2019-ssb-fd

September 15, 2019

1 DL0ABT IARU Region 1 FD SSB

Es handelt sich hier um eine Auswertung der QSOs der Amateurfunk-Klubstation DL0ABT während der Teilnahme am IARU Region 1 SSB Fieldday am 7-8.9.2019: Op, Bänder, QSO-Raten.

Wir haben in der Klasse "fixed" teilgenommen mit Stromanschluss, bis zum Abend des 7.9. mit 100 W, anschließend mit einer Endstufe mit Sendeleistung im Bereich 200-500 W PEP, beides an einer zwischen zwei Bäumen in ca 10..15 m Höhe aufgespannten G5RV-Dipolantenne.

QTH war JO62nf, genauer 52,2419° N und 13,1544° Ost in Glau südlich von Berlin, im Gebiet DLFF-0095. Wir sind dort mit unserer Clubstation seit Jahren immer wieder gerne Gäste der Johannischen Kirche.

Das hier veröffentliche Material enthält personenbezogene Daten von zwei teilnehmenden Funkamateuren. Beide haben der Veröffentlichung zugestimmt (Danke, Jan!).

1.1 Grenzen der Auswertung

Ich habe es aus Zeitnot leider nicht geschafft, irgendwelche Auswertungen über Multiplikatoren zu machen.

1.2 Benutzte Software

Es handelt sich um ein Jupyter Notebook (entwickelt und gerendert mit jupyter lab 1.1.3). Außer Jupyter und Python habe ich Matplotlib benutzt. Wer nur die PDF-Datei hat und den Code nicht abtippen will: Der Sourcecode ist auch komplett verfügbar.

1.3 Open Source

oder der VERWENDBARKEIT FÜR EINEN BESTIMMTEN ZWECK. Details finden Sie in der GNU General Public License.

Sie erhalten ein Exemplar der GNU General Public License von http://www.gnu.org/licenses/.

1.4 Software importieren

[1]:
```
import re
from datetime import timedelta, datetime, timezone
import functools
import collections

import matplotlib as mpl
import matplotlib.pyplot as plt # https://matplotlib.org/contents.html
from matplotlib.dates import DateFormatter
```

