

# Script Programming

January 10, 2007

## 1 Introduction

Phoenix' script language is a complete programming language allowing one to not only program simulation runs but all kinds of repetitive tasks one might encounter. It being a dynamic object-oriented programming language reduces the development time substantially as no compilation step is needed like for instance for java/C/C++. As scripts can include other scripts, modular user libraries can easily be created.

In this application note a rootfinder will be implemented. This very common problem is chosen in order to keep the focus on the programming techniques rather than on the mathematics which in this case are rather simple.

The more advanced topics that will be addressed are:

**classes** The primary unit of abstraction.

**inheritance** Class hierarchies capture the commonality among similar—yet distinct—types.

## 2 Algorithm Description

We will implement a rootfinder that combines bisection and Muller's method. Given a function  $f$ , continuous on the interval  $[a, b]$  and  $f(a) \cdot f(b) < 0$ , at least one root,  $\zeta$ , is known to lie in  $[a, b]$ . Bisection will be used to successively half the interval until either  $\max\{|f(a_i)|, |f(b_i)|\} < \gamma$  or  $|a_i - b_i| < \delta$ . The first condition is used as a switch to start the more efficient method by Muller assuming that once  $|f(x)|$  is smaller than some user specified threshold value  $\gamma$ ,  $f(x)$  can be reasonably well approximated by a quadratic polynomial. The second condition is a terminal condition preventing bisection to run forever in case the first condition is never met.

In Muller's method, a quadratic polynomial,  $P$ , is constructed that passes through three given points  $[x_1, f(x_1)]$ ,  $[x_2, f(x_2)]$  and  $[x_3, f(x_3)]$ . One of the roots of this polynomial is used as an improved estimate of the root,  $\zeta$  of  $f(x)$ . If this new approximation is denoted by  $x_4$  then if  $x_4 \notin [a_i, b_i]$  another bisection step is taken otherwise  $[x_4, f(x_4)]$  is used to further reduce  $[a_i, b_i]$ . After assigning  $[x_1, f(x_1)] \leftarrow [x_2, f(x_2)]$ ,  $[x_2, f(x_2)] \leftarrow [x_3, f(x_3)]$ ,  $[x_3, f(x_3)] \leftarrow [x_4, f(x_4)]$



the next Muller step is attempted. See figure 1 for a graphical illustration. The interval  $[a_i, b_i]$  is reduced by Muller steps until  $|a_i - b_i| < \delta$  or  $|f(x_4)/P'(x_4)| < \delta/2$ . The latter case indicates that the  $x_4$  is a very good approximation of  $\zeta$ . Using  $f(x_4 + \Delta x) \approx f(x_4) + \Delta x f'(x_4) \approx f(x_4) + \Delta x P'(x_4)$ ,  $\Delta x$  is calculated such that  $f(x_4 + \Delta x) \approx -f(x_4)$  which leads to  $\Delta x = -2f(x_4)/P'(x_4)$ . So  $\zeta \in [x_4, x_5]$  with  $x_5 = x_4 + \Delta x$  should hold and if so, since  $|\Delta x/2| = |f(x_4)/P'(x_4)| < \delta/2$ ,  $|x_4 - x_5| < \delta$  and thus the first condition is met and the problem is solved.

