By Philippe Nguyen

In [1]:
```
import csv
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import time
%matplotlib inline
```

First data set - Simple linear regression

In [2]:
```
d = np.genfromtxt('data.txt', delimiter='\t', defaultfmt='.3f')
x,y = [np.array([x[i] for x in d if x[0] > 0]) for i in [0,1]]
```

In [3]:
```
def leastSq(x1,y1):
    """Simple linear regression function"""
    N = len(x1)
    xx = np.sum(x1**2)
    xy = np.sum(x1*y1)
    delta = (N * xx) - np.sum(x1)**2

    # A = y-intercept, B = slope
    A = (1/delta) * (xx*np.sum(y1) - np.sum(x1)*xy)
    B = (1/delta) * (N*xy - np.sum(x1)*np.sum(y1))
    yErr = np.sqrt(np.sum((y1 - A - B*x1)**2) / (N-2))
    AErr = yErr * np.sqrt(xx/delta)
    BErr = yErr * np.sqrt(N/delta) # uncertainty in slope

    return A, B, yErr, AErr, BErr

chiSq = lambda obs, exp: 1/(len(obs)-2) * np.sum((obs - exp)**2/exp)
```

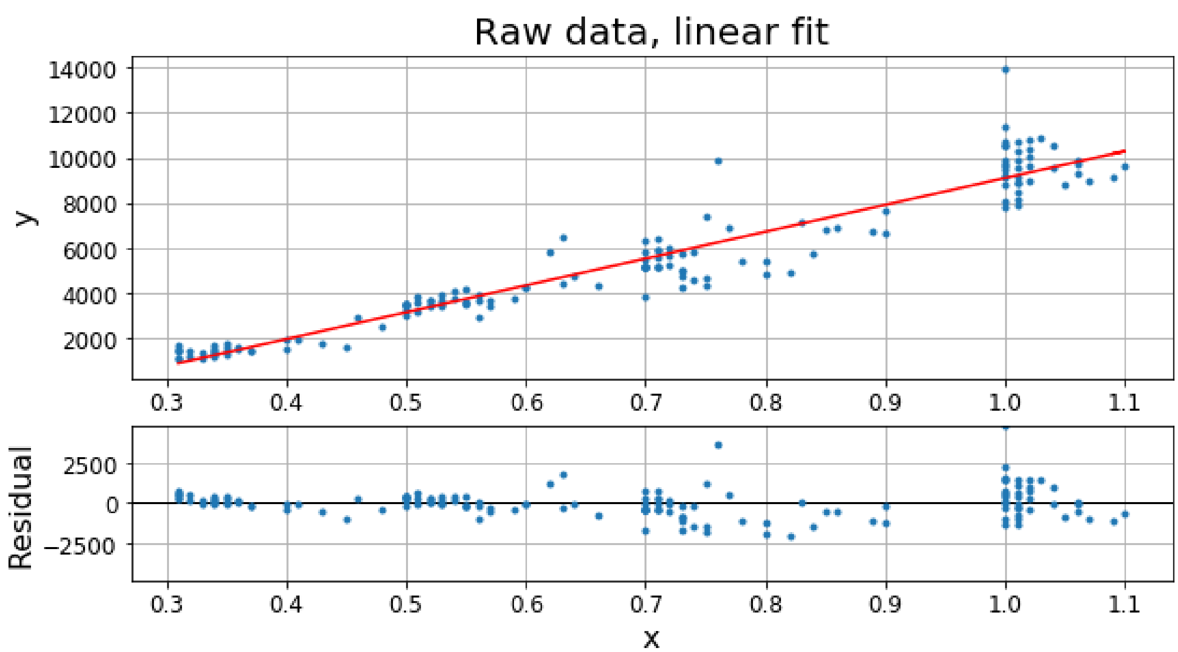
Fit applied to full data set

In [4]:

```
# Apply fitting function, get residuals
A, B, yErr, AErr, BErr = leastSq(x,y)
res = y - (A + x*B)

# Print fit results
print('Linear regression results:',
      '\n\nSlope = {:.0f}, \ny-intercept = {:.0f},\
      \nsigma = {:.0f}, \nslope error = {:.0f},\
      \ny-intercept error = {:.0f}'
      .format(B, A, yErr, BErr, AErr))
print('\nAt x = 1.2, y = {:.0f} +/- {:.0f}'
      .format(A + 1.2*B, yErr))
print('\nChi-Squared = {:.0f}'.format(chiSq(y, A + x*B)))
```

```
Linear regression results:

Slope = 11916,
y-intercept = -2811,
sigma = 906,
slope error = 303,
y-intercept error = 222

At x = 1.2, y = 11488 +/- 906

Chi-Squared = 134
```

In [5]:
```
plt.figure(figsize=(10,7))
plt.subplot(2,1,1)
plt.plot(x, y, '.')
plt.plot(x, A + x*B, 'r-')
plt.xticks(size=12)
plt.yticks(size=12)
plt.title('Raw data, linear fit', size=20)
plt.ylabel('y', size=16)
plt.grid()

plt.subplot(4,1,3)
plt.axhline(y=0, c='k', lw=1)
plt.plot(x, res, '.')
plt.ylim(-max(np.abs(res)), max(np.abs(res)))
plt.xticks(size=12)
plt.yticks(size=12)
plt.ylabel('Residual', size=16)
plt.xlabel('x', size=16)
plt.grid()

plt.show()
```



