» Gradient Descent

* Recall general iterative minimisation algorithm:

```
x=x0
for k in range(num_iters):
    step = calcStep(fn,x)
    x = x − step
```

* We know one way to choose the step, namely:

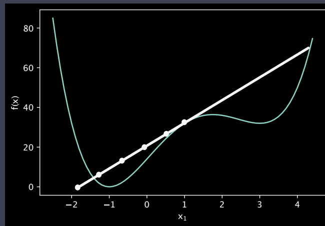
$$step = \alpha[\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ldots, \frac{\partial f}{\partial x_n}(x)] = \alpha \nabla f(x)$$

where α is the *step size* or *learning rate*

* How to choose automate choice of α?

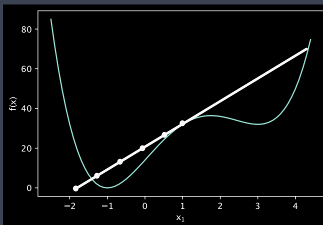
» Line search

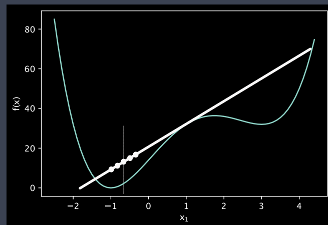
* Update x to $x - \alpha \nabla f(x)$
* *Exact line search* $\rightarrow$ select α that minimises $f(x - \alpha \nabla f(x))$ i.e. $\alpha \in \arg\min_{\alpha'} f(x - \alpha' \nabla f(x))$
* In practice an α that makes $f(x - \alpha \nabla f(x))$ decrease by a reasonable amount is good enough.
* How to carry out this optimisation? Remember α is a scalar.
  * *Grid search.* Calculate $f(x - \alpha \nabla f(x))$ over a grid of values for α and pick smallest $\rightarrow$ simple, robust, but how to choose grid when we don't know the approx magnitude of α to use?



» Line search

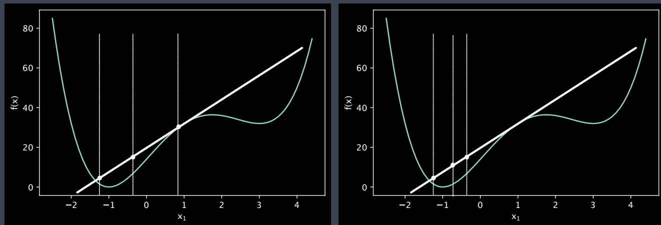
* *Iterative grid search*. Select initial grid, e.g. [0.001,0.01,0.1,1,10]. Suppose $\alpha = 0.01$ gives $f(x - \alpha \nabla f(x))$, then create a new grid around this point e.g. $[0.005, 0.0075, 0.01, 0.02, 0.05, 0.075]$ and repeat. Again, simple and robust.





» Line search

* *Bracketing methods* e.g. Golden-section search[1], Brents method[2]



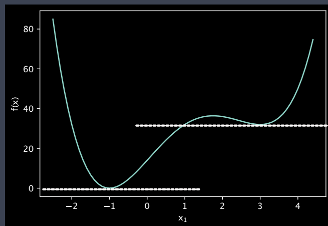
* Hopefully fewer function evaluations than grid search (so faster/cheaper)
* When there are multiple minima, may converge to local minimum.

---

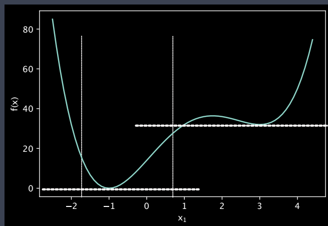
[1]https://en.wikipedia.org/wiki/Golden-section_search
[2]https://en.wikipedia.org/wiki/Brent's_method

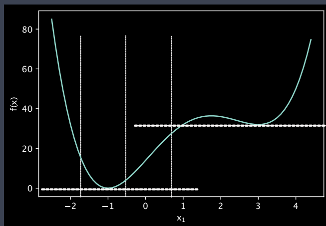
» Line search

* *Derivative-based methods* $\rightarrow$ search for point where derivative of $\frac{df}{dx}(x - \alpha\nabla f(x)) = 0$.



* Derivative changes sign on either side of minimum $\rightarrow$ use that to bracket the solution





» Line search

* *Backtracking search*[3]. Idea: try an initial large step α and then reduce ("backtrack").
* Recall $f(x+\delta) \approx f(x) + \nabla f(x)^T \delta$ with $\nabla f(x)^T \delta = \sum_{i=1}^n \frac{\partial f}{\partial x_i}(x)\delta_i$. Choosing $\delta = -\alpha \nabla f(x)$ then

$$f(x - \alpha \nabla f(x)) \approx f(x) - \alpha \nabla f(x)^T \nabla f(x)$$

with $\nabla f(x)^T \nabla f(x) = \sum_{i=1}^n \frac{\partial f}{\partial x_i}(x)^2$
* Idea: try to select α such that

$$f(x - \alpha \nabla f(x)) \leq f(x) - c\alpha \nabla f(x)^T \nabla f(x)$$

with parameter $0 \leq c \leq 1$.

