Computational Projects

Lecture 2: Solution of transcendental equations

Dr Rob Jack, DAMTP

Note: this lecture covers material useful for the introductory project

http://www.maths.cam.ac.uk/undergrad/catam/part-ia-lectures

Basic idea

Given a continuous function $f : \mathbb{R} \to \mathbb{R}$, we want to solve

$$f(x) = 0$$

(the relevant cases are those without any closed form solution, eg $f(x) = \mathrm{e}^x - 4x$, etc. . .)

Iterative approach: We are going to compute a sequence $x_0, x_1, x_2, \ldots$ such that as $n \to \infty$,

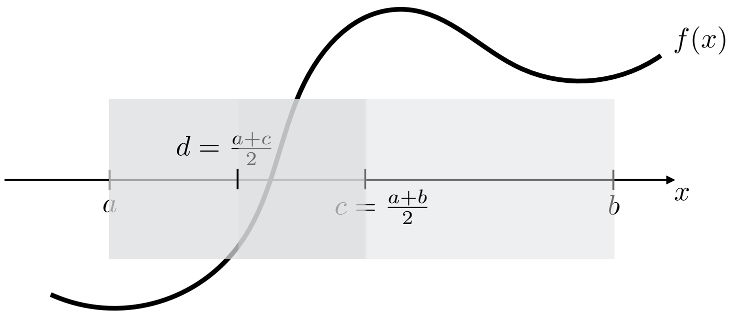
$$x_n \to x_*, \qquad \text{with} \qquad f(x_*) = 0$$

As the algorithm proceeds, we accumulate information, which can be used in computation of the rest of the sequence.

Eg, in some simple methods $x_n = g(x_{n-1}, x_{n-2})$ for some function g (which is sometimes called the *iteration rule*).

Bisection method -- key idea

(also known as interval halving or binary search)



$f(x)$ changes sign between a and b, and $f(x)$ is continuous, hence there is a root between a and b (intermediate value thm.)

$f(x)$ changes sign between a and c, there is a root between a and c

Compute $d = \frac{a+c}{2}$ and repeat. . .

Bisection method -- algorithm

Given: a function f and two numbers a, b such that $f(a)f(b) < 0$ and $a < b$.

Let $a_0 = a$ and $b_0 = b$. Let $k = 0$.

Iterate the following loop for $k = 0, 1, 2, \ldots$

> There is surely a root in $[a_k, b_k]$
>
> Compute $c_k = \frac{a_k + b_k}{2}$ and $f(c_k)$.
>
> If $f(b_k)f(c_k) > 0$ then let $(a_{k+1}, b_{k+1}) = (a_k, c_k)$,
> otherwise let $(a_{k+1}, b_{k+1}) = (c_k, b_k)$

After n iterations, we know that there is a root in $[a_n, b_n]$ which is an interval of size $2^{-n}(b - a)$

The sequence $c_0, c_1, c_2, \ldots$ converges to a root of f

Bisection method

There is a root x^* such that

$$|c_n - x^*| \le 2^{-(n+1)}(b-a)$$

Efficiency / complexity: to be sure that $|c_n - x_*| < \zeta$, we insist that $|c_n - x_*| \le 2^{-(n+1)}(b-a) < \zeta$, which requires

$$n > \frac{1}{\ln 2} \ln\left(\frac{b-a}{\zeta}\right) - 1$$

Loosely speaking, "complexity" is $O(\ln(1/\zeta))$.
. . . see also rate/order of convergence (later)

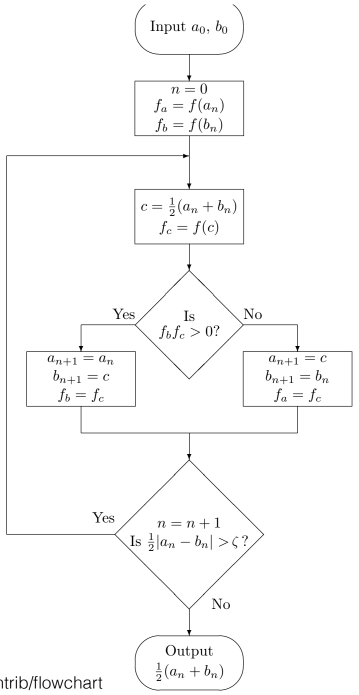
Notes: (i) we need a suitable initial pair (a_0, b_0); (ii) we always find one root but we don't know about other possible roots

Flowcharts

Before writing your program...
... one way to check that an algorithm makes sense is to construct a flow chart

You can see the "loops", and you can check the possible sequences of operations that the algorithm will require

It's often a good idea to check that the system will not get stuck in an infinite loop...



Wikipedia's page on flowcharts
http://en.wikipedia.org/wiki/Flowchart
Package for creating flowcharts in LaTeX
http://www.ctan.org/tex-archive/graphics/pgf/contrib/flowchart

Code and pseudocode

Pseudocode is a way to sketch out programs without worrying about the details of : ; ~=, etc

Pseudocode for bisection

```
Fix some ζ and a suitable a,b

loop over n, until 0.5*|b - a| < ζ :
    set c = 0.5*(a + b)
    if f(b)*f(c) > 0
        set b = c
    else
        set a = c
    end if
end loop

estimate root as 0.5*(a+b)
```

MATLAB code

```
zeta = 1e-7;
a = 0.0; b = 1.0;

while abs(b - a)/2 > zeta
   c = (a+b)/2;
   if f(b)*f(c) > 0
        b = c;
   else
        a = c;
   end
end

estRoot = (a+b)/2
```


MATLAB implementation

```
%  set f to be a (mathematical) function
% (not the same as a MATLAB function...)
f = @(x) exp(x)-4*x;
% plot the function
fplot( f, [0,1] )

% now we aim to solve exp(x)-4x == 0
% to 6 decimal places
zeta = 1e-7;
a = 0.0; b = 1;

while abs(b - a)/2 > zeta
   c = (a+b)/2;
   if f(b)*f(c) > 0
        b = c;
   else
        a = c;
   end
end

estRoot = (a+b)/2

% check that f(estRoot) is indeed small
display( f(estRoot) )
```

Example: `root_simple.m`

MATLAB function

```
function [ root ] = binarySearch( func, xlow, xhigh, tol )
%binarySearch method to find root of a function (called func)
  % the output is root, initial guesses xlow and xhigh
  % the tolerance (tol) is such that there is a root between
  % xroot(1+tol) and xroot(1-tol), this is "relative error"
  % (see lecture 3)

  % Use this to solve exp(x) - 4x == 0 by running
  % binarySearch( @(x) exp(x)-4*x, 0,1, 1e-7)

  a=xlow;
  b=xhigh;

  while abs(b - a)/2 > tol*abs(a+b)/2
    c = (a+b)/2;

    if func(b)*func(c) > 0
        b = c;
   else
        a = c;
    end
  end % of the "while loop"

  root = (a+b)/2;
end % of the function
```

Example: `binarySearch.m`

Bisection method

Good points
Always finds a root (for any continuous function)
Even for finite n, we know that there is definitely a root in $[a_n, b_n]$.

Non-good points
Requires a suitable initial interval
...can't find double roots, eg no suitable interval if $f(x) = (x-1)^2$
Other methods may converge faster

General caveat about root finding

We want to solve $f(x) = 0$.

...but even if $|x_n - x_*| < \zeta$, we might still have $|f(x_n)|$ quite large (especially if $f'(x_*)$ is large, or does not exist...)

A note on efficiency

You can see that `binarySearch` evaluates both $f(c)$ and $f(b)$ in each iteration

At step n, the value of $f(b_n)$ has already been calculated (in a previous step)

If we keep track of this, we can reduce the computational effort.

If evaluating the function f is expensive then this can reduce the time to find the root by up to a factor of 2

Replace the while loop in `binarySearch` by:

```
fb = func(b);
while abs(b - a)/2 > tol*abs(a+b)/2
  c = (a+b)/2;
  fc = func(c);

  if fb*fc > 0
      b = c;
      fb = fc;
  else
      a = c;  % (fb stays the same)
  end
end % of the "while loop"
```

Example: `binarySearchV2.m , binaryTest.m`

Order of convergence

We want to characterise the efficiency of our algorithms.
Define

$$\delta_n = x_n - x_*$$

We say that the *order of convergence* is p if we can find constants $p \geq 1$ and c such that

$$\lim_{n\to\infty} \frac{|\delta_{n+1}|}{|\delta_n|^p} = c$$

(if $p = 1$ then we require $c < 1$)

The *asymptotic error constant* is c

Algorithms with larger p converge faster, as long as c is not too large/small.

Order of convergence

An alternative definition is that the order of convergence is p if there is a sequence $y_1, y_2, \ldots$ such that $|\delta_n| < y_n$ for all n and

$$\lim_{n\to\infty} \frac{|y_{n+1}|}{|y_n|^p} \le c$$

Using this definition, it is easy to analyse the bisection method: we have $y_n = 2^{-n-1}(b_0 - a_0)$ so that $p = 1$ and $c = 1/2$.

The case $p = 1$ is called *linear convergence*, while $p = 2$ is quadratic convergence, etc

Order of convergence -- efficiency

Suppose we require $|\delta_n| < \zeta$. How many iterations are needed?

Assume that $|\delta_{n+1}| \le c|\delta_n|^p$ for all n.

(This is a bit stronger than just having order of convergence p.)

For $p = 1$ we must have $c < 1$; then $|\delta_n| \le c^n|\delta_0|$.

