# SF1545 - Lab 1

**Applying Numerical Methods to Ramsey's "A Mathematical Theory Of Saving"**

Johan Björklund, u1480nqu, johbjo09@kth.se

November 17, 2015

## 1 Minimize a series

$$\min_{\alpha_n \in \mathbb{R}} \sum_{n=0}^{\infty} \left[ \frac{\tilde{U}\left(\tilde{\alpha}_n\right)}{f\left(X_n\right) - \tilde{\alpha}_n} \Delta X \right]$$

To minimize the series, we need to take the partial derivate, and find critical points:

$$\frac{\partial}{\partial \tilde{\alpha}_n} \left( \sum_{n=0}^{\infty} \left[ \frac{\tilde{U}\left(\tilde{\alpha}_n\right)}{f\left(X_n\right) - \tilde{\alpha}_n} \Delta X \right] \right) = 0$$

Expanding, we see:

$$\frac{\partial}{\partial \tilde{\alpha}_n} \left( \frac{\tilde{U}\left(\tilde{\alpha}_1\right)}{f\left(X_1\right) - \tilde{\alpha}_1} + \frac{\tilde{U}\left(\tilde{\alpha}_2\right)}{f\left(X_2\right) - \tilde{\alpha}_2} + \frac{\tilde{U}\left(\tilde{\alpha}_3\right)}{f\left(X_3\right) - \tilde{\alpha}_3} + \ldots + \frac{\tilde{U}\left(\tilde{\alpha}_n\right)}{f\left(X_n\right) - \tilde{\alpha}_n} \right) = 0$$

As the series expand, only one term in the series is dependent on the corresponding index of alpha. Taking the derivative with respect to a constant yields zero, so:

$$\frac{\partial}{\partial \tilde{\alpha}_i} \left( \frac{\tilde{U}\left(\tilde{\alpha}_j\right)}{f\left(X_j\right) - \tilde{\alpha}_j} \right) = 0$$

The derivative of all other terms is zero. Therefore, we may say:

$$\frac{\partial}{\partial \tilde{\alpha}_n} \left( \sum_{n=0}^{\infty} \left[ \frac{\tilde{U}\left(\tilde{\alpha}_n\right)}{f\left(X_n\right) - \tilde{\alpha}_n} \Delta X \right] \right) = \sum_{n=0}^{\infty} \left[ \frac{\partial}{\partial \tilde{\alpha}_n} \left( \frac{\tilde{U}\left(\tilde{\alpha}_n\right)}{f\left(X_n\right) - \tilde{\alpha}_n} \right) \Delta X \right]$$

To find the inner derivative, we apply the quotient rule:

$$\frac{\partial}{\partial \tilde{\alpha}_n} \left( \frac{\tilde{U}\left(\tilde{\alpha}_n\right)}{f\left(X_n\right) - \tilde{\alpha}_n} \right) = \frac{\tilde{U}'\left(\tilde{\alpha}_n\right)\left(f\left(X_n\right) - \tilde{\alpha}_n\right) + \tilde{U}\left(\tilde{\alpha}_n\right)}{\left(f\left(X_n\right) - \tilde{\alpha}_n\right)^2}$$

$$\frac{\tilde{U}'\left(\tilde{\alpha}_n\right)\left(f\left(X_n\right) - \tilde{\alpha}_n\right) + \tilde{U}\left(\tilde{\alpha}_n\right)}{\left(f\left(X_n\right) - \tilde{\alpha}_n\right)^2} = \frac{\tilde{U}'\left(\tilde{\alpha}_n\right)}{f\left(X_n\right) - \tilde{\alpha}_n} + \frac{\tilde{U}\left(\tilde{\alpha}_n\right)}{\left(f\left(X_n\right) - \tilde{\alpha}_n\right)^2} = 0$$

$$\frac{\tilde{U}'(\tilde{\alpha}_n)}{f(X_n)-\tilde{\alpha}_n} = \frac{-\tilde{U}(\tilde{\alpha}_n)}{(f(X_n)-\tilde{\alpha}_n)^2} \implies \tilde{U}'\left(\tilde{\alpha}_n\right) = \frac{-\tilde{U}(\tilde{\alpha}_n)}{f(X_n)-\tilde{\alpha}_n}$$

$$f\left(X_n\right) - \tilde{\alpha}_n = \frac{-\tilde{U}\left(\tilde{\alpha}_n\right)}{\tilde{U}'\left(\tilde{\alpha}_n\right)}$$