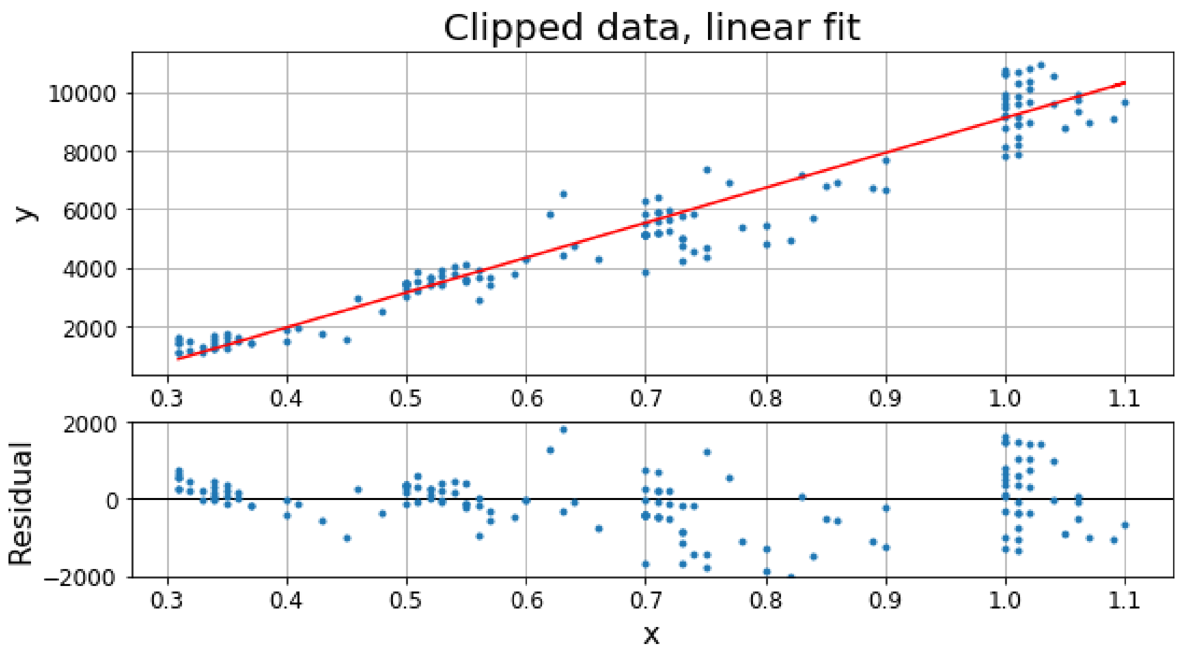
Fit applied to clipped (res < 2.5σ) data set

In [6]:

```
# Data clipped at 2.5*sigma
idx = np.where(res < 2.5*yErr)
x1,y1 = [x[idx], y[idx]]

# Apply fitting function, get chi-sq and residuals
A1, B1, yErr1, AErr1, BErr1 = leastSq(x1,y1)
res1 = y1 - (A + x1*B)

# Print fit results
print('Linear regression results:',
      '\n\nSlope = {:.0f}, \ny-intercept = {:.0f},\
      \nsigma = {:.0f}, \nslope error = {:.0f},\
      \ny-intercept error = {:.0f}'
      .format(B1, A1, yErr1, BErr1, AErr1))
print('\nAt x = 1.2, y = {:.0f} +/- {:.0f}'
      .format(A + 1.2*B, yErr1))
print('\nChi-Squared = {:.0f}'.format(chiSq(y1, A + x1*B)))
```

```
Linear regression results:

Slope = 11627,
y-intercept = -2687,
sigma = 734,
slope error = 249,
y-intercept error = 181

At x = 1.2, y = 11488 +/- 734

Chi-Squared = 100
```

In [7]:
```
plt.figure(figsize=(10,7))
# plt.subplots_adjust(hspace=0.1)
plt.subplot(2,1,1)
plt.plot(x1, y1, '.')
plt.plot(x1, A + x1*B, 'r-')
plt.xticks(size=12)
plt.yticks(size=12)
plt.title('Clipped data, linear fit', size=20)
plt.ylabel('y', size=16)
plt.grid()

plt.subplot(4,1,3)
plt.axhline(y=0, c='k', lw=1)
plt.plot(x1, res1, '.')
plt.ylim(-max(np.abs(res1)), max(np.abs(res1)))
plt.xticks(size=12)
plt.yticks(size=12)
plt.ylabel('Residual', size=16)
plt.xlabel('x', size=16)
plt.grid()

plt.show()
```



Comparison

In [8]:
```
print(yErr1 < yErr)
print(chiSq(x1, A + x1*B) < chiSq(x, A + x*B))
```

```
True
True
```

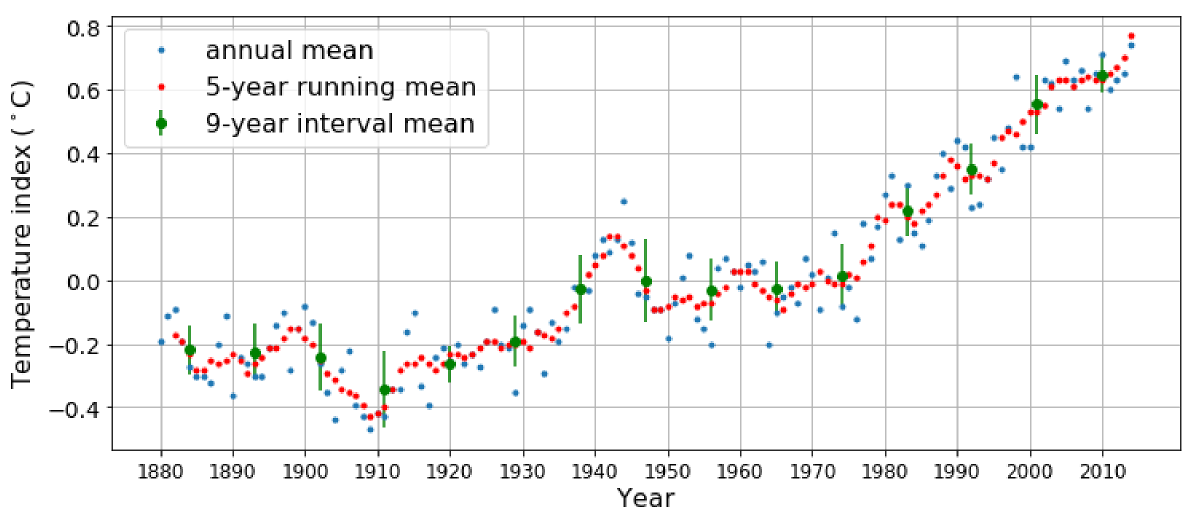
We get a smaller uncertainty in the value at x = 1.2 (y error = 734 instead of 906) and smaller χ^2 value. The changes overall are fairly small, since the clipping didn't remove a whole lot of points.

