Introduction to High Performance Computing

Autumn, 2017

Lecture 6

Clarifications/corrections

- **1-D Random walk (lecture 4 slides):**

  - Var{X(t+n*dt)} = v t **where** v = dx^2/dt = constant   **Incorrect!**

  - Var{X(n*dt)} = v (n*dt) = v t **where** v = (dx)^2/dt = constant **Correct**

- **Lists vs. arrays**

  - **Lecture 5: lists are easier to grow/shrink than arrays Incorrect**

  - **Lists can be grown/shrunk more efficiently than arrays Correct**

  - L.append() **for a list,** L**, is more efficient than** np.append(A) **for an array,** A**. The numpy command will make a new copy of the array, the list will grow** *in place*

Clarifications/corrections

- **Lists vs. arrays**
  - **There are other subtle differences between lists and arrays**
    - **For example, indexing of 1D lists and arrays is identical**
    - **For multidimensional lists however, cannot use i,j style indexing:**

```
In [1]: L = [[1,2,3],[4,5]]

In [2]: L
Out[2]: [[1, 2, 3], [4, 5]]

In [3]: L[0,1]
---------------------------------------------------------------------------
TypeError Traceback (most recent call last)
<ipython-input-3-8cf514cef5ce> in <module>()
----> 1 L[0,1]
TypeError: list indices must be integers or slices, not tuple

In [4]: L[0][1]
Out[4]: 2
```
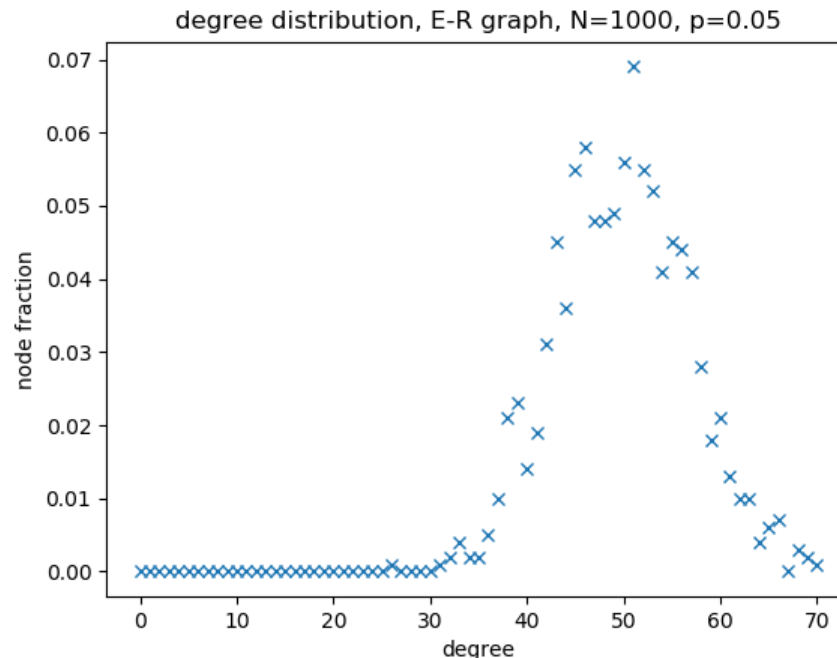
Today

- **Wrapping up networks**

- **(Classical) optimization with Numpy**

Getting started with NetworkX

- **Last time: looked at Erdos-Renyi $G_{N,P}$ graph:**

  - **$G_{N,P}$: a network with N nodes. The probability of a link being placed between any 2 nodes is P**

