1

Effect of the Guess Function & Continuation Method on the Run Time of MATLAB BVP Solvers

Fikri Serdar Gökhan
*Gazikent University, Faculty of Engineering and Architecture,
Department of Electrical and Electronic Engineering, Gaziantep
Turkey*

1. Introduction

The MATLAB computing environment is a package used extensively throughout industry, research and education by users of a complete range in proficiency. MATLAB provides then an ideal platform to introduce such an item of Boundary Value Problem (BVP) software and indeed, Kierzenka and Shampine (Kierzenka & Shampine, 2001) developed the core BVP Ordinary Differential Equation (ODE) software bvp4c to solve a large class of two-point boundary value problems of the form;

$$y'(x) = f(x, y(x), p) \tag{1}$$

$$g(x_L, x_R, y(x_L), y(x_R), p) = 0 \tag{2}$$

where f is continuous and Lipschitz function in y and p is a vector of unknown parameters. Their view was that a user solving a BVP of form (1) in MATLAB would be most interested in the graphical representation of a solution, and as such a solver with a MIRK4-based Simpson Method would be appropriate for graphical accuracy.
If information is specified at more than one point the problem (1) becomes a Boundary Value Problem. The most common types of BVP are those for which information given at precisely two points. These are known as two-point boundary value problems.
The MATLAB BVP solver of bvp4c is introduced as a Residual control based, adaptive mesh solver. An adaptive mesh solver is an alternative approach to that of a uniform mesh, which would specify a uniform grid of data points x_i over the interval $[x_i, x_{i+1}]$ and solve accordingly. The adaptive solver will adjust the mesh points at each stage in the iterative procedure, distributing them to points where they are most needed. This can lead to obvious advantages in terms of computational and storage costs as well as allowing control over the grid resolution. The concept of a residual is the cornerstone of the bvp4c framework; being responsible for both errors control and mesh selection (Hale, 2006).
The most difficult part for the solution of BVPs is to provide an initial estimation to the solution. In order to direct the solver for the solution of interest, it is necessary to assist the solver by informing it with a guess. Not only for the computation of the solution of interest

but also whether any solution is achieved or not depends strongly on the initial guess. Therefore, depending of the guess function, BVPs may have no solution or a single solution, or multiple solutions. Moreover, the quality of the initial guess can be critical to the solver performance, which reduces or augments the run time. However, coming up with a sufficiently good guess can be the most challenging part of solving a BVP. Certainly, the user should apply the knowledge of the problem's physical origin (MATLAB Documentation).

In MATLAB, when solving BVPs the user must provide a guess to assist the solver in computing the desired solution (Kierzenka & Shampine, 2001). MATLAB BVP solvers call for users to provide guesses for the mesh and solution. Although MATLAB BVP solvers take an unusual approach to the control of error in case of having poor guesses for the mesh and solution, especially for the nonlinear BVP, a good guess is necessary to obtain convergence (Shampine *et al.*, 2003).

Whatever intuitive guess values/functions are imposed, eventually the BVP solver fails for some parameters or for some lengths. If any guess values works for the range of length, the rest of the length may be extended using *continuation*. The method of continuation exploits the fact that the solution obtained for one input will serve as the initial guess for the next value tried. In case of any difficulty in finding a guess for the interval of interest, generally it will be easier to solve the problem on a shorter interval. Then the solution of the sequence of BVPs on the shorter interval will be used as a guess for the next section. With modest increases in the interval, this will continue until the interval of interest is spanned (Shampine *et al.*, 2003).

The cost of the continuation method is the increased run time. How the guess value good is, the less computation time it takes with the continuation method. This is due the fact that, the remaining length depends of the convergence length (based on the guess value) which its higher value reduces the computation time.

2. Initial setup

The first step in solving a problem is defining it in a way the software can understand. The bvp4c framework uses a number of subfunctions which make it as simple as possible for the user to enter the ODE function, initial data and parameters for a given problem. By way of the following example we see exactly how a problem is supplied and solved by bvp4c. For the evaluation of the guess value /function, the steady-state Brillouin equation is exploited. The coupled ODEs for the evolution of the intensities of pump I_p and Stokes I_s can be written as (Agrawal, 2001),

$$\frac{dIp}{dz} = -g_B I_P I_S - \alpha I_p \tag{3}$$

$$\frac{dIs}{dz} = -g_B I_P I_S + \alpha I_S \tag{4}$$

where $0 \le z \le L$ is the propagation distance along the optical fiber of the total length L, α is the fiber loss coefficient, g_B is the Brillouin gain coefficient, respectively. Here, it is assumed that, Stokes wave is launched from the rear end of the fiber. Then the known values of the input pump power $I_p(0)$ and the Stokes wave power I_s (L) are referred as the boundary values.

The first task is to define the ODEs in MATLAB as a function to return these equations. Similarly the user then rewrites the boundary conditions to correspond to this form of the problem. We may code the ODEs for scalar evaluation and boundary conditions, respectively as,

```
function dydx = bvpode(x,y)
global alpha_s gb K

dydx =  [ -gb *y(1)*y(2)-alpha_s*y(1)
          -gb *y(1)*y(2)+alpha_s*y(2) ];

------------------------------------------------------------

function res = bvpbc(ya,yb)
 global Ip0 IsL

 res = [ya(1)- Ip0
        yb(2)- IsL ];
```

The next step is to create an initial guess for the form of the solution using a specific MATLAB subroutine called bvpinit . The user passes a vector x and an initial guess on this mesh in the form bvpinit (x, Yinit), which is then converted into a structure useable by bvp4c. Aside from a sensible guess being necessary for a convergent solution the mesh vector passed to bvpinit will also define the boundary points of the problem, i.e. x_L = x[1] and x_R = x[end].
The initial guess for the solution may take one of two forms. One option is a vector where Yinit(i) is a constant guess for the i-th component y(i,:) of the solution at all the mesh points in x. The other is as a function of a scalar x, for example bvpinit(x,@yfun) where for any x in [a, b], yfun(x) returns a guess for the solution y(x). It must be pointed out that even when a function is supplied that approximates the solution everywhere in the interval; the solver uses its values only on the initial mesh. The guess can be coded as a function of a scalar x as,

```
function v = guess(x)

global alpha_s L gb k Pp0 PsL Aeff
a=alpha_s*L;
k=gb/Aeff*Pp0*L;
epsilon=PsL/Pp0;
kappa=-log(gb/Aeff*PsL*L);
T=log(kappa*(1-kappa/k));
c0=-(PsL + PsL*k - 1)/((PsL*k^2)/2 + 1);
A=c0./(1-(1-c0).*exp(-c0*k.*x));
B=c0*(1-c0)./(exp(c0*k.*x)-1+c0);
w=(A.^(exp(-a))).*exp(-a.*x);
u=(B.*exp(a.*(x-1)));

v=[w*Pp0; u*Pp0];
```