Global land-ocean temperature index

In [9]:
```
d = np.genfromtxt('data2.txt', delimiter=(5,10,10))
ti = pd.DataFrame(d, columns=['year','annual', '5-year'])
ti1 = ti[ti['year']<2015] # data w/o 2015-16
yearsDec1 = np.arange(1880, 2015, 9.) + 4
tiDec1 = [np.mean(ti[(ti['year']>=1880+i*9) & (ti['year']<1880+9*(i+1))]['annual']) for i in range(len(yearsDec1))]
tiDec1Std = [np.std(ti[(ti['year']>=1880+i*9) & (ti['year']<1880+9*(i+1))]['annual']) for i in range(len(yearsDec1))]
```

9-year means, without 2015-16

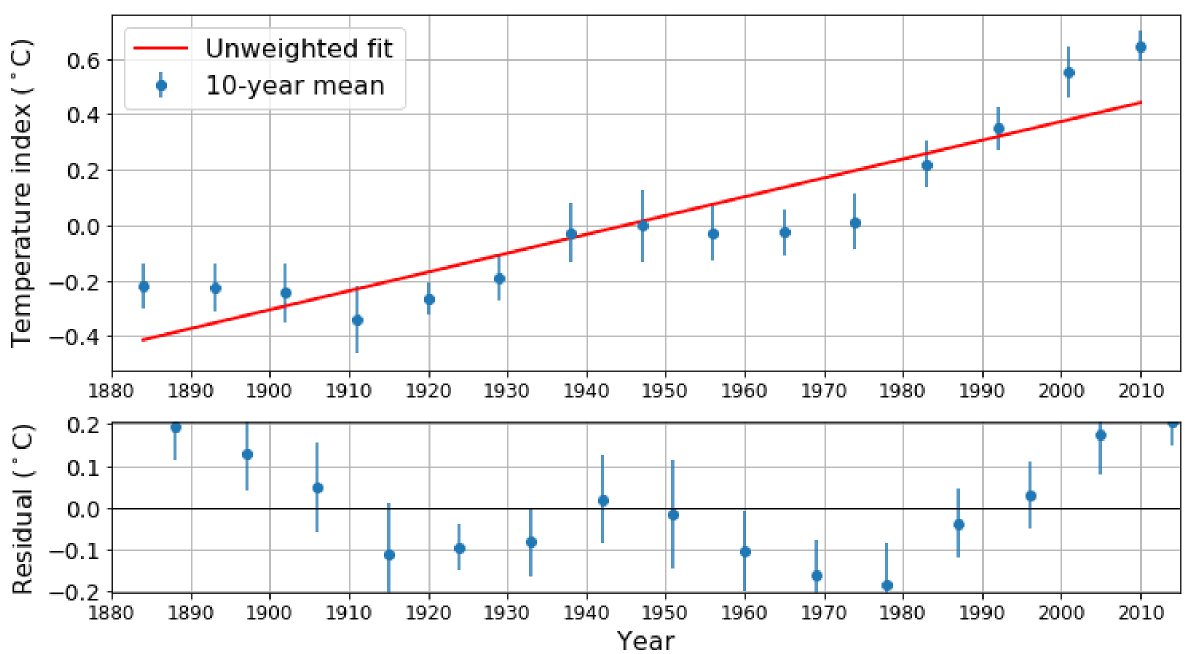
In [10]:
```
plt.figure(figsize=(12,5))
plt.plot(ti1['year'], ti1['annual'], '.', label='annual mean')
plt.plot(ti1['year'], ti1['5-year'], 'r.', label='5-year running mean')
plt.errorbar(yearsDec1, tiDec1, tiDec1Std, fmt='o', color='g', label='9-year interval mean')
plt.xticks(np.arange(1880, 2020, 10), size=12)
plt.yticks(size=14)
plt.xlabel('Year', size=16)
plt.ylabel('Temperature index ($^\circ$C)', size=16)
plt.legend(loc='upper left', fontsize=16)
plt.grid()
plt.show()
```



Unweighted least-squares, excluding 2015-16

In [11]:
```
A1, B1, yErr1, AErr1, BErr1 = leastSq(yearsDec1,tiDec1)
res1 = tiDec1 - (A1 + yearsDec1*B1)
```