1.5 Adif log einlesen

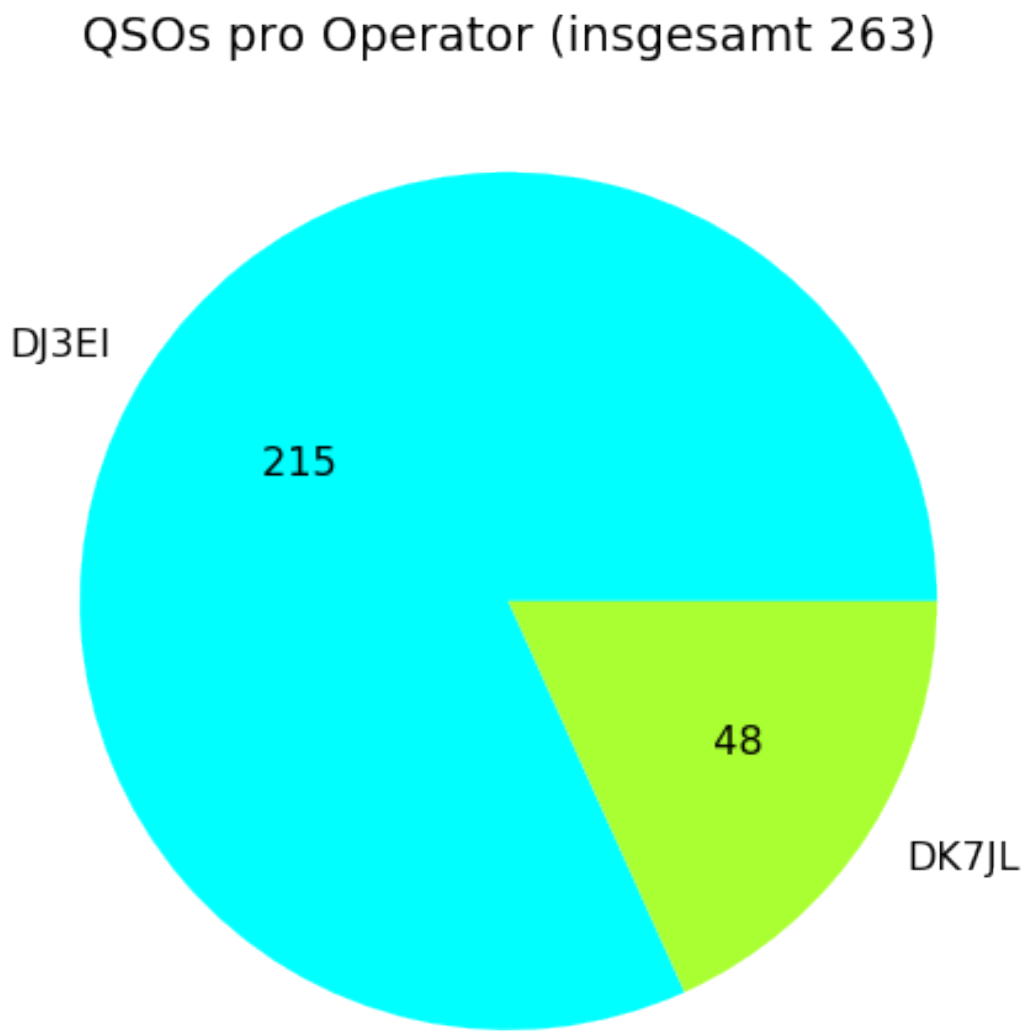
[2]:
```
# This is an ADIF parser in Python.

# It knows nothing about ADIF data types or enumerations,
# so it is fairly simple.

# But it does correcly handle things like:
# <notes:66>In this QSO, we discussed ADIF and in particular the <eor> marker.
# So, in that sense, this parser is somewhat sophisticated.

# Main result of parsing: List of QSOs.
# Each QSO is one Python dict.
# Keys in that dict are ADIF field names in lower case,
# value for a key is whatever was found in the ADIF.
# Order of QSOs is same as in ADIF file.
qsos = []

# The ADIF file header keys and values, if any. (Not needed by this notebook.)
adif_headers = {}

header_field_re = re.compile(r'<((eoh)|(\w+)\:(\d+)(\:[^>]+)?)>', re.IGNORECASE)
field_re = re.compile(r'<((eor)|(\w+)\:(\d+)(\:[^>]+)?)>', re.IGNORECASE)

with open("DL0ABT-ssb-fd-2019-09.adif.adi") as adif_file:
    adif = adif_file.read()
    cursor = 0
    if adif[0] != '<':
        # Has ADIF header.
```

```
        eoh_found = False
        while(not eoh_found):
            header_field_mo = header_field_re.search(adif, cursor)
            if header_field_mo.group(2):
                eoh_found = True
                cursor = header_field_mo.end(0)
            else:
                field = header_field_mo.group(3).lower()
                value_start = header_field_mo.end(0)
                value_end = value_start + int(header_field_mo.group(4))
                value = adif[value_start:value_end]
                adif_headers[field] = value
                cursor = value_end

    qso = {}
    field_mo = field_re.search(adif, cursor)
    while(field_mo):
        if field_mo.group(2):
            # <eor> found:
            qsos.append(qso)
            qso = {}
            cursor = field_mo.end(0)
        else:
            # Field found:
            field = field_mo.group(3).lower()
            value_start = field_mo.end(0)
            value_end = value_start + int(field_mo.group(4))
            value = adif[value_start:value_end]
            qso[field] = value
            cursor = value_end
        field_mo = field_re.search(adif, cursor)
```

1.6 QSOs pro Operator

[3]:
```
mpl.rcParams['font.size'] = 16
mpl.rcParams['legend.fontsize'] = 'large'
mpl.rcParams['figure.titlesize'] = 'large'


def filter_by_op(qso):
    return 'DK7JL' if qso['operator'] == 'DL0ABT' else qso['operator']

def plot_by_selection(qsos, select, title, sort_sel=None):

    def inc_sel2num(sel2num, qso):
        sel = select(qso)
```

