced and
                displayed
returns:        [[mean,sdmean,var,sd],[a,siga,b,
                  sigb]] (fit ax+b)
Remarks:
  report of properties (average, msd, rmsd) and of
  estimates (mean, variance, sd, skewness, excess)
  and their accuracies is printed (skewness and
  excess only if relevant). Figures produced:
    figdata.svg: data points with error bars and
    linear fit;
    figcum.svg: cumulative plot on probability
    scale.
  Outliers are identified. If s.d. sig are given, a
  chi-squared analysis is produced. A linear
  regression  drift analysis is done.
'''
    import os
    from scipy import stats
    # unify data structure:
    data=array(data)
    dimension=array(data).ndim
    weights=False
    if (dimension==1):
        n=len(data)
        xy=array([arange(n),data])
```

```
elif (dimension==2):
    n=len(data[0])
    if (len(data)==2):
        xy=array(data)
    elif (len(data)==3):
        xy=array(data[:2])
        weights=True
        w=1./array(data[2])**2
    else:
        print 'ERROR: wrong data length'
        print 'report aborted'
        return 0
else:
    print 'ERROR: wrong data dimension'
    print 'report aborted'
    return 0
# compute properties
if weights:
    wtot=w.sum()
    xav=(xy[0]*w).sum()/wtot
    yav=(xy[1]*w).sum()/wtot
else:
    wtot=float(n)
    xav=xy[0].mean()
    yav=xy[1].mean()
xdif=xy[0]-xav
ydif=xy[1]-yav
if weights:
    ssq=(w*ydif**2).sum()
else:
    ssq=(ydif**2).sum()
msd=ssq/wtot
rmsd=sqrt(msd)
var=msd*n/(n-1.)
sd=sqrt(var)
sdmean=sd/sqrt(n)
ymin=xy[1].min()
yminindex=xy[1].argmin()
ymax=xy[1].max()
ymaxindex=xy[1].argmax()
# linear regression:
if weights:
    xmsd=(w*xdif**2).sum()/wtot
    a=(xdif*ydif*w).sum()/wtot/xmsd
```

```
        b=yav-a*xav
        S=(w*((xy[1]-a*xy[0]-b)**2)).sum()
        siga=sqrt(S/(wtot*(n-2.)*xmsd))
        sigb=siga*sqrt((w*(xy[0]**2)).sum()/wtot)
    else:
        xmsd=(xdif**2).mean()
        a=(xdif*ydif).mean()/xmsd
        b=yav-a*xav
        S=((xy[1]-a*xy[0]-b)**2).sum()
        siga=sqrt(S/(n*(n-2.)*xmsd))
        sigb=siga*sqrt((xy[0]**2).mean())
    # produce figures
    if figures:
        def fun0(x): return yav
        def fun1(x): return yav-sd
        def fun2(x): return yav+sd
        def fun3(x): return yav-2.*sd
        def fun4(x): return yav+2.*sd
        def fun5(x): return a*x+b
        f=Figure()
        f.frame([[xy[0,0],xy[0,-1]],[ymin,ymax]],\
                 title='input data')
        f.plotf(fun0,color=Red)
        f.plotf(fun1,color=Red)
        f.plotf(fun2,color=Red)
        f.plotf(fun3,color=Red)
        f.plotf(fun4,color=Red)
        f.plotf(fun5,color=Green)
        if weights:
            f.plotp(xy,ybars=data[2],symbolfill=\
                    Blue, barcolor=Blue)
        else:
            f.plotp(xy,lines=Blue,symbol='')
        f.addtext([890,4140],\
            '<small>red lines: mean, &#177; &#963;,\
            &#177; 2&#963; </small>',fill=Red)
        f.addtext([4890,4140],\
            '<small>green line: linear\
              regression</small>', align='r',\
              fill=Green)
        f.show(filename='figdata.svg')
        os.startfile('figdata.svg')
        print 'figdata.svg is now displayed by\
               your browser'
```