The next subroutine to look at is bvpset, that specifies which options bvp4c should be use in solving it. The function is called options = bvpset('name1',value1,...) and since MATLAB documentation gives an in depth account of each of the options only a brief outline of those notable is given here (Hale, 2006).

```
options = [];  % default
%options = bvpset('Stats','on','RelTol',1e-5,'abstol',1e-4);
%options = bvpset(options,'Vectorized','on');
%options = bvpset(options,'FJacobian',@odeJac);
%options = bvpset(options,'BCJacobian',@bcJac);
```

RelTol - Relative tolerance for the residual [positive scalar 1e-3]
The computed solution $S(x)$ is the exact solution of $S'(x) = F(x, S(x)) + res(x)$. On each subinterval of the mesh, component i of the residual must satisfy norm

$$norm\left(\frac{res(i)}{\max(abs(F(i)),\ AbsTol(i)/\mathrm{Re}lTol)}\right) \le \mathrm{Re}lTol$$

AbsTol - Absolute tolerance for the residual [positive scalar or vector 1e-6]
Elements of a vector of tolerances apply to corresponding components of the residual vector. AbsTol defaults to 1e-6.
FJacobian \ BCJacobian - Analytical partial derivatives of ODEFUN \ BCFUN
Computation of the Jacobian matrix at each mesh point can be a very expensive process. By passing an analytic derivative of the ODE and BC functions the user can greatly reduce computational time. For example when solving y′ = f(x, y), setting FJacobian to `@FJAC` where `∂f/∂y = FJAC(x, y)` evaluates the Jacobian of f with respect to y.
Stats - Display computational cost statistics [on — off]
Vectorized - Vectorized ODE function [on — off]
As will be discussed in section 6, bvp4c is able to accept a vectorised function which can markedly increase the efficiency of calculating local Jacobians over using finite differences with the odenumjac subroutine. Hence in the following programs, we will define

```
options = bvpset('Stats','on','RelTol',1e-5,'abstol',1e-4);
solinit = bvpinit(linspace(0,L,2), @guess);
```

And call the bvp4c routine with:

```
sol = bvp4c(@ode,@bc,solinit,options);
```

The above essentially ends the user input in solving the BVP system and the rest is left to bvp4c. Within the framework there are several notable steps which should be expounded.

3. Derivation of the guess

In this chapter, four guess functions are derived for the assistance of the MATLAB BVP solvers with the help of MATLAB symbolic toolbox.

3.1 1st Guess

If the constant guess is used as the initial values i.e., for the pump "Ip0" and for the Stokes "IsL" and `L = 10000;`

```
solinit = bvpinit(linspace(0,L,2),@guess);
options = bvpset('Stats','on','RelTol',1e-5);
function v = guess(x)
global Ip0  IsL
v=[Ip0 ; IsL];
```

It prompts as, “Unable to solve the collocation equations -- a singular Jacobian encountered”. However, if the computation length is decreased as, `L = 1000`, the solver *is able* to solve coupled equations with these poor guesses v=[Ip0 ; IsL]. Therefore, with these guess values the convergence length (the maximum length which the solver is able to converge) is 1000 meter. The evolution of the guess values (estimate) with the real solution is shown in Fig. 1

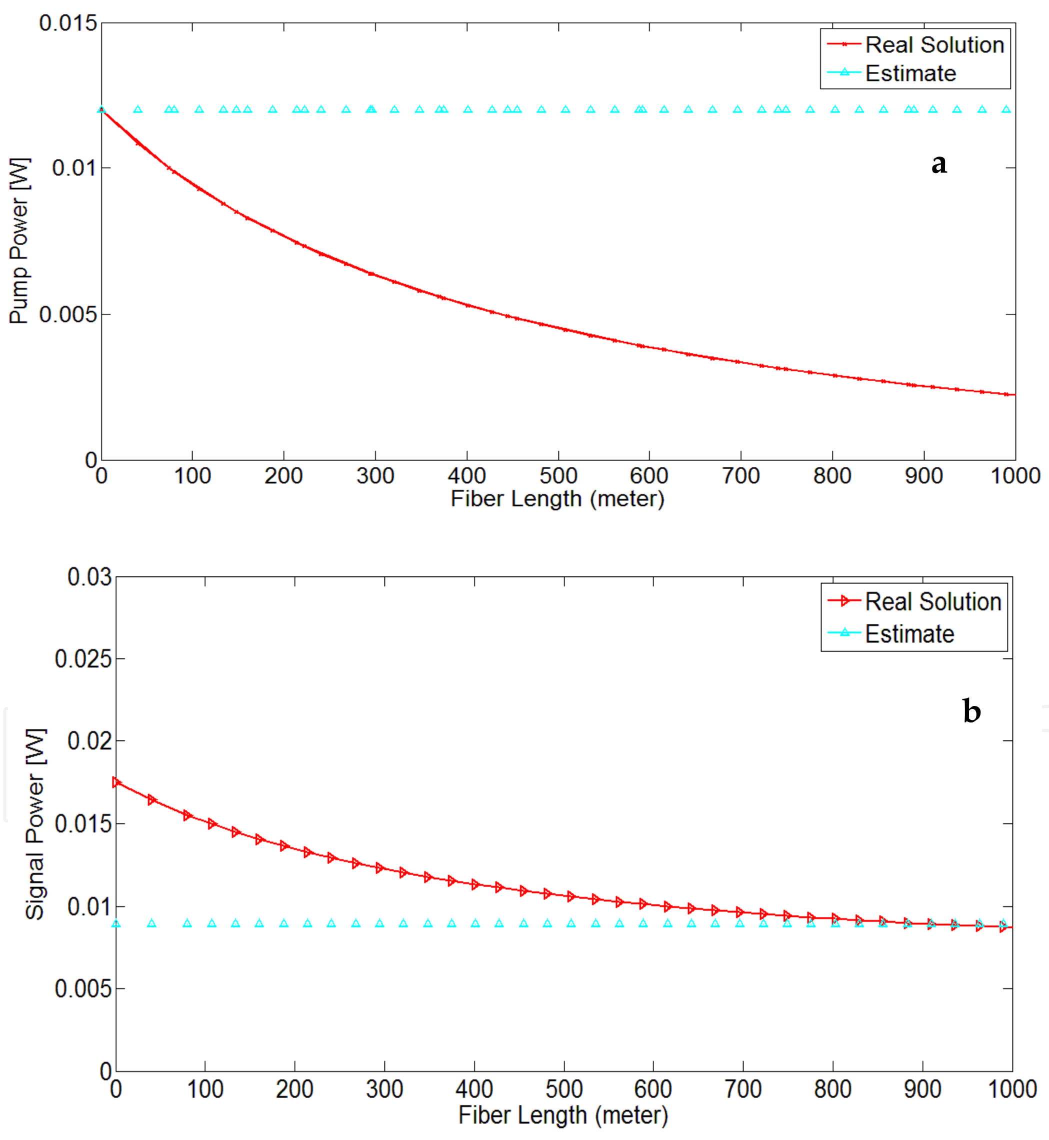

Fig. 1. Evolution of the guess values and real solution according to 1st guess

3.2 2nd Guess

If we guess that I_p and I_s is linearly changing as,

```
Ip ~ Ip0+A*z ;
Is ~ Is0+B*z ;
```