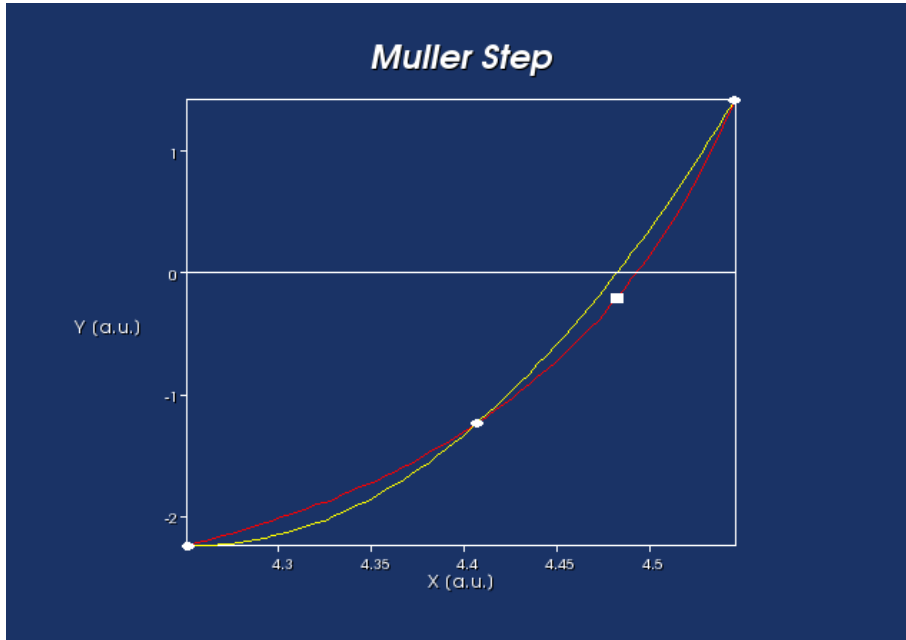


Figure 1: Illustration of a Muller step. The red line is the function of which the root is sought. The yellow line denotes the parabolic fit through the points  $P_1, P_2$  and  $P_3$  marked by a  $\bullet$ . The point marked  $\blacksquare$  denotes the resulting point  $P_4$ .

Given  $f(x)$  continuous on  $[a, b]$ , with  $f(a) \cdot f(b) < 0$ , desired tolerance  $t$ , user specified threshold parameter,  $\gamma$ , and  $\delta = t + \epsilon \cdot |b_i|$ , with  $\epsilon$  the unit round, the algorithm proceeds as:

#### 1 Bisection

- a If  $|a - b| < \delta$  goto step 3.
- b  $c \leftarrow (a + b)/2$ .
- c If  $f(b) \cdot f(c) \leq 0$ ,  $a \leftarrow c$ ; else  $b \leftarrow c$ .
- d If  $\max\{|f(a_i)|, |f(b_i)|\} < \gamma$  goto step 2.
- e goto step 1a.



## 2 Muller

- a  $P_1 \leftarrow [a, f(a)], P_2 \leftarrow [b, f(b)], P_3 \leftarrow [(a+b)/2, f((a+b)/2)]$ .
- b If  $|a-b| < \delta$  goto step 3.
- c  $P(\hat{x}) \leftarrow$  quadratic polynomial through  $P_1, P_2$  and  $P_3$ , with  $\hat{x} = x - x_3$ .
- d Solve  $P(\hat{x}) = 0, \Rightarrow \hat{x}_1, \hat{x}_2$ .
- e If  $|\hat{x}_1| < |\hat{x}_2|$  then  $x_4 \leftarrow x_3 + \hat{x}_1$  else  $x_4 \leftarrow x_3 + \hat{x}_2$ .
- f If  $x_4 \notin \langle a, b \rangle$  goto step 1.
- g If  $f(b) \cdot f(x_4) \leq 0, a \leftarrow x_4$ ; else  $b \leftarrow x_4$ .
- h If  $|f(x_4)/P'(x_4)| < \delta/2, x_5 \leftarrow x_4 - 2f(x_4)/P'(x_4 - x_3)$ .
- i If  $x_5 = a \vee x_5 = b$ , goto step 3.
- j If  $x_5 \in \langle a, b \rangle$ 
  - If  $f(b) \cdot f(x_5) \leq 0, a \leftarrow x_5$ ; else  $b \leftarrow x_5$ .
  - $P_2 \leftarrow P_3, P_3 \leftarrow P_4, P_4 \leftarrow P_5$ .
- k  $P_1 \leftarrow P_2, P_2 \leftarrow P_3, P_3 \leftarrow P_4$ .
- l goto step 2b.

