# Pyphant's ExtremumFinder worker

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

```python
# Copyright (c) 2007-2008, Rectorate of the University of Freiburg
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are met:
#
# * Redistributions of source code must retain the above copyright
#   notice, this list of conditions and the following disclaimer.
# * Redistributions in binary form must reproduce the above copyright
#   notice, this list of conditions and the following disclaimer in the
#   documentation and/or other materials provided with the distribution.
# * Neither the name of the Freiburg Materials Research Center,
#   University of Freiburg nor the names of its contributors may be used to
#   endorse or promote products derived from this software without specific
#   prior written permission.
#
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
# IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
# TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
# PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
# OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
# EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
# PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
# PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
# LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
# NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
# SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

u"""Pyphant module computing the local extrema of one-dimensional sampled fields. If a two-dimensional field is provided as input, the algorithm loops over the 0th dimension denoting the y-axis, which corresponds to an iteration over the rows of the data matrix.
"""

__id__ = "Id"
__author__ = "Author"
__version__ = "Revision"
# Source

import numpy
from pyphant.core import (Worker, Connectors,
                          Param, DataContainer)
```

**Error:**

No handlers could be found for logger "pyphant"

```python

import scipy.interpolate
from Scientific.Physics import PhysicalQuantities
import copy

class ExtremumFinder(Worker.Worker):
    API = 2
    VERSION = 1
    REVISION = "Revision"[11:-1]
```

```python
    name = "Extremum Finder"

    _sockets = [("field", Connectors.TYPE_IMAGE)]
    _params = [("extremum", u"extremum", [u"minima",u"maxima",u"both"], None)
              ]

    @Worker.plug(Connectors.TYPE_IMAGE)
    def locate(self, field, subscriber=0):
        #Determine the number of rows $N_{rows}$ for which the local extrema have to be found.
        if len(field.dimensions)==1:
            Nrows = 1
        else:
            Nrows = len(field.dimensions[0].data)
        #Find local extrema $\vec{x}_0$
        x0, extremaCurv, extremaError, extremaPos = findLocalExtrema(field, Nrows)
        #Map roots and curvatures to arrays
        if Nrows == 1:
            X0 = numpy.array(x0)
            XCurv = numpy.array(extremaCurv[0])
            XError= numpy.array(extremaError[0])
        else:
            maxLen = max(map(len,extremaPos))
            X0 = numpy.zeros((Nrows,maxLen),'f')
            X0[:] = numpy.NaN
            XCurv = X0.copy()
            XError= X0.copy()
            for i in xrange(Nrows):
                numExt = len(extremaPos[i])
                if numExt == 1:
                    X0[i,0]=extremaPos[i][0]
                    XCurv[i,0]=extremaCurv[i]
                    XError[i,0]=extremaError[i]
                else:
                    for j in xrange(numExt):
                        X0[i,j]=extremaPos[i][j]
                        XCurv[i,j]=extremaCurv[i][j]
                        XError[i,j]=extremaError[i][j]
        extremaType = self.paramExtremum.value
        if extremaType == u'minima':
            X0 = numpy.where(XCurv > 0,X0,numpy.nan)
            error = numpy.where(XCurv > 0,XError,numpy.nan)
        elif extremaType == u'maxima':
            X0 = numpy.where(XCurv < 0,X0,numpy.nan)
            error = numpy.where(XCurv < 0,XError,numpy.nan)
        else:
            extremaType = u'extrema'
            error = XError
        xName = field.dimensions[-1].longname
        xSym  = field.dimensions[-1].shortname
        yName = field.longname
        ySym  = field.shortname
        if numpy.sometrue(numpy.isnan(X0)):
            if len(field.dimensions) == 1:
                roots = DataContainer.FieldContainer(numpy.extract(numpy.logical_not(numpy.isnan(X0)),X0),
                                                     unit = field.dimensions[-1].unit,
```

```
                                                       longname="%s of the local %s of %s" % (xName, extremaType, yName),
                                                       shortname="%s_0" % xSym)
            else:
                roots = DataContainer.FieldContainer(X0.transpose(),
                                                       mask = numpy.isnan(X0).transpose(),
                                                       unit = field.dimensions[-1].unit,
                                                       longname="%s of the local %s of %s" % (xName, extremaType, yName),
                                                       shortname="%s_0" % xSym)
        else:
            roots = DataContainer.FieldContainer(X0.transpose(),
                                                       unit = field.dimensions[-1].unit,
                                                       longname="%s of the local %s of %s" % (xName, extremaType, yName),
                                                       shortname="%s_0" % xSym)
        if field.error != None:
            if len(field.dimensions)==1:
                roots.error = numpy.extract(numpy.logical_not(numpy.isnan(X0)),error)
            else:
                roots.error = error.transpose()
        else:
            roots.error = None
        if len(field.data.shape)==2:
            roots.dimensions[-1] = field.dimensions[0]
        roots.seal()
        return roots

def findLocalExtrema(field, Nrows):
    #Init nested lists ll_x0, ll_sigmaX0 and ll_curv which are going to hold one list per
    #analysed data row.
    ll_x0 = []       #Nested list for $\vec{x}_{0,i}$ with $i=0,N_{rows}-1$
    ll_sigmaX0 = [] #Nested list for $\sigma_{x_{0,i}}$ with $i=0,N_{rows}-1$
    ll_curv = []    #Nested list indicating local maximum (-1) or local minimum (1)
    #Because we are looking for local extrema of one-dimensional sampled
    #fields the last dimension is the sampled abscissa $\vec{x}$.
    x = field.dimensions[-1].data
    #Loop over all rows $N_{rows}$.
    for i in xrange(Nrows):
        #If a $1\times N_{rows}$ matrix is supplied, save this row to vector $\vec{y}$.
        #Otherwise set vector $\vec{y}$ to the i^th row of matrix field.data
        #and handle vector of errors $\vec{\sigma}_y$ accordingly. It is None, if no error is given.
        sigmaY= field.error
        if Nrows == 1:
            y = field.data
        else:
            y = field.data[i]
            if field.error != None:
                sigmaY= field.error[i]
        x0, l_sigmaX0, dyy = findLocalExtrema1D(y, x, sigmaY)
```

```python
            ll_x0.append(numpy.array(x0))
            ll_sigmaX0.append(numpy.array(l_sigmaX0))
            ll_curv.append(numpy.array(dyy))
    return x0, ll_curv, ll_sigmaX0, ll_x0

def findLocalExtrema1D(y, x, sigmaY=None):
    #Compute differences Δ_y of data vector y.
    #The differencing reduces the dimensionalty of the vector by one: dimΔ_y = dimx − 1.
    DeltaY    = numpy.diff(y)
    #Test if the sign of successive elements of DeltaY change sign. These elements are candidates for the
    #estimation of local extrema. The result is a vector b_x0 of booleans with dimx_{0,b} = dimx − 2.
    #From b_x0[j]==True follows x_{0,j} ∈ [x_{j+1}, xj + 2].
    #Note, that b_x0[j]==b_x0[j+1]=True indicate a special case,
    #which maps to one local extremum x_{0,j} ∈ [x_{j+1}, x_{j+2}].
    b_x0= numpy.sign(DeltaY[:-1])!=numpy.sign(DeltaY[1:])
    #Init list l_x0 for collecting the local extrema.
    l_x0 = []
    #Init list l_sigmaX0 for collecting the estimation errors of locale extrema positions x_0.
    l_sigmaX0 = []
    #Init list l_curv_sign for collecting the sign of the curvature at the position of the locale extrema.
    l_curv_sign   = []
    #If one or more local extrema have been found, estimate its position, otherwise set its position to NaN.
    if numpy.sometrue(b_x0):
        #Remove successive True values, which occur do to symmetrically boxed or exact local extrema.
        b_x0[1:]=numpy.where(numpy.logical_and(b_x0[:-1],b_x0[1:]),False,
            b_x0[1:])
        #Compute vector index referencing the True elements of b_x0.
        index = numpy.extract(b_x0,numpy.arange(len(DeltaY)-1))
        skipOne = False
        for j in index:
            if skipOne:
                skipOne = False
            else:
                if sigmaY == None:
                    x0,sigmaX0,curv_sign = estimateExtremumPosition(y[j:j+3]
                        ,x[j:j+3])
                else:
                    x0,sigmaX0,curv_sign = estimateExtremumPosition(y[j:j+3]
                        ,x[j:j+3], sigmaY=sigmaY[j:j+3])
                l_x0.append(x0)
                l_sigmaX0.append(sigmaX0)
                l_curv_sign.append(curv_sign)
    else: #No local extremum found.
        l_x0.append(numpy.NaN)
        l_sigmaX0.append(numpy.NaN)
        l_curv_sign.append(numpy.NaN)
    return l_x0, l_sigmaX0, l_curv_sign
```