Exploiting with the MATLAB Symbolic Toolbox using the following script,

```
syms Is Ip Is0 Ip0 IsL gb L alpha A B z

 Ip = Ip0+A*z ;
 Is = Is0+B*z ;

 eqn1 = collect(diff(Ip,'z') + (gb)*Ip*Is+alpha*Ip);
 eqn2 = collect(diff(Is,'z') + (gb)*Ip*Is-alpha*Is);

 eqn3=collect(taylor(eqn1,1,0),z)
 eqn4=collect(taylor(eqn2,1,L),z)
```

The below output is produced;

```
eqn3 =   A + Ip0*alpha + (Ip0*Is0*gb)
eqn4 =   B - Is0*alpha - L*(B*alpha - (gb*(A*Is0 + B*Ip0))) +
         (Ip0*Is0*gb) + (A*B*L^2*gb)

Is0=IsL-B*L;
```

Here, there are two equations and two unknowns A and B. With the substitution of Is0, It can be solved simultaneously by the below script,

```
eqn=

solve(A + Ip0*alpha + (Ip0*(IsL-B*L)*gb),...
    B - (IsL-B*L)*alpha - L*(B*alpha - (gb*(A*(IsL-B*L) + B*Ip0)))+
    (Ip0*(IsL-B*L)*gb) + (A*B*L^2*gb),...
    'A','B')
eqn =

    A: [1x1 sym]
    B:  [1x1 sym]
```

Here A and B can be obtained as,

```
A=-(Ip0*alpha_s + Ip0^2*IsL*L*gb^2 + Ip0*IsL*gb -
    Ip0*IsL*L*alpha_s*gb)/(Ip0*IsL*L^2*gb^2);

B=(IsL*alpha_s + Ip0*IsL^2*L*gb^2 - Ip0*IsL*gb +
   Ip0*IsL*L*alpha_s*gb)/(Ip0*IsL*L^2*gb^2)

Ip ~ Ip0+A.*x ;
```

```
Is ~ (IsL-B.*L)+B.*x ;
```

The evolution of the 2nd guess values with the real solution is shown in Fig. 2

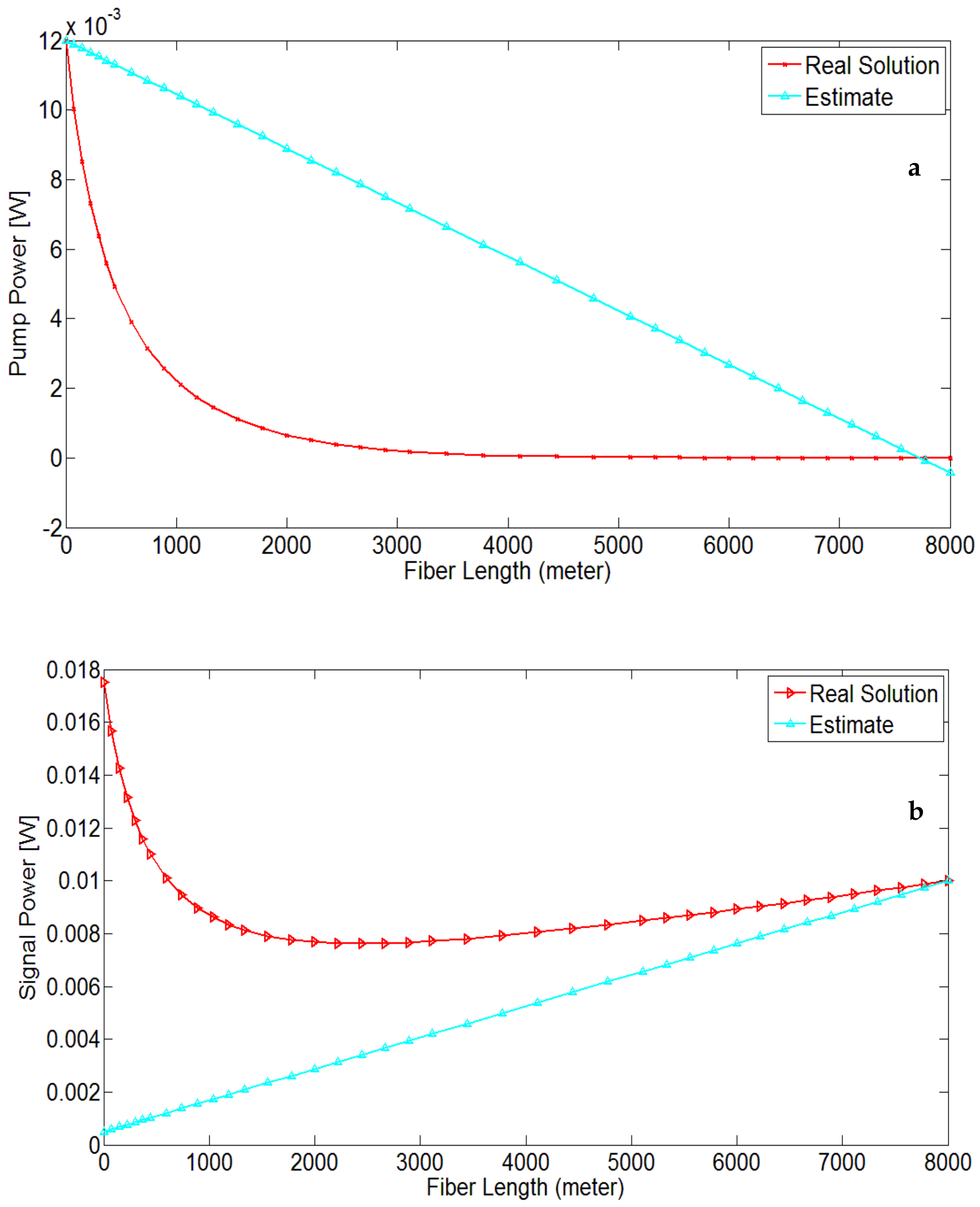

Fig. 2. Evolution of the guess values and real solution according to 2nd guess

3.3 3rd Guess

If it is guessed that I_p and I_s is exponentially changing as,

```
Ip ~ Ip0*exp(gamma1*z) ;
Is ~ Is0*exp(kappa1*z) ;
```

with using the following script,

```
syms Ip Is Is0 Ip0 gb z L IsL alpha gamma1 kappa1

Ip = Ip0*exp(gamma1*z) ;
Is = Is0*exp(kappa1*z) ;

eqn1 = collect(diff(Ip,'z') + (gb)*Ip*Is+alpha*Ip);
eqn2 = collect(diff(Is,'z') + (gb)*Ip*Is-alpha*Is);

eqn3=taylor(eqn1,1,0)
eqn4=taylor(eqn2,1,L)
```

The below output is produced;

```
eqn3 =Ip0*alpha + Ip0*gamma1 + (Ip0*Is0*gb)
eqn4 =Is0*kappa1*exp(L*kappa1) - Is0*alpha*exp(L*kappa1) +
       (Ip0*Is0*gb*exp(L*gamma1)*exp(L*kappa1))
```