## 3 Refine

$$\zeta \leftarrow a + f(a) \cdot (b-a)/(f(a) - f(b)).$$

4 return  $\zeta$ .

### 3 Implementation

Reducing the interval enclosing the root,  $\zeta$ , is a basic part of the algorithm. Therefore a class named Bracket is created as:

```
class Bracket()
{
    // data
    Point lb,rb;
    // methods
    /// Am I really a bracket
    function Ok() { return lb.y*rb.y < 0.;}
    /// is coordinate within current range
    function InRange(double x) {
        return (x-lb.x)*(x-rb.x) < 0.;
    }
    /// is coordinate within current range boundaries included
    function InRangeInclusive(double x) {
        return (x-lb.x)*(x-rb.x) <= 0.;
    }
    /// (re)set bracket checking input
    function Set(Point lb_, Point rb_) {
        lb=lb_; rb=rb_;
    }
}
```



```
if (!Ok())
    runtime("Input isn't a bracket!\n");
}
/// adjust according to new internal point
function Update(Point p) {
    if (!InRange(p.x))
        runtime("Update point outside current bracket!\n");
    if (p.y*rb.y <= 0) lb = p;
    else rb = p;
}
/// What's my size
function size() { return abs(lb.x-rb.x);}
/// What's my left boundary value
function LeftBoundaryValue() { return lb.y;}
/// What's my left boundary coordinate
function LeftBoundaryCoordinate() { return lb.x;}
/// What's my left boundary point
function LeftBoundary() { return lb;}
/// What's my right boundary value
function RightBoundaryValue() { return rb.y;}
/// What's my right boundary coordinate
function RightBoundaryCoordinate() { return rb.x;}
/// What's my right boundary point
function RightBoundary() { return rb;}
/// linear interpolate
function BestGuess() {
    return lb.x + lb.y*(rb.x-lb.x)/(lb.y-rb.y);
}
}
```

Classes can be extended like java to form new classes e.g.

listing

```
class RootFinder(double tolerance=1e-14,int maxIter=25)
    extends Bracket()
{
    /// functions that implement the parts of the algorithm
    function DoPrepare() {...}
    function DoStep() {...}
    function DoFinalize(int code) {...}
    function DoWhoAml() {...}
    /// the function of which the root is sought
    function DoF(double x) {...}
    // the interface
    /// set/get
    function SetTolerance(double tol) {...}
    function Tolerance() {...}
    function SetMaxIter(maxit) {...}
    function MaxIter() {...}
    /// d=t+eps.*|b|
    function delta() {...}
    function f(double x) {...}
    function Prepare() {...}
    function Step() {...}
    function Finalize(int code) {...}
    function WhoAml(int code) {...}
    /// find the root of f(x) in (a,b) with a tolerance, tol.
    function Root(double a, double b, double tol=1e-14) {...}
}
```



This defines a class `RootFinder`<sup>1</sup> which compared to the class `Bracket` has additional data members `tolerance` and `maxiter` and the listed functions as additional member functions.

The skeleton of iterative rootfinders is implemented in the function `Root` as:

```
function Root(double a, double b, double tol=1e-14) {  
    // initialize bracket  
    Set((a, f(a)), (b, f(b))); // calls Bracket.Set  
    // specify tolerance  
    SetTolerance(tol);  
    // perform preperation steps if necessary  
    Prepare();  
    int retval = 0;  
    // step  
    for (int it=0; !retval && (it<MaxIter()); ++ it)  
        retval=Step();  
    // do finalization steps if necessary  
    return Finalize(retval);  
}
```

where the functions `Prepare`, `Step` and `Finalize` are implemented as:

```
function Prepare() {  
    return DoPrepare();  
}  
function Step() {  
    if (size() < delta()) return 1;  
    else return DoStep();  
}  
function Finalize(int code) {  
    return DoFinalize(code);  
}
```

i.e. they delegate the work to functions that do the actual work and are intended to be overridden by subclasses that implement the base `RootFinder` class. For the base class the implementation is simply

```
function DoPrepare() {  
    return; // do nothing  
}  
function DoStep() {  
    return 0; // not done  
}  
function DoFinalize(int code) {  
    return BestGuess(); // lin. interpolation done by Bracket  
}
```