# 1 Function estimateExtremumPosition(y,x,sigmaY=None)

Estimate the extremum position from three sample points $(x_0, y_0)$, $(x_1, y_1)$, and $(x_1, y_1)$ by a linear model. The sample points are provided as vectors $\vec{x} = (x_0, x_1, x_2)$ and $\vec{y} = (y_0, y_1, y_2)$. The middle sample point

$(x_1, y1)$ separates two bins. For each bin the slope $y'$ is calculated as finite difference. These slopes are assumed to be located at the centre of each bin, such that the slopes $((x_0+x_1)/2, (y_1-y_0)/(x_1-x_0))$ and $((x_1+x_2)/2, (y_2-y_1)/(x_2-x_1))$ can be compiled to a linear equation, whose root is an estimate for the position of the local extremum:

$$\tilde{x}_0 = \frac{1}{2}(x_0+x_1) - \frac{\frac{1}{2}(x_2-x_0)}{\frac{y_2-y_1}{x_2-x_1} - \frac{y_1-y_0}{x_1-x_0}} \frac{y_1-y_0}{x_1-x_0} \tag{1}$$

and its error

$$\sigma_{\tilde{x}_0} = R \cdot (\sigma_{y,0}|y_2-y_1| + \sigma_{y,1}|y_2-y_0| + \sigma_{y,3}|y_1-y_0|) \tag{2}$$

with

$$R = \frac{1}{2}\left|\frac{(x_1-x_0)(x_2-x_0)(x_2-x_1)}{[y_0(x_2-x_1)+y_1(x_0-x_2)+y_2(x_1-x_0)]^2}\right|.$$

```python
def estimateExtremumPosition(y, x, sigmaY = None):
    """Estimate the extremum position from three sample points, whose x- and y-coordinates
    are given as numpy arrays x and y. The middle sample
    point separates two bins. For each bin a slope is calculated as finite difference.
    Both slopes are assumed to be located at the centre of each bin, which leads
    to a linear equation for the estimation of the position of the local extremum.
    If an y-error is specified an estimation error is computed from error propagation.
    """
    #Compute the width of left and right bin. The bin width has to be finite.
    deltaXleft = x[1]-x[0]
    deltaXright= x[2]-x[1]
    if deltaXleft == 0 or deltaXright == 0:
        raise ValueError, "Both bins need to have a finite width."
    #Compute the centres of left and right bin. The centre should not be identical.
    xCleft = 0.5*(x[0]+x[1])
    xCright= 0.5*(x[1]+x[2])
    if xCleft == xCright:
        raise ValueError, "The centres of the left and the right bin cannot be identical."
    #Compute the difference of the sampled values.
    deltaYleft = y[1]-y[0]
    deltaYright= y[2]-y[1]
    #If the difference is zero in both bins, a constant region has been detected and the
    #algorithm should return NaN. If the difference of the right bin is greater or lower
    #than zero a local minimum or local maximum has been detected, respectively.
    if deltaYleft == 0.0:
        if deltaYright == 0.0: #constant region
            return numpy.NaN,numpy.NaN,numpy.NaN
        elif deltaYright > 0:  #local minimum
            curv_sign = 1.0
        else:                  #local maximum
            curv_sign = -1.0
    else:
        curv_sign = -numpy.sign(deltaYleft)
    # Estimate position of local extrema according to Eq. (1).
    x0 =xCleft-(xCright-xCleft)/(deltaYright/deltaXright-deltaYleft/deltaXleft)*deltaYleft/deltaXleft
```

```
    # If an y-error has been provided, compute the estimation error according to Eq. (2).
    if sigmaY != None:
        numerator = 0.5*deltaXleft*deltaXright*(x[2]-x[0])
        R = numerator / (y[0]*deltaXright+y[1]*(x[0]-x[2])+y[2]*deltaXleft)**2
        partError = numpy.array([-deltaYright,y[2]-y[0],-deltaYleft])
        sigmaX0 = R * numpy.dot(sigmaY,numpy.abs(partError))
    else:
        sigmaX0=numpy.NaN
    return x0,sigmaX0,curv_sign
```