`gamma1` and `kappa1` can be obtained as;

```
eqn5=solve(Ip0*alpha + Ip0*gamma1 + (Ip0*Is0*gb),'gamma1')
gamma1= -(alpha + Is0*gb)
```

using the same way;

```
kappa1 = (alpha - Ip0*gb*exp(L*gamma1));
```

Here, Is0 can be readily found by,

```
Is(z) = Is0*exp(kappa1*z) ;
For z=L
IsL= Is0*exp(kappa1*L);
Is0= IsL/exp(kappa1*L);
```

Therefore,

```
Ip ~ Ip0*exp(gamma1*z) ;
Is ~ IsL/exp(kappa1*L)*exp(kappa1*z) ;
```

The evolution of the 3rd guess values with the real solution is shown in Fig. 3

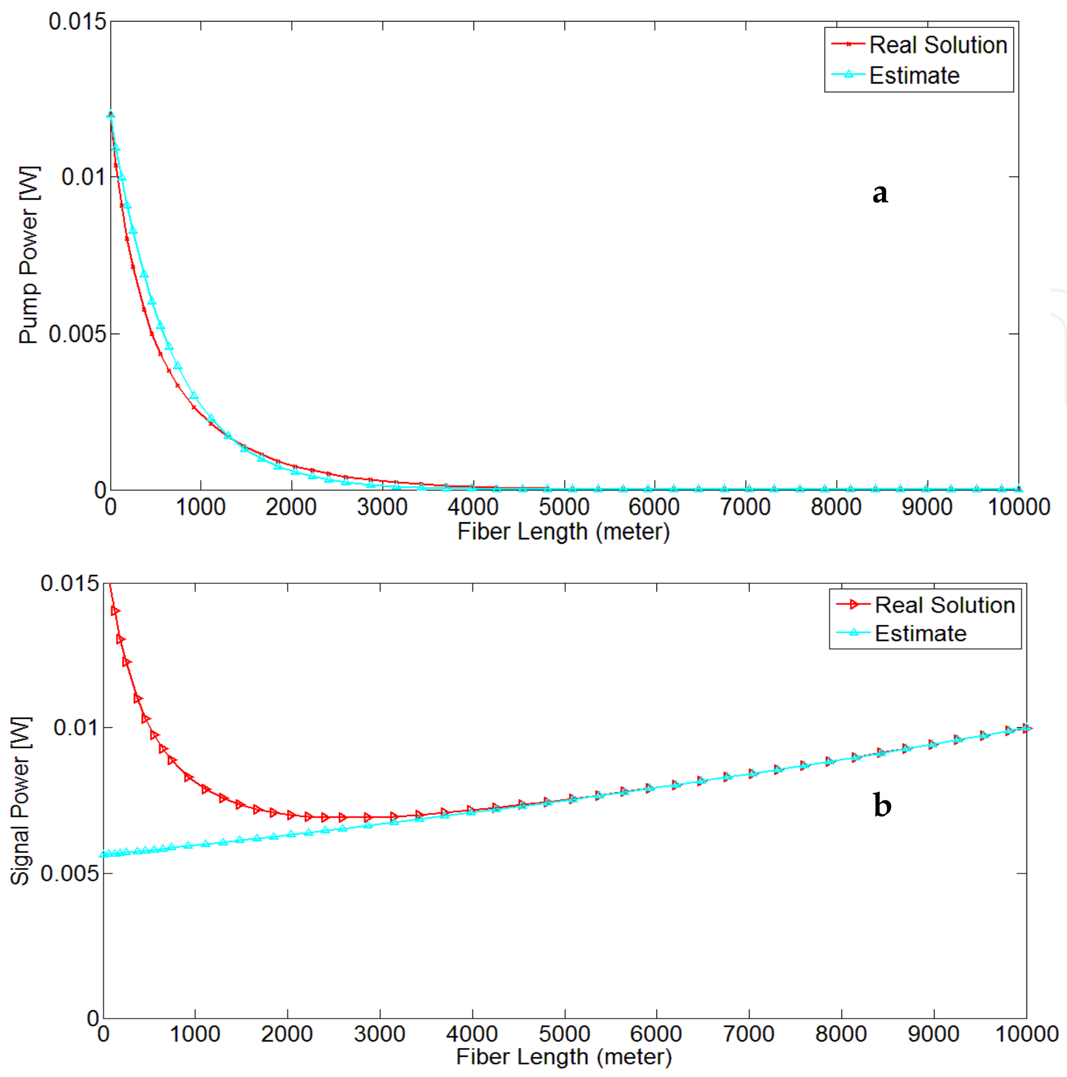

Fig. 3. Evolution of the guess values and real solution according to 3rd guess

3.4 4th Guess

Highly intuitive guess function may be derived the using the solution of lossless system, i.e., with eliminating the α coefficient in the Eq (3) and Eq. (4),

$$\frac{dIp}{dz} = -g_B I_P I_S \tag{5}$$

$$\frac{dIs}{dz} = -g_B I_P I_S \tag{6}$$

With neglecting the attenuation coefficient, the solution of the Eq.(5) and Eq.(6) is found as (Kobyakov *et al.*, 2006),

$$A(\zeta) = c_0[1 - (1 - c_0)\exp(-c_0 k\zeta)]^{-1} P_P(0) \tag{7}$$

$$B(\zeta) = c_0(1 - c_0)[\exp(c_0 k\zeta) - 1 + c_0]^{-1} P_P(0) \tag{8}$$

where,

$$c_0 \approx \frac{1}{k}\left\{\Lambda + \ln\left[\Lambda(1-\frac{\Lambda}{k})\right]\right\},\ k = \frac{g_B}{A_{eff}}P_P(0)L\ ,\ \ \zeta = \frac{z}{L} \tag{9}$$

where,

$$\Lambda = -\ln(\frac{P_{SL}}{P_{p0}}k) = -\ln[\frac{g_B}{A_{eff}}P_{SL}L]\ ,\ \ \frac{P_{SL}}{P_{p0}}k \ll 1 \tag{10}$$