```
        f=Figure()
        f.size=[5500,6400]
        f.frame([[(1.1*ymin-0.1*ymax),(-0.1*ymin\
                  +1.1*ymax)],\
                 [0,100]], title='cum.distribution\
                 of data', yscale='prob')
        f.plotc(xy[1])
        f.show(filename='figcum.svg')
        os.startfile('figcum.svg')
        print 'figcum.svg is now displayed by your\
               browser'
    print '\nStatistical report on uncorrelated\
           data series'
    print '\nProperties:'
    print 'nr of elem. = %5d' % n
    print 'average     = %10.6g' % (yav)
    print 'msd         = %10.6g' % (msd)
    print 'rmsd        = %10.6g' % (rmsd)
    print '\nEstimates'
    print 'mean        = %10.6g +/- %8.4g' % (yav,\
        sdmean)
    if weights: print '*)'
    print '\nvariance    = %10.6g +/- %8.4g' %\
        (var, var*sqrt(2./(n-1.)))
    print 'st. dev     = %10.6g +/- %8.4g' %\
        (sd, sd/sqrt(2.*(n-1.)))
    if weights:
        print '*) this standard uncertainty in the
              mean is\
              derived from the data variance'
        print "   derived from the supplied sigma's
              it is", "%8.4g" % (wtot**(-0.5))
        print '  Choose the more reliable, or else\
                the larger value.'
        print '   See also the chi-square analysis\
                below.'
    # skewness and excess only if weights=False
    if not weights:
        if (n>=20):
            skew=(xy[1]**3).sum()/(n*var*sd)
            print 'skewness    = %10.6g +/- %8.4g'\
                   % (skew, sqrt(15./n))
        else: print 'skewness: insufficient\
        statistics'
```

```
    if (n>=100):
        exc=(xy[1]**4).sum()/(n*var*var)-3.
        print 'excess      = %10.6g +/- %8.4g'\
              % (exc, sqrt(96./n))
    else: print 'excess: insufficient\
    statistics'
# outliers and their probabilities
ydevmax=(ymax-yav)/sd
ydevmin=(yav-ymin)/sd
Fmax=stats.norm.cdf(-ydevmax)
probmax=100.*(1.-(1.-Fmax)**n)
Fmin=stats.norm.cdf(-ydevmin)
probmin=100.*(1.-(1.-Fmin)**n)
print '\nPossible outliers:',
if ((probmax>5.) and (probmin>5.)):
    print '  (there are no significant\
    outliers with p<5%)'
else:
    print '(there are significant outliers\
    with p<5%)'
print 'largest element y[%d]=%10.6g deviates\
    +%5.2g stand.',\
      'dev. from mean'  % (ymaxindex,ymax,\
      ydevmax)
print 'prob. to obtain a higher value at least\
    once is',\
       '%4.3g %%' % (probmax)
print 'smallest element y[%d]=%10.6g deviates\
    -%5.2g stand.',\
       'dev from mean' % (yminindex,ymin,\
       ydevmin)
print 'prob. to obtain a lower value at least\
once is',\
       '%4.3g %%' % (probmin)
# chi-square analysis if weight=True:
if weights:
    nu=n-1
    F=stats.chi2.cdf(S,nu)
    print '\nChi-square analysis:'
    print 'chi^2 (sum of weighted square dev.)\
          = %10.6g' % (ssq)
    print 'cum. prob. for chi^2 = %5.3g %%' %\
          (100.*F)
    if (F<.1):
```

```
            print "chi^2 is low! Did you\
                   overestimate the\
                   supplied sigma's?"
            print 'Or did you fit the original\
                   data too closely\
                   with too many parameters?'
        elif (F>.9):
            print "chi^2 is high! Did you neglect\
                   an error source\
                   in the supplied sigma's?"
            print 'Or did the data result from a\
                   bad fitting procedure?'
        else:
            print 'cum. probab. of chi^2 is\
                   reasonable (between\
                   10% and 90%).'
            print "The spread in the data agrees\
                   with the supplied sigma's."
    # Significance of drift
    print '\nLinear regression: y=a*x+b'
    print 'a=%10.6g +/- %10.6g; b=%10.6g +/-\
           %10.6g' % (a,siga,b,sigb)
    Pdrift=2.*(1.-stats.norm.cdf(abs(a)/siga))
    print '\nNormal test on significance of\
           slope a'
    print 'Probability to obtain at least this\
           drift by random\
           fluctuation is %8.3g %%' %\
           (100.*Pdrift)
    print '\nF-test on significance of linear\
           regression:'
    print 'sum of square deviations reduced from\
           %7.5g to %7.5g'\
           % (ssq,S)
    ypred=a*xy[0]+b
    ypmean=ypred.mean()
    if weights:
        SSR=(w*(ypred-ypmean)**2).sum()
    else:
        SSR=((ypred-ypmean)**2).sum()
    Fratio=SSR/(S/(n-1))
    Fcum=stats.f.cdf(Fratio,1,n-1)
    print 'The F-ratio SSR/(SSE/(n-1)) = %7.3g' %\
           (Fratio)
```

