S1. Python Code of the Interpolation Algorithms

S1.1. Interpolation algorithm IA1

```python
def IA1(g):
    """
    ***************************************************************************
    * Copyright 2017
    * Sabine Hittmeir, Anne Philipp, Petra Seibert
    * This work is licensed under the Creative Commons Attribution 4.0
    * International License. To view a copy of this license, visit
    * http://creativecommons.org/licenses/by/4.0/ or send a letter to
    * Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.
    ***************************************************************************

    @Description
    The input to this python function is a 1d array of data passed as
    parameter g. If g has n elements, it returns an array of size
    (n-1)*3 + 1 as the result for IA1 where additional supporting points
    have been added in each interval.
    A reconstruction by linear interpolation between these points is
    conservative with respect to the input (where we consider each list
    element to represent a mean value for the preceding interval) and
    positive definite. The function includes a monotonicity filter applied
    in a second sweep.

    For more information see article:
    reconstruction scheme for the interpolation of extensive quantities
    in the Lagrangian particle dispersion model FLEXPART.
    Geoscientific Model Development

    @Input
    g: numpy array, shape(nt), float
       Original data series which will be interpolated.
       nt is the length of the time series.

    @Return
    f: list of float values
       The reconstructed input data series with the two additional
       sub-grid points for each interval of the input data.
    """
```
import numpy as np

############### Non-negative Geometric Mean Based Algorithm ###############

# at the left boundary, the following boundary condition is assumed: 
# output value at t=0 is equal to the first input data value, 
# corresponding to the assumption of persistence 
f=[g[0]]

# go through the data series and extend each interval by two sub-grid 
# points and calculate the corresponding data values 
# except for the last interval due to boundary conditions 
for i in range(0, len(g)-1):
    # as a requirement: 
    # if there is a zero data value such that g[i]=0, then the whole 
    # interval in f has to be zero to such that f[i+1]=f[i+2]=f[i+3]=0 
    # according to Eq. (12) 
    if g[i]==0.:
        f.extend([0.,0.,0.])
    else:
        # temporal save of last value in interpolated list 
        # since it is the left boundary and hence the new (fi) value 
        fi = f[-1]

        # the value at the end of the interval (fip1) is prescribed by the 
        # geometric mean, restricted such that non-negativity is guaranteed 
        # according to Eq. (26) 
        fip1=min( 3.*g[i] , 3.*g[i+1] , np.sqrt(g[i+1]*g[i]) )

        # the function value at the first sub-grid point (fi1) is determined 
        # according to the equal area condition with Eq. (20) 
        fi1=3./2.*g[i]-1./12.*f[i-5./12.*fip1]

        # the function value at the second sub-grid point (fi2) is determined 
        # according to the equal area condition with Eq. (21) 
        fi2=3./2.*g[i]-5./12.*f[i-1./12.*fip1]

        # add next interval of interpolated (sub-)grid values 
        f.append(fi1)
f.append(fi2)
f.append(fip1)

# separate treatment of the final interval

# as a requirement:
# if there is a zero data value such that g[i]=0, then the whole
# interval in f has to be zero to such that f[i+1]=f[i+2]=f[i+3]=0
# according to Eq. (12)
if g[-1]==0.:
    f.extend([0.,0.,0.])

# otherwise the sub-grid values are calculated and added to the list
# using the persistence assumption as boundary condition
else:
    # temporal save of last value in interpolated list
    fi = f[-1]
    # since last interval in series, last value is also (fip1)
    fip1 = g[-1]
    # the function value at the first sub-grid point (fi1) is determined
    # according to the equal area condition with Eq. (20)
    fi1 = 3./2.*g[-1]-1./12.*fi-5./12.*fip1
    # the function value at the second sub-grid point (fi2) is determined
    # according to the equal area condition with Eq. (21)
    fi2 = 3./2.*g[-1]-5./12.*fi-1./12.*fip1
    # add next interval of interpolated (sub-)grid values
    f.append(fi1)
f.append(fi2)
f.append(fip1)