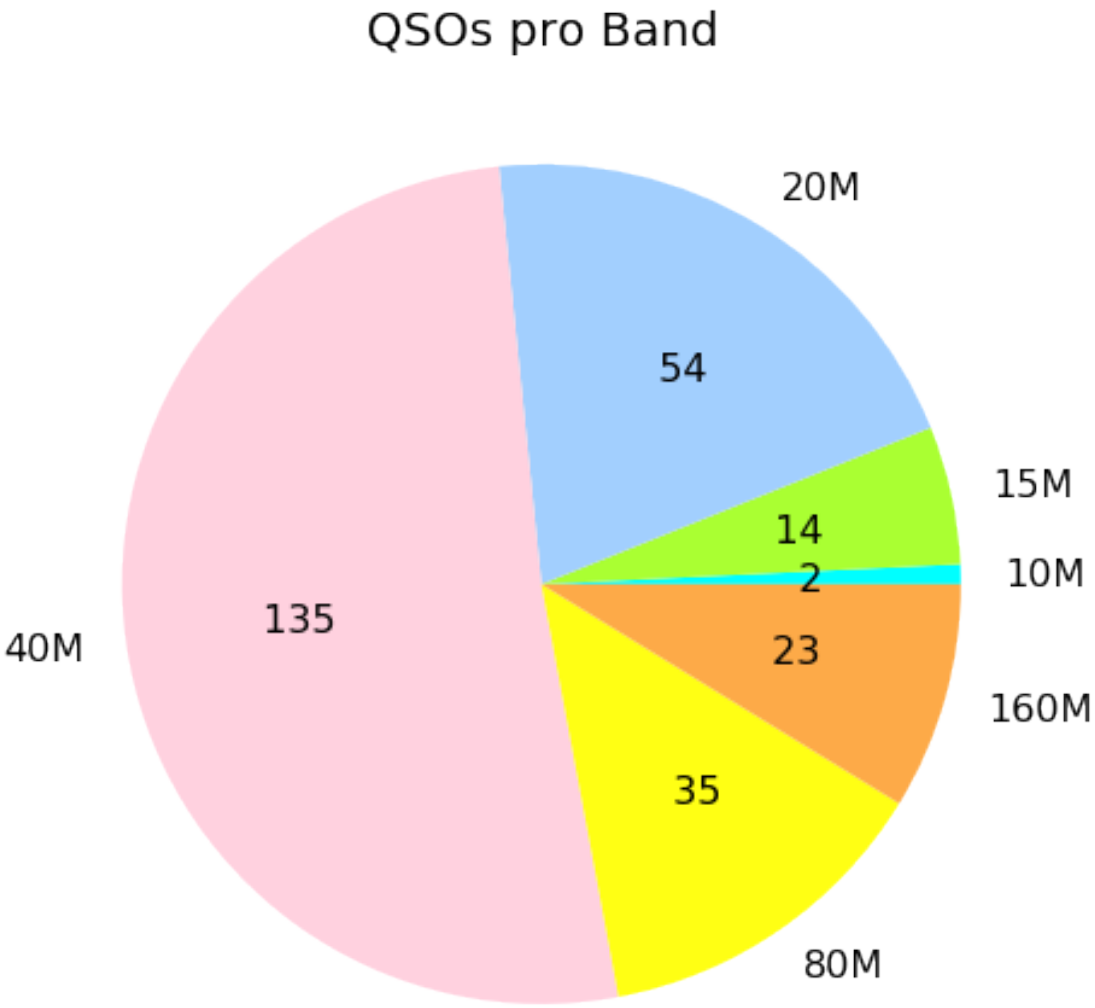
```
        sel2num[sel] = sel2num[sel] + 1
        return sel2num

    sel2num = functools.reduce(inc_sel2num, qsos, collections.Counter())
    sels = sorted(sel2num.keys(), key=sort_sel)
    nums = [sel2num[sel] for sel in sels]
    fig, ax = plt.subplots(figsize = [8,8])

    ax.pie(nums, labels = sels, autopct = lambda x: "{:4.0f}".format(x *␣
↪len(qsos) / 100),
           colors = ['xkcd:cyan', 'xkcd:lime', 'xkcd:baby blue',
                     'xkcd:light pink', 'xkcd:yellow', 'xkcd:light orange' ])
    ax.set_title(title)
    plt.show(block = True)
    return sel2num

title = 'QSOs pro Operator (insgesamt {})'.format(len(qsos))
plot_by_selection(qsos, filter_by_op, title)
pass
```