```
    print 'The cum. prob. of the F-distribution is\
           %8.5g' % (Fcum)
    print 'Probability to obtain this fit (or\
           better) by random',\
          'fluctuation is %8.3g %%' % (100.*\
           (1.-Fcum))
    if ((Fcum>0.9) and (Pdrift<0.1)):
        print '\nThere is a significant drift (90%\
              conf. level)'
    else:
        print '\nThere is no significant drift\
              (90% conf. level)'
    print
    return [[yav,sdmean,var,sd],[a,siga,b,sigb]]
```

Comments:
This program can be downloaded from www.hjcb.nl/python). Look for recent updates. Two plots are automatically generated and displayed by the standard browser. Make sure that the .svg mime type starts your SVG-enabled browser.

Python code 6.1 (page 81) *Fit a number of harmonics to data points*

```
from scipy import optimize
# data from compass corrections:
x=arange(0.,365.,15.)
y=array([-1.5,-0.5,0.,0.,0.,-0.5,-1.,-2.,-3.,-2.5,\
         -2.,-1.,0., 0.5,1.5,2.5,2.0,2.5,1.5,0.,\
         -0.5,-2.,-2.5,-2.,-1.5])
def fitfun(x,p):
    phi=x*pi/180.
    result=p[0]
    for i in range(1,5,1):
        result=result+p[2*i-1]*sin(i*phi)+p[2*i]\
        *cos(i*phi)
    return result                  # result is
                                   # array like x
def residuals(p): return y-fitfun(x,p)
pin=[0.]*9                         # initial
                                   # parameter guess
output=optimize.leastsq(residuals,pin)
```

```
pout=output[0]                          # optimized
                                        # parameters
def fun(x): return fitfun(x,pout)       # suitable for
                                        # plotting
```

Comments:
For simplicity a general least-squares optimization is used, although the optimization problem is linear here. The function to be fitted is $p_0 + p_1 \sin\phi + p_2 \cos\phi + p_3 \sin 2\phi + p_4 \cos 2\phi + p_5 \sin 3\phi + p_6 \cos 3\phi + p_7 \sin 4\phi + p_8 \cos 4\phi$. In view of the inaccuracy of the corrections, a fit with still higher harmonics is an overkill. The minimizer `leastsq` of the scipy package `optimize` is used.

Python code 7.1 (page 94) *Nonlinear least-squares fit, urease kinetics*

```
from scipy import optimize
S = array([30.,60.,100.,150.,250.,400.])
v = array([3.09,5.52,7.59,8.72,10.69,12.34])
```

A. Minimization using leastsq:

```
lsq = optimize.leastsq
def residuals(p):
    [vmax,Km]=p
    return v-vmax*S/(Km+S)
output = lsq(residuals,[15,105])
pout = output[0]
```

B. Minimization using fmin_powell:

```
def fun(S,p):
    [vmax,Km]=p
    return vmax*S/(Km+S)
def SSQ(p): return ((v-fun(S,p))**2).sum()
pin = [15,105]
pout = optimize.fmin_powell(SSQ,pin)
```

Comments:
The function `leastsq` requires as input an array of residues as a function of the parameters; it minimizes its sum of squares. The function `fmin_powell` adjusts the parameters in `fun` such that SSQ is a minimum. The new parameters `pout` are returned. The last line may be repeated with the new parameters as input. These minimization procedures do not need any derivatives. Check SSQ by the command `print SSQ(pout)`. In this example method A gives a more accurate result than method B.

Python code 7.2 (page 96) *Generate the cumulative probability for given* χ^2

