Coursework 2 for MATH0070
Newton's method in complex n space

Vassilis Kostakos

May 3, 2001

The purpose of this coursework was to implement Newton's method in complex n space using Maple.

1 Using the procedure

The call to the `newton` procedure should be something like `newton(f,v,a,k,max)`, where

- `f` is a list of functions.
- `v` is the list of variables which appear in `f`.
- `a` is a list of initial values for the variables in `v`.
- `k` is an integer, such that any number ϵ, with $|\epsilon| < 10^{-k}$, will be regarded as insignificantly small.
- `max` is the maximum number of iterations.

The procedure will return a list `r` such that $|\texttt{f}(\texttt{r})| < 10^{-k}$.

2 Description of the code

First of all, I should say that the `newton` procedure requires the `linalg` package. No check is made to see if this package is loaded.

Some checks are initially made to `f,v,a` in order to make sure that they match, or in other words, they have the same number of items. I should point out that the procedure doesnt' check if all the variables in `v` do appear in `f`, or if all the variables used in `f` are listed in `v`.

After the initial checks, the procedure enters the main loop by calculating the Jackobian of `f` using the variables in `v`, giving `J`. Then, the initial values in `a` are substituted into the Jackobian. This means that the computation is no more symbolic. However, this is the only way of performing the calculations, since a symbolic computation would require vast amounts of memory after a couple of iterations.

At the same time, the variables in `f` are replaced with the values in `a`, giving `t`. Then, the procedure computes the `norm` of `t`, checking if the desired precision has been achieved. (Note: The `norm` is defined as the square root of the sum of squares of the coordinates). If the precision is adequate, the procedure exits, giving as the result `a`.

If the procedure doesn't exit, it prepares for generating the next `a`. In order to do this, it must make sure that the Jackobian computed earlier has an inverse. If not, the procedure prints an error message and exits.

If the Jackobian has an inverse, the procedure computes the next `a` using the formula

$$\texttt{a := a - J}^{-1}\texttt{t}$$

Note that `t`, which was computed earlier, is equal to `f(a)`. At this point, if the number of iterations has reached `max` then the procedure exits, returning `a` as the result. Otherwise, the loop repeats.

3 Remarks

There is a number of points which I should mention about my procedure

- My procedure works on lists. This can be a good thing, and a bad thing. The good thing is that it is very flexible with data types. For instance, it works perfectly good with real numbers, as well as irrational and complex nubmers. What's bad about this is that the type checking is not good, which means that different data types could get mixed together, resulting in a small chaos.

- The calculations which take place within `newton` are not symbolic, but use floating point arithmetic. This avoids the huge amounts of memory requirements that a symbolic calculation would have. Although floating point arithmetic is used, the procedure makes no attempt to specify the precisions that the calculations should use. This should be done by the user of the procedure, by assigning a value to the global variable "Digits". However, the procedure does allow the user to specify how good of an approximation is acceptible, through the use of the arguement `k`.

- As I mentioned earlier, the procedure doesn't make any intellegent effort in order to verify the correctness of the input data. The user should make sure that the list of functions, variables, and initial values are correct (matching). Also, the user should make sure that the `linalg` package has been loaded.

- My procedure uses the `norm` function for evaluating the approximation at every iteration. The `norm` function may be used with a number of different paramenters. My procedure uses it with the parameter `2`, which means that `norm` calculates the square root of the sum of squares of the given list of values. I supposes this could be done differently, or I could even let the user of my procedure specify how it should be done. But for the sake of simplicity, I decided to implement my procedure this way.

4 Sample Tests

At the end of this report I have included a printout of Maple, which demonstrates the use of my procedure.

5 Source code

```
newton:=proc(f::list, v::list, _a::list, k::integer, max::integer)
  # f : list of functions
  # v : list of variables
  #_a : initial values for variables (respectively)
  # k : decimal digits of required precision
  #max: number of iterations before giving up

  local a,t,j,i,n,size;
  a := _a;

  size := nops(f);                 #Number of items in list
  if ((size <> nops(v)) or (size <> nops(a))) then
    print("Mismatch of list lengths.");
    return;
  fi;

  for i from 1 to max do                        #main loop
      t := f;               #we need original f every time
      j := jacobian(f,v);              #calculate jacobian
      for n from 1 to size do           #replace variables
          j:=subs(v[n]=a[n],evalm(j));        #... in j ...
          t:=subs(v[n]=a[n],evalm(t));       #... and t (f)
      od;

      #Check if the desired  precision has been achieved
      if (simplify(norm(t,2))<10^(-k)) then
         return(evalm(a));
      fi;

      if det(j)=0 then             #check if inverse exists
         print("Jacobian doesn't have inverse.  Cannot continue");
         return evalm(a);
      fi;

      a := evalm(a - inverse(j)&*evalm(t));
  od;
  evalm(a);                                      #return a

end:
```