Exploiting the solution of Eq.(7) and Eq. (8), general expression of $P_P(z)$ and $P_S(z)$ can be derived as,

$$P_P(z) = A(z)\cdot[1 - a\cdot(A\cdot x + B\cdot x^2)] \tag{11}$$

$$P_S(z) = B(z)\cdot[1 - a\cdot(C\cdot x + D\cdot x^2)] \tag{12}$$

If a→0, then $P_P(z) = A(z)$ and $P_S(z) = B(z)$

```
syms z k A B C D a AA BB c0 x

AA=c0/(1-(1-c0)*exp(-c0*k*z));
BB=c0*(1-c0)/(exp(c0*k*z)-1+c0);
w=(AA*(1-a*(A*x+B*x^2)));

u=(BB*(1-a*(C*x+D*x^2)));
eqn1 = collect(diff(w,1,'z') + (k*u*w)+a*w)
eqn2 = collect(diff(u,1,'z') + (k*u*w)-a*u)
The output produces,

eqn1 =
 …+ ((c0^2*k*(A*a + C*a)*(c0 - 1))/(((c0 - 1)/exp(c0*k*z) + 1)*(c0 +
exp(c0*k*z) - 1)) - (A*a^2*c0)/((c0 - 1)/exp(c0*k*z) + 1) -
(2*B*a*c0)/((c0 - 1)/exp(c0*k*z) + 1) - (A*a*c0^2*k*(c0 -
1))/(exp(c0*k*z)*((c0 - 1)/exp(c0*k*z) + 1)^2))*x …
+ (a*c0)/((c0 - 1)/exp(c0*k*z) + 1) - (A*a*c0)/((c0 - 1)/exp(c0*k*z)
+ 1) + (c0^2*k*(c0 - 1))/(exp(c0*k*z)*((c0 - 1)/exp(c0*k*z) + 1)^2)
- (c0^2*k*(c0 - 1))/(((c0 - 1)/exp(c0*k*z) + 1)*(c0 + exp(c0*k*z) -
1))

 eqn2 =
 … + ((2*D*a*c0*(c0 - 1))/(c0 + exp(c0*k*z) - 1) - (C*a^2*c0*(c0 -
1))/(c0 + exp(c0*k*z) - 1) + (c0^2*k*(A*a + C*a)*(c0 - 1))/(((c0 -
1)/exp(c0*k*z) + 1)*(c0 + exp(c0*k*z) - 1)) -
(C*a*c0^2*k*exp(c0*k*z)*(c0 - 1))/(c0 + exp(c0*k*z) - 1)^2)*x …
+ (a*c0*(c0 - 1))/(c0 + exp(c0*k*z) - 1) + (C*a*c0*(c0 - 1))/(c0 +
exp(c0*k*z) - 1) - (c0^2*k*(c0 - 1))/(((c0 - 1)/exp(c0*k*z) + 1)*(c0
+ exp(c0*k*z) - 1)) + (c0^2*k*exp(c0*k*z)*(c0 - 1))/(c0 +
exp(c0*k*z) - 1)^2
```

We are interested in the behavior as z → 0 and so, the higher the power of x, the less effect it has in these expansions. Our goal is to satisfy the equations as well as possible, so we want to choose coefficients that make as many successive terms zero as possible, starting with the lowest power. To eliminate the constant terms, we see from the expansions that we must take

```
A= 1; C=-1;
B= -(a - a/exp(c0*k*z) + (a*c0)/exp(c0*k*z) - (c0*k)/exp(c0*k*z) +
    (c0^2*k)/exp(c0*k*z))/((2*c0)/exp(c0*k*z) - 2/exp(c0*k*z) + 2)
D=  (a - a/exp(c0*k*z) + (a*c0)/exp(c0*k*z) - (c0*k)/exp(c0*k*z) +
    (c0^2*k)/exp(c0*k*z))/((2*c0)/exp(c0*k*z) - 2/exp(c0*k*z) + 2)
```

The evolution of the 4rd guess values with the real solution is shown in Fig. 4

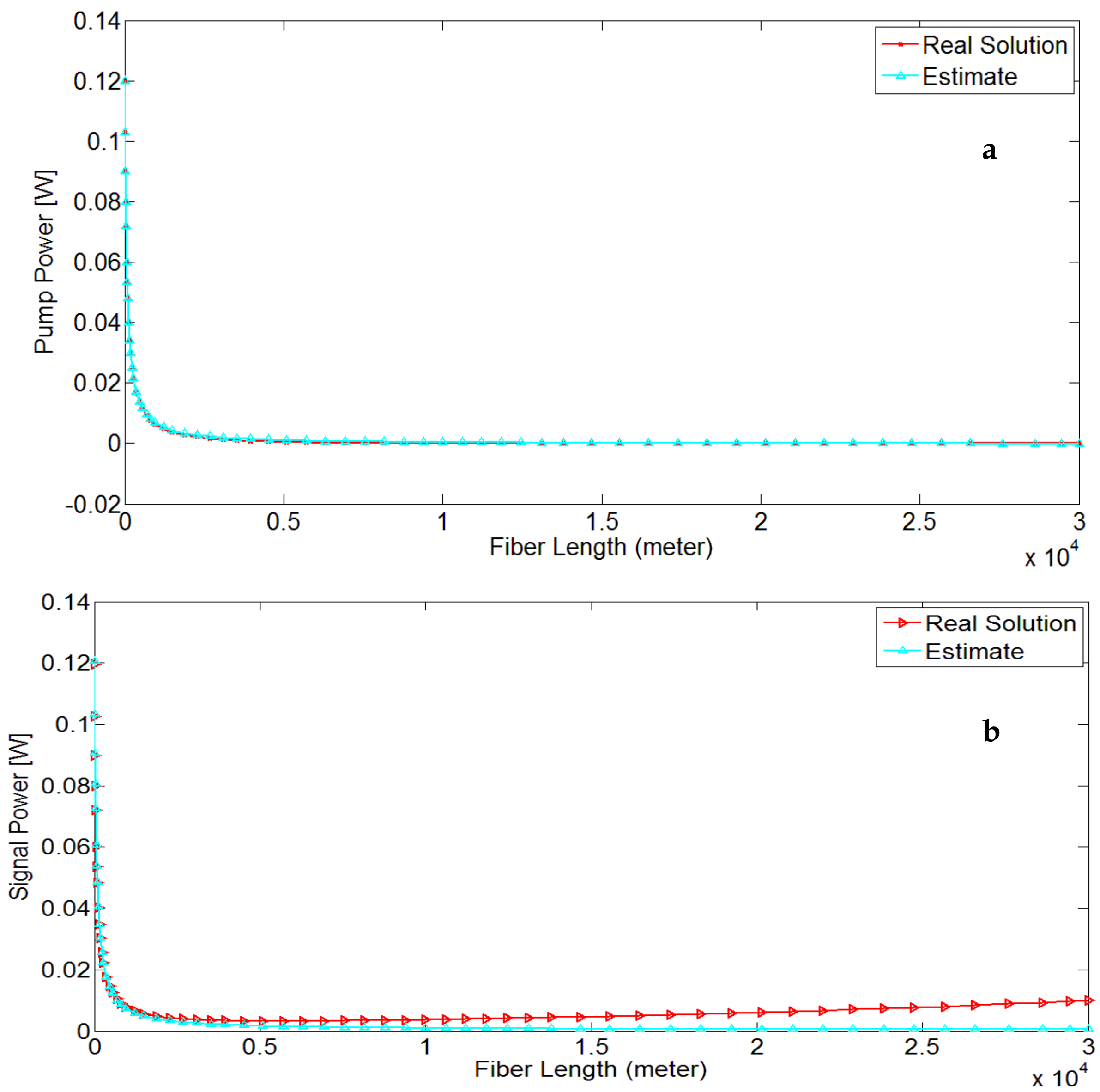

Fig. 4. Evolution of the guess values and real solution according to 4th guess

Guess	Length(mt) /mesh	Convergence length/mesh	Computation Time (with bvp4c) at 8000 meter
1st Guess - values (Ip0,IsL)	8000/ 40	8000/40	~1.18 sec
2nd Guess - functions (linear)	8000/ 40	9000/453	~1.13 sec
3rd Guess -functions (exponential)	8000/ 40	15000/56	~1.13 sec
4th Guess functions (modified exponential)	8000/ 35	30000/61	~1.0 sec

Table 1. Guess values/functions versus convergence length/mesh and run time.