```
from scipy import stats
cdf=stats.chi2.cdf
ppf=stats.chi2.ppf
```

Comments:
The function `cdf(x,ν)` gives the probability that χ^2, i.e., the sum of ν squares of random samples from a normal distribution, is less than x. For example, for 15 degrees of freedom, the probability of finding $\chi^2 \leq 10.5$ is given by
`print cdf(10.5,15)`
and the probability of finding $\chi^2 \geq 18.3$ is given by
`1.-cdf(18.3,15)`
The values of χ^2 for which the probabilities that the sum of 15 squares does not exceed χ^2 are 1,2,5,10 % are given by
`ppf(array([1.,2.,5.,10.])*0.01,15)`
The values of χ^2 for which the probabilities that the sum of 15 squares exceeds χ^2 are 1,2,5,10 % are given by
`ppf(array([99.,98.,95.,90.])*0.01,15).`

Python code 7.3 (page 103) *Generate and plot a contour for a two-dimensional function*

```
from scipy import *
from scipy import optimize
def contour(fxy,z,xycenter,xyscale=[1.,1.],\
  radius=0.05,nmax=500):
# construct contour f(x,y)=z by succession of
# circular intersects
# input:
#   fxy(x,y): defined function;
#   z:        level
#   xycenter: [xc,yc] point within contour
#   xyscale:  [xscale,yscale] approximate
#             coordinate ranges
#   radius:   radius of circle in units of
#             coordinate range
#   nmax:     maximum number of points (for open
#             contours)
    from scipy import optimize
    x0=xycenter[0]; y0=xycenter[1]
    xscale=xyscale[0]; yscale=xyscale[1]
```

```
def funx(x):
    return fxy(x,y0)-z
def funphi(phi):
    # uses xa,xb; dxs,dys (scaled)
    sinphi=sin(phi); cosphi=cos(phi)
    x=xa+(dxs*cosphi+dys*sinphi)*xscale
    y=ya+(-dxs*sinphi+dys*cosphi)*yscale
    return fxy(x,y)-z
# find first point on x-axis
xx=optimize.brentq(funx,x0,x0+5.*xscale)
xlist=[xx]; ylist=[y0]
# find second point
dxs=radius; dys=0.
xa=xx; ya=y0
phi=optimize.brentq(funphi,-pi,0.)
sinphi=sin(phi); cosphi=cos(phi)
xb=xa+(dxs*cosphi+dys*sinphi)*xscale
yb=ya+(-dxs*sinphi+dys*cosphi)*yscale
xlist += [xb]; ylist += [yb]
# find next point
radsq=radius*radius
dsq=4.*radsq
n=0
while (dsq>radsq) and (n<nmax):
    n +=1
    dxs=(xb-xa)/xscale
    dys=(yb-ya)/yscale
    xa=xb; ya=yb
    phi=optimize.brentq(funphi,-0.5*pi,0.5*pi)
    sinphi=sin(phi); cosphi=cos(phi)
    xb=xa+(dxs*cosphi+dys*sinphi)*xscale
    yb=ya+(-dxs*sinphi+dys*cosphi)*yscale
    xlist += [xb]; ylist += [yb]} \\
    dsq=((xb-xx)/xscale)**2+((yb-y0)/yscale)**2
xlist += [xx]; ylist += [y0]
data=array([xlist,ylist])
return data
```

Comments:
This function produces an array of coordinate values $[x, y]$ along a contour for which $f(x, y) = z$. Here $f(x, y)$ is a predefined function and z is a prescribed level. This array can be simply plotted by connecting the points with straight lines, e.g.:

```
autoplotp(data,symbol='',lines=Black)
```

The points are generated as follows. The first point is located on a line parallel to the x-axis, starting from the point `[xc,yc]`, searching in positive direction. Thus the input point `[xc,yc]` should be located inside the contour. The second point is searched on a half-circular (positive y) contour around the first point, with radius `radius`. Subsequent points are searched on a half-circular contour around the previous point, searching in the forward direction. Thus `radius` is the distance between subsequent points, which determines the resolution of the plot. The search is done in *scaled* x, y coordinates in order to prevent uneven distribution of points. The input `xyscale` is used for scaling: x-values are divided by `xyscale[0]` and y-values by `xyscale[1]`. You can use the total width of the plotted scales for `xyscale`, but the choice is not critical. The default radius 0.05 then means that the distance between points is 5 percent of the plot size. The optional parameter `nmax` limits the number of points generated on the contour, preventing infinite search along open contours. If a closed contour appears to be incomplete, either increase `nmax` or increase `radius`.

Python code 7.4 (page 104) *Generate a* $\Delta\chi^2 = 1$ *contour and derive uncertainties for the urease kinetics example*

Start from python code 7.1, which defines `SSQ(p)`, `p[0]`$= v_{\max}$; `p[1]`$= K_{\mathrm{m}}$, `pout` = *best parameter values.*

