Gavin Fielder
Math 461 Numerical Analysis
Prof. Sergei Fomin
May 1st, 2017

Description of My Nonlinear Shooting Program

My nonlinear shooting program takes an arbitrary nonlinear second order boundary value problem and solves it numerically with the shooting method with the chosen methods of approximation.

Initial setup and form of expected input

Problem. Given:

$y'' = f(x, y, y')$ for $a \leq x \leq b$ and $y(a) = \alpha$, $y(b) = \beta$, approximate the solution $y(x)$.

Change to initial value problem by letting $y'(a) = t$, and change to a system of 2 first order equations by letting $y = y_1$ and $y' = y_2$. The BVP becomes:

$$y_1{}' = y_2$$
$$y_2{}' = f(x, y_1, y_2)$$
$$a \leq x \leq b, \qquad y_1(a) = \alpha\,, \qquad y_2(a) = t$$

I henceforth refer to this system as the y-system.

For the sake of clarity to the user when declaring the function `f`, the variables y_1 and y_2 are renamed such that y_1 = `y` and y_2 = `yprime`. Thus f appears in the main file as:
`f(x, y, yprime)`.

As per the nonlinear shooting method used in the text, we need a second initial value problem to help iterate the parameter t, involving $z'' = \frac{\partial f}{\partial y}(x, y, y') \cdot z + \frac{\partial f}{\partial y'}(x, y, y') \cdot z'$. Changing this to a system of first order equations, we have

$$z_1{}' = z_2$$
$$z_2' = g(x, z_1, z_2) = \frac{\partial f}{\partial y}(x, y, y') \cdot z_1 + \frac{\partial f}{\partial y'}(x, y, y') \cdot z_2$$
$$a \leq x \leq b, \qquad z_1(a) = 0\,, \qquad z_2(a) = 1$$

I henceforth refer to this IVP system as the z-system.

Note $g(x, z_1, z_2)$ is actually a linear function in z_1 and z_2, and so only the coefficient functions $\frac{\partial f}{\partial y}(x, y, y')$ and $\frac{\partial f}{\partial y'}(x, y, y')$ are needed from the user to define it.

Finally, once these functions are defined, the user can select the number of subintervals, the methods of approximation they wish to use, the desired tolerance, as well as some other minor miscellaneous program settings such as output display precision.

Things required from the user

1. Interval endpoints `a` $= a$ and `b` $= b$
2. Dirichlet boundary values `LBC` $= \alpha = y(a)$ and `RBC` $= \beta = y(b)$
3. The function definition `f(x, y, yprime)` $= f(x, y, y')$
4. The partial derivative function definitions $\frac{\partial f}{\partial y}(x, y, y') =$ `dfdy(x, y, yprime)` and $\frac{\partial f}{\partial y'}(x, y, y') =$ `dfdyprime(x, y, yprime)`
5. Number of subintervals `N`
6. Desired tolerance `TOLERANCE`
7. Various program settings
    a. Default methods are Eulers method for generating meshpoints and Newtons method for iterating t. For meshpoint generation, Runge-Kutta of order 4 is available as a program setting.

Basic Overview of Program Function

1. Perform iterations of t until tolerance of target error of $\beta - y(b)$ has been reached.
2. Print full output to a file, including approximations for y, y′, z, and z′ for each mesh point of each iteration.
3. Provide a summary of each iteration to the console, including t-values and target errors.