  - **Generated using** nx.gnp_random_graph

  - **Degree distribution follows a binomial distribution:**

degree distribution, E-R graph, N=1000, p=0.05

Getting started with NetworkX

- **Last time: looked at Erdos-Renyi $G_{N,P}$ graph:**

  - **Degree distribution follows a binomial distribution:**

degree distribution, E-R graph, N=1000, p=0.05
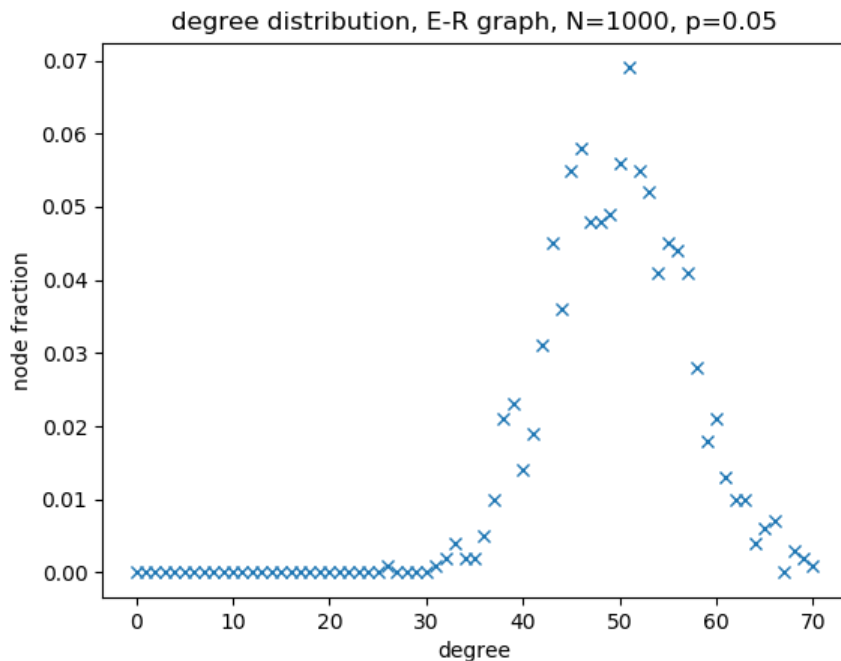
- Should compute degree distributions for several graphs (with fixed N,P) and average

- Generally, when there is randomness in the problem, statistics are the quantities of interest (mean, variance, etc…)

- For large degree, distribution decays away exponentially – most real complex networks have large-degree hubs

Getting started with NetworkX

- **Last time: looked at Erdos-Renyi $G_{N,P}$ graph:**

- **Two other important quantities are the *clustering coefficient* and *shortest path***

- **Clustering coefficient for node *i* with degree q_i:**
$$C_i = \text{\# of links between neighbors}/(q_i/2*(q_i-1))$$

In [**16**]: nx.clustering(G,500)
Out[**16**]: 0.044096728307254626

In [**17**]: nx.clustering(G,100)
Out[**17**]: 0.06464646464646465

In [**18**]: nx.clustering(G,0)
Out[**18**]: 0.04645760743321719

For $G_{N,P}$ graph, expect $C_i = P$

Getting started with NetworkX

- **Last time: looked at Erdos-Renyi $G_{N,P}$ graph:**

- **Two other important quantities are the *clustering coefficient* and *shortest path***

- **Shortest path: find route between two nodes traversing fewest number of links**

In [**20**]: nx.shortest_path(G,source=0,target=500)
Out[**20**]: [0, 233, 15, 500]

Getting started with NetworkX

- Last time: looked at Erdos-Renyi $G_{N,P}$ graph:

- Two other important quantities are the *clustering coefficient* and *shortest path*

- Shortest path: find route between two nodes traversing fewest number of links

```
In [20]: nx.shortest_path(G,source=0,target=500)
Out[20]: [0, 233, 15, 500]
```

→ **Very important in study** *algorithms,* **see** *BFS, Dijkstra's algorithm*

Notes: GNP graph is not a good model for large complex networks

- Degree distribution should include large-degree nodes, power-law decay for large q