```
from scipy import *
from plotsvg import *
S0=SSQ(pout)
def fxy(x,y): return 4./S0*(SSQ([x,y])-S0)
data=contour(fxy,1.,pout,xyscale=[0.4,7.])
# plot the contour:
f=Figure()
f.frame([[15.25,16.25],[105,125]]
f.plotp(data,lines=Black,symbol='')
f.show()
# compute sig1,sig2, rho from contour:
sig1=0.5*(data[0].max()-data[0].min())
sig2=0.5*(data[1].max()-data[1].min())
ratio=(data[0,0]-pout[0])/sig1
rho=sqrt(1.-ratio**2)
print 'sigma1= %5.2f, sigma2=%5.2f, rho=%5.2f' %\
       (sig1,sig2,rho)
```

Comments:
The function `fxy` defines $\Delta\chi^2$ as a function of the parameters. The input `xyscale` in the function `contour` (see python code 7.3) is taken as

estimates of the standard deviations. The contour data array `data` contains 122 points; you may increase the resolution by setting a smaller `radius`. The standard uncertainties are derived from the extrema of the contour data; the correlation coefficient is found from the x-intercept `data[0,0]` by using the rule that the intercept occurs at a fraction $\sqrt{1-\rho^2}$ of the standard deviation.

Python code 7.5 (page 105) *Generate the covariance matrix by minimization (urease kinetics example)*

Start from python code 7.1, which defines `residuals(p)`, `SSQ(p)`, `p[0]`$= v_{\max}$; `p[1]`$= K_{\mathrm{m}}$. *We redo the minimization with full output:*

```
from scipy import optimize
lsq=optimize.leastsq
output = lsq(residuals,[15,105],full_output=1)
pout = output[0]
S0=SSQ(pout)
C=S0/(n-m)*output[1]
sig1=sqrt(C[0,0])
sig2=sqrt(C[1,1])
rho=C[0,1]/sig1/sig2
print 'sigma1= %5.2f, sigma2=%5.2f, rho=\%5.2f' %\
       (sig1,sig2,rho)
```

Comments:
The routine `leastsq` has a *full output* option, which produces as second element the covariance matrix $\boldsymbol{C}$, be it without proper scaling. The output matrix is only equal to the covariance matrix if all standard uncertainties σ_y are equal to 1. Correct results are obtained if the output matrix is scaled by $S_0/(n-m)$.

Python code 7.6 (page 105) *Generate the covariance matrix from the B-matrix (urease kinetics example)*

First construct matrix B in general, given function `delchisq(p)`

```
from scipy import *
def matrixB(delchisq, delta):
# delchisq(delp) = chisq(p-p0)-chisq(p0)
# delta = array of test deviations
    m=len(delta)
    B=zeros((m,m))
```

```
    d=zeros(m)
    fun=zeros(m)
    if (abs(delchisq(d)) > 1.e-8):
        print 'definition delchisq incorrect'
    for i in range(m):
        di=delta[i]
        d[i]=di
        fun[i]=delchisq(d)
        B[i,i]=fun[i]/(di*di)
        for j in range(i):
            dj=delta[j]
            d[j]=dj
            funij=delchisq(d)
            B[j,i]=B[i,j]=0.5*(funij-fun[i]-\
            fun[j])/(di*dj)
            d[j]=0.
        d[i]=0.
    return B
```

Start from python code 7.1, which defines `residuals(p), SSQ(p), p[0]=` v_{max} `; p[1]=` K_m `, pout =` *best parameter values. First construct B, then invert B and print results.*