## 2 Fitting functions with least-squares

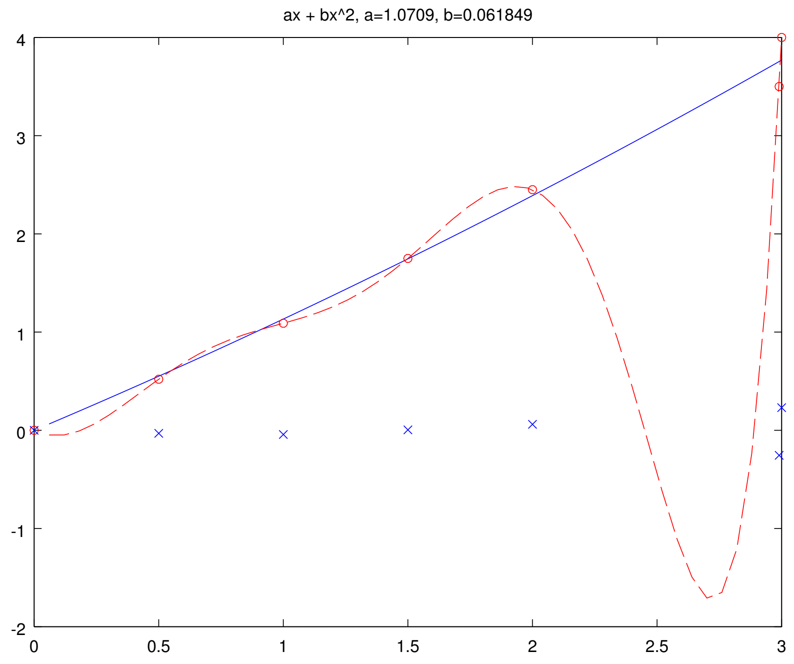
The given problem is to fit a second degree polynomial to certain data points using the method of least squares. The method finds the linear coefficients in a system of equations by factoring the coefficients of an equation into a vector, and arranging into a matrix equation. The system is so-called over-determined, as the number of equations exceed the number of coeffecients. Solving this system using matlab's \-operator is equivalent of minimizing the square norm of the residual $\left\|Ac - x\right\|$ (hence "least squares"), by setting the derivative with respect to the coefficients to zero, and then solving for $c$:

$$\frac{d}{dc}\left\|Ac - x\right\| = 2A^T\left(Ac - x\right) = 0 \quad \Rightarrow \quad A^T\left(Ac - x\right) = 0 \quad \Rightarrow \quad A^TAc = A^Tx$$

### Notes on the code

In the function `computeLeastSquaresFit` we make us of variable argument lists (`varargin` in matlab), which allows us to call the function with linear combinations of any functions. It returns two arrays; first the coefficients corresponding to the ordered input functions, and the residual. This is necessary for the function to be *testable* which is fundamental.

### Results



Firstly, we see that polynomials are unsuitable to model this data. We see from the residual plot that the two last points are the most important. The small value of b makes the function almost linear, and the least squared distance from these points is "between them", rather than crossing them. Sine the polynomial is always continous and "smooth", two successive data-points whose vector/line is almost orthogonal to the other data points will force the polynomial into "wide swings". Runges phenomenon is related, but also points out the wide swings outside of the data boundaries. Without the two last data points, the polynomial could have been a good fit, within the boundaries of the data.

## 3 Fitting non-linear functions

Since the non-linear functions are not linear combinations, we can not directly use linear systems to solve for the constants. One way to transform exponential functions to linear systems is to use logarithms. As a solution, this places narrow restrictions on the functions and the code is not reusable.

### Gauss-Newton method

Our solution is to use Gauss-Newton, which achieves linearisation through a jacobian matrix. The advantage over logarithms is that our code can be used for any function for which we can find a jacobian. If we can not find a jacobian analytically, we could use approximations. This allows us to test several candidate functions with minimal manual calculation.

Gauss-Newton is a type of fixed point iterative method. By linearisation, each equation in the system corresponds geometrically to a plane (though number of dimensions may vary.) At each iteration, we exploit the linearisation to solve the system for the coefficients using matlab's \-operator, and update the coefficients with the residual, until the residual is smaller than a set tolerance.

### Notes on the code

We make use of matlab's cell array as a container to place functions, the partial derivatives, so we can factor the computation of jacobians. Watch for `{}-brackets` in the code.