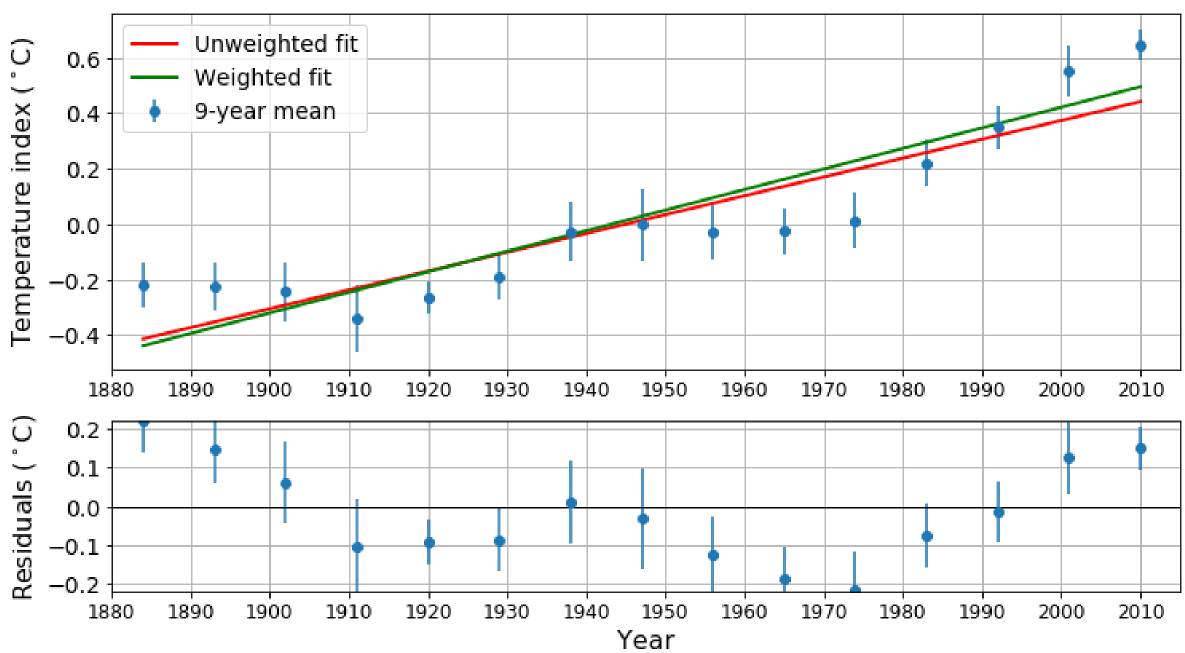
In [12]:
```
plt.figure(figsize=(12,9))
# plt.subplots_adjust(hspace=0.1)
plt.subplot(2,1,1)
plt.errorbar(yearsDec1, tiDec1, yerr=tiDec1Std, fmt='o', label='10-year mean')
plt.plot(yearsDec1, A1 + yearsDec1*B1, 'r', lw=2, label='Unweighted fit')
plt.xlim(1880, 2015)
plt.xticks(np.arange(1880, 2020, 10), size=12)
plt.yticks(size=14)
# plt.xlabel('Year', size=16)
plt.ylabel('Temperature index ($^\circ$C)', size=16)
plt.legend(loc='upper left', fontsize=16)
plt.grid()

plt.subplot(4,1,3)
plt.errorbar(yearsDec1+4, res1, yerr=tiDec1Std, fmt='o')
plt.axhline(y=0, color='k', lw=1)
plt.xlim(1880, 2015)
plt.ylim(-max(np.abs(res1)), max(np.abs(res1)))
plt.xticks(np.arange(1880, 2020, 10), size=12)
plt.yticks(size=14)
plt.xlabel('Year', size=16)
plt.ylabel('Residual ($^\circ$C)', size=16)
plt.grid()
plt.show()
```



Weighted least squares, still excluding 2015-16

In [13]:
```python
def leastSqW(x1, y1, w):
    """Weighted least squares"""
    N = len(x1)
    wxx = np.sum(w*x1**2)
    wxy = np.sum(w*x1*y1)
    delta = np.sum(w)*wxx - np.sum(w*x1)**2

    # A = y-intercept, B = slope
    A = (1/delta) * (wxx*np.sum(w*y1) - np.sum(w*x1)*wxy)
    B = (1/delta) * (np.sum(w)*wxy - np.sum(w*x1)*np.sum(w*y1))
    yErr = np.sqrt(np.sum((w**2)*(y1 - A - B*x1)**2) / (N-2))
    AErr = yErr * np.sqrt(wxx/delta)
    BErr = yErr * np.sqrt(np.sum(w)/delta) # uncertainty in slope

    return A, B, yErr, AErr, BErr
```

In [14]:
```
weights = 1/np.array(tiDec1Std)**2 # Weight by inverse-squared-noise
Aw, Bw, yErrw, AErrw, BErrw = leastSqW(yearsDec1,tiDec1, weights)
resw = tiDec1 - (Aw + yearsDec1*Bw)

plt.figure(figsize=(12,9))
plt.subplot(2,1,1)
plt.errorbar(yearsDec1, tiDec1, yerr=tiDec1Std, fmt='o', label='9-year mean')
plt.plot(yearsDec1, A1 + yearsDec1*B1, 'r', lw=2, label='Unweighted fit')
plt.plot(yearsDec1, Aw + yearsDec1*Bw, 'g', lw=2, label='Weighted fit')
plt.xlim(1880, 2015)
plt.xticks(np.arange(1880, 2020, 10), size=12)
plt.yticks(size=14)
# plt.xlabel('Year', size=16)
plt.ylabel('Temperature index ($^\circ$C)', size=16)
plt.legend(loc='upper left', fontsize=14)
plt.grid()

plt.subplot(4,1,3)
plt.errorbar(yearsDec1, resw, yerr=tiDec1Std, fmt='o')
plt.axhline(y=0, color='k', lw=1)
plt.xlim(1880, 2015)
plt.ylim(-max(np.abs(resw)), max(np.abs(resw)))
plt.xticks(np.arange(1880, 2020, 10), size=12)
plt.yticks(size=14)
plt.xlabel('Year', size=16)
plt.ylabel('Residuals ($^\circ$C)', size=16)
plt.grid()
plt.show()
```



The unweighted fit is noticably steeper, which results in a higher estimate for the later years.