```
delta=array([0.2,3.5]) # displacements near
                       # delchisq = 1
def delchisq(delp): return 4.*(SSQ(pout+delp)/\
  S0-1.)
B=matrixB(delchisq,delta)
from numpy import linalg
C=linalg.inv(B)
sig1=sqrt(C[0,0])
sig2=sqrt(C[1,1])
rho=C[0,1]/sig1/sig2
print 'sigma1=%5.2f, sigma2=%5.2f, rho=%5.2f' %\
       (sig1,sig2,rho)
```

Comments:
The construction of ***B*** proceeds by stepping `delta[i]` in all directions (which yields the diagonal elements) and stepping all pairs `delta[i],delta[j]` (which yields the off-diagonal elements). This is a simple procedure that could be made more sophisticated by involving steps in the opposite directions as well. Matrix inversion is done by the routine `inv` contained in the `numpy` module `linalg`.

Python code 7.7 (page 106) **Fit**: *a program that reports a general least-squares fit of a predefined function to a set of independent data.*

```
from scipy import *
from plotsvg import *
def fit(function,data,parin,figures=True):
    '''
Function. Non-linear least-squares fit of function
--------------------------------------------------
(x,par) to data
---------------
arguments:
  function          predefined function(x,par),
                      where x=independent variable
                      (called with an array x=data
                      [0]); par is a list of
                      parameters, e.g. [a,b]
  data              list (or 2D array) [x,y] or
                      [x,y,sig]. sig contains
                      standard deviations of y
                      (if known). If sig is given, a
                      chi-squared test is done; if
                      not given, equal weights are
                      assumed.
  parin             list of initial values for the
                      parameters, e.g. [0.,1.]
  figures=True      if True, two figures are
                      produced and displayed
returns:            [parout,sigma] (final parameters
                      with s.d.)

Remarks:
  The sum of weighted square deviations
  chisq=sum(((y-function(x))/sig)**2) [or, if no
  sig is given, SSQ=sum(((y-function(x)))**2)] is
  minimized by the nonlinear Scipy routine
  leastsq., using function values only. After
  successful determination of the best fit,
  uncertainties (s.d. and correlation
  coefficients) are computed, including the full
  covariance matrix. Plots of the fit and the
  residuals are produced.
```

```
Example: fit exponential function to data [x,y]
with sd sig:
  >>>def f(x,par):
         [a,k,c]=par
         return a*exp(-k*x)+c
  >>>[[a,k,b],[siga,sigk,sigc]]=fit(f,[x,y,sig],
  [1.,0.1,0.])
'''
    import os
    from scipy import optimize,stats
    lsq=optimize.leastsq
    if (len(data)==2):
        weights=False
    elif (len(data)==3):
        weights=True
    else:
        print 'ERROR: data should contain 2 or\
               3 items'
        print 'fit aborted'
        return 0
    m=len(parin)
    x=array(data[0])
    n=len(x)
    y=array(data[1])
    if (len(y)!=n):
        print 'ERROR: x and y have unequal length'
        print 'fit aborted'
        return 0
    if weights:
        sig=array(data[2])
        if (len(sig)!=n):
            print 'ERROR: x and sig have unequal\
                   length'
            print 'fit aborted'
            return 0
        def residuals(p): return (y-\
          function(x,p))/sig
    else:
        def residuals(p): return (y-function(x,p))
    def SSQ(p): return (residuals(p)**2).sum()
    SSQ0=SSQ(parin)
    # print results after minimization:
    print '\n Report on least-squares parameter\
           fit'
```

```
if weights:
    print 'chisq = sum of square reduced dev.\
           (y-f(x))/sig'
else:
    print 'SSQ = sum of square deviations\
           (y-f(x))'
print '\nnr of data points:        %5d' % (n)
print   'nr of parameters:         %5d' % (m)
print   'nr of degrees of freedom: %5d' % (n-m)
print '\nInitial values of parameters: '
print parin
if weights:
    print 'Initial chisq = %10.6g' % (SSQ0)
else:
    print 'Initial SSQ = %10.6g' % (SSQ0)
output=lsq(residuals,parin,full_output=1)
parout=output[0]
SSQout=SSQ(parout)
print 'Results after minimization:'
if weights:
    print 'Final chisq = %10.6g' % (SSQout)
else:
    print 'Final SSQ = %10.6g' % (SSQout)
print 'Final values of parameters'
print parout
# covariance matrix C
C=SSQout/(n-m)*output[1]
sigma=arange(m,dtype=float)
for i in range(m): sigma[i]=sqrt(C[i,i])
print 'Standard inaccuracies of parameters,:'
print sigma
print '\nMatrix of covariances'
print C
SR=zeros((m,m),dtype=float)
for i in range(m):
    SR[i,i]=sigma[i]
    for j in range(i+1,m):
        SR[j,i]=SR[i,j]=C[i,j]/(sigma[i]*
        sigma[j])
print '\nMatrix of sd (diagonal) and corr.
      coeff. (off-diag)'
print SR
# chisq analysis
if weights:
```