As can be seen from Table 1 and Fig.4 the best estimation is the 4th guess. Because its' convergence length (30000) is more than the others (15000, 9000, 8000, respectively). The performance of the 2nd guess is approximately same as the first one. Because it hardly converge the solution using 453 points at 9000 meter. However, its performance is same as the first one with 40 points at 8000 meter.

4. Continuation

The method of continuation exploits the fact that generally the solution of one BVP is a good guess for the solution of another with slightly different parameters. If you have difficulty in finding a guess for the solution that is good enough to achieve convergence for the interval of interest, it is frequently the case that the problem is easier to solve on a shorter interval. The idea is then to solve a sequence of BVPs with the solution on one interval being used as a guess for the problem posed on a longer interval. Of course, this technique cannot be used to extend the interval *ad infinitum*; no matter how good the guess is, eventually the solver will not be able to distinguish the different kinds of solutions (Shampine *et al.*, 2003).
For the range of interested lengths bigger than the convergence lengths, the continuation process can be applied. Some types of snippets of continuation are illustrated below,

```
=================#1#=========================
options = bvpset('FJacobian',@sampleJac,...
                 'BCJacobian',@sampleBCJac,...
                 'Vectorized','on');

sol = bvpinit([-1 -0.5 0 0.5 1],[1 0]);

c = 0.1;
for i=2:4
    c = c/10;
    sol = bvp4c(@sampleODE,@sampleBC,sol,options);
end
=================#2#========================

infinity = 3;
maxinfinity = 6;

solinit = bvpinit(linspace(0,infinity,5),[0 0 1]);
sol = bvp4c(@fsode,@fsbc,solinit);
```

```
eta = sol.x;
f = sol.y;

for Bnew = infinity+1:maxinfinity

  solinit = bvpinit(sol,[0 Bnew]); % Extend solution to Bnew.
  sol = bvp4c(@fsode,@fsbc,solinit);
  eta = sol.x;
  f = sol.y;
=================#3#======================

L=30000;
for i = 1:1000:L
D = 5*i/1000; d = 1/D;
if i == 1
solinit = bvpinit(linspace(1,i,10),@guess);
else
solinit = bvpinit(sol,[d,D]);
end

sol = bvp4c(@odes,@bcs,solinit);
end
=================#4#======================
function Boundary_value_increment
global a b

a= XL;    %[a
b= XR;    %   b]
sol = bvpinit(linspace(0,L,2),[Boundary values for each pump and
signal power]);

options = bvpset('Stats','on','RelTol',1e-5);

for k=1:Desired_Power
b=b+k;
sol = bvp4c(@bvpode,@bvpbc,sol,options);
end
```

If the interested length is bigger than the convergence length then continuation can be applied with the **bvpxtend** function. In the recent version of MATLAB, bvpinit function is simplified to a new function **bvpxtend**. Besides offering new possibilities, this function permits extension of length of interval to only one end point at a time (Kierzenka & Shampine, 2008). Briefly,

`solinit = bvpxtend(sol,xnew,ynew)` uses solution sol computed on [a,b] to form a solution guess for the interval extended to xnew. The extension point xnew must be outside the interval [a,b], but on either side. The vector ynew provides an initial guess for the solution at xnew.

For example, if it is assumed that the convergence and interested lengths are 15 and 30 km, respectively, the continuation can be applied via the below codes (Gokhan & Yilmaz, 2011a),

```
=================#5#======================
L= 15000;
Interested=30000;
solinit = bvpinit(linspace(0,L,2),[Guess expression running with 15
km]);
sol = bvp4c(@bvpode,@bvpbc,solinit,options);
.
for Bnew=L:2000:Interested
    solinit=bvpxtend(sol,Bnew);
    sol = bvp4c(@bvpode,@bvpbc,solinit,options);
end
```

In the above codes, 2000 is the step size which is the modest increment range. In case of bigger step size, the solver may fail but the computation time reduces. On the other hand, if this increment is kept little, it takes more time to reach the end of the computation. Therefore, selecting the step size is important factor for the efficiency of the computation for continuation. One advantage of using bvpxtend function is the reduced computation time. Because, bvp solvers try to use mesh points as few as possible, the step size is incremented automatically depending on the previous mesh points. In bvpxtend, after obtaining convergence for the mesh, the codes adapt the mesh so as to obtain an accurate numerical solution with a modest number of mesh points. Here it must be emphasized that, for BVPs the most difficult part is providing an initial approximation to the solution.

5. Effect of the step size on the run time

Using the snippets 5 and the 1st Guess of Table 1, the performance of the continuation over step size is illustrated in Table 2.

Convergence Length (km)	Computation Length (km)	Linspace (0,L,**N**) Optimal *N*	Step Size (mt.)	Computation Time at Convergence Length (with bvp4c)	Total Computation time	Mesh number (between) (with bvp4c)
40	50	10	50	~3.2 sec	~110.0 sec	482-541
40	50	10	100	~3.2 sec	~60.0 sec	482-541
40	50	10	200	~3.2 sec	~32.8 sec	482-553
40	50	10	300	~3.2 sec	~25.3 sec	482-560
40	50	10	400	~3.2 sec	~23.6 sec	482-663
40	50	10	500	~3.2 sec	~44.5 sec	482-1984
40	50	10	600	Computation fails.		

Table 2. The performance of continuation method versus step size

If the step size is increased over the 600 meter, the computation fails with the below warning message;

```
Warning: Unable to meet the tolerance without using more than 5000
mesh points.
```

```
The last mesh of 4106 points and the solution are available in the
output  argument.

The maximum residual is 0.00018931, while requested accuracy is 1e-
005.
```

As can be seen on Table 2, for some step size over the modest increment, computation blows up (i.e. 600 m). When the step size is between 50 and 400, the number of used mesh is slightly different from each other. However, when it is 500 m, abruptly increase in the number of used mesh is a sign of lack of confidence.

In Fig.5, it can be seen that the distance between some mesh points, especially near the boundaries are denser than the others. This is because the solver tries to control the residual of the interpolating polynomial: r(x) = S′(x) – f(x,S(x)). The behavior of this residual depends on the behavior of some high derivatives of the solution (that the solver does not have access to). In the solver, the residual is estimated at each mesh subinterval, and additional mesh points are introduced if the estimate is bigger than the tolerance.

The mesh selection algorithm is 'localized', which means that if the residual is just above the tolerance, the interval will be divided into two (and likely on each of those subintervals, the residual will be much smaller than the tolerance). Also, the algorithm for removing mesh points is quite conservative, so there could be regions where the residual will be quite a bit smaller that the tolerance (i.e., the mesh could be quite a bit denser than necessary)(Kierzenka & Shampine, 2001).