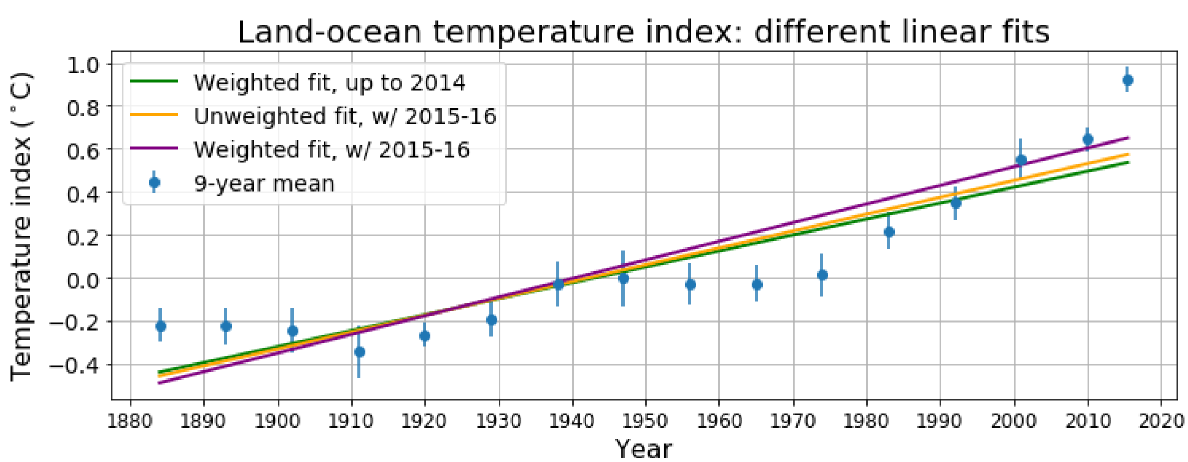
Unweighted and weighted fits, now including 2015-16

In [15]:
```
yearsDec = np.append(yearsDec1, 2015.5)
tiDec = np.append(tiDec1, np.mean(ti[(ti['year']>2014) & (ti['year']<2017)]['a
nnual']))
tiDecStd = np.append(tiDec1Std, np.std(ti[(ti['year']>2014) & (ti['year']
<2017)]['annual']))
```

In [16]:
```
A2, B2, yErr2, AErr2, BErr2 = leastSq(yearsDec,tiDec)
res2 = tiDec - (A2 + yearsDec*B2)
print('2050 forecast, unweighted: {:.3f} deg C'.format(A2 + 2050*B2))

weights2 = 1/np.array(tiDecStd)**2
A2w, B2w, yErr2w, AErr2w, BErr2w = leastSqW(yearsDec,tiDec, weights2)
res2w = tiDec - (A2w + yearsDec*B2w)
print('2050 forecast, weighted: {:.3f} deg C'.format(A2w + 2050*B2w))

print('Weighted fit chi-sq = {:.2e}'.format(chiSq(tiDec, A2w + yearsDec*B2w)))
```

```
2050 forecast, unweighted: 0.844 deg C
2050 forecast, weighted: 0.949 deg C
Weighted fit chi-sq = 4.08e-02
```

In [17]:
```
plt.figure(figsize=(12,4))
plt.errorbar(yearsDec, tiDec, yerr=tiDecStd, fmt='o', label='9-year mean')
plt.plot(yearsDec, Aw + yearsDec*Bw, 'g', lw=2, label='Weighted fit, up to 201
4')
plt.plot(yearsDec, A2 + yearsDec*B2, 'orange', lw=2, label='Unweighted fit, w/
 2015-16')
plt.plot(yearsDec, A2w + yearsDec*B2w, 'purple', lw=2, label='Weighted fit, w/
 2015-16')
plt.xticks(np.arange(1880, 2030, 10), size=12)
plt.yticks(size=14)
plt.title('Land-ocean temperature index: different linear fits', size=20)
plt.xlabel('Year', size=16)
plt.ylabel('Temperature index ($^\circ$C)', size=16)
plt.legend(loc='upper left', fontsize=14)
plt.grid()
plt.show()
```



Including the 2015-16 average bumps the later end higher, and weighting by y^2 pushes it up even more!

Sigmoidal fit, using brute force χ^2 minimization

We will now fit a symmetric sigmoidal function of the form

$$y = D + \frac{A - D}{1 + (x/C)^B}$$

which is non-linear and has 4 parameters. We'll either need something far more sophisticated or something very simple yet reliable. For now let's go for the latter and apply a brute-force, guess-and-check approach by looping over many values in the full 4-dimensional parameter space, an selecting the values that minimize χ^2 as defined before.

In [18]:
```
sigm = lambda x, A, B, C, D: D + (A - D)/(1 + (x/C)**B)
```

In [19]:
```
Avals = np.arange(-.4, -.199, .005)
Bvals = np.arange(40, 80.1, .5)
Cvals = np.arange(2000, 3010, 25)
Dvals = np.arange(300000, 401000, 2500)

csMin = 1000
t0 = time.clock()
for a in Avals:
    for b in Bvals:
        for c in Cvals:
            for d in Dvals:
                cs = chiSq(tiDec, sigm(yearsDec,a,b,c,d))
                if np.abs(cs) < np.abs(csMin):
                    A,B,C,D = [a,b,c,d]
                    csMin = cs
dt = time.clock()-t0

print('Chi-square minimization completed in {:.2f} min. Final results:\
\nA = {:.3f}, B = {:.1f}, C = {}, D = {},\nchi-sq = {}'.format(dt/60,A,B,C,D,c
sMin))
```