Files and their algorithm-related functions:

- nonlinearshooting.cpp : main file
    - `main()` : main function. An outline of `main()` is given below.
    - `f(x, y, yprime)` : Evaluates $y'' = f(x, y, y')$ at the given x, y, and y'. User-defined prior to compilation.
    - `dfdy(x, y, yprime)` : Evaluates $\frac{\partial f}{\partial y}$ at the given x, y, and y'. User-defined prior to compilation.
    - `dfdyprime(x, y, yprime)` : Evaluates $\frac{\partial f}{\partial y'}$ at the given x, y, and y'. User-defined prior to compilation.
    - `g(x, z, zprime)` : Evaluates z′′ = $\frac{\partial f}{\partial y}(x, y, y') \cdot z + \frac{\partial f}{\partial y'}(x, y, y') \cdot z'$ at the given x, z, and z'. Includes functionality for obtaining estimates for y and y'.
    - `solution(x)` : Evaluates actual solution at a given x
    - `analyzeResults(shootingIterator)` : Calculates error at each meshpoint at the current iteration by comparing the meshpoint with results from `solution(x)`.
- shootingIterator.h / shootingIterator.cpp : Defines the class `shootingIterator`, which handles the nonlinear shooting algorithm.

- Constructor : Called on creation. Accepts and stores boundary value constants, calculates initial parameter t and step size h, links numerical evaluation functions to `f(x, y, yprime)` and `g(x, y, yprime)` in main file, sets default program settings, and initializes the first mesh point with alpha and initial *t*.
- `startNextIteration()` : Iterates a new *t*-value and generates meshpoints for both IVP systems (y-system and z-system).
- `meshPointFunction(x, u1, u2, u1_next, u2_next, h, system)` : Takes a given mesh point and a pointer to an IVP system to generate a new mesh point for that system. This is a wrapper function: depending on program settings, this function passes the parameters to either `eulers` or `rungeKutta`.
- `eulers(x, u1, u2, u1_next, u2_next, system)` : Generates a new mesh point using euler's method for the system based passed mesh point `u1` and `u2`. Stores the results in `u1_next` and `u2_next`. This is generalized to be used for either the y-system or the z-system.
- `rungeKutta(x, u1, u2, u1_next, u2_next, h, system)` : Generates a new mesh point using Runge-Kutta method of order 4 for the system based on passed mesh point `u1` and `u2`. Stores the results in `u1_next` and `u2_next`. This is generalized to be used for either the y-system or the z-system.
- `y_system(i, x, y, yprime)` : Evaluates the ith component of y-system. i = 0 simply returns the value passed to `yprime`. i = 1 refers back to `f(x, y, yprime)` defined in the main function through the use of a function pointer.
- `z_system(i, x, z, zprime)` : Evaluates the ith component of z-system. i = 0 simply returns the value passed to `zprime`. i = 1 refers back to `g(x, z, zprime)` defined in the main function through the use of a function pointer.
- `tValueIterator()` : Wrapper function to call the selected function for iterating *t*. Currently the program has only one option for iterating *t*, and so this function simply calls `newtonsMethod()`.
- `newtonsMethod()` : Reassigns parameter *t* according to Newton's method, using approximations obtained during the iteration.
- `error()` : returns the difference of `RBC` $= \beta$ and the approximated $y(b)$

Peculiarity regarding the use of Runge-Kutta Method

When the program is set to use Runge-Kutta method of order 4, it does so for approximating both the y-system and the z-system. However, the Runge-Kutta method of order 4 requires evaluations at $x + \frac{h}{2}$, that is, it requires function evaluations between mesh points. This is not a problem for the y-system, as the y-system is completely defined mathematically. But the z-system depends on approximations from the y-system. Hence, if Runge-Kutta is used, then the program must generate twice as many mesh points for the y-system as the z-system.

The functionality to do this is shared between two functions:

1. `startNextIteration()`, which loops meshpoint generation, will detect if Runge-Kutta is being used and if so will populate 2x-sized arrays to hold the meshpoints. This will cover half-measure approximations for the y-system.
    a. Note also that when calling rungeKutta for the y-system meshpoints, it must also pass h/2 as a modified step size substitution.
2. `g(x, z, zprime)` in the main file includes functionality for obtaining approximations from the y-system. Because of this peculiarity, it also detects whether Runge-Kutta is being used and if so, checks whether the approximations it requires are half-measures. If so, it retrieves the data from the larger arrays.