An implementation of the bisection algorithm could then be:

```
class BiSectionRootFinder extends RootFinder() {  
    function DoWhoAmI() { return "Bisection Method"; }  
    function DoStep() {  
        double c =  
            0.5*(LeftBoundaryCoordinate()+RightBoundaryCoordinate());  
        Update((c, f(c)));  
    }  
}
```

<sup>1</sup>For brevity the actual implementation has been left out as indicated by {...}. The full listing can be viewed by clicking on the link in the margin.



```
    return 0;
  }
}
```

Calls to the Root-function of instances of the BiSectionRootFinder class will invoke the extended version of the DoStep function instead of the one provided by the base class which does nothing. Note that the member functions of Bracket are available to BiSectionRootFinder.

Muller's method as described in step 2 of the intended algorithm needs three given points to start and might fail to yield a new point within the bracket. Initialization of the three points will be done in DoPrepare, whereas abnormal DoStep execution is indicated by a non-zero return value and dealt with in the DoFinalize step.

listing

```
class MullerRootFinder() extends RootFinder() {
  // MullerRootFinder specific
  // get the parabolic fit coefficients: ax^2 +bx+c
  function CalcABC(P1, P2, P3) {...}
  function NextPoint(double x) {...}
  // data
  Point P1, P2, P3, P4, P5;
  function DoWhoAml() { return "Muller's Method";}
  // the preparation step
  function DoPrepare() {
    P1=LeftBoundary();P2=RightBoundary();
    double xc=0.5*(P1.x+P2.x);
    P3=(xc, f(xc));
  }
  // A Muller step return codes:
  // -999 : fail
  // 2 : solution is on boundary
  // 0 : normal execution
  function DoStep() {
    double x4, x5, y3, y4, y5, dx;
    double abc(3);
    abc=CalcABC(P1, P2, P3);
    var sol()=RealABCformula(abc(0), abc(1), abc(2));
    if (!sol.NumSols || sol.IsComplex) // no real solution
      return -999;
    ...
    P1=P2; P2=P3; P3=P4;
    return 0; // normal execution
  }
  // check for failure
  function DoFinalize(int code) {
    if (code == -999)
      runtime("Muller failed\n");
    else
      return super.DoFinalize(code);
  }
}
```

The method as outlined in section 2 can be looked upon as Muller's method with an extended preparation step and a backup strategy in case of failure. We therefore implement it as an extension to the MullerRootFinder class. This presents a minor subtlety as MullerRootFinder already implements DoPrepare. Since we



don't want to reimplement that part, which would be impossible if we would not know its implementation details anyway, what we would like is a means to call MullerRootFinder.DoPrepare after the additional preparation steps are finished. Member functions of direct base classes can be called by prepending 'super.' in front of the function name. MullerRootFinder being the direct base of our new class this would mean calling `super.DoPrepare()`.

```
class ImprovedMuller(gamma=0.1) extends MullerRootFinder() {
  // ImprovedMuller specific
  function SetThreshold(double gam) { gamma=gam;}
  // overrides
  function DoWhoAml() { return "ImprovedMuller";}
  function DoPrepare() {
    do
    {
      double c =
        0.5*(LeftBoundaryCoordinate()+RightBoundaryCoordinate());
      Update((c,f(c)));
    }
    while ( max(abs(LeftBoundaryValue()),
                abs(RightBoundaryValue())) > gamma
            );
    super.DoPrepare(); // proceed with MullerRootFinder's DoPrepare
  }
  function DoFinalize(int code) {
    if (code == -999) // restart
      return Root(LeftBoundaryCoordinate(),
                  RightBoundaryCoordinate(),
                  Tolerance());
    else
      return super.DoFinalize(code);
  }
}
```