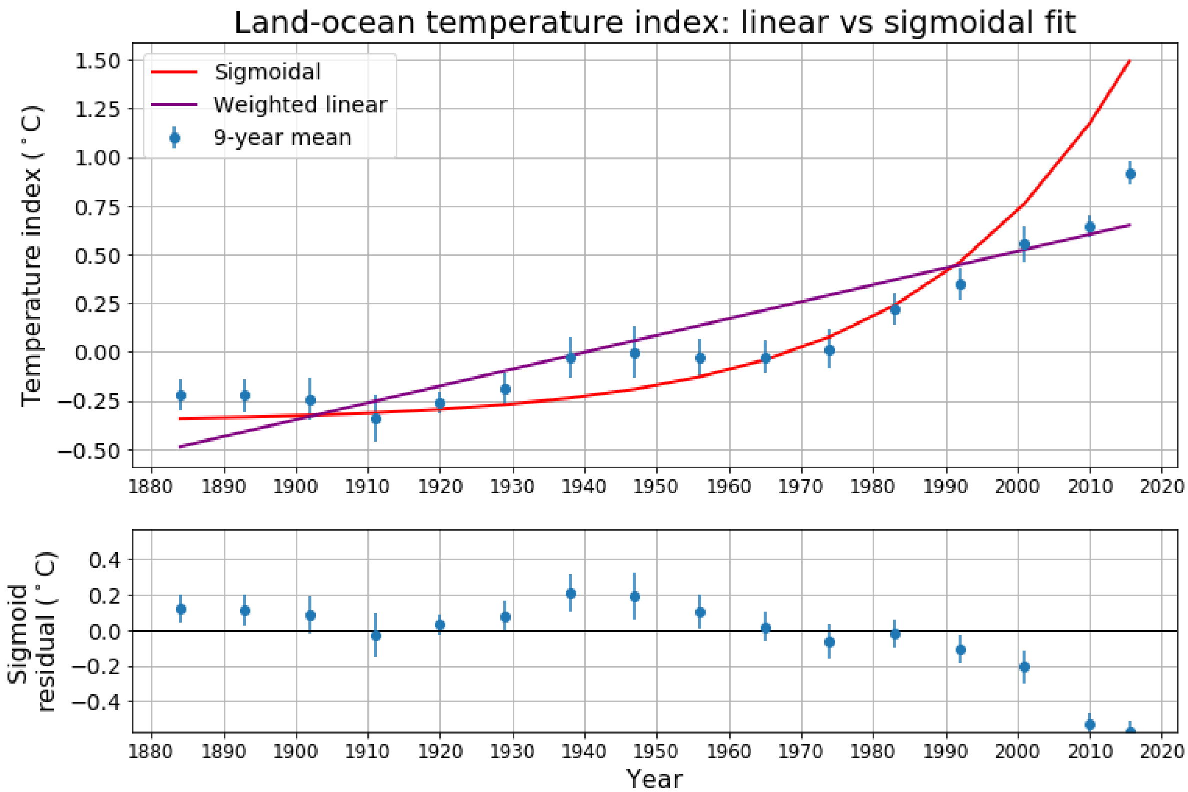
```
Chi-square minimization completed in 6.91 min. Final results:
A = -0.360, B = 69.5, C = 2400, D = 345000,
chi-sq = -7.670252874514283e-08
```

Wow that was slow, but it looks like it succeeding in finding a very small χ^2. Let's see what we got:

In [20]:

```
tiDecFit = sigm(yearsDec,A,B,C,D)
resSigm = tiDec - tiDecFit
print('2050 forecast, weighted-linear fit:\t{:.3f} deg C'.format(A2w + 2050*B2
w))
print('2050 forecast, sigmoidal fit:      \t{:.3f} deg
C'.format(sigm(2050,A,B,C,D)))
print('Weighted-linear fit chi-sq =', chiSq(tiDec, A2w + yearsDec*B2w))
print('Sigmoidal fit chi-sq       =', chiSq(tiDec, tiDecFit))

plt.figure(figsize=(12,11))
plt.subplot(2,1,1)
plt.errorbar(yearsDec, tiDec, yerr=tiDecStd, fmt='o', label='9-year mean')
plt.plot(yearsDec, tiDecFit, 'r', lw=2, label='Sigmoidal')
plt.plot(yearsDec, A2w + yearsDec*B2w, 'purple', lw=2, label='Weighted
linear')
plt.xticks(np.arange(1880, 2030, 10), size=12)
plt.yticks(size=14)
plt.title('Land-ocean temperature index: linear vs sigmoidal fit', size=20)
plt.ylabel('Temperature index ($^\circ$C)', size=16)
plt.legend(loc='upper left', fontsize=14)
plt.grid()

plt.subplot(4,1,3)
plt.axhline(y=0, lw=1, color='k')
plt.errorbar(yearsDec, resSigm, yerr=tiDecStd, fmt='o')
plt.ylim(-max(np.abs((resSigm))), max(np.abs((resSigm))))
plt.xticks(np.arange(1880, 2030, 10), size=12)
plt.yticks(size=14)
plt.title('')
plt.xlabel('Year', size=16)
plt.ylabel('Sigmoid\nresidual ($^\circ$C)', size=16)
plt.grid()
plt.show()
```

```
2050 forecast, weighted-linear fit:     0.949 deg C
2050 forecast, sigmoidal fit:           5.666 deg C
Weighted-linear fit chi-sq = 0.0408122061888
Sigmoidal fit chi-sq       = -7.67025287451e-08
```



This got us a much lower χ^2 value, but it also gives us a very alarming forecast of over 5 degress Celsius, as opposed to the 1 degree Celsius forecast from the weighted linear fit. Visually, we have reason to suspect that this isn't so good a fit despite the deceptively low χ^2. Perhaps a smarter algorithm would help us here...