Es war auch mal ganz schön, so eine große Station über lange Zeit ”für mich” zu haben. Aber noch mehr hätte mich gefreut, wenn ich nicht über weite Strecken alleine gefunkt hätte, sondern mehr Op über längere Zeit mitgemacht hätten. DJ3EI

1.7 QSOs pro Band

[4]:
```
def sort_key_band(band):
    return "0{}".format(band) if len(band) == 3 else band
plot_by_selection(qsos, lambda qso: qso['band'], 'QSOs pro Band', sort_key_band)
pass
```

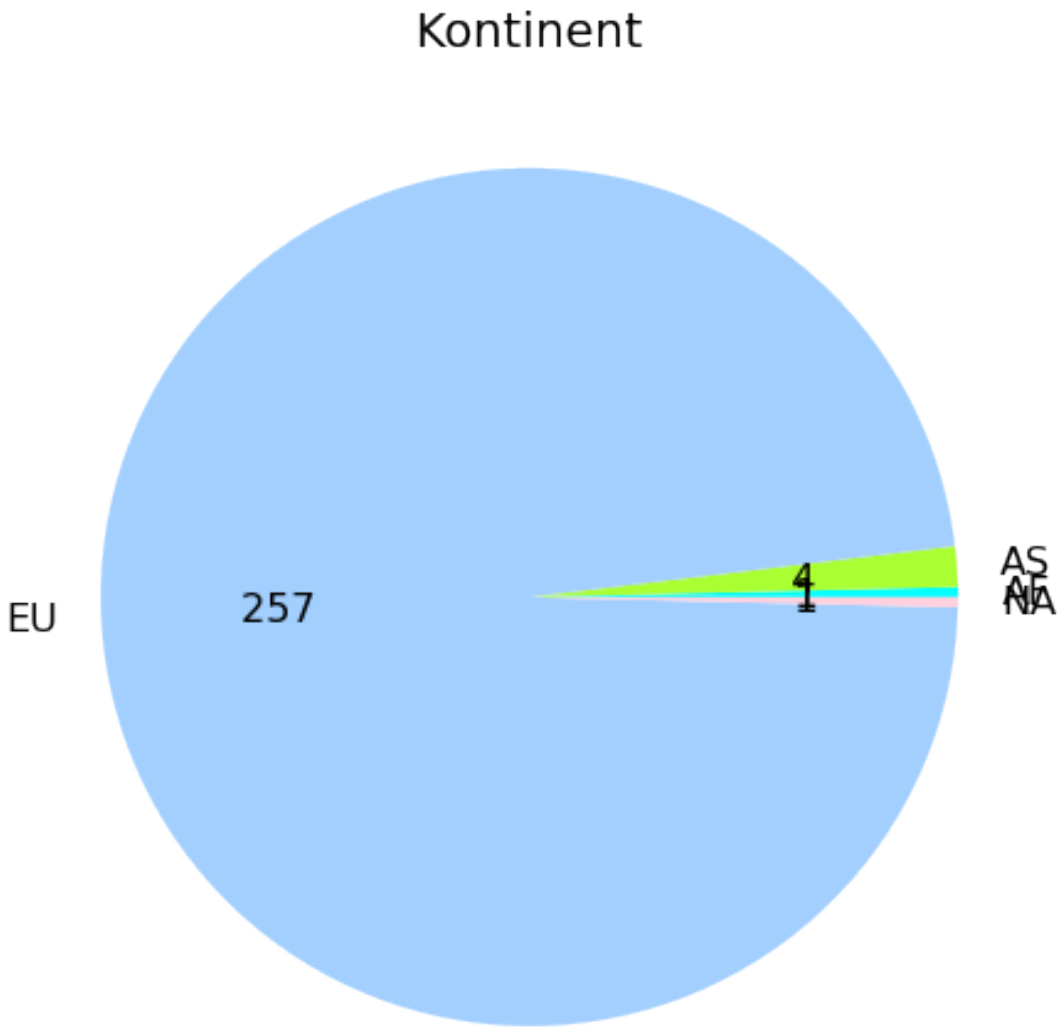


1.8 Erreichte Kontinente

Das derzeitige Sonnenfleckenminimum hat kräftig zugeschlagen, wir sind nur sehr wenig über Europa hinausgekommen.

Daher sind die Zahlen ist in der Graphik schlecht zu erkennen. Dieselben Daten folgen daher als Texttabelle, wie Python sie ausgibt.