- Clustering coefficient should be large and the average degree should be small

- Will consider a more-realistic model in this week's lab

Optimization with Python

We will focus almost entirely on the scipy.optimize **package**

- optimize **contains all standard methods for unconstrained optimization**

- **Choices for constrained optimization are more limited**
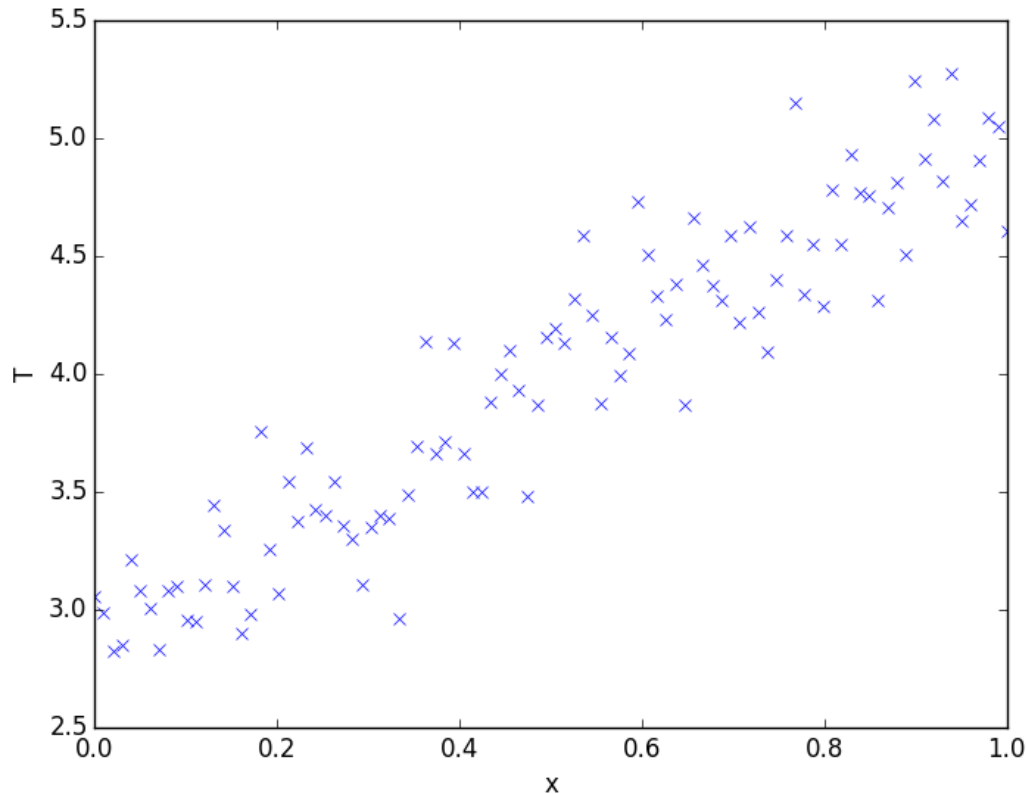 - **Other packages are available (*e.g.* pyopt)**

Getting started – finding trends in data

Consider the noisy data:
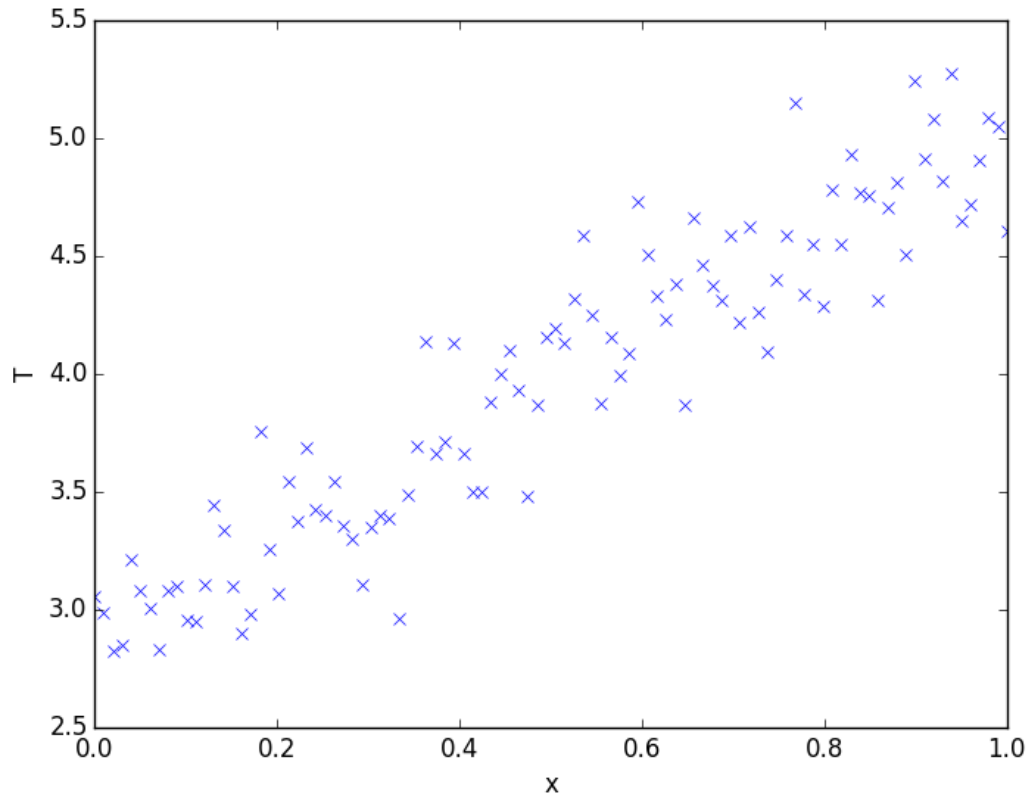
Getting started – finding trends in data