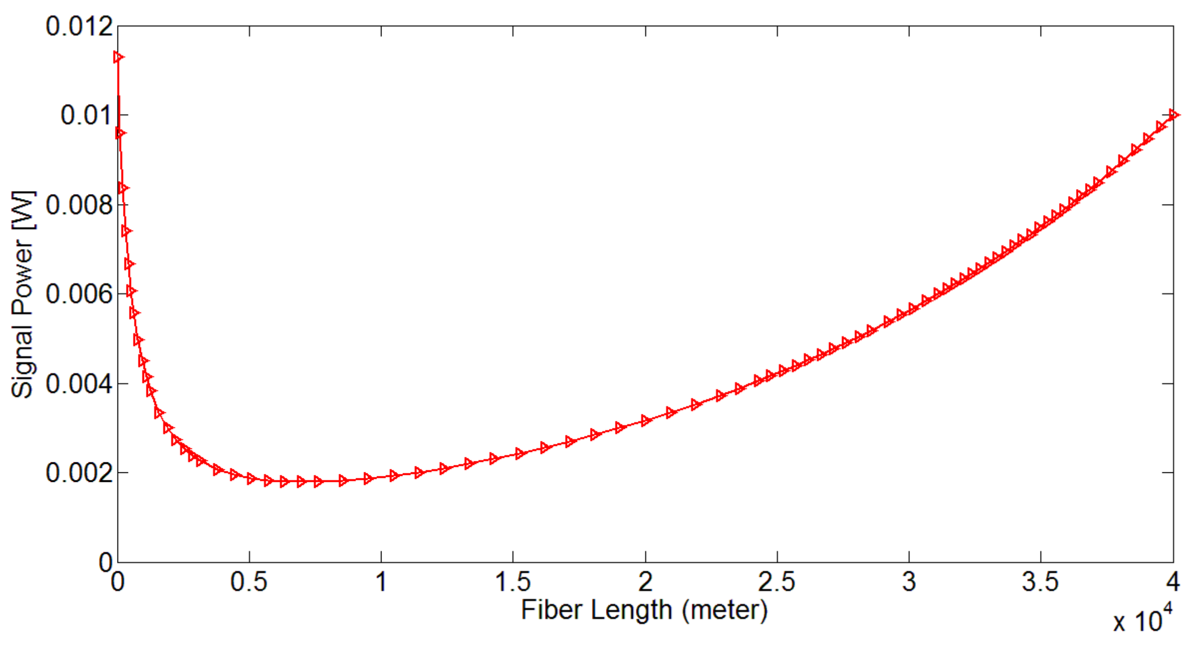

Fig. 5. Evolution of mesh density along the computation

5.1 Effect of the constructing an initial guess with bvpinit function on the run time

As can be seen from Table 3, when constructing an initial guess using `bvpinit(linspace(0,L,N)`, starting with a mesh of 5-10 points could often result in a more efficient run. It must be pointed out that with adaptive collocation solvers, using that many points (N=50,100) with a poor guess could often be counterproductive. In the case of N=100, the solver still achieved the sufficient accuracy as it is between 2 and 10.

Linspace (0,L,N) Optimal *N*	Computation Time (with bvp4c)	Mesh number (with bvp4c)	Maximum Residual
2-4	Singular Jacobian encountered		
5	~3.20 sec	498	9.791e-006
6	~2.85 sec	475	9.967e-006
7	~3.10 sec	483	9.983e-006
8	~2.75 sec	487	9.913e-006
9	~2.87 sec	464	9.848e-006
10	~2.68 sec	469	9.817e-006
50	~2.81 sec	478	9.938e-006
100	~3.10 sec	485	9.959e-006

Table 3. Performance of equally spaced N points for the mesh of a guess

6. Speeding up the run time of BVP solvers

The first technique which is used to reduce run time is vectorizing the evaluation of the differential equations. Vectorization is a valuable tool for speeding up MATLAB programs and this greatly reduces the run time (Shampine *et al.*, 2003). By vectorization, the function $f(x,y)$ is coded so that when given a vector $x=[x_1,x_2,...]$ and a corresponding array of column vectors $y=[y_1,y_2,...]$, it returns an array of column vectors $[f(x_1,y_1),f(x_2,y_2),...])$. By default, bvp4c and bvp4c approximate a Jacobian using finite differences. The evaluation of the ODEs is vectorized by changing the vectors to arrays and changing the multiplication to an array multiplication. It can be coded by changing scalar quantities like y(1) into arrays like y(1,:) and changing from scalar operations to array operations by replacing * and ^ with .* and .^, respectively. When vectorizing the ODEs, the solver must be informed about the presence of vectorization by means of the option `'Vectorized','on'`.

```
options = bvpset('Stats','on','RelTol',1e-3,'Vectorized','on');
```

The second technique is that of supplying analytical partial derivatives or to supply a function for evaluating the `Jacobian` matrix. This is because, in general, BVPs are solved much faster with analytical partial derivatives. However, this is not an easy task since it is too much trouble and inconvenient, although MATLAB Symbolic Toolbox can be exploited when obtaining analytical Jacobians. The third technique is to supply analytical partial derivatives for the boundary conditions. However, it has less effect on the computation time compared with supplying analytical Jacobians and vectorization. The solver permits the user to supply as much information as possible. It must be emphasized that supplying more

information for the solvers results in a shorter computation run time (Gokhan & Yilmaz, 2011b).
The set of equations (3) and (4) is vectorized by changing the vectors to arrays and changing the multiplication to an array multiplication as seen below,

```
function dydx = bvpodevectorized(x,y)
global alpha_s gb K

dydx =  [ -gb *y(1,:).*y(2,:)-alpha_s*y(1,:)
          -gb *y(1,:).*y(2,:)+alpha_s*y(2,:) ];
```

Using vectorized ODEs with N=8, the performance of the vectorization is illustrated in Table 4.

Length (mt)	Computation Time with scalar evaluation (with bvp4c)	Computation Time with vectorization (with bvp4c)
5000	~1.83 sec	~1.79 sec
10000	~2.26 sec	~2.15 sec
20000	~2.60 sec	~2.45 sec
30000	~2.70 sec	~2.58 sec
40000	~3.00 sec	~2.85 sec

Table 4. Comparison of the computation time with scalar evaluation and with vectorization

The code bvp4c permits you to supply analytical partial derivatives for either the ODEs or the boundary conditions or both. It is far more important to provide partial derivatives for the ODEs than the boundary conditions. The solver is informed that a function is written for evaluating ∂f/∂y by providing its handle as the value of the FJacobian option. Similarly, the solver can be informed of a function for evaluating analytical partial derivatives of the boundary conditions with the option BCJacobian (Shampine *et al.*, 2003). FJacobian and BCJacobian can be introduced as with the below codes,

```
%options = bvpset(options,'FJacobian',@odeJac);
%options = bvpset(options,'BCJacobian',@bcJac);
```

The MATLAB Symbolic Toolbox has a function jacobian that can be very helpful when working out partial derivatives for complicated functions. Its use is illustrated with a script for the partial derivatives of the ODEs of this example.

```
syms res y1 y2 y1 y2 alpha_s gb

res = [ -gb*y1*y2 - alpha_s*y1
        -gb*y1*y2 + alpha_s*y2];
dFdy = jacobian(res,[y1; y2])

dFdy =
[ - alpha_s - gb*y2,        -gb*y1]
[          -gb*y2, alpha_s - gb*y1]
```

The performance of the insertion of analytical partial derivatives and vectorization, and both are compared in Table 5. As can be seen from Table 5, with vectorization and analytical partial derivatives, the computation time is reduced approximately 15 %. The calculations are simulated using the MATLAB 7.9 (R2009b) on an Intel Core i5 2.53 GHz laptop computer.

