4. Optimisation

LECTURE 4

2024/25
SOPHIE.STEVENS@BRISTOL.AC.UK

Workshop 3 summary:

4. **First order inhomogeneous equations.**

   (a) $f'(t) + 5f(t) = 1$ with initial condition $f(0) = 2$.

   (b) $f'(t) = t - f(t)$ with initial condition $f(1) = 3e^{-1}$.

   (c) $f'(t) + 2f(t) = \sin(t)$ with initial condition $f(0) = 9/5$.

   (d) $f'(t) - 2f(t) + t^2 = 0$ with initial condition $f(2) = 13/4 + 6e^4$.

Check your work!

Form of $f(x)$	Form of $P.I.$
p	λ
$p + qx$	$\lambda + \mu x$
$p + qx + rx^2$	$\lambda + \mu x + \varphi x^2$
pe^{kx}	λe^{kx}
$p\cos(\omega x) + q\sin(\omega x)$	$\lambda\cos(\omega x) + \mu\sin(\omega x)$

https://pmt.physicsandmathstutor.com/download/Maths/A-level/Further/Core-Pure/Edexcel/CP2/Cheat-Sheets/Ch.7%20Methods%20in%20Differential%20Equations.pdf

What we'll cover today:

- Optimising functions via calculus
- Gradient descent method
- Simplex method

We can optimise with calculus

Example [board]: $\mathrm{E}(x, y) = (1 - x)^2 + 100(y - x^2)^2$

Aside: how to calculate the trace

$$Tr\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} = a_{11} + a_{22} + \cdots + a_{nn}$$

Gradient descent

- Algorithm that aims to find the minimum of a function $E(x_1, \dots, x_m) = E(\boldsymbol{x})$
- To use the algorithm, we need to be able to calculate the gradient operator $\nabla E(\boldsymbol{x})$
- Iterative algorithm: at each step, walk in the `down' direction
- Parameter: $\eta > 0$ determines the size of the step
- $\boldsymbol{x_{n+1}} = \boldsymbol{x_n} - \eta \nabla E(\boldsymbol{x_n})$
- $\boldsymbol{x_n} = (x_{n,1}, \dots, x_{n,m})$ is the input for the nth step

Gradient descent example

$$\mathrm{E}(x,y) = (1-x)^2 + 100(y-x^2)^2$$

- $\nabla E(x,y) = (-2(1-x) - 400x(y-x^2), 200(y-x^2))$
- Pick $\eta = 0.000001$
- Pick a random starting point: $\mathbf{x_0} = (2,3)$
- Run algorithm:
  - $\boldsymbol{x_1} = \boldsymbol{x_0} - \eta\nabla(\mathbf{x_0})$
  - $\boldsymbol{x_2} = \boldsymbol{x_1} - \eta\nabla(\mathbf{x_1})$
  - …

Python code (for documentation)

```
def E(x,y):
    return (1-x)**2 + 100*(y-x**2)**2

def grad(x,y):
    return [-2*(1-x) -400*x*(y-x**2), 200*(y-x**2)]

vec = [2,3]

eta = 0.000001

iters = 0
limit = 4

while iters < limit:
    [x,y] = [vec[0],vec[1]]
    newgrad = grad(x,y)
    vec = [x - eta*newgrad[0], y - eta*newgrad[1]]
    iters += 1
    try:
        print("Iteration number: ", iters, "\n--- Vector: ", vec,  "\n--- E(x,y):", E(vec[0],
vec[1]))
    except:
        print("iteration", iters, "failed")
```

Gradient descent: observations

- η determines the size of the step. If it's too big, our optimisation is too coarse and we might miss the minimum; if it's too small, we might waste resources
- We can get stuck in local minima
- We need a stopping condition
- We need to determine what the output is

Gradient descent: stopping conditions

- After a set number of iterations
- When the difference in the output between steps is very small
- When the gradient is very small (i.e. approaching a critical point)

Gradient descent: output

Let $\boldsymbol{x_n}$ be the final value of the algorithm.

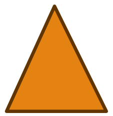
- We could output $\boldsymbol{E}(\boldsymbol{x_n})$
- We could output the value at an average of the values that we have seen:
  - $\overline{\boldsymbol{x}} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x_n}$
  - Output $\boldsymbol{E}(\overline{\boldsymbol{x}})$

Gradient descent: variations

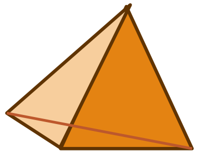
- We could take different step sizes depending on what we discover
- We probably want a way to automate out of loops/bad areas
- More powerful is *stochastic gradient descent*
- Not possible if we can't easily calculate the gradient…
- … although still possible if the gradient doesn't exist

Simplex

- 0-dimensional:
- 1-dimensional:
- 2-dimensional:



- 3-dimensional:



- n-dimensional:
  - convex hull of n+1 linearly independent vertices
  - i.e. pick n+1 linearly independent vertices (e.g. not all on a line/plane). Find the smallest shape that you can draw with straight lines that contains all points

Downhill simplex algorithm/Nelder-Mead

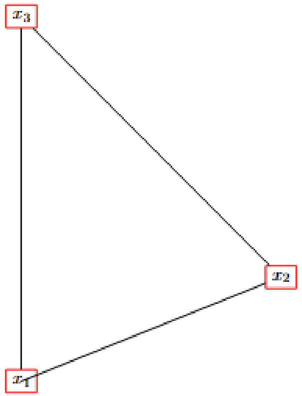
We want to minimise $E(x_1, x_2, \dots, x_n)$

1. Randomly pick n+1 (linearly independent) vectors. They look like $\boldsymbol{v} = (v_1, \dots, v_n)$

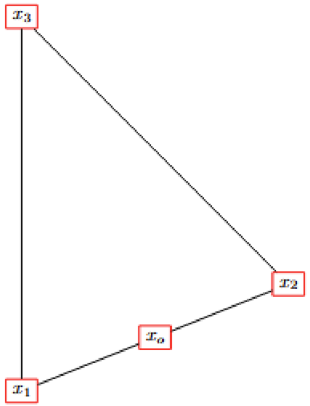
2. Order the vectors according so that $E(\boldsymbol{v}_{(1)}) \leq E(\boldsymbol{v}_{(2)}) \leq \cdots \leq E(\boldsymbol{v}_{(n+1)})$
   Goal: replace $\boldsymbol{v}_{(\boldsymbol{n}+1)}$
   - **Centroid**: $\boldsymbol{v}_{(0)} = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{v}_{(\boldsymbol{i})}$
   - **Reflection point**: $\boldsymbol{v}_{\boldsymbol{r}} = \boldsymbol{v}_{(0)} + (\boldsymbol{v}_{(0)} - \boldsymbol{v}_{(\boldsymbol{n}+1)})$
     reflect $\boldsymbol{x}_{(\boldsymbol{n}+1)}$ *in n-dimensional simplex formed by first n vertices*

3. If $E(\boldsymbol{v}_{(1)}) \leq E(\boldsymbol{v}_{\boldsymbol{r}}) \leq E(\boldsymbol{v}_{(\boldsymbol{n})})$ then replace $\boldsymbol{v}_{(\boldsymbol{n}+1)} \leftarrow \boldsymbol{v}_{\boldsymbol{r}}$

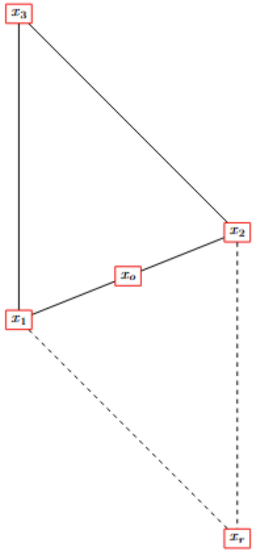
Visually



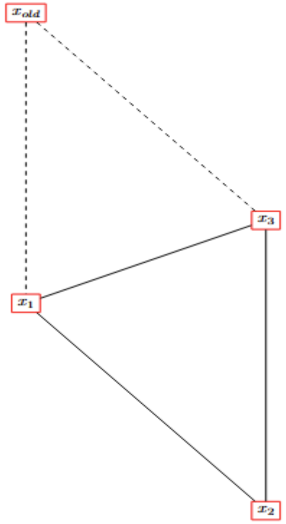
1. Order points



2. Find centroid. Here it’s the midpoint.



3. Reflection



4. Replace old vertex and reorder points

What if we don't have $E(\boldsymbol{v}_{(1)}) \leq E(\boldsymbol{v}_{\boldsymbol{r}}) \leq E(\boldsymbol{v}_{(\boldsymbol{n})})$?

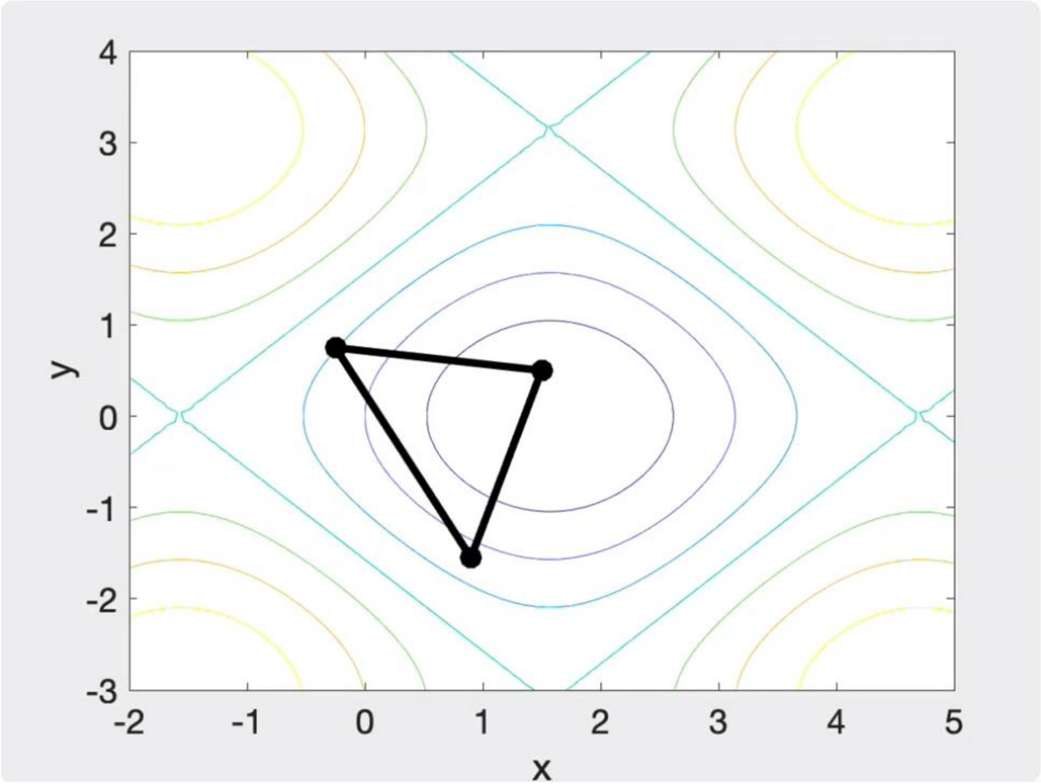
$E(\boldsymbol{v}_r) \leq E(\boldsymbol{v}_{(1)})$

- $\boldsymbol{v}_r$ is actually really good!
- Can we make an even better vertex?
- Expansion point: $\boldsymbol{v}_e = \boldsymbol{v}_{(0)} + 2(\boldsymbol{v}_{(0)} - \boldsymbol{v}_{(n+1)})$
- If $E(\boldsymbol{v}_e) \leq E(\boldsymbol{v}_r)$ then $\boldsymbol{v}_{(n+1)} \leftarrow \boldsymbol{v}_e$
- Else $\boldsymbol{v}_{(n+1)} \leftarrow \boldsymbol{v}_r$

$E(\boldsymbol{v}_r) > E(\boldsymbol{v}_{(n)})$

- If we swapped in $\boldsymbol{v}_r$, it would become our worst point
- Contraction point: $\boldsymbol{v}_c = \boldsymbol{v}_{(0)} - \frac{\boldsymbol{v}_{(0)} - \boldsymbol{v}_{(n+1)}}{2}$
- If $E(\boldsymbol{v}_c) < E(\boldsymbol{v}_{(n)})$ then $\boldsymbol{v}_{(n+1)} \leftarrow \boldsymbol{v}_c$
- Else [emergency move]: $\boldsymbol{x}_{(i)} = \frac{\boldsymbol{x}_{(i)} + \boldsymbol{x}_{(1)}}{2}$ for $i \neq 1$

Video



https://www.youtube.com/watch?v=XfdHAcHat7M

Nelder-Mead: when to stop?

- When the evaluation of the function at the simplex vertices is `close’
- After a maximum number of iterations

Summary

- Sometimes calculus is helpful to optimise a function
- Gradient descent is useful if the derivative is calculable
- Simplex method is always possible, especially if we can’t find the derivative