Consider the noisy data:

- **How can we construct a linear fit?**
  i.e. **find** c_1 **and** c_2 **such that**
  $T = c_1 x + c_2$

- **Python provides several options:**

  - scipy.stats
  - numpy.linalg.lstsq
  - scipy.optimize.curve_fit
  - numpy.polyfit
  - Pandas, statsmodel, **probably quite a few more!**

- **As a start, let's use** polyfit

Getting started – finding trends in data

x and T are simple arrays:

```
In [8]: x = np.linspace(0,1,100)

In [9]: T = 2*x + 3 + 0.25*np.random.randn(100)
```

Getting started – finding trends in data

x and T are simple arrays:

```
In [8]: x = np.linspace(0,1,100)

In [9]: T = 2*x + 3 + 0.25*np.random.randn(100)
```

And we use polyfit **to fit a 1st order polynomial (i.e. a line) to the data:**
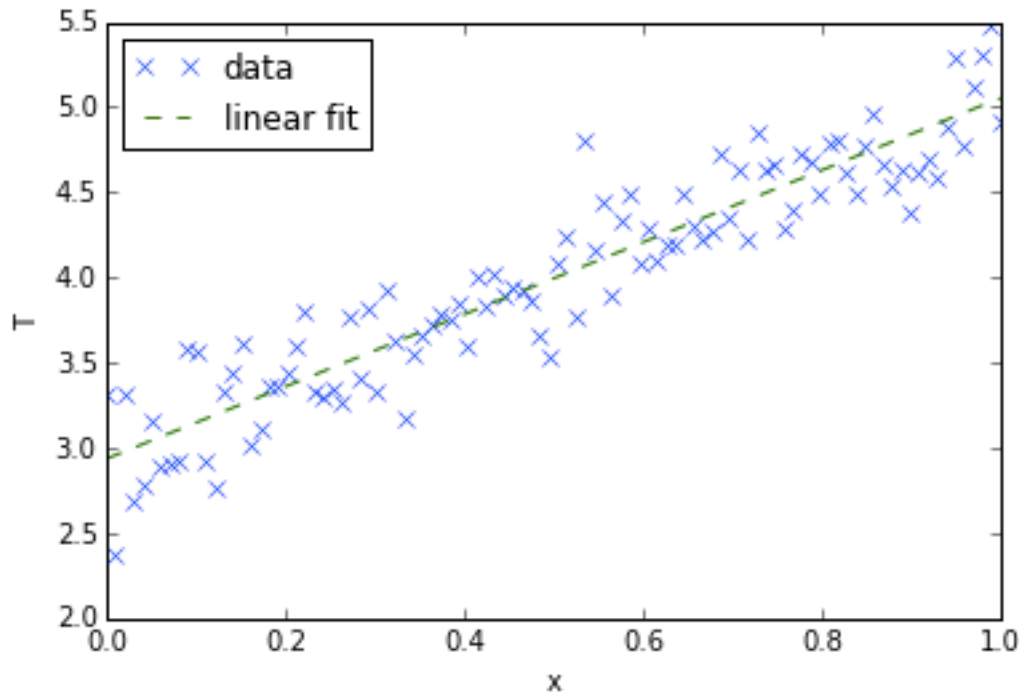
```
In [10]: C = np.polyfit(x,T,1)

In [11]: print "C1=",C[0],", C2=",C[1]
C1= 1.99587384666 , C2= 2.9703841544

In [13]: Tf = C[0]*x + C[1]
```

So the constants are close to what we expect, let's now display the fit:

Getting started – finding trends in data

- polyfit **constructs a linear least-squares fit to a nth-order polynomial:**

$$f(x) \approx c_1 x + c_2 x^2 + c_3 x^3 + ... + c_n x^n$$

so, it is only the fitting parameters that must be linear

- **The functional form for least-squares fits can be arbitrary (polynomial, trigonometric, etc…)**

General linear least squares

- **Let's say you have data T_1, T_2, …, T_n at the points x_1, x_2, … x_n and expect the data to fit a function of the form:**

$$T = c_a + c_b g_b(x) + c_c g_c(x) + \ldots$$

- **Here, g_b, g_c ,… can be any "well-behaved" functions. For** polyfit**, they have to be of the form x^n (with n an integer)**