```
        nu=n-m
        F=stats.chi2.cdf(SSQout,nu)
        print '\nChi-square analysis:'
        print 'chi^2 (sum of weighted square\
           deviations) =%10.6g' % (SSQout)
        print 'cum. prob. for chi^2  = %5.3g %%' %\
            (100.*F)
        if (F<.1):
            print "chi^2 is low! Did you\
               overestimate the supplied\
               sigma's?"
            print 'Or did you fit the original\
               data too closely with too many\
               parameters?'
        elif (F>.9):
            print "chi^2 is high! Did you neglect\
               an error source in the supplied\
               sigma's?"
            print 'Or did the data result from a\
               bad fitting procedure?'
        else:
            print 'cum. probab. of chi^2 is\
               reasonable (between 10% and 90%).'
            print "The spread in the data agrees\
               with the supplied sigma's"
    # produce two plots (data and fitting curve;
    #  residuals)
    if figures:
        xmin=x.min(); xmax=x.max()
        ymin=y.min(); ymax=y.max()
        if weights:
            maxsigy=sig.max()
            ymin=ymin-maxsigy
            ymax=ymax+maxsigy
        y1=1.05*ymin-0.05*ymax; y2=1.05*ymax
        -0.05*ymin
        f=Figure()
        f.frame([[xmin,xmax],[y1,y2]],title=\
          'Least-squares fit')
        if weights:
            f.plotp([x,y],symbolfill=Blue,ybars=\
            sig, barcolor=Blue)
        else:
            f.plotp([x,y],symbolfill=Blue)
```

```
        def fun(x): return function(x,parout)
        f.plotf(fun,color=Red)
        f.show(filename='figfit.svg')
        os.startfile('figfit.svg')
        print 'figfit.svg is now displayed by your\
              browser'
        residuals=y-fun(x)
        minres=residuals.min(); maxres=residuals.
        max()
        if weights:
            minres=minres-maxsigy
            maxres=maxres+maxsigy
        y1=1.05*minres-0.05*maxres; y2=1.05*maxres\
           -0.05*minres
        f=Figure()
        f.size=[5500,3400]
        f.frame([[xmin,xmax],[y1,y2]],
        title="residuals")
        if weights:
            f.plotp([x,residuals],symbolfill=Blue,\
            ybars=sig, barcolor=Blue)
        else:
            f.plotp([x,residuals],symbolfill=Blue)
        f.show(filename='figresiduals.svg')
        os.startfile('figresiduals.svg')
        print 'figresiduals.svg is now displayed\
               by your browser'
    return [parout,sigma]
```

Comments:
This program can be downloaded from www.hjcb.nl/python. Look for recent updates. Two plots are automatically generated and displayed by the standard browser. Make sure that the .svg mime type starts your SVG-enabled browser.

Python code 7.8 (page 108) *Compute various sum of squared deviations and perform an F-test on the urease kinetics example*

Start from python code 7.1 and code 7.5, which defines independent variable `S`, *dependent variable* `v` *and best parameters* `pout`.

```
y=S
def fun(x,p): return p[0]*x/(p[1]+x)
def ssq(x): return (x**2).sum()
```

```
f=fun(v,pout)
SST=ssq(y-y.mean())
SSR=ssq(f-f.mean())
SSE=ssq(y-f)
Fratio=SSR/(SSE/4.)
from scipy import stats
Fcum=stats.f.cdf(Fratio,1,4)
print 'SST=%7.3f SSR=%7.3f SSE=%7.3f' % (SST,SSR,\
      SSE)
print 'Fratio=%7.3f Fcum=%7.3f' % (Fratio,Fcum)
```

Comments:
The function `ssq(x)` computes the sum of squares of the elements of a 1D-array `x`. The array f gives the best-fitted function values. The function `f.cdf(Fratio, nu1,nu2)` of the `scipy` module `stats` gives the cumulative F-distribution.

Python code A5.1 (page 150) *Generate pdf of sum of n homogeneously distributed random numbers*