Explanation of Algorithm-related Functions

`main()`

Main() begins by setting up the program:

```
//Declare the shooting method handler (see shootingIterator.h)
shootingIterator shooter(a, b, N, LBC, RBC, &f, &g);
h = shooter.stepSize();

//Set program settings
if (USE_RUNGE_KUTTA) shooter.useRungeKutta();
else shooter.useEulers();
shooter.useNewtons();
shooter.setPrecision(PRECISION);

//Begin data output to file
shooter.beginOutput("output.txt");
```

The first line declares a shootingIterator, which will handle the actual algorithm and its computational steps. The last line will open a file, output.txt, for output streaming. By calling this, every iteration that the shootingIterator performs will be added to this file, including t-values and approximations for y, y', z, z' at each mesh point, as well as the target error.

```
//Welcome message
cout << endl;
stringstream output;
output << "************************************************\n";
output << "****** Nonlinear Shooting Method results: *******\n";
output << "************************************************\n\n";
output << "Approximating the solution of second-order BVP\n";
output << "with y(" << a << ") = " << LBC << ", y(" << b << ") = "
       << RBC << ", with N = " << N << "\n";
output << "subintervals,  " << a << " <= x <= " << b << " :\n";
```

```
if (USE_RUNGE_KUTTA) output << "Using Runge-Kutta method of order 4 and ";
else output << "Using Euler's method and ";
output << "\nNewton's Method for iterative parameter t\n";
output << "Target tolerance is " << TOLERANCE << "\n";
output << "Calculations are being streamed to: " << shooter.getFileName()
      << ".\n\n";
cout << output.str();
shooter.addToFile(output.str());
cout << "Basic information:\n\n";
```

This creates a header that is both output to the console, and added at the top of the output file. It contains basic information about the problem being approximated as well as what methods will be used to approximate it.

Now the important part:

```
//Start iterating
do {
      shooter.startNextIteration();
      shooter.printSummary();
} while ((abs(shooter.error()) > TOLERANCE)
            && (shooter.iterationNumber() < MAX_ITERATIONS));
```

This loops over successive t-values until either the tolerance has been reached or the maximum number of iterations has been exceeded. Each iteration, it prints a summary to the console. Since I previously called `shooter.beginOutput("output.txt")`, the shootingIterator also automatically outputs all of the complete generated data to this file after each iteration.

The end of the function holds analysis functionality:

```
//Done iterating, figure out why
if (shooter.iterationNumber() >= MAX_ITERATIONS) {
      cout << "\nMaximum number of iterations reached. Recommend switching\n";
      cout << "  approximation methods for faster convergence.\n";
}
else {
      //Tolerance reached
      shooter.addToFile("Tolerance achieved. Stopping Iterations.");
      cout << "\nReached desired tolerance at " << shooter.iterationNumber();
      cout << " iterations. ";
      //Check if the user provided a solution to check against
      if (SOLUTION_KNOWN) {
            cout << "Exact solution detected; analyzing meshpoint errors.\n";
            long double maxError = analyzeResults(shooter,
                              ANALYSIS_X_PREC, ANALYSIS_Y_PREC);
      }
```

```
      else cout << endl;
}
cout << endl << endl;

shooter.endOutput();
shooter.deallocate();
```

The first if statement determines whether the maximum number of iterations was reached, or if alternatively the tolerance was satisfied. If the tolerance was satisifed, it checks if the user indicated that the solution was known, and if so, it calls `analyzeResults`. It passes the shootingIterator by reference, which allows it access to the data of the last iteration performed. `analyzeResults` will return the largest absolute error over the meshpoints, which is then printed to the console.