# Monotonicity Filter

for i in range(3, len(f)-4, 3):
    j = int(i/3.)-1

    # otherwise, if the monotonicity property is violated (test Eq. (27))
    # we replace the interpolated function values according to the
    # specifications in the actual and consecutive interval
    if np.sign(f[i-1]-f[i-2]) * np.sign(f[i]-f[i-1])===-1 \
        and np.sign(f[i+1]-f[i]) * np.sign(f[i]-f[i-1])===-1 \
        and np.sign(f[i+2]-f[i+1]) * np.sign(f[i+1]-f[i])===-1:

        # the monotonicity filter corrects the value at (fip1) by
        # substituting (fip1) with (fmon), see Eq. (28), (29) and (30)
        fmon = min([3.*g[j], \n            3.*g[j+1], \n            np.sqrt(max(0,(18./13.*g[j] -5./13.*f[i-3]) * 
                (18./13.*g[j+1]-5./13.*f[i+3]))))])

        # re-computation of the sub-grid interval values while the
        # interval boundaries (fi) and (fip2) remains unchanged
        # see Eq. (20) and (21)
        f[i-2]=3./2.*g[j]-1./12.*f[i-3]-5./12.*fmon
        f[i-1]=3./2.*g[j]-5./12.*f[i-3]-1./12.*fmon
        f[i]=fmon
        f[i+1]=3./2.*g[j+1]-5./12.*f[i+3]-1./12.*fmon
        f[i+2]=3./2.*g[j+1]-1./12.*f[i+3]-5./12.*fmon

return f
def IA2(gnp):
    '''
    Copyright 2017
    Sabine Hittmeir, Anne Philipp, Petra Seibert
    * This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.
    **************************************************************************
    @Description
    The input to this python function is a 1d array of data passed as parameter g. If g has n elements, it returns an array of size (n-1)*3 + 1 as the result for IA2 where additional supporting points have been added in each interval. A reconstruction by linear interpolation between these points is conservative with respect to the input (where we consider each list element to represent a mean value for the preceding interval) and positive definite. The function includes a monotonicity filter which is already incorporated in the reconstruction algorithm by making an educated guess for the f_{i+2} function value.
    For more information see article:
    @Input
    g: numpy array, shape(nt), float
    Original data series which will be interpolated. nt is the length of the time series.
    @Return
    f: list of float values
    The reconstructed input data series with the two additional sub-grid points for each interval of the input data.
    **************************************************************************
    # variable description
    #
    # i - index variable for looping over the data series
    # gnp - input data series as numpy array
    # g - input data series
    # f - output data series with additional grid points
    # f1 - function value f_i (at position i)
    # f11 - first sub-grid function value f_{i-1}
    # f12 - second sub-grid function value f_{i-2}
    # fip1 - next function value f_(i+1) (at position i+1)
    # fip2 - educated guess for function value f_(i+2)(at position i+2)
    # dt - time step
    # fmon - monotonicity filter
    # gm1 - g minus 1; last element in input data series
    # f[-1] - sub-grid function value f_{i}
import numpy as np

# create and extended data list by coping the last value two times
# to avoid separate treatment of the penultimate interval
gm1 = gnp[-1] # temporal save of last element of list
g = []
g.extend(gnp)
g.extend([gm1,gm1])

############################ Advanced Geometric Mean based algorithm ############################

# at the left boundary, the following boundary condition is assumed:
# output value at t=0 is equal to the first input data value,
# corresponding to the assumption of persistence
f=[g[0]]