Another (better) method: gradient descent

The brute-force chi-square minimization does a decent job, but it's a dumb and slow way to do things. I've heard of gradient descent algorithms being far superior for linear regression problems, so let's give it a try and we can compare the outcomes.

Gradient descent (GD) begins by taking some initial parameters, and iteratively changes them, like the brute force method above; however GD is "smarter" in how it updates the parameters. Instead of just cycling through the entire parameter space (which is very inefficient when there are 4 parameters to fit), we can compute the "gradients" of the cost-function (in this case we'll use the mathematically simpler mean-squared-error, $\frac{1}{N-4}\sum_{i=1}^{N}(O_i - E_i)^2$ as opposed to χ^2) by finding derivative with respect to each parameter. Each gradient will then be used to update current parameters. For example, the parameter A in the j-th iteration of this algorithm is assigned the value $A_j = A_{j-1} - \alpha\nabla_A$, where α is a "learning rate" that allows us to fine-tune the pace of the algorithm, and ∇_A is the gradient given by

$$\nabla_A = \frac{1}{N-4}\sum_{i=1}^{N}\frac{\partial}{\partial A}\left[y_i - \left(D + \frac{A-D}{1+(x/C)^B}\right)\right]^2$$

$$\nabla_A = -\frac{2}{N-4}\sum_{i=1}^{N}\frac{y_i - \left(D + \frac{A-D}{1+(x/C)^B}\right)}{1+(x/C)^B}$$

and likewise for B, C, and D. The point here is to always update the parameters by moving each of them in the direction of decreasing mean-squared-error, terminating the routine once we're under some threshold error.

Stochastic gradient descent (SGD) is a useful variation for dealing with much larger datasets. Instead of summing over all N data points per iteration, SGD only looks at a randomly-selected subset of the data per iteration, re-sampling every time. The smaller the subset, the lower the computational-cost-per-iteration, but at the expense of accurate error calculation.

In [21]:
```
def sigUpdateParams(x, y, A, B, C, D, rate):
    "Gradient descent parameter-updating. This is the same for normal and stoc
hastic GD."
    N = len(x)
    A_grad = -(2/(N-4)) * np.sum((y - (D + (A-D)/(1+(x/C)**B))) / (1 +
(x/C)**B))
    B_grad = (2/(N-4)) * np.sum((A-D) * ((x/C)**B) * np.log(x/C) * (y - (D +
(A-D)/(1+(x/C)**B))) / (1 + (x/C)**B)**2)
    C_grad = -(2/(N-4)) * np.sum(B * (A-D) * ((x/C)**B) * (y - (D + (A-D)/(1+
(x/C)**B))) / (C * (1 + (x/C)**B)**2))
    D_grad = (2/(N-4)) * np.sum((1/(1+(x/C)**B) - 1) * (y - (D + (A-D)/(1+
(x/C)**B))))
    A_new = A - rate*A_grad
    B_new = B - rate*B_grad
    C_new = C - rate*C_grad
    D_new = D - rate*D_grad
    return A_new, B_new, C_new, D_new

def sigGD(x, y, A_old, B_old, C_old, D_old, max_steps, rate, converge):
    """Gradient descent"""
    for i in range(max_steps):
        A_new, B_new, C_new, D_new = sigUpdateParams(x, y, A_old, B_old,
C_old, D_old, rate)
        cs = chiSq(tiDec, sigm(yearsDec, A_new, B_new, C_new, D_new))
        if np.abs(cs) < converge:
            print('Converged in', i, 'iterations')
            break
        elif i == max_steps-1:
            print('No convergence')
        A_old = A_new
        B_old = B_new
        C_old = C_new
        D_old = D_new
    return A_new, B_new, C_new, D_new, cs

def sigSGD(x, y, A_old, B_old, C_old, D_old, max_steps, rate, converge, sample
_size=10):
    """Stochastic gradient descent"""
    xy = pd.DataFrame({'x':x,'y':y})
    for i in range(max_steps):
        xy_sample = xy.sample(sample_size)
        x, y = [xy_sample['x'], xy_sample['y']]
        A_new, B_new, C_new, D_new = sigUpdateParams(x, y, A_old, B_old,
C_old, D_old, rate)
        cs = chiSq(y, sigm(x, A_new, B_new, C_new, D_new))
        if np.abs(cs) < converge:
            print('Converged in', i, 'iterations')
            break
        elif i == max_steps-1:
            print('No convergence')
        A_old = A_new
        B_old = B_new
        C_old = C_new
        D_old = D_new
    return A_new, B_new, C_new, D_new, cs
```

In [22]:
```
# Define some parameters we'll use for both GD and SGD
max_steps = int(1e4); rate = 0.1; converge = 1e-4; sample_size = 5
```