The remaining lines simply close the file stream and deallocate memory.

f(x, y, y'), dfdy(x, y, y'), dfdyprime(x, y, y'), and solution(x)

These are the user-defined functions. A variety of mathematical functions, such as `sin(x)`, are available from the `<cmath>` library.

```
//Here is f(y1, y2, x)
long double f(long double x, long double y, long double yprime) {

      //Edit starting here v
      long double result = (1 - pow(yprime, 2) - y * sin(x)) / 2;
      //End editing        ^
      return result;
}
//Here is the partial derivative of f with respect to y
long double dfdy(long double x, long double y, long double yprime) {

      //Edit starting here v
      long double result = sin(x) / -2.00;
      //End editing       ^
      return result;
}
//Here is the partial derivative of f with respect to y'
long double dfdyprime(long double x, long double y, long double yprime) {

      //Edit starting here v
      long double result = yprime * -1;
      //End editing        ^
      return result;
}
```

```
//This is the actual solution, if known. This allows checking meshpoint error
long double solution(long double x) {

	//Edit starting here v
	long double result = 2 + sin(x);
	//End editing	^
	return result;
}
```

g(x, z, z')

This is the function for the second component of the z-system. It is different from f in that it requires the ability to retrieve approximations for y and y'

```
long double g(long double x, long double z, long double zprime,
		long double * y_approx, long double * yprime_approx,
		long double * y_full, long double * yprime_full) {
```

`y_approx` is a pointer to an array of approximations for y -- similar for `yprime_approx`. The `y_full` `and yprime_full` are pointers to the 2x-sized arrays for when Runge-Kutta is used and half-measure approximations are needed.

```
//Get approximation mesh points from y-system
int i = 0;
long double y, yprime;
if (!(USE_RUNGE_KUTTA)) {
	i = lround((x-a)/h);
	y = y_approx[i];
	yprime = yprime_approx[i];
}
```

The above is when Runge-Kutta is not used. An x is passed in, and we know $x = a + ih$, so we can reverse construct the integer i from this relationship. Once we have this i, we can set `y` equal to the relevant approximation for y, and `yprime` equal to the relevant approximation for y'

```
else { //Runge-kutta used. Need to determine whether requested
	//  x-value is a half-measure in approximation arrays
	if (lround(((2*(x-a))/h)) % 2 == 0) {
		//not a half measure, use y_approx
		i = lround((x-a)/h);
		y = y_approx[i];
		yprime = yprime_approx[i];
	}
	else {
		//half measure, get data from full array
```

```
            i = lround((((x-a)/h) * 2));
            y = y_full[i];
            yprime = yprime_full[i];
      }
}
```

If the program is set to use Runge-Kutta, it must use the above block to appropriately assign y and yprime. Generally speaking, $i = (x - a)/h$ will either be an integer, or an integer and a half. So by checking whether `(2*(x-a))/h` is even or odd (via the modulus operation `% 2`), we can determine if the passed x wants to refer to an actual meshpoint, or a point between meshpoints. Based on this, we know which array to access the data from.

```
long double result = dfdy(x, y, yprime) * z
                        + dfdyprime(x, y, yprime) * zprime;
return result;
```

Finally, the numeric result for g(x, z, z') is computed by referencing the user defined functions `dfdy(x, y, yprime)`, `dfdyprime(x, y, yprime)` together with the passed values of `z` and `zprime`.

analyzeResults(shootingIterator)

This function takes a shootingIterator by reference, and compares its approximations for y against the user-defined `solution(x)`.

It formats a table of data and prints the table to the output file, if one is open, and depending on program settings, it also prints the table to the console.

Finally, it returns the maximum absolute error over all meshpoints.

```
//Calculates and prints errors for each meshpoint
long double analyzeResults(shootingIterator& shooted, int x_precision,
                    int y_precision, bool suppressConsoleOutput) {

stringstream output;
long double error = 0.00;
long double maxError = 0.00;
output << "************************************************\n";
output << "*************** Mesh point errors ****************\n";
output << "************************************************\n";
output << endl;
output << "Iteration number: " << shooted.iterationNumber() << endl;
output << "t = " << fixed << setprecision(y_precision) << shooted.getT()
        << endl << endl;
for (int i = 0; i < N + 1; i++) {
      output << "  x = " << fixed << setprecision(x_precision)
```

```
                << shooted.getDataPoint(0, i);
        output << ", \tw = " << fixed << setprecision(y_precision)
                << shooted.getDataPoint(1, i);
        output << ", \ty = " << fixed << setprecision(y_precision)
                << solution(shooted.getDataPoint(0,i));
        error = shooted.getDataPoint(1, i) -
                        solution(shooted.getDataPoint(0,i));
        output << ", \tError = " << scientific << error << endl;
        if (abs(error) > maxError) maxError = abs(error);
}
output << endl;
output << "Maximum absolute meshpoint error is " << scientific
        << maxError << "\n";

//Add this to the file if streaming
shooted.addToFile(output.str());

//Print analysis to console
if (!(suppressConsoleOutput)) {
        cout << endl << output.str();
}
else {
        cout << "Maximum absolute meshpoint error is " << scientific << maxError
             << "\n";
        cout << "Meshpoint error analysis provided with "
             << shooted.getFileName();
        cout << endl;
}

return maxError;
```

`getDataPoint(0,i)` retrieves the x-value of the ith mesh point, and `getDataPoint(1, i)` retrieves the approximated y-value of the ith mesh point.

`shootingIterator` Functions

I will cover the functions of shootingIterator that pertain to the nonlinear shooting algorithm. A full list of the functions and their descriptions are provided in shootingIterator.h

```
//Constructor
// simple form: shootingIterator(a, b, N, LBC, RBC, &f, &g)
```

Declare a shootingIterator takes the above 5 parameters a, b, N, `LBC` $= \alpha$, and `RBC` $= \beta$. `&f` and `&g` are the address of functions `f(x, y, yprime)` and `g(x, z, zprime)` in the main file.

The constructor includes these important lines:

```
f = f_in;
g = g_in;
 ...
h = (b - a) / N;
t = (RBC - LBC) / (b - a);
 ...
//Push initial conditions for y_system and z_system
y_approximation[0] = LBC;
yprime_approximation[0] = t;
z_approximation[0] = 0.00;
zprime_approximation[0] = 1.00;
y_full[0] = LBC;
yprime_full[0] = t;
```

The first two assign global function pointers f and g. This allows shootingIterator.cpp to link the `f(x, y, yprime)` and `g(x, z, zprime)` functions defined in the main file into the shootingIterator functions.

The next two lines assign a step size h and initial value for *t*.

The remaining lines shown above set up initial conditions in the data arrays, to prepare for the first iteration.

startNextIteration()

The basic functionality of startNextIteration() is as follows:

1. Assign a new t-value
2. Clear and reallocate the memory for y, y', z, and z' approximations
3. Push initial conditions (identical to code above from constructor)
4. Generate meshpoints for the y-system
5. Calculate target error $\beta - y(b)$
6. Generate meshpoints for the z-system

Assigning a new t-value is accomplished with the line:

```
t = tValueIterator();
```

`tValueIterator()` in turn `calls newtonsMethod()`, which will use the existing data and t-value to return a new t-value.

Items 2 and 3 are straightforward. Item 4 starts the brunt of the computation. First, supposing we're not using Runge-Kutta,

```
//Calculate the y_system meshpoints
// This is the normal method--if set to runge-kutta, we must do extra
if (selectedMeshPointFunction != RUNGE_KUTTA) {
	for (int i = 0; i < N; i++) {
		meshPointFunction(x[i], y_approximation[i],
			yprime_approximation[i], y_approximation[i+1],
			yprime_approximation[i+1], h, &y_system);
	}
}
```

This iterates i = 0, 1, 2, … N - 1. For each i, it calls meshPointFunction, passing in a value for x, an approximated value for y, and an approximated value for y'. meshPointFunction generates a new meshpoint approximation for y and y'. It stores the y result into the next position in the y approximation array, and stores the y' result into the next position in the y' approximation array. The final two arguments tell it to use the normal step size h, and to reference the y-system to perform all the necessary calculations.