As before (for bisection) insist that $|\delta_n| \le c^n|\delta_0| < \zeta$

This requires

$$n > \frac{\log(|\delta_0|/\zeta)}{\log(1/c)}$$

. . . can think of this as $O(\log(1/\zeta))$ but one would usually just quote the order of convergence (linear in this case).

Order of convergence

For $p > 1$ we have:

$$|\delta_n| < c^{\frac{p^n-1}{p-1}} |\delta_0|^{p^n}$$

Assuming $n \gg 1$, we get $|\delta_n| < (1/\zeta)$ if

$$n \gtrsim \frac{1}{\log p} \log \left[\frac{\log(1/\zeta)}{\log(1/|\delta_0|) + (p-1)^{-1}\log(1/c)} \right]$$

The number of iterations grows as $\log\log(1/\zeta)$ – few iterations are needed even for very small ζ

Again the order of convergence characterises the efficiency of the algorithm, this is better than writing $O(\log\log(1/\zeta))$

Secant method

An alternative method for root-finding:

Given two points x_0, x_1 (not necessarily with $f(x_0)f(x_1) < 0$):

Iterate $n = 1, 2, \ldots$ and compute

$$x_{n+1} = x_n - \left[\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \right] f(x_n)$$

Unlike bisection, the resulting sequence is *not guaranteed* to converge to a root of f

However, for “nice enough” functions f, it does converge to a root. In this case, the order of convergence is (usually) $p = (1 + \sqrt{5})/2 \approx 1.6$

Secant vs bisection

Good points for bisection
Always finds a root (for any continuous function)
Even for finite n, we know that there is definitely a root in $[a_n, b_n]$.

Good points for secant
Does not require a suitable initial interval
Often converges faster than bisection

Common trade-offs...

Prior information (eg initial interval) helps to guarantee convergence
Faster methods (eg secant) may not guarantee convergence but are useful in those cases where they work...

Termination criteria

Remember, at stage n, bisection guarantees that $a_n \leq x^* \leq b_n$

This means that we can specify the tolerance ζ required for our estimate, and stop our computation once $|b_n - a_n| < \zeta$

In the secant method, we get an estimate for x^* but we don't get exact upper/lower bounds.

How do we know when our estimate is "good enough"?

Mathematics can't answer this question, we need to define "good enough"

Typically, one would fix some ξ and stop when $|f(x_n)| < \xi$ or $|x_{n+1} - x_n| < \xi$. Of course, $|x_n - x^*|$ might still be large, depending on the function

Introductory project

- Based on this lecture
- Published online after exams
- Not submitted to Maths Faculty (no marks for it)
- Opportunity to try a full project (computing + write-up) and get feedback from a supervisor
- Model answer published in Michaelmas term

Now: introduce the main mathematical idea(s)

Fixed point iteration

(or Picard iteration)

As before we want to solve $f(x) = 0$.

Rewrite this equation as $x = g(x)$ for some g
(of course there are many ways to do this)

Choose some x_0, iterate $n = 1, 2, \ldots$ and compute $x_n = g(x_{n-1})$

If $f(x^*) = 0$ then $g(x^*) = x^*$ so the root x^* is a fixed point of this iteration scheme. . . can use this method to search for roots

This is a very simple scheme but of course there is no guarantee that the sequence $x_0, x_1, \ldots$ will converge to a fixed point

What would be a sensible choice for g?

Newton-Raphson iteration

A nice example is

$$g(x) = x - \frac{f(x)}{f'(x)}$$

(For reasonable functions, $g(x) = x$ implies $f(x) = 0$)

Hence we can iterate as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

No guarantee of convergence but for a (sufficiently nice) class of functions and suitable initial points x_0, can prove quadratic convergence (order $p = 2$).

In-built routines

MATLAB has built-in routines for finding roots

```
>> help fzero
[...]

>> fzero( @(x) x^2 - cosh(x), 1.0)
ans = 1.621347946103253

>> fsolve( @(x) exp(x) - 4*x , 0.0 )
[...]
```

"In real life", you would always use a built-in routine instead of writing your own. They are efficient, reliable, etc

However, for CATAM projects, we ask you to write your own code and not to use built-in routines (unless they have been approved by CATAM)

In-built routines

From the introduction to the project manuals:

- As a rule of thumb, do not use a built-in function if there is no equivalent MATLAB routine that has been approved for use, or if use of the built-in function would make the programming considerably easier than intended. For example, use of a command to test whether an integer is prime would not be allowed in a project which required you to write a program to find prime numbers. The CATAM Helpline (see §4 below) can give clarification in specific cases.

The reason is (of course) is that solving relatively simple problems will *help you to learn* how to design and implement computer programs

Next...

... how do computers manipulate numbers, and what implications does this have for mathematics?

... important for all CATAM projects

http://www.maths.cam.ac.uk/undergrad/catam/part-ia-lectures