Phoenix' script language currently excludes user defined types(classes) and functions as function arguments. As one may have noted the DoF-function was not implemented by any of the subclasses defined sofar. In order to find the root of some function embed it in a class with a member function DoF that implements your function and extends e.g. ImprovedMuller like:

```
class MyFunc extends ImprovedMuller() {
  function DoF(double x) {
    return cos(X); //
  }
}
```

instances of this class than have a Root member function that can be called, e.g.

```
MyFunc() mf;
printf(mf.Root(0,3,1e-8),"\n");
```

## 4 Results

In order to show the workings of the algorithm graphically we further decorate the DoStep function of our ImprovedMuller class by making graphs as in figure 1,



at each step. Apart from the details of creating the actual plots, the process is similar to what we have shown before:

listing

```
class GraphedImprovedMullerMethod(int Np=65)
  extends ImprovedMuller()
{
  // data and functions specific to GraphedImprovedMullerMethod
  ...
  // fill f(x) and P(x) curves annotated with P1,P2 and P3
  function FillFuncAndFit() {...}
  // annotate curves with P4
  function AddPrediction() {...}
  // Plot the curves
  function PlotIterate() {...}
  // decorated version
  function DoStep() {
    FillFuncAndFit(); // P1, P2 and P3 are up to date
    // need MullerRootFinder.DoStep which is base of base
    var retval=super.super.DoStep();
    if (retval != -999) { // P4 is up to date
      AddPrediction(); // fill prediction point
      PlotIterate(); // plot the combined curves
    }
    return retval;
  }
}
```

Note the `super.super.DoStep()` which is needed to access the correct implementation of this function.

Figures 2-4 show the first couple of Muller steps for locating the root of  $f(x) = \tan x - x$  in the range  $[1.6, 4.6]$  with a desired tolerance and threshold parameter of  $t = 1e - 8$  and  $\gamma = 20$  respectively. Table 1 shows the bracket after each call to Update.

## 5 Conclusion

A non-trivial root finder has been implemented demonstrating the use of **classes** and **inheritance** in Phoenix' script language. These script features allow one to create class hierarchies of increasing complexity and functionality while keeping the added code small and local and thus easy to comprehend, maintain and extend further.

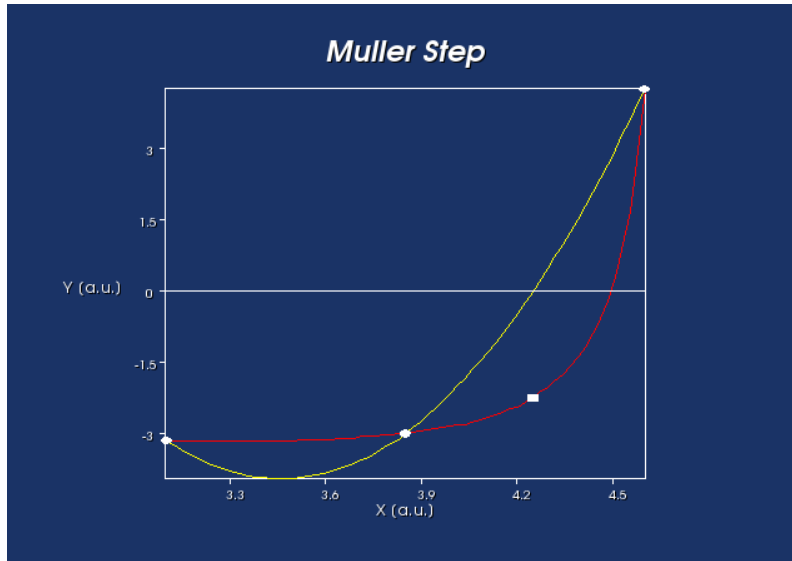


Figure 2: Step 1, bracket = [3.1, 4.6]