If Runge-Kutta method of order 4 is used, we have to create twice as many meshpoints. Here is the algorithm:

```
else { //runge-kutta requires half measures for the y-system, so
	// populate over twice as many mesh points
	//Note for this one only we must pass an h/2 instead of h as
	// the step size substitution
	for (int i = 0; i < 2*N; i++) {
		meshPointFunction(a + ((h*i)/2), y_full[i], yprime_full[i],
				y_full[i+1], yprime_full[i+1],
				(h/2), &y_system);
	}
	//Copy over full measures to the normal array
	for (int i = 0; i < N+1; i++) {
		y_approximation[i] = y_full[i*2];
		yprime_approximation[i] = yprime_full[i*2];
	}
}
```

The first loop is a near-equivalent to the normal loop defined for not using Runge-Kutta. The differences are that it passes values and references to the larger `y_full` and `yprime_full` arrays, and passes h/2 as a modified step size.

Once finished, it copies the non-half-measure meshpoint values over to the normal arrays, so the rest of the program can access them normally.

After calculating and recording the error, it calculates meshpoints for the z-system. This is precisely similar to the first loop (for non-Runge-Kutta y-system meshpoints), except that it uses and populates

approximation arrays for z and z', and passes the z-system to meshPointFunction for all necessary calculations.

```
//Calculate the z_system meshpoints
for (int i = 0; i < N; i++) {
	meshPointFunction(x[i], z_approximation[i], zprime_approximation[i],
			z_approximation[i+1], zprime_approximation[i+1],
			h, &z_system);
}
```

meshPointFunction(x, u1, u2, ... , system_ptr)

This function is a wrapper which calls the selected mesh point generating function, either `rungeKutta(x, u1, u2, … , system_ptr)`, or `eulers(x, u1, u2, … , system_ptr)`. It is straightforward and all parameters (except for h, in the case of eulers--euler's method being used means only the standard step size determined upon declaring the shootingIterator will ever be needed) are simply passed through to the correct function.

```
//Calls the selected meshpoint generator function
void shootingIterator::meshPointFunction(long double x, long double u1,
				long double u2, long double& u1_next,
				long double& u2_next, long double hh,
				long double (*system_ptr)(int,
				long double, long double, long double)) {

	if (selectedMeshPointFunction == RUNGE_KUTTA)
		rungeKutta(x, u1, u2, u1_next, u2_next, hh, system_ptr);

	else if (selectedMeshPointFunction == EULERS_METHOD)
		eulers(x, u1, u2, u1_next, u2_next, system_ptr);

	else //default
		eulers(x, u1, u2, u1_next, u2_next, system_ptr);
}
```

eulers(x, u1, u2, u1_next, u2_next, system_ptr)

This function generates a new meshpoint using Euler's method for systems. It takes an existing meshpoint (`u1`, `u2`) for the given value of x, and generates a new mesh point, which it stores into (`u1_next`, `u2_next`).

```
//Generate a mesh point for a system using eulers method
void shootingIterator::eulers(long double x, long double u1, long double u2,
	long double& u1_next, long double& u2_next,
	long double (*system_ptr)(int, long double, long double, long double)) {
```

```
	u1_next = u1 + (h * system_ptr(0, x, u1, u2));
	u2_next = u2 + (h * system_ptr(1, x, u1, u2));
}
```

system_ptr can either refer to `y_system(i, x, y, yprime)`, or `z_system(i, x, z, zprime)`, where the i refers to the component of the system to evaluate (i = 0 returns the first component value of the system, i = 1 returns the second).

rungeKutta(x, u1, u2, … , system_ptr)