Rearrange:

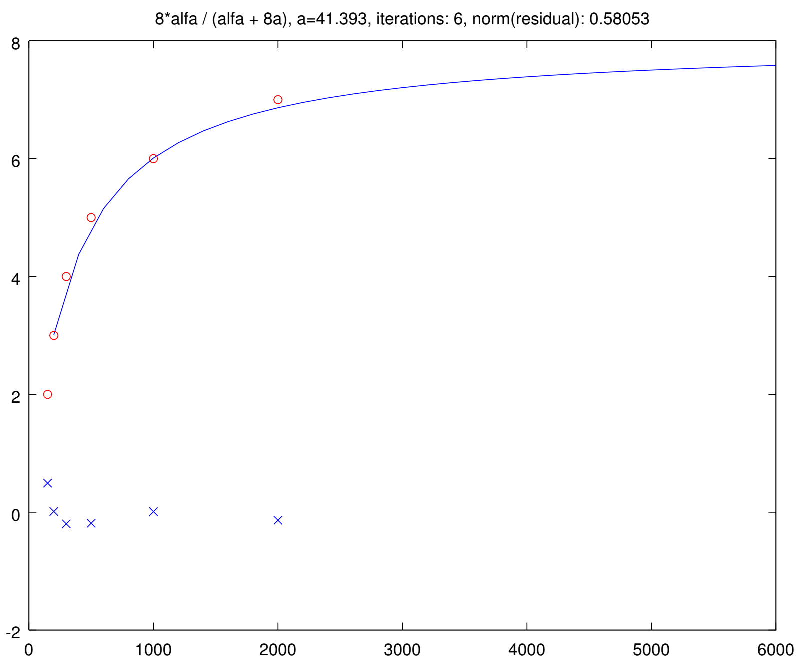
$$\frac{1}{\tilde{U}\left(\alpha\right)} = \frac{1}{8} + \frac{a}{\alpha} \Longrightarrow \tilde{U}(\alpha) = \frac{8\alpha}{\alpha + 8a}$$

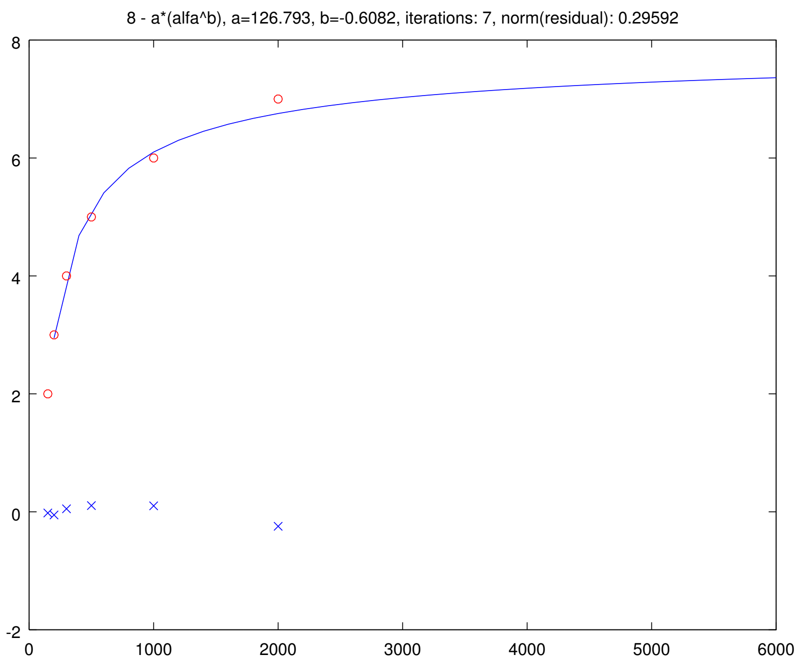
Find partial derivatives:

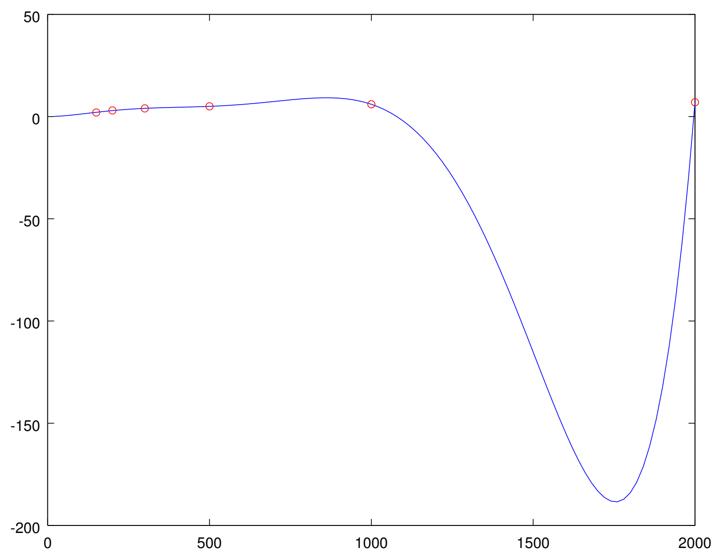
$$\frac{\partial \tilde{U}}{\partial a} = 8\alpha \frac{-8}{\left(\alpha + 8a\right)^2}$$

### Results

Tolerance is set to $10^{-6}$. Starting values are found by experimentation. Blue crosses are residuals.


8*alfa / (alfa + 8a), a=41.393, iterations: 6, norm(residual): 0.58053
8
6
4
2
0
-2
0
1000
2000
3000
4000
5000
6000


8 - a*(alfa^b), a=126.793, b=-0.6082, iterations: 7, norm(residual): 0.29592
8
6
4
2
0
-2
0
1000
2000
3000
4000
5000
6000



## Analysis

The exponential function $U(\tilde{\alpha}) = 8 - a\tilde{\alpha}^b$ is sensitive to variations in the constant $b$ and very sensitive to starting values, sometimes diverging with positive values for $b$. In this example the residual is smaller, but seems to grow. If accuracy close to zero is important, the exponential function is favourable.

For the polynomial, the residual is exactly zero (from the definition of polynomial fitting) but it is a bad choice, crossing into negative asymptically diverging to infinity.

The quotient function $\frac{8\alpha}{\alpha+8a}$ converges faster than the exponential, is less sensitive to small varitions, and is less sensitive to starting values, and might be faster to compute. All of these factors might have more relevance in real-world applications (for example realtime systems with realtime data, possibly with limited computational resources) than in lab-environments using scientific software. We see from the plot that the residual is largest close to zero, which might be acceptable under circumstances. We could attempt to improve the fit, for example by modeling the residual and adding terms that cancel the residual.

The choice of modeling function depends on application, and without knowing test cases it is meaningless to decide.

In Ramsey's model, accuracy as $\alpha$ grows is more important, so the quotient is probably favourable over the exponential.

## Maximum relative error

Find the maximum relative error in $\alpha$: $\frac{\Delta\alpha}{\alpha}$, given relative errors in $U$ and $U'$. First, rearrange our equation:

$$f\left(X_n\right) - \tilde{\alpha}_n = \frac{-\tilde{U}\left(\tilde{\alpha}_n\right)}{\tilde{U}'\left(\tilde{\alpha}_n\right)} \Longrightarrow \alpha_n = f\left(X\right) + \frac{U}{U'}$$

From the definition of partial derivatives, we remember that linearisation means "variation in $z$ depending on variation of $x$ and $y$" when $z = f(x, y)$ - this would also be the equation of a tangent plane to $z = f(x, y)$. We recognize our problem of finding dependent errors as being similar:

$$\Delta\alpha = \frac{\partial\alpha}{\partial U}\Delta U + \frac{\partial\alpha}{\partial U'}\Delta U'$$

After finding partial derivatives, and the given relative errors:

$$\tfrac{\partial\alpha}{\partial U} = \tfrac{1}{U'} \qquad \tfrac{\partial\alpha}{\partial U'} = -\tfrac{U}{(U')^2} \qquad \tfrac{\Delta U}{U} = \tfrac{\Delta U'}{U'} = \pm\tfrac{1}{10}$$

the problem of solving for $\frac{\Delta\alpha}{\alpha}$ is now a matter of algebra:

$$\Delta\alpha = \tfrac{1}{U'}\Delta U - \tfrac{U}{(U')^2}\Delta U' \qquad \Delta\alpha \;=\; \tfrac{1}{U'}\left(\Delta U - \tfrac{U}{U'}\Delta U'\right)$$