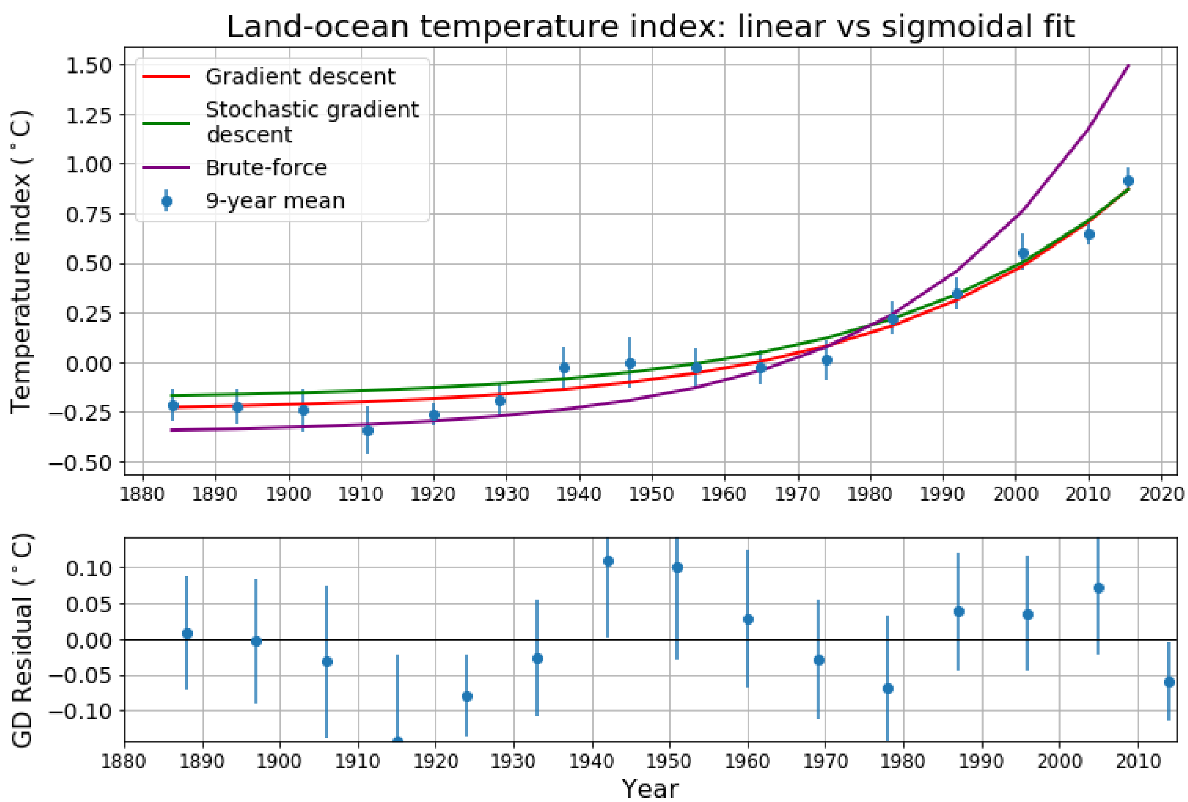
In [23]:
```
t0 = time.clock()
A_GD, B_GD, C_GD, D_GD, cs_GD = sigGD(yearsDec, tiDec, -.3, 60, 2500, 350000,
max_steps, rate, converge)
dt = time.clock() - t0
print('Computation time: {:.2f} seconds.'.format(dt))
print('A={:.2f}, B={:.2f}, C={:.0f}, D={:.0f},\ncs={:.2e}'.format(A_GD, B_GD,
C_GD, D_GD, cs_GD))
print('2050 forecast: {:.2f} deg C'.format(sigm(2050, A_GD, B_GD, C_GD,
D_GD)))
```

```
Converged in 1612 iterations
Computation time: 0.81 seconds.
A=-0.25, B=58.75, C=2500, D=350000,
cs=-9.42e-05
2050 forecast: 2.79 deg C
```

In [24]:
```
t0 = time.clock()
A_SGD, B_SGD, C_SGD, D_SGD, cs_SGD = sigSGD(yearsDec, tiDec, -.3, 60, 2500, 35
0000, max_steps, rate, converge, sample_size)
dt = time.clock() - t0
print('Computation time: {:.2f} seconds.'.format(dt))
print('A={:.2f}, B={:.2f}, C={:.0f}, D={:.0f},\ncs={:.2e}'.format(A_SGD,
B_SGD, C_SGD, D_SGD, cs_SGD))
print('2050 forecast: {:.2f} deg C'.format(sigm(2050, A_SGD, B_SGD, C_SGD, D_S
GD)))
```

```
Converged in 277 iterations
Computation time: 4.42 seconds.
A=-0.19, B=59.00, C=2500, D=350000,
cs=5.49e-05
2050 forecast: 2.69 deg C
```

In [25]:
```
resGD = tiDec - sigm(yearsDec, A_GD, B_GD, C_GD, D_GD)

plt.figure(figsize=(12,11))
plt.subplot(2,1,1)
plt.errorbar(yearsDec, tiDec, yerr=tiDecStd, fmt='o', label='9-year mean')
plt.plot(yearsDec, sigm(yearsDec, A_GD, B_GD, C_GD, D_GD), 'r', lw=2, label='G
radient descent')
plt.plot(yearsDec, sigm(yearsDec, A_SGD, B_SGD, C_SGD, D_SGD), 'g', lw=2, labe
l='Stochastic gradient\ndescent')
plt.plot(yearsDec, tiDecFit, 'purple', lw=2, label='Brute-force')
plt.xticks(np.arange(1880, 2030, 10), size=12)
plt.yticks(size=14)
plt.title('Land-ocean temperature index: linear vs sigmoidal fit', size=20)
plt.ylabel('Temperature index ($^\circ$C)', size=16)
plt.legend(loc='upper left', fontsize=14)
plt.grid()

plt.subplot(4,1,3)
plt.errorbar(yearsDec+4, resGD, yerr=tiDecStd, fmt='o')
plt.axhline(y=0, color='k', lw=1)
plt.xlim(1880, 2015)
plt.ylim(-max(np.abs(resGD)), max(np.abs(resGD)))
plt.xticks(np.arange(1880, 2020, 10), size=12)
plt.yticks(size=14)
plt.xlabel('Year', size=16)
plt.ylabel('GD Residual ($^\circ$C)', size=16)
plt.grid()

plt.show()
```



For this small a dataset, GD ran a lot quicker than SGD, but visually both get us a better fit than the brute-force method. The residuals shown here are for the GD result, and they are all around half the values of the residuals from the brute-force method. The new estimates provided is considerably more moderate as well: 2.8 $^\circ$C for both.