- **As needed for *linear* least squares, the parameters c_a, c_b, … appear in a linear form**

- **Substituting each of the data points into the functional form (for the case with three parameters), we expect:**

$$
\begin{bmatrix}
1 & g_b(x_1) & g_c(x_2) \\
1 & g_b(x_2) & g_c(x_2) \\
\vdots & \vdots & \vdots \\
1 & g_b(x_n) & g_c(x_n)
\end{bmatrix}
\begin{bmatrix}
c_a \\
c_b \\
c_c
\end{bmatrix}
\approx
\begin{bmatrix}
T_1 \\
T_2 \\
\vdots \\
T_n
\end{bmatrix}
$$

- **And we can use linear least-square to find the "best" c_a, c_b, and c_c.**

General linear least squares

$$
\begin{bmatrix} 1 & g_b(x_1) & g_c(x_2) \\ 1 & g_b(x_2) & g_c(x_2) \\ \vdots & \vdots & \vdots \\ 1 & g_b(x_n) & g_c(x_n) \end{bmatrix} \begin{bmatrix} c_a \\ c_b \\ c_c \end{bmatrix} \approx \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_n \end{bmatrix}
$$

- np.linalg.lstsq **requires the** n x 3 **matrix on the left and the** n x 1 **column vector on the right as input**

General linear least squares

- **What does linear least-squares have to do with optimization?**

- **Rewrite matrix "equation" on previous slide as:** $M\mathbf{c} \approx \mathbf{t}$

- **Least-squares problem is:** $\text{Find } \mathbf{c} \text{ such that } \epsilon = (M\mathbf{c} - \mathbf{t})^2 \text{ is } minimized$

- **How do we minimize this?**

General linear least squares

- **What does linear least-squares have to do with optimization?**

- **Rewrite matrix "equation" on previous slide as:** $M\mathbf{c} \approx \mathbf{t}$

- **Least-squares problem is:** $\text{Find } \mathbf{c} \text{ such that } \epsilon = (M\mathbf{c} - \mathbf{t})^2 \text{ is } minimized$

- **How do we minimize this?**

- **Single-variable calculus: set derivative with respect to** c **to zero, and check that the second derivative at the minimum is positive**

- **Generalizing to n-dimensions:**

$$\begin{aligned} \epsilon &= (M\mathbf{c} - \mathbf{t})^T (M\mathbf{c} - \mathbf{t}) \\ &= \mathbf{c}^T M^T M \mathbf{c} - 2\mathbf{c}^T M^T \mathbf{t} + \mathbf{t}^T \mathbf{t} \\ \frac{\partial \epsilon}{\partial \mathbf{c}^T} &= 2M^T M \mathbf{c} - 2M^T \mathbf{t} \end{aligned}$$

General linear least squares

- **What does linear least-squares have to do with optimization?**

- **Rewrite matrix "equation" on previous slide as:** $M\mathbf{c} \approx \mathbf{t}$

- **Least-squares problem is:** $\text{Find } \mathbf{c} \text{ such that } \epsilon = (M\mathbf{c} - \mathbf{t})^2 \text{ is } minimized$

- **How do we minimize this?**

- **Single-variable calculus: set derivative with respect to** c **to zero, and check that the second derivative at the minimum is positive**

- **Generalizing to n-dimensions:**

$$\begin{aligned} \epsilon &= (M\mathbf{c} - \mathbf{t})^T (M\mathbf{c} - \mathbf{t}) \\ &= \mathbf{c}^T M^T M \mathbf{c} - 2\mathbf{c}^T M^T \mathbf{t} + \mathbf{t}^T \mathbf{t} \\ \frac{\partial \epsilon}{\partial \mathbf{c}^T} &= 2M^T M \mathbf{c} - 2M^T \mathbf{t} \end{aligned}$$

- **Now setting the derivative to zero,** c **must satisfy:** $\boxed{M^T M \mathbf{c} = M^T \mathbf{t}}$

- **All linear least-squares routines are solving this system of equations.**
    - **Routines for nonlinear least squares are also available**

Optimization: basic ideas

- **For linear least squares, we don't need to check the second derivative (due to the general "shape" of norms like ε).**

- **For more general functions it is (often) useful to look at the second derivative**

- **Consider "general" optimization problem:**

$$\text{Find } \mathbf{x} \text{ so that } f(\mathbf{x}) \text{ is minimized}$$

- **To start, consider behavior near minimizing point, x***

  - **In 1-D:** $f(x^* + h) = f(x^*) + \frac{df}{dx}|_{x^*} h + \frac{d^2 f}{dx^2}|_{x^*} \frac{h^2}{2} + O(h^3)$

  and since the derivative is zero at a minimum:

  $$f(x^* + h) = f(x^*) + \frac{d^2 f}{dx^2}|_{x^*} \frac{h^2}{2} + O(h^3)$$

  So, the second derivative must be positive at a minimum (otherwise f is smaller at x*+h than it is at x*)

Optimization: basic ideas

- **To start, consider behavior near minimizing point, x***

  - **In 1-D:** $f(x^* + h) = f(x^*) + \frac{d^2 f}{dx^2}\big|_{x^*} \frac{h^2}{2} + O(h^3)$

  - **Now in n-dimensions:**

    $$f(x_i^* + h_i) = f(x_i^*) + \sum_{j=1}^{n} \frac{\partial f}{\partial x_j}\big|_{x_i^*} h_j + \sum_{j=1}^{n} \sum_{k=1}^{n} \frac{\partial^2 f}{\partial x_j x_k}\big|_{x_i^*} \frac{h_j h_k}{2} + O(|h|^3)$$

    or in vector notation:

    $$f(\mathbf{x}^* + \mathbf{h}) = f(\mathbf{x}^*) + \mathbf{h}^T \nabla f\big|_{\mathbf{x}^*} + \frac{1}{2}\mathbf{h}^T H\big|_{\mathbf{x}^*}\mathbf{h} + O(|h|^3)$$

    Here, H is the _Hessian_, $H_{j,k} = \frac{\partial^2 f}{\partial x_j x_k}$ **, a n x n symmetric matrix.**

  - **At the minimum, the _gradient_, is zero, so we have:**

    $$f(\mathbf{x}^* + \mathbf{h}) = f(\mathbf{x}^*) + \frac{1}{2}\mathbf{h}^T H\big|_{\mathbf{x}^*}\mathbf{h} + O(|h|^3)$$

  - **The Hessian must be positive for any (small) h**

Optimization: basic ideas

- **All optimization routines require the user to specify the function to be minimized (usually called the *cost* function or the *objective* function)**

- **Most optimization routines use the gradient to search for a minimum (starting from an user-specified "guess")**

- **Some routines use the Hessian, some don't.**

Optimization: basic ideas

- **All optimization routines require the user to specify the function to be minimized (usually called the *cost* function or the *objective* function)**

- **Most optimization routines use the gradient to search for a minimum (starting from an user-specified "guess")**

- **Some routines use the Hessian, some don't**

- **Today, we'll look at two methods:**

    1. **(Truncated) Newton's method: uses gradient and Hessian**

    2. **BFGS method: uses gradient**

Newton's method

- **Classical method:**
    1. **Evaluate function, gradient and Hessian at a guess,** x_0
    2. **Use these values to fit a quadratic to the function:**

    $$g(\mathbf{h}) = f(\mathbf{x}_0) + \mathbf{h}^T \nabla f|_{\mathbf{x}_0} + \tfrac{1}{2}\mathbf{h}^T H|_{\mathbf{x}_0}\mathbf{h}$$

    3. **Setting** $\frac{\partial g}{\partial \mathbf{h}} = 0$ **:**

    $$H|_{\mathbf{x}_0}\mathbf{h} = -\nabla f|_{\mathbf{x}_0}$$

    4. **This can be solved for** h **and the new guess for the minimum is then:**
    $x_1 = x_0 + h$

- **Classical method only works well if guess is good**

- **In practice, a quadratic fit may be inaccurate or inappropriate**
    - **Truncated-Newton and BFGS methods both address this**

Truncated Newton's method

- **Classical method:**
    1. **Evaluate function, gradient and Hessian at a guess, x_0**
    2. **Use these values to fit a quadratic to the function:**

    $$g(\mathbf{h}) = f(\mathbf{x}_0) + \mathbf{h}^T \nabla f|_{\mathbf{x}_0} + \tfrac{1}{2}\mathbf{h}^T H|_{\mathbf{x}_0}\mathbf{h}$$

    3. **Setting $\frac{\partial g}{\partial \mathbf{h}} = 0$:**

    $$H|_{\mathbf{x}_0}\mathbf{h} = -\nabla f|_{\mathbf{x}_0}$$

    4. **This can be solved for** h **and the new guess for the minimum is then:**
    $x_1 = x_0 + h$

Truncated approach:
1. **Solve for** h **as in the classical approach**
2. **Then search in the direction of** h **for the minimum along this direction**
- **Step 2 is called a** *line search***, idea is similar to finding the zero of a 1-d function (e.g. Newton-Rhapson, secant, Brent methods)**

Optimizers in Scipy

- **We will use the** scipy.optimize **package**

- **And particularly, the function,** scipy.optimize.minimize

- **The optimization** *method* **can be specified when calling this function, e.g.:**

  method = 'Newton-CG'

- **This will use truncated Newton's method**

- **The CG indicates the conjugate gradient method is used to solve for** h

- **There is a separate CG method for optimization**

Optimizers in Scipy

- Newton-CG **requires the user to specify:**

  - **A python function which computes the cost function**

  - **A function for computing the gradient**

  - **A function for computing the Hessian (or instruct the optimizer to approximate it)**

  - **A guess,** $x0$**, where the optimizer starts its search for a minimum**

Simple example

- **Simple illustrative example:**

  Find x and y that minimize $f = -exp(-\alpha(x - x_0)^2 - \beta(y - y_0)^2)$
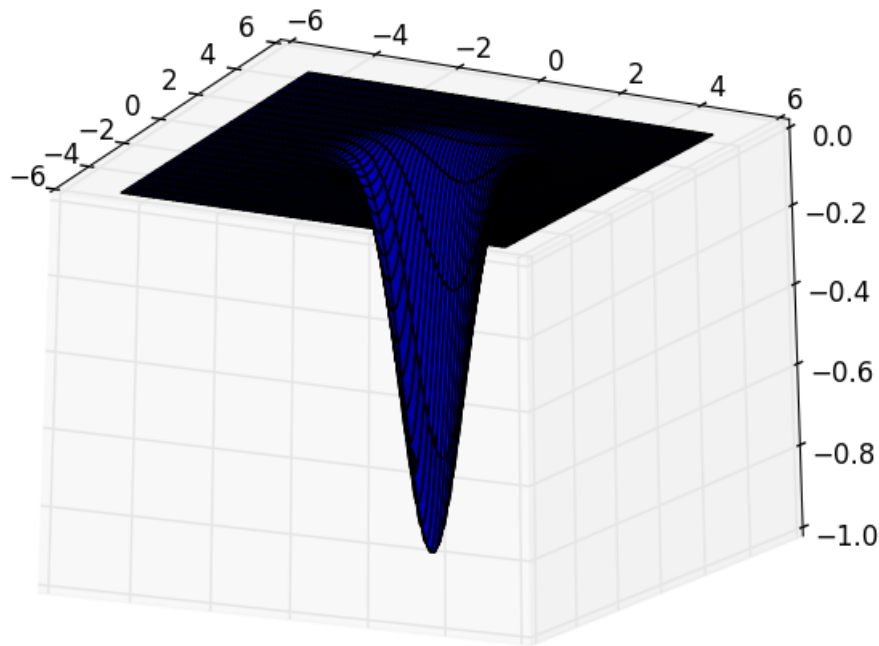
Simple example

The code, *gauss2d.py*, applies a few different approaches to this problem

- **It has three functions:** gauss2d, gauss2d_grad, gauss2d_hess
- gauss2d_grad **returns the two components of the gradient:**

$$\nabla f = (\tfrac{\partial f}{\partial x}, \tfrac{\partial f}{\partial y}) = (-2\alpha(x - x_0)f, -2\beta(y - y_0)f)$$

- gauss2d_hess **returns the 2 x 2 Hessian matrix:**

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y \partial y} \end{bmatrix} = \begin{bmatrix} -2\alpha(f + (x - x_0)\frac{\partial f}{\partial x}) & 4\alpha\beta(x - x_0)(y - y_0)f \\ 4\alpha\beta(x - x_0)(y - y_0)f & -2\beta(f + (y - y_0)\frac{\partial f}{\partial y}) \end{bmatrix}$$

- **The minimizer is called near the bottom with different inputs**

Simple example

- **The code,** *gauss2d.py***, applies a few different approaches**

- **Let's first look at the Newton-CG, approximate gradient case (ncg1=True)**

- **The call to the minimizer is:**

  minimize(gauss2d,xguess,args=parameters,method='Newton-CG',jac=gauss2d_grad)