Length (mt)	Computation Time with scalar evaluation (with bvp4c)	Computation Time with only vectorization (with bvp4c)	Computation Time with only analytical partial derivatives (with bvp4c)	Computation Time with both vectorization & analytical partial derivatives (with bvp4c)
5000	~1.83 sec	~1.79 sec	~1.69 sec	~1.59 sec
10000	~2.26 sec	~2.15 sec	~2.05 sec	~1.80 sec
20000	~2.60 sec	~2.45 sec	~2.26sec	~1.96sec
30000	~2.70 sec	~2.58 sec	~2.40 sec	~2.04 sec
40000	~3.00 sec	~2.85 sec	~2.60 sec	~2.18 sec

Table 5. Comparison of the computation time of bvp4c with vectorization, with analytical partial derivatives and with both

In Table 6, the performance of the bvp5c is illustrated. In terms of scalar evaluation, the performance of bvp5c solver is better than bvpc4 and it is evident as the length is increased. This improvement is about 47 % at 40 km. As in the case of bvp4c, the performance can be increased with vectorization and analytical partial derivatives or with both. Compared with the scalar evaluation, only with vectorization and only with analytical partial derivatives this improvement is 8% and 13 %, respectively. If both is used this improvement is about 24 %.

Length (mt)	Computation Time with scalar evaluation (with bvp5c)	Computation Time with only vectorization (with bvp5c)	Computation Time with only analytical partial derivatives (with bvp5c)	Computation Time with both vectorization & analytical partial derivatives (with bvp5c)
5000	~1.32 sec	~1.32 sec	~1.30 sec	~1.23 sec
10000	~1.38 sec	~1.38 sec	~1.35 sec	~1.27 sec
20000	~1.44 sec	~1.42 sec	~1.38sec	~1.30sec
30000	~1.55 sec	~1.50 sec	~1.43 sec	~1.34 sec
40000	~1.60 sec	~1.52 sec	~1.47 sec	~1.36 sec

Table 6. Comparison of the computation time of bvp5c with vectorization, with analytical partial derivatives and with both

If the comparison among two solvers has made, it could be expressed that bvp5c "looks" exactly like bvp4c. However, bvp5c controls scaled residual and true error but bvp4c controls residual in a different norm. And, bvp5c is more efficient at stringent tolerances.

Also, bvp5c solves singular BVPs, but not multipoint BVPs. Moreover, bvp5c handles unknown parameters in a different way. And also, bvp5c was added to MATLAB at R2007b (Shampine, 2008)

7. Conclusion

Within the chapter, in order to analyze the effect of guess functions on the computation time, four guess functions are derived. For better understanding, while exploiting physical origin, guess functions are derived with the help of MATLAB Symbolic toolbox. Continuation method with functional snippets is presented to cope with poor guesses. Effect of the step size and bvpinit function on the computation time is analyzed. Speeding up the run time with vectorization and analytical partial derivatives are discussed and the comparison between bvp4c and bvp5c has been made.
As a conclusion, it is illustrated that, intuitive guess values/functions improves the convergence length, leads the computation with fewer mesh points and consequently lessens the computation time. On the other hand, regarding with the continuation, adjusting the step size is important for the reduction of run time. It is illustrated that, over the modest step size, the solver fails and below the optimum step size, the computation time is increased. Moreover, it is showed that when constructing an initial guess using bvpinit(linspace(0,L,N), starting with a mesh of 5-10 points could often result in a more efficient run. Another outcome of the chapter is the illustration of the efficiency of the vectorization and analytical partial derivatives. It is showed specifically with an example and with bvp4c that, with the application of vectorization and analytical partial derivatives, the computation time is reduced approximately 15 %. The performance of the bvp4c and bvp5c is also compared. In terms of scalar evaluation, the performance of bvp5c solver is better than bvpc4 and it is evident as the computation length is increased. Compared with the scalar evaluation, for the bvp5c, only with vectorization and only with analytical partial derivatives this improvement is 8% and 13 % respectively. If both is used this improvement is about 24 %.

8. Acknowledgments

The author gratefully acknowledges the financial support from the head of scientific research projects commission of Gazikent University.

9. References

Kierzenka J. & Shampine Lawrence F. (2001), *A BVP Solver Based on Residual Control and the MATLAB PSE*, ACM TOMS, vol. 27, No. 3, pp. 299-316.
Hale N.P., (2006), *A Sixth-Order Extension to the MATLAB bvp4c Software of J. Kierzenka and L. Shampine*, Msc Thesis. Accessed 06.04.2011, Available from: < http://people.maths.ox.ac.uk/hale/files/hale_mastersthesis.pdf >
MATLAB Documentation, *Continuation*, Accessed 06 April 2011, Available from: < http://www.mathworks.com/help/techdoc/math/f1-713877.html>
Shampine, L.F., Gladwell, I. & Thompson, S. (2003), *Solving ODEs with MATLAB*, 1st ed., Cambridge University Press, ISBN, 978-0-521-82404-4, New York
Agrawal G. P. (2001), *Nonlinear Fiber Optics*, 3rd ed. Academic, Chap. 9.

Kobyakov A., Darmanyan S., Sauer M. & Chowdhury D (2006), *High-gain Brillouin amplification: an analytical Approach*, Opt.Lett. 31 1960

Kierzenka, J. & Shampine, L.F. (2008), *BVP solver that controls residual and error*, Journal of Numerical Analysis, Industrial and Applied Mathematics (JNAIAM), Vol. 3 Nos 1/2, pp. 27-41, available at: www.jnaiam.org/downloads.php?did=44 (Accessed 06 April 2011).

Gokhan F. S., & Yilmaz G, (2011a), *Solution of Raman fiber amplifier equations using MATLAB BVP solvers*, COMPEL: The International Journal for Computation and Mathematics in Electrical and Electronic Engineering, Vol. 30 Iss: 2, pp.398 – 411

Gokhan F. S., & Yilmaz G, (2011b), *Novel Guess Functions for Efficient Analysis of Raman Fiber Amplifiers*, COMPEL: The International Journal for Computation and Mathematics in Electrical and Electronic Engineering, *accepted in 2010*

Shampine L.F, (2008). *Control of Residual and Error*, In: CAIMS-Canadian Mathematical Society, Accessed 06 April 2011,
Available from: < http://faculty.smu.edu/shampine/Montreal.pdf>