# go through the data series and extend each interval by two sub-grid
# points and calculate the corresponding data values
# except for the last interval due to boundary conditions
# loop until -3 instead of -1 because of two additional sub-grid points
for i in range(0,len(g)-3):
    # as a requirement:
    # if there is a zero data value such that g[i]=0, then the whole
    # interval in f has to be zero to such that f[i+1]=f[i+2]=f[i+3]=0
    # according to Eq. (12)
    if g[i]==0.:
        f.extend([0.,0.,0.])
    # otherwise the sub-grid values are calculated and added to the list
    else:
        # temporal save of last value in interpolated list
        # since it is the left boundary and hence the new (fi) value
        fi = f[-1]
        # calculate an educated guess for position (fip2) to apply the
        # monotonicity filter according to Eq. (31)
        fip2=min(3.*g[i+1],
                 3.*g[i+2],
                 np.sqrt(g[i+1]*g[i+2]))

        # the value at the end of the interval (fip1) is prescribed by the
        # geometric mean, restricted such that non-negativity is guaranteed
        # according to Eq. (33)
        fip1 = min(3.*g[i],
                   3.*g[i+1],
                   np.sqrt(max(0, (18./13.*g[i] -5./13.*fi)*
                              (18./13.*g[i+1] -5./13.*fip2))))

        # the function value at the first sub-grid point (fi1) is determined
        # according to the equal area condition with Eq. (20)
        fi1=3./2.*g[i]-1./12.*fi-5./12.*fip1

        # the function value at the second sub-grid point (fi2) is determined
        # according to the equal area condition with Eq. (21)
        fi2=3./2.*g[i]-5./12.*fi-1./12.*fip1

        # add next interval of interpolated (sub-)grid values
        f.append(fi1)
        f.append(fi2)
f.append(fip1)

# separate treatment of the final interval

# as a requirement:
# if there is a zero data value such that g[i]=0, then the whole
# interval in f has to be zero to such that f[i+1]=f[i+2]=f[i+3]=0
# according to Eq. (12)
if g[-1]==0.:
    f.extend([0.,0.,0.])

# otherwise the sub-grid values are calculated and added to the list
# using the persistence hypothesis as boundary condition
else:
    # temporal save of last value in interpolated list
    # since it is the left boundary and hence the new (fi) value
    fi = f[-1]
    # since last interval in series, last value is also (fip1)
    fip1 = g[-1]

    # the function value at the first sub-grid point (fi1) is determined
    # according to the equal area condition with Eq. (20)
    fi1 = 3./2.*g[-1]-1./12.*fi-5./12.*fip1

    # the function value at the second sub-grid point (fi2) is determined
    # according to the equal area condition with Eq. (21)
    fi2 = 3./2.*g[-1]-5./12.*fi-1./12.*fip1

    # add next interval of interpolated (sub-)grid values
    f.append(fi1)
    f.append(fi2)
    f.append(fip1)

    return f
S1.3. Interpolation algorithm IA1m

```python
def IA1m(g):
    """
    ***************************************************************************
    * Copyright 2017
    * Sabine Hittmeir, Anne Philipp, Petra Seibert
    *
    * This work is licensed under the Creative Commons Attribution 4.0
    * International License. To view a copy of this license, visit
    * http://creativecommons.org/licenses/by/4.0/ or send a letter to
    * Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.
    ***************************************************************************
    @Description
    The input to this python function is a 1d array of data passed as parameter g. If g has n elements, it returns an array of size (n-1)*3 + 1 as the result for IA1m where additional supporting points have been added in each interval. A reconstruction by linear interpolation between these points is conservative with respect to the input (where we consider each list element to represent a mean value for the preceding interval) and positive definite. The function includes a monotonicity filter applied directly in the first sweep.
    
    For more information see article: Hittmeir, S.; Philipp, A.; Seibert, P. (2018): A conservative reconstruction scheme for the interpolation of extensive quantities in the Lagrangian particle dispersion model FLEXPART. Geoscientific Model Development
    @Input
    g: numpy array, shape(nt), float
    Original data series which will be interpolated.
    nt is the length of the time series.
    @Return
    f: list of float values
    The reconstructed input data series with the two additional sub-grid points for each interval of the input data.
    """
```

@Description
The input to this python function is a 1d array of data passed as parameter g. If g has n elements, it returns an array of size (n-1)*3 + 1 as the result for IA1m where additional supporting points have been added in each interval. A reconstruction by linear interpolation between these points is conservative with respect to the input (where we consider each list element to represent a mean value for the preceding interval) and positive definite. The function includes a monotonicity filter applied directly in the first sweep.