[3] https://en.wikipedia.org/wiki/Backtracking_line_search

» Line search

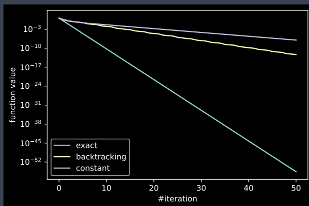
- Python implementation of backtracking linesearch:

```
import numpy as np
alpha=1;
beta=0.5; c = 0.5 # design parameters
df=fn.df(x)
while fn.f(x-alpha*df) > fn.f(x)-c*alpha*np.dot(df,df):
    alpha=beta*alpha
```

- Start with $\alpha = 1$, if $f(x - \alpha \nabla f(x))$ too large then decrease α to $\beta\alpha$ (with $\beta < 1$ obviously), repeat.
- E.g. when $\beta = 0.5$ then try sequence of α's $1, 0.5, 0.5^2, 0.5^3, 0.5^4, \ldots$ $=1, 0.5, 0.25, 0.125, 0.0625, \ldots$
- When $\beta = 0.8$ then try sequence of α's $1, 0.8, 0.8^2, 0.8^3, 8^4, \ldots$ $=1, 0.8, 0.64, 0.512, 0.4096, \ldots$

» Example: Quadratic

* $f(x) = 0.5(x_1^2 + 10x_2^2)$, $x = [x_1, x_2]$.
* Starting value $x0 = [1.5, 1.5]$, step size $alpha = 0.15$



* Increasing α to 0.2 for constant step sizes causes output to diverge.
* So exact line search gives convergence in fewer iterations, but is that actually *faster*?

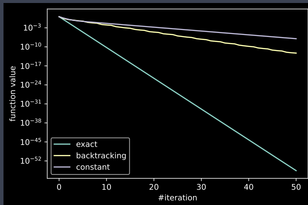
» Example: Quadratic

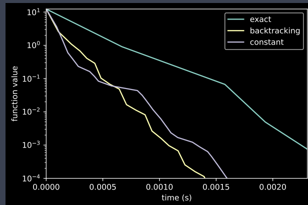
* Time for each iteration is larger with exact line search since need to perform an optimisation to find α. Similarly for backtracking.

* When considering performance, accuracy vs #iterations is not enough, also need to take account of time taken for each iteration $\rightarrow$ its really *accuracy vs wall-clock time* that we're usually interested in.

» Example: Quadratic

* Time for each iteration is larger with exact line search since need to perform an optimisation to find α. Similarly for backtracking.

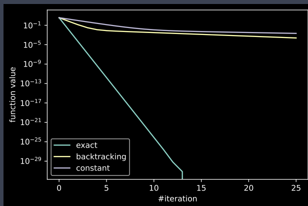


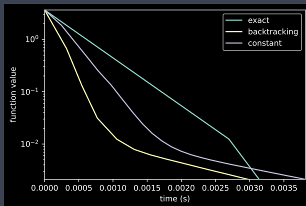


* In this example, constant step size converges faster vs wall-clock time than eact line search, and about the same as backtracking. But constant step size was hand-tuned ...

» Example: Linear Regression Quadratic Loss

- $y^{(i)} = \theta^T x^{(i)}$, $\theta = [-0.5, 0.2]$, $m = 1000$ training data points with random $x^{(i)}$, $i = 1, \ldots, n$
- Cost function $J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)})^2$
- Starting value $x0 = [1, 1]$, step size $alpha = 0.5$

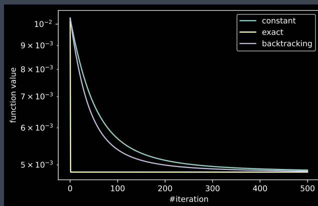


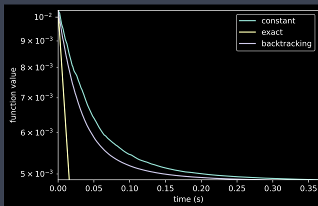


- Again, exact linesearch slower wrt wall-clock time.
- But remember quadratic functions/linear regression are the “easy” cases ...