This function generates a new meshpoint using Runge-Kutta of order 4 for systems. It takes an existing meshpoint (`u1`, `u2`) for the given value of x, and generates a new mesh point, which it stores into (`u1_next`, `u2_next`).

```
//Generate a mesh point for a system using runge-kutta method
void shootingIterator::rungeKutta(long double x, long double u1, long double u2,
	long double& u1_next, long double& u2_next, long double hh,
	long double (*system_ptr)(int, long double, long double, long double)) {
	long double k1[2];
	long double k2[2];
	long double k3[2];
	long double k4[2];
	//Note: all the k1's must be computed before any of the k2's can be computed
	for (int i = 0; i < 2; i++)
		k1[i] = hh * system_ptr(i, x, u1, u2);
	//All the k1's computed, now we can compute the k2's
	for (int i = 0; i < 2; i++)
		k2[i] = hh * system_ptr(i, x + (hh/2), u1 + 0.5*k1[0], u2 + 0.5*k1[1]);
	//All the k2's computed, now we can compute the k3's
	for (int i = 0; i < 2; i++)
		k3[i] = hh * system_ptr(i, x + (hh/2), u1 + 0.5*k2[0], u2 + 0.5*k2[1]);
	//All the k3's computed, now we can compute the k4's
	for (int i = 0; i < 2; i++)
		k4[i] = hh * system_ptr(i, x + hh, u1 + k3[0], u2 + k3[1]);
	//Now lets construct the meshpoint
	u1_next = u1 + ((k1[0] + 2*k2[0] + 2*k3[0] + k4[0]) / 6.00);
	u2_next = u2 + ((k1[1] + 2*k2[1] + 2*k3[1] + k4[1]) / 6.00);
}
```

The only peculiarity in this function is `hh` versus `h`--Since in using Runge-Kutta the shootingIterator needs to create twice as many meshpoints for the y-system as it does the z-system, it needs to be able to accept different values for the step size. Since `h` is the member variable for the normal step size, `hh` is locally-used step size for this function only, which will either have a value of h, or h/2.

y_system(i, x, y, y') and z_system(i, x, z, z')

These two functions handle functionality emulating y-system and z-system respectively. They reference `f` and `g` in the main file (via through the global function pointers `f` and `g` defined in shootingIterator.cpp)

```
long double shootingIterator::y_system(int index, long double x,
                                       long double y1, long double y2) {
      long double result = 0.00;
      switch (index) {
            case 0:
                  result = y2;
                  break;
            case 1:
                  result = f(x, y1, y2);
                  break;
      }
      return result;
}
long double shootingIterator::z_system(int index, long double x,
                                       long double z1, long double z2) {
      long double result = 0.00;
      switch (index) {
      case 0:
            result = z2;
            break;
      case 1:
            result = g(x, z1, z2, y_approximation,
                        yprime_approximation, y_full, yprime_full);
            break;
      }
      return result;
}
```

By passing in 0 to i, it returns the first component, which will in both cases return the -prime value itself. By passing in 1 to i, it returns the second component, which is defined by `f` and `g` in the main file, which we can access from here via the global function pointers `f` and `g`. Note that `g(x, z, zprime)` also requires access to approximations for y and y', so we pass the names of the arrays (as pointers) to `g`.

tValueIterator() and newtonsMethod()

```
//Calls the selected t-value iterator function
long double shootingIterator::tValueIterator() {
      //right now there's only one option
      return newtonsMethod();
}
```

```
//Use newton's method to find the next t-value
long double shootingIterator::newtonsMethod() {
	long double result =
		t - ((y_approximation[N] - RBC) / z_approximation[N]);
	return result;
}
```

`tValueIterator()` is a wrapper function which calls the selected iteration function for the parameter t. There's only one option implemented right now, so it calls `newtonsMethod()`.

`newtonsMethod()` calculates a new t value with the calculation shown above. It uses the approximations for y(b) and z(b) in the current data set. It returns the next *t*-value, to overwrite the existing *t*, which happens in `startNextIteration()` when a new iteration is being started.

error()

```
//Calculates the distance between the final y approximation and the RBC
long double shootingIterator::error() const {
	long double result = 0.00;
	if (allocated) result = RBC - y_approximation[N];
	else cout << "Error calculating target error: no memory allocated\n";
	return result;
}
```

`error()` returns the difference between `RBC` $= \beta$ and the y(b) approximation for the current iteration.