[5]:
```
plot_by_selection(qsos, lambda qso: qso['app_n1mm_continent'], 'Kontinent')
```

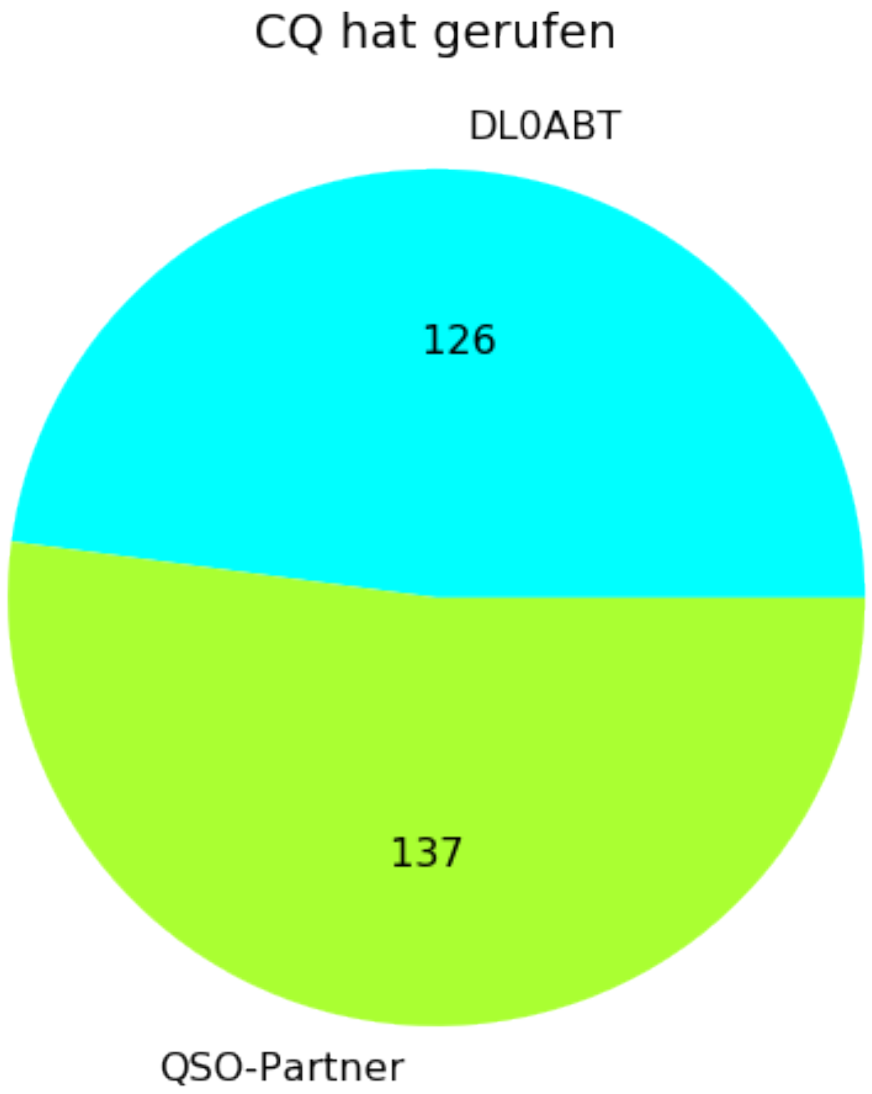


[5]: Counter({'AF': 1, 'AS': 4, 'EU': 257, 'NA': 1})

1.9 Suchen vs. CQ rufen

[6]:
```
l = len(qsos)
if 2 < l:

    def b2wer(b):
        return 'DL0ABT' if b else 'QSO-Partner'
```

```
    qsos[0]['cq'] = b2wer(qsos[0]['freq'] == qsos[1]['freq'])
    qsos[l-1]['cq'] = b2wer(qsos[l-1]['freq'] == qsos[l-2]['freq'])
    for i in range(1, l-1):
        freq = qsos[i]['freq']
        qsos[i]['cq'] = b2wer((qsos[i-1]['freq'] == freq) or (qsos[i+1]['freq']␣
↪== freq))
    plot_by_selection(qsos, lambda qso: qso['cq'], 'CQ hat gerufen')
```



1.10 QSO-Rate

Die QSO-Rate (in QSOs pro Stunde) berechne ich wie folgt: Ein einzelnes QSO erzeugt ein "Signal" bei der QSO-Rate, das einem umgekehrten V gleicht: Ab einer Viertelstunde vorher fängt die QSO-Rate an, linear zu steigen. Zum Zeitpunkt des QSOs erreicht sie den Wert 4. Dann sinkt sie wieder linear ab, genau eine Viertelstunde nach dem QSO ist sie bei Null angekommen.