» Example: Toy Neural Net Quadratic Loss

* $z = f(\theta_1 x + \theta 1), \hat{y} = g(z)$, $\theta = [1, 5]$, f ReLu, g sigmoid, $m = 1000$ training data points with random $x^{(i)}$, quadratic loss
* Starting value $x0 = [1, 1]$, step size $alpha = 0.75$

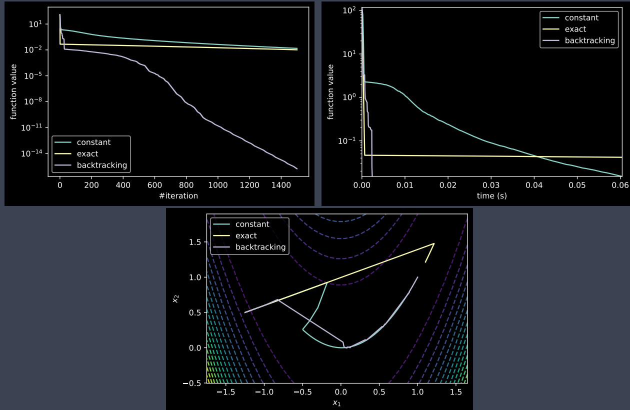




* In this “harder” problem (non-smooth, flat surfaces) exact line search is faster even though it involves more computation per iteration

» Example: Rosenbrock Function

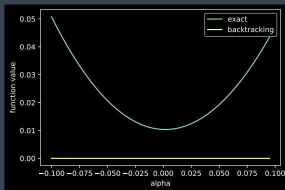
* Another "hard" problem, although in a different way (narrow curved valley with flat valley floor).
* Starting value $x0 = [-1.25, 0.5]$, step size *alpha* $= 0.002$



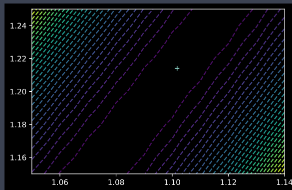
* In this "harder" problem backtracking line search is faster
* Exact linesearch gets "stuck", constant step size slow

» Example: Rosenbrock Function

- Exact linesearch gets "stuck". Plot of function value $f(x - \alpha\nabla f(x))$ vs α at terminal point of linesearch:



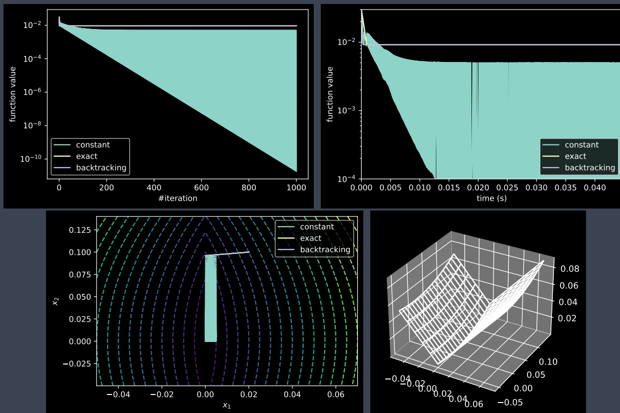
- Plot of function around point (marked by '+') where exact linesearch terminates:



- Valley floor is v flat here, so direction of steepest descent is across valley and doesn't allow function to be decreased $\rightarrow$ try another direction, or perturb ourselves away from '+' point (e.g. constant step would do that)

» Example: Non-Smooth Function

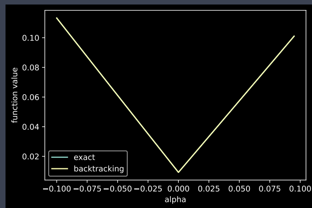
* Another "hard" problem (non-smooth), $f(x) = |x_1| + x_2^2$
* Starting value $x0 = [0.02, 0.1]$, step size *alpha* $= 0.005$



* Linesearch methods get "stuck" away from minimum, constant step size oscillates around min (size of oscillation depends on step size) but beats linesearch methods

» Example: Non-Smooth Function

* Linesearch methods get "stuck" away from minimum. Plot of function value $f(x - \alpha \nabla f(x))$ vs α at terminal point of linesearch:



* The kink in the function has trapped the linesearch methods at a poor point $\rightarrow$ a bit like in Rosenbrock example.

» Summary

* Performance of line search depends on the function being minimised $\rightarrow$ no general rules (we'll see this is true for most methods for choosing the step size unfortunately)
* Need to take account of increased computation with linesearch $\rightarrow$ convergence in fewer iterations might not mean faster converges wrt wall-clock time (which is what usually matters)
* For quadratic-like functions a constant step size is already v good so limited gain from using line search
* For "harder" functions line search can speed up convergence (e.g. toy neural net example), but can also lead to getting stuck at a sub-optimal point (e.g. rosenbrock and non-smooth example).