Step	$a$	$f(a)$	$b$	$f(b)$	size
start	1.6	-35.8325	4.6	4.26017	3.0
1	3.1	-3.14162	4.6	4.26017	1.5
2	4.25089	-2.24008	4.6	4.26017	3.49e-01
3	4.40733	-1.23156	4.6	4.26017	1.93e-01
4	4.40733	-1.23156	4.54622	1.41632	1.39e-01
5	4.48266	-0.206513	4.54622	1.41632	6.36e-02
6	4.49253	-0.0176359	4.54622	1.41632	5.37e-02
7	4.4934	-0.000266535	4.54622	1.41632	5.28e-02
8	4.4934	-0.000266535	4.49341	5.13917e-08	1.32e-05
9	4.49341	-5.14119e-08	4.49341	5.13917e-08	5.09e-09

Table 1: Convergence history. After step 8 the special case 2h of the algorithm description holds. Step 9 confirms that after updating the bracket with  $[x_5, f(x_5)]$ ,  $|a - b| < \delta$  so the algorithm stops.

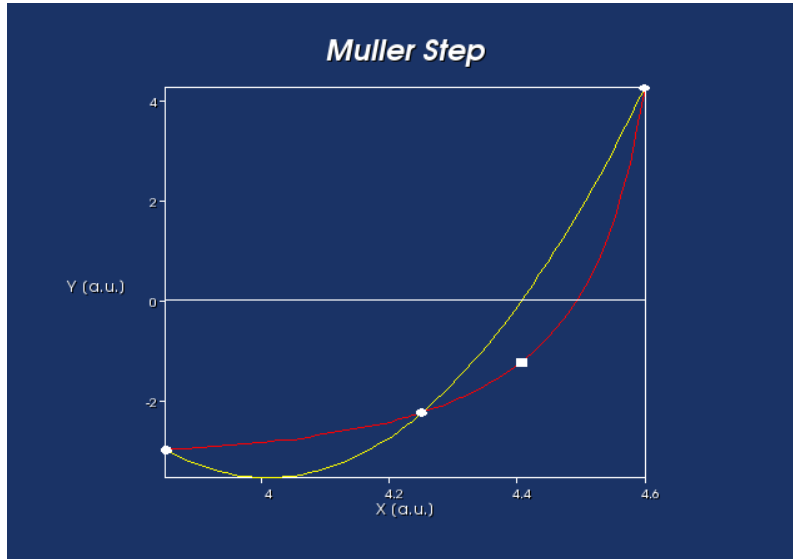


Figure 3: Step 2, bracket = [4.25089, 4.6]

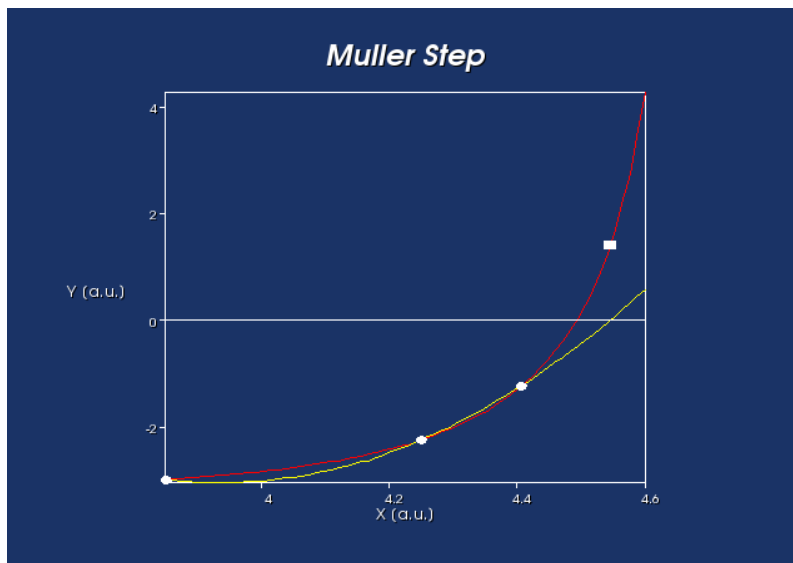


Figure 4: Step 3, bracket = [4.40733, 4.6]