Diese einzelnen Signale aller QSOs werden aufaddiert. Das ist die unten dargestellte QSO-Raten-Kurve.

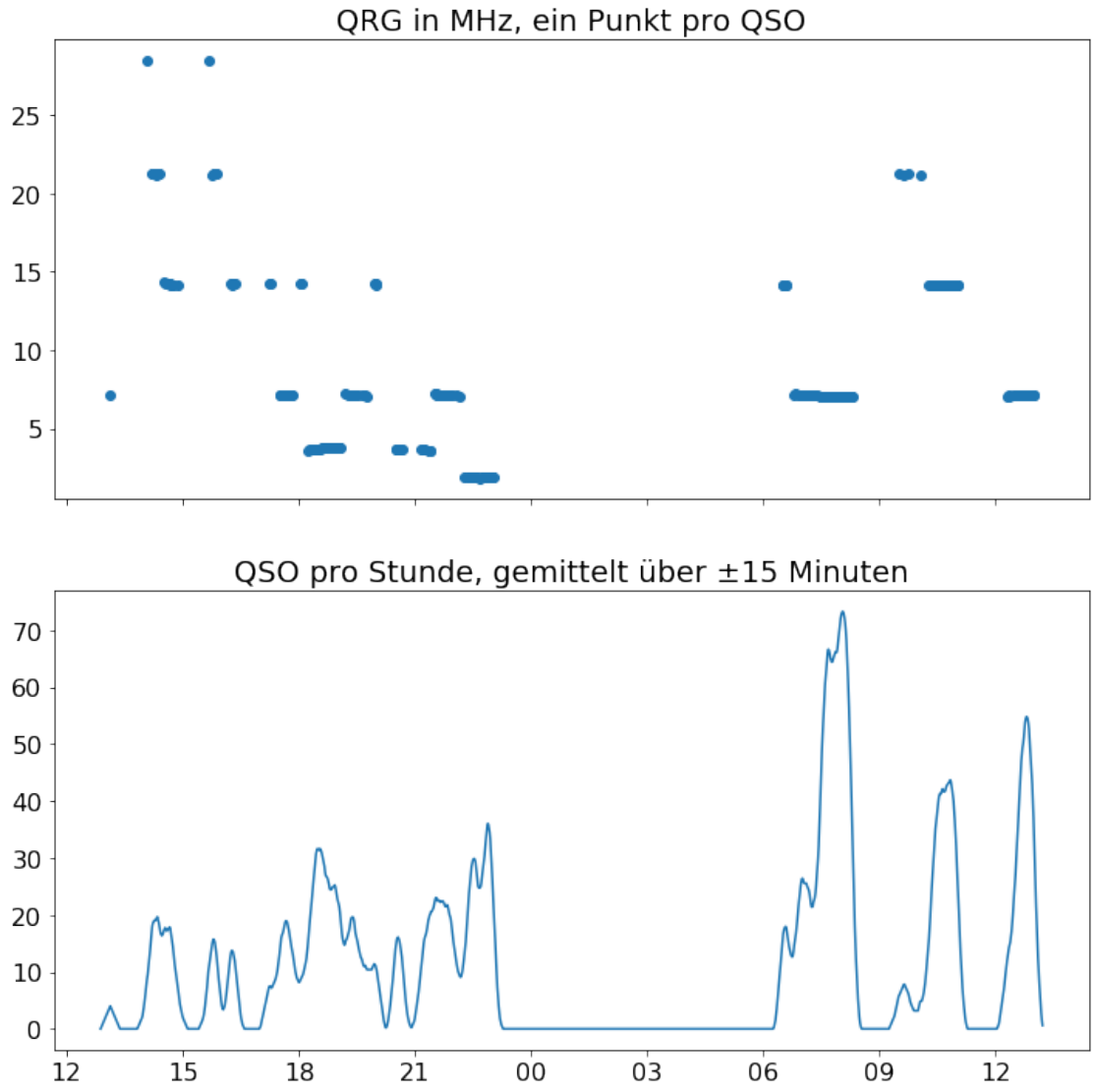
1.10.1 Warum gerade der Wert 4?