  Note: since the name of the Hessian function is not specified, it is approximated from the gradient

Simple example

- **Running the code:**

```
In [44]: run gauss2d
Gauss2d, Newton–CG, approximate Hessian
optimum: [ 0.5   0.25]
info:     status: 0
 success: True
    njev: 17
    nfev: 6
     fun: −1.0
       x: array([ 0.5 ,  0.25])
 message: 'Optimization terminated successfully.'
    nhev: 0
     jac: array([ −5.37315725e−09,  −2.68657863e−09])
```

Notes:
- gauss2d **sets** (x0,y0) **to** (0.5,0.25) **so the answer is correct**
- **17 gradient evaluations and 6 function evaluations were needed**
- **What happens if we now specify the exact Hessian (ncg2=True)?**

Simple example

Running the code with:

minimize(gauss2d,xguess,args=parameters,method='NewtonCG',
                                jac=gauss2d_grad,hess=gauss2d_hess)

```
Gauss2d, Newton-CG, exact Hessian
optimum: [ 0.5    0.25]
info:      status: 0
 success: True
    njev: 9
    nfev: 6
     fun: -1.0
       x: array([ 0.5 ,  0.25])
 message: 'Optimization terminated successfully.'
    nhev: 4
     jac: array([ -2.28865316e-09,  -1.14432658e-09])
```

Notes:
- **The number of gradient evaluations reduced from 17 to 9**
- gauss2d **also has options for BFGS and Nelder-Mead methods – how do these work?**

Overview of BFGS

- BFGS is generally the method-of-choice for unconstrained optimization problems involving smooth functions

- Works surprisingly well for non-smooth/noisy functions as well

- Can be memory-intensive, use L-BFGS-B for large problems

- Motivating idea:
  - Computing and inverting the Hessian can be very expensive
  - Instead, approximate the (inverse) Hessian, and update this approximation at each step (quasi-Newtonian methods)

Overview of BFGS

- **Motivating idea:**
  - **Computing and inverting the Hessian can be very expensive**
  - **Instead, approximate the (inverse) Hessian, and update this approximation at each step (quasi-Newtonian methods)**
- **Newton-CG:**
1. **Solve for** h: $H|_{\mathbf{x}_0}\mathbf{h} = -\nabla f|_{\mathbf{x}_0}$
2. **Search along direction of** h **for minimum** → x$_1$ **and repeat step 1**

- **BFGS:**
1. **Approximate inverse Hessian and solve for** h: $M^i \approx H^{-1}$, $\mathbf{h} = -M^i\nabla f|_{\mathbf{x}_i}$
2. **Similar to Newton-CG**
3. **Update** M: $M^{i+1} = f(M^i, h)$

- **The update should result in a positive-definite (approximate) Hessian, and the BFGS update formula produces particularly good results**

- **Now back to** *gauss2d*

Simple example -- BFGS

Results for BFGS with approximate gradient

minimize(gauss2d,xguess,args=parameters,method='BFGS',jac=False)

```
Gauss2d, BFGS, approximate gradient
optimum: [ 0.49999556  0.24999778]
info:      status: 0
  success: True
     njev: 5
     nfev: 20
 hess_inv: array([[ 0.60077592, -0.19961208],
       [-0.19961208,  0.90019394]])
      fun: -0.9999999999753861
        x: array([ 0.49999556,  0.24999778])
  message: 'Optimization terminated successfully.'
      jac: array([ -8.85874033e-06,  -4.43309546e-06])
```

Simple example -- BFGS

Results for BFGS with exact gradient

minimize(gauss2d,xguess,args=parameters,method='BFGS',jac=gauss2d_grad)

```
Gauss2d, BFGS, exact gradient
optimum: [ 0.49999557  0.24999779]
info:      status: 0
  success: True
     njev: 5
     nfev: 5
 hess_inv: array([[ 0.60077589, -0.19961205],
       [-0.19961205,  0.90019397]])
      fun: -0.9999999999754784
        x: array([ 0.49999557,  0.24999779])
  message: 'Optimization terminated successfully.'
      jac: array([ -8.85828506e-06,  -4.42914253e-06])
```

Notes:
1. **BFGS requires less function, gradient evaluations than Newton-CG**
2. **For large (slow) problems, specifying the gradient (and Hessian if appropriate) can make a big difference**