```
from scipy import fftpack
def symmetrize(x): # adds mirror to x
    n=alen(x)
    half=n/2
    for i in range(1,half): x[n-i] += x[i]
    return 1
def FT(Fx,delx):   # produces real FT of symmetric
                   # Fx
    Gy=fftpack.fft(Fx).real*delx
    return Gy
def IFT(Gy,delx):  # produces real inverse FT of
                   # symmetric Gy
    Fx=fftpack.ifft(Gy).real/delx
    return Fx
n=10               # number of random numbers to be
                   # added
a=sqrt(3./n)       # [-a,a] is range of random
                   # numbers
nft=4096           # array length for FT
xm=50.             # maximum of x-scale
delx=2.*xm/nft     # delta x between points in Fx
dely=pi/xm         # delta y between points in Gy
ym=nft*dely/2.     # maximum of y-scale
Fx=zeros((nft),dtype=float)
```

```
# set rectangular function Fx:
for i in range(int(a/delx)): Fx[i]=0.5/a
symmetrize(Fx)      # this makes the FT real
corr=1./Fx.sum()/delx  # correction to make
                       # integral exactly = 1
Fx=Fx*corr
Gy=FT(Fx,delx)      # Fourier transform of
                    # rectangular function
Gyn=Gy**n           # FT of convolution of 10
                    # rectangular functions
Fxn=IFT(Gyn,delx)   # Inverse FT gives the
                    # convolution function
m=4./delx           # [-4,4] is interesting plot
                    # range
yn=concatenate((Fxn[-m:],Fxn[:m]))
                    # yn is the useful output
```

Comments:
This example computes the probability density function of the sum of $n = 10$ homogeneously distributed random numbers from an interval $[-a, a\rangle$, where a is chosen such that the resulting variance of the sum equals 1. Such a distribution is a convolution of 10 rectangular functions, which is most easily computed by inverse FT of the n-th power of the FT of the original rectangular function. In the last lines the result is recast into a smaller range, symmetrical around zero ($-4 < x < 4$).

Python code A7.1 (page 157) *Variance of the mean by block averages*

```
def block(data,n):
# block-average data in blocks of length n
# data: input [x,y] (x,y: 1D-arrays of same length)
# n: number of points in each block
# returns array of block averages of both x and y
    ntot=len(data[0])
    nnew=ntot/n
    x=zeros(nnew,dtype=float)
    y=zeros(nnew,dtype=float)
    for i in range(nnew):
        x[i]=sum(data[0][i*n:(i+1)*n])/float(n)
        y[i]=sum(data[1][i*n:(i+1)*n])/float(n)
    return [x,y][1ex]
def blockerror(data,blocksize=[10,20,40,60,80,100,\
  125]):
```

```
# make list of s.d of the mean for given blocksizes
# data: input [x,y] (x,y: 1D-arrays of same length)
# blocksize: list of lengths of blocks,
#   assuming independent block averages
# returns [blocksize, stderror, ybars]
# ybars is rms inaccuracy of stderror
    n=len(data[1])
    delt=(data[0][-1]-data[0][0])/float(n-1)
    xout=[]
    yout=[]
    ybars=[]
    for nb in blocksize:
        xyblock=block(data,nb)
        number = len(xyblock[1])
        std=xyblock[1].std()
        stderror = std/sqrt(number-1.)
        xout += [nb*delt]
        yout += [stderror]
        ybars += [stderror/sqrt(2.*(number-1.))]
    return [xout,yout,ybars]
```

Comments:
It is assumed that a set of `data=[x,y]` is available (x and y being arrays). The function `block(data,n)` returns a new set of data consisting of the averages over blocks of length n. The blocks start at the first data item; if the number of data points does not fit an integer number of blocks, the remaining points are not used. The function `blockerror` calls the function `block` for each of the elements in the optional argument `blocksize`. For each blocksize it computes the standard error in the mean and outputs it as `yout`. The output values `xout` are the block sizes expressed in units of x. The output values `ybars` are the standard deviations expected for `yout` on the basis of the limited number of averages; it can be used to draw error bars in a plot of the output data.