Wenn man über lange Zeit regelmäßig jede 15 Minuten ein QSO führt, bleibt die Linie konstant bei 4 stehen. Denn die absteigende Flanke vom letzten QSO und die steigende vom kommenden addieren sich zu einer konstanten Linie der Höhe 4. Wer genau alle Viertelstunde ein QSO macht, macht 4 QSOs pro Stunde.

Allgemeineres Argument: Die Fläche unter der QSO-Raten-Linie (mathematisch: das Integral der QSO-Rate über der Zeit) entspricht genau der Anzahl der QSOs.

[7]:
```
def timeon(qso):
    date = qso['qso_date']
    y = int(date[0:4])
    mo = int(date[4:6])
    d = int(date[6:8])
    time = qso['time_on']
    h = int(time[0:2])
    mi = int(time[2:4])
    s = int(time[4:6])
    return datetime(y, mo, d, h, mi, s, tzinfo = timezone.utc)

qsotimes = [timeon(qso) for qso in qsos]

# Integrating over simple inverted - V function
# spanning +/- 15 minutes.
d = timedelta(minutes=15)
def qsos_per_hour(qsotimes, t):
    start = t - d
    end = t + d
    def nearness_to_t(time):
        if time <= start:
            return 0.0
        elif end <= time:
            return 0.0
        else:
            return (1.0 - (abs(t-time) / d)) * 4.0
    return functools.reduce(lambda x, t_qso: x + nearness_to_t(t_qso),␣
↪qsotimes, 0.0)
```

```
step = timedelta(minutes=1)
overall_start = qsotimes[0] - d
overall_end = qsotimes[len(qsotimes) - 1] + d
ts = [overall_start + i * step for i in range(0, int((overall_end -␣
 ↪overall_start)/step))]
# This is incredibly inefficient:
rates = [qsos_per_hour(qsotimes, t) for t in ts]

# As a test, a crude integration:
# print("total: {}".format(functools.reduce(lambda x, r: x + r, rates, 0.0)/60.
 ↪0))

fig, axs = plt.subplots(nrows = 2, figsize = [12,12], sharex = 'col')
ax = axs[1]
ax.set_title("QSO pro Stunde, gemittelt über ±15 Minuten")
ax.plot_date(ts, rates, '-', xdate = True, ydate = False)
axs[0].set_title("QRG in MHz, ein Punkt pro QSO")
axs[0].plot_date(qsotimes, [float(qso['freq']) for qso in qsos], 'o', xdate =␣
 ↪True, ydate = False)
ax.xaxis.set_major_formatter(DateFormatter('%H'))
plt.show(block = True)
pass
# Maximale QSO-Rate:
# functools.reduce(lambda max, v: v if max < v else max, rates)
```



1.11 Tatsächliche QSO-Zahlen pro Stunde

Den Maximalwert von 73,3 QSO/Stunde haben wir natürlich nur für kurze Zeit erreicht, sozusagen im Sprint. Als Realitätscheck hier die tatsächlichen QSO-Zahlen pro (voller) Stunde:

[8]:
```
one_hour = timedelta(minutes=60)
test_start = datetime(2019, 9, 7, 13, 0, 0, tzinfo = timezone.utc)
ind = [i for i in range(0, 25)]
hours = [test_start + one_hour * i for i in ind]
def qsos_that_hour(hour):
    def inc_if_in_hour(count, qsotime):
        if hour <= qsotime and qsotime < hour + one_hour:
```

```
            return count + 1
        else:
            return count
    return functools.reduce(inc_if_in_hour, qsotimes, 0)
qsos_per_hour = [qsos_that_hour(hour) for hour in hours]
fig, axs = plt.subplots(figsize = [12, 6])
plt.bar([i + 0.5 for i in ind], qsos_per_hour)
plt.title("QSO Anzahl pro voller Stunde")
plt.xticks(ind, [h.strftime('%H') for h in hours])
plt.show()
pass
```