For more information see article: Hittmeir, S.; Philipp, A.; Seibert, P. (2018): A conservative reconstruction scheme for the interpolation of extensive quantities in the Lagrangian particle dispersion model FLEXPART. Geoscientific Model Development
```python
import numpy as np

# Non-negative Geometric Mean Based Algorithm

# at the left boundary, the following boundary condition is assumed:
# output value at t=0 is equal to the first input data value,
# corresponding to the assumption of persistence
f = [g[0]]

# compute two first sub-grid intervals without monotonicity check
# go through the data series and extend each interval by two sub-grid
# points and calculate the corresponding data values
# except for the last interval due to boundary conditions
for i in range(0, 2):
    # as a requirement:
    # if there is a zero data value such that g[i]=0, then the whole
    # interval in f has to be zero to such that f[i+1]=f[i+2]=f[i+3]=0
    # according to Eq. (12)
    if g[i] == 0.:
        f.extend([0., 0., 0.])
    else:
        # temporal save of last value in interpolated list
        # since it is the left boundary and hence the new (fi) value
        fi = f[-1]

        # the value at the end of the interval (fip1) is prescribed by the
        # geometric mean, restricted such that non-negativity is guaranteed
        # according to Eq. (26)
        fip1 = min(3.*g[i], 3.*g[i+1], np.sqrt(g[i+1]*g[i]))

        # the function value at the first sub-grid point (fi1) is determined
        # according to the equal area condition with Eq. (20)
        fi1 = 3./2.*g[i]-1./12.*fi-5./12.*fip1

        # the function value at the second sub-grid point (fi2) is determined
        # according to the equal area condition with Eq. (21)
        fi2 = 3./2.*g[i]-5./12.*fi-1./12.*fip1

        # add next interval of interpolated (sub-)grid values
        f.append(fi1)
        f.append(fi2)
        f.append(fip1)

# compute rest of the data series intervals
# go through the data series and extend each interval by two sub-grid
# points and calculate the corresponding data values
# except for the last interval due to boundary conditions
for i in range(2, len(g)-1):
    # as a requirement:
    # if there is a zero data value such that g[i]=0, then the whole
    # interval in f has to be zero to such that f[i+1]=f[i+2]=f[i+3]=0
    # according to Eq. (12)
    if g[i] == 0.:
        # apply monotonicity filter for interval before (Eq. (27))
```

---

```python
# Code execution
```
# check if there is "M" or "W" shape
if np.sign(f[-5]-f[-6]) * np.sign(f[-4]-f[-5])==-1 \
    and np.sign(f[-4]-f[-5]) * np.sign(f[-3]-f[-4])==-1 \
    and np.sign(f[-3]-f[-4]) * np.sign(f[-2]-f[-3])==-1:

    # the monotonicity filter corrects the value at (fim1) by
    # substituting (fim1) with (fmon), see Eq. (28), (29) and (30)
    fmon = min(3.*g[i-2], \n            3.*g[i-1], \n            np.sqrt(max(0,(18./13.*g[i-2] - 5./13.*f[-7]) * \n                     (18./13.*g[i-1] - 5./13.*f[-1]))))

    # re-computation of the sub-grid interval values while the
    # interval boundaries (fi) and (fip2) remains unchanged
    # see Eq. (20) and (21)
    f[-6]=3./2.*g[i-2]-1./12.*f[-7]-5./12.*fmon
    f[-5]=3./2.*g[i-2]-5./12.*f[-7]-1./12.*fmon
    f[-4]=fmon
    f[-3]=3./2.*g[i-1]-1./12.*fmon-5./12.*f[-1]
    f[-2]=3./2.*g[i-1]-5./12.*fmon-1./12.*f[-1]

f.extend([0.,0.,0.])

# otherwise the sub-grid values are calculated and added to the list
else:
    # temporal save of last value in interpolated list
    # since it is the left boundary and hence the new (fi) value
    fi = f[-1]

    # the value at the end of the interval (fip1) is prescribed by the
    # geometric mean, restricted such that non-negativity is guaranteed
    # according to Eq. (26)
    fip1=min( 3.*g[i] , 3.*g[i+1] , np.sqrt(g[i+1]*g[i]) )

    # the function value at the first sub-grid point (fi1) is determined
    # according to the equal area condition with Eq. (20)
    fi1=3./2.*g[i]-1./12.*fi-5./12.*fip1

    # the function value at the second sub-grid point (fi2) is determined
    # according to the equal area condition with Eq. (21)
    fi2=3./2.*g[i]-5./12.*fi-1./12.*fip1

    # apply monotonicity filter for interval before (Eq. (27))
    # check if there is "M" or "W" shape
if np.sign(f[-5]-f[-6]) * np.sign(f[-4]-f[-5])==-1 \
    and np.sign(f[-4]-f[-5]) * np.sign(f[-3]-f[-4])==-1 \
    and np.sign(f[-3]-f[-4]) * np.sign(f[-2]-f[-3])==-1:

    # the monotonicity filter corrects the value at (fim1) by
    # substituting (fim1) with fmon, see Eq. (28), (29) and (30)
    fmon = min(3.*g[i-2], \n            3.*g[i-1], \n            np.sqrt(max(0,(18./13.*g[i-2] - 5./13.*f[-7]) * \n                     (18./13.*g[i-1] - 5./13.*f[-1]))))

    # re-computation of the sub-grid interval values while the
    # interval boundaries (fi) and (fip2) remains unchanged
    # see Eq. (20) and (21)
    f[-6]=3./2.*g[i-2]-1./12.*f[-7]-5./12.*fmon
    f[-5]=3./2.*g[i-2]-5./12.*f[-7]-1./12.*fmon
    f[-4]=fmon
\[
\begin{align*}
    f[-3] &= 3./2.*g[i-1] - 1./12.*fmon - 5./12.*f[-1] \\
    f[-2] &= 3./2.*g[i-1] - 5./12.*f[-1]
\end{align*}
\]

# add next interval of interpolated (sub-)grid values
f.append(fi1)
f.append(fi2)
f.append(fip1)

# separate treatment of the final interval

# as a requirement:
# if there is a zero data value such that \( g[i] = 0 \), then the whole
# interval in \( f \) has to be zero to such that \( f[i+1] = f[i+2] = f[i+3] = 0 \)
# according to Eq. (12)
if g[-1]==0.:
    # apply monotonicity filter for interval before (Eq. (27))
    # check if there is "M" or "W" shape
    if np.sign(f[-5]-f[-6]) * np.sign(f[-4]-f[-5])==-1 \
        and np.sign(f[-4]-f[-5]) * np.sign(f[-3]-f[-4])==-1 \
        and np.sign(f[-3]-f[-4]) * np.sign(f[-2]-f[-3])==-1:
        # the monotonicity filter corrects the value at \( (fim1) \) by
        # substituting \( (fim1) \) with \( (fmon) \), see Eq. (28), (29) and (30)
        fmon = \min(3.*g[-3], \ 
                    3.*g[-2], \ 
                    np.sqrt(\max(0 ,(18./13.*g[-3] - 5./13.*f[-7]) * \\
                                (18./13.*g[-2] - 5./13.*f[-1]))))

    # re-computation of the sub-grid interval values while the
    # interval boundaries \( (fi) \) and \( (fip2) \) remains unchanged
    # see Eq. (20) and (21)
    f[-6]=3./2.*g[-3]-1./12.*f[-7]-5./12.*fmon
    f[-5]=3./2.*g[-3]-5./12.*f[-7]-1./12.*fmon
    f[-4]=fmon
    f[-3]=3./2.*g[-2]-1./12.*fmon-5./12.*f[-1]
    f[-2]=3./2.*g[-2]-5./12.*fmon-1./12.*f[-1]

f.extend([0.,0.,0.])

# otherwise the sub-grid values are calculated and added to the list
# using the persistence hypothesis as boundary condition
else:
    # temporal save of last value in interpolated list
    # since it is the left boundary and hence the new \( (fi) \) value
    fi = f[-1]

    # since last interval in series, last value is also fip1
    fip1 = g[-1]

    # the function value at the first sub-grid point \( (fi1) \) is determined
    # according to the equal area condition with Eq. (20)
    fi1 = 3./2.*g[-1]-1./12.*fi-5./12.*fip1

    # the function value at the second sub-grid point \( (fi2) \) is determined
    # according E to the equal area condition with Eq. (21)
    fi2 = 3./2.*g[-1]-5./12.*fi-1./12.*fip1

    # apply monotonicity filter for interval before (Eq. (27))
    # check if there is "M" or "W" shape
    if np.sign(f[-5]-f[-6]) * np.sign(f[-4]-f[-5])==-1 \
        and np.sign(f[-4]-f[-5]) * np.sign(f[-3]-f[-4])==-1 \

10
and np.sign(f[-3]-f[-4]) * np.sign(f[-2]-f[-3])== -1:

# the monotonicity filter corrects the value at (fim1) by
# substituting (fim1) with (fmon), see Eq. (28), (29) and (30)
fmon = min(3.*g[-3], \
    3.*g[-2], \
    np.sqrt(max(0,(18./13.*g[-3] - 5./13.*f[-7]) * 
        (18./13.*g[-2] - 5./13.*f[-1]))))

# re-computation of the sub-grid interval values while the
# interval boundaries (fi) and (fip2) remains unchanged
# see Eq. (20) and (21)
f[-6]=3./2.*g[-3]-1./12.*f[-7]-5./12.*fmon
f[-5]=3./2.*g[-3]-5./12.*f[-7]-1./12.*fmon
f[-4]=fmon
f[-3]=3./2.*g[-2]-1./12.*fmon-5./12.*f[-1]
f[-2]=3./2.*g[-2]-5./12.*fmon-1./12.*f[-1]

# add next interval of interpolated (sub-)grid values
f.append(fi1)
f.append(fi2)
f.append(fip1)

return f
S1.4. Interpolation algorithm IFP

```python
def IFP(alist):
    """
    *******************************************************************************
    * Copyright 2017
    * Paul James, Leopold Haimberger
    * 
    * This work is licensed under the Creative Commons Attribution 4.0
    * International License. To view a copy of this license, visit
    * http://creativecommons.org/licenses/by/4.0/ or send a letter to
    * Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.
    *******************************************************************************
    @Author: 
    Originally written by P. James 2000 in Fortran
    for use in flex_extract.
    Transferred to Python by L. Haimberger
    
    @Description:
    This program interpolates de-accumulated fluxes of an
    ECMWF FG rainfall field using a modified linear solution.
    The interpolated values are output at the central point
    of the 4 accumulation time spans. (P. James)
    
    This is the original disaggregation algorithm from
    flex_extract. A list of 4 values is passed to it.
    From them, a new value for the 2nd position is
    reconstructed and returned as function value.
    (A. Philipp)
    
    @Input
    alist: list of float
    List of 4 float values.
    @Return
    newval: float
    New value which substitutes the old value at
    position 2 of the passed list.
    """
    xa=alist[0]
    xb=alist[1]
    xc=alist[2]
    xd=alist[3]
    
    if (xa+xc>0.):
        xac=xb*xc/(xa+xc)
    else:
        xac=0.5*xb
    
    if (xb+xd>0.):
        xbd=xb*xc/(xb+xd)
    else:
        xbd=0.5*xc
    
    newval=xac+xbd
    
    return newval
```

12
S2. ECMWF MARS Extraction Code

The following code can be used to retrieve the data used from ECMWF’s MARS archive. Note that interactive access to ecaccess/ecgate is needed for this form of the code, and for the access to the operational data streams. It should not be too difficult, however, to modify this code in a way that it can be used to retrieve reanalysis data through the python API interface, an option that is available to the public after registration.

The 1-hourly and the 3-hourly data sets are extracted separately. Furthermore, precipitation data are not analysed parameters, therefore only forecasts can be used. We use the forecasts based on the 00 UTC analysis for the data between 01 and 12 UTC, and forecasts based on the 12 UTC analysis for the data between 12 and 24 UTC (00 UTC next day). A retrieval consists of the introductory code word retrieve and the parameters which specify the data. Parameter from the first retrieval not specified again with subsequent retrieve commands remain valid. Precipitation data for the 1st of January, 00 UTC, need to be fetched separately. Data are retrieved as GRIB files. The sample code below extracts convective and large-scale precipitation at once.

S2.1. MARS retrieval for 3-hourly data

mars_request_3h <<EOF
retrieve,
# retrieve the first field separately
# because midnight is not included in
# general daily extraction (will be the forecast
# from the day before)
class=od,
expver=1,
stream=oper,
levtype=sfc,
type=fc,
time=0000,
step=0,
date=20140101/to/20140101,
grid=0.5/0.5,
resol=319,
# N/W/S/O
area=90/-180/-90/179.5,
param=LSP/CP,
target="ecmwf_lsp-cp_3h_2014"
retrieve,
# retrieve data every 3 h between 00 and 12 UTC
# excluding 00 UTC
# including forecast for 12 UTC
time=0000,
step=3/TO/12/BY/3,
date=20140101/to/20141231
retrieve,
# retrieve data every 3 h between 12 and 00 UTC
# excluding 12 UTC
# including forecast for 00 UTC on the next day
time=1200,
step=3/TO/12/BY/3,
date=20140101/to/20141231
EOF

mars_request_3h # execute the request
S2.2. MARS retrieval for 1-hourly data

mars_request_1h <<EOF

retrieve,
# retrieve the first field separately
# because midnight is not included in
# general daily extraction (will be the forecast
# from the day before)
class=od,
expver=1,
stream=oper,
levtype=sfc,
type=fc,
time=0000,
step=0,
date=20140101/to/20140101,
grid=0.5/0.5,
resol=319,
# N/W/S/O
area=90/-180/-90/179.5,
param=LSP/CP,
target="ecmwf_lsp-cp_1h_2014"
retrieve,
# retrieve data every 1 h between 00 and 12 UTC
# excluding 00 UTC
# including forecast for 12 UTC
time=0000,
step=1/TO/12/BY/1,
date=20140101/to/20141231
retrieve,
# retrieve data every 1 h between 12 and 00 UTC
# excluding 12 UTC
# including forecast for 00 UTC on the next day
time=1200,
step=1/TO/12/BY/1,
date=20140101/to/20141231

EOF

mars mars_request_1h # execute the request

S2.3. Splitting the GRIB files

The code provided above collects the data in a single big GRIB file. In order to split it into daily files, create a file `gfilter` containing the following lines:

```
# gribfilter for splitting data into daily files
# change <STEP> to 1h for 1-hourly 3h for the 3-hourly data file
write "ecmwf_lsp-cp_<STEP>_[dataDate]";
```

and execute

```
grib_filter gfilter <infilename>.grib
```

`<STEP>` and `<infilename>` have to be replaced by appropriate values.