The maximum error occurs when $f(X) = 0$ in the denominator:

$$\tfrac{\Delta\alpha}{\alpha} = \tfrac{1}{f(X)+\frac{U}{U'}}\tfrac{1}{U'}\left(\Delta U - \tfrac{U}{U'}\Delta U'\right) \;\Rightarrow\; \tfrac{\Delta\alpha}{\alpha} = \tfrac{1}{\frac{U}{U'}}\tfrac{1}{U'}\left(\Delta U - \tfrac{U}{U'}\Delta U'\right)$$

$$\tfrac{\Delta\alpha}{\alpha} = \tfrac{U'}{U}\tfrac{1}{U'}\left(\Delta U - \tfrac{U}{U'}\Delta U'\right) \;\Rightarrow\; \tfrac{\Delta\alpha}{\alpha} = \tfrac{\Delta U}{U} - \tfrac{1}{U'}\Delta U'$$

$$\tfrac{\Delta\alpha}{\alpha} = \tfrac{\Delta U}{U} - \tfrac{\Delta U'}{U'} \;\Rightarrow\; \tfrac{\Delta\alpha}{\alpha} = \pm\tfrac{1}{10} \mp \tfrac{1}{10} = \pm\tfrac{1}{5}$$

Maximum absolute relative error is $\frac{2}{10} = \frac{1}{5}$.

## 4 Constrained optimization with lagrange multipliers

The given problem is to find the optimum level of salaries (consumption) and re-investments (savings) in a company. This is modeled as the maximisation problem:

$$\max_{\alpha\in\mathbb{R}^{\mathbb{N}}} \int_0^T U(\alpha(t))\,dt + g(X(T))$$

subject to the constraint:

$$\tfrac{d}{dt}X(t) \;=\; f(X(t)) - \alpha(t)$$

Where $X(t)$ is our production function.

The strategy is to convert this optimization problem into a linear system of equations that can be solved using familiar methods.

We formulate the problem into a lagrange function:

$$\mathcal{L}(X,\alpha,\lambda) = \int_0^T U(\alpha)\,dt + g(X) - \lambda\left(f(X(t)) - \alpha(t) - \frac{dX}{dt}(t)\right)$$

The principle of linearisation allows us to use linear systems to solve this problem (normal lagrange multiplier method). However, we see the difficulty in proceeding to find the gradient of $\mathcal{L}$ and finding roots, and consider the suggestion that a numerical approach might be easier.

**Euler approximation** Our solution to the differential in the constraint condition is to use euler approximation, and approximating the integral with piecewise constant functions. Occurences of $dt$ transforms to $\Delta t$, the integral transforms into a riemann-sum, the constraint turns into an euler approximation:

$$\mathcal{L}\left(X,\alpha,\lambda\right)=\sum_{n=0}^{N-1}U\left(\alpha_n\right)\Delta t+g\left(X_n\right)-\sum_{n=0}^{N-1}\lambda_{n+1}\left(X_{n+1}-X_n-\Delta t\left(f(X_n)-\alpha_n\right)\right)$$

Setting the gradient to zero $\nabla\mathcal{L}=<0,0,0>$ and rearranging, the lagrange system becomes:

$$\begin{aligned}\lambda_{n+1}&=U'\left(\alpha_n\right)\\ \lambda_n&=\lambda_{n+1}+\Delta t\, f'\left(X_n\right)\,\lambda_{n+1}\\ X_{n+1}&=X_n+\Delta t\left(f(X_n)-\alpha_n\right)\end{aligned}$$

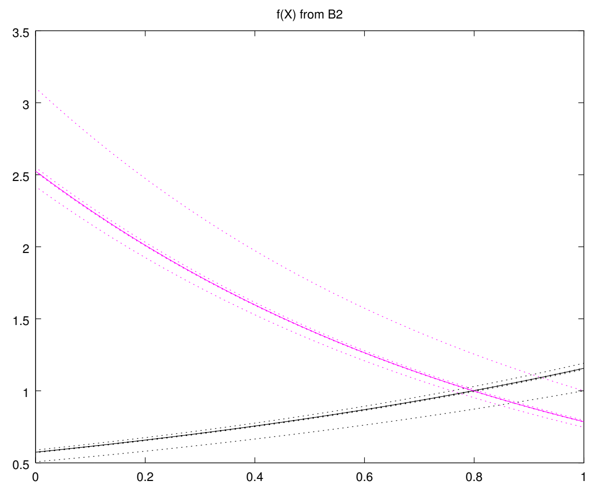
suitable for an iterative algorithm. Since this gives us $U'=\lambda_{n+1}$, we can solve for $\alpha$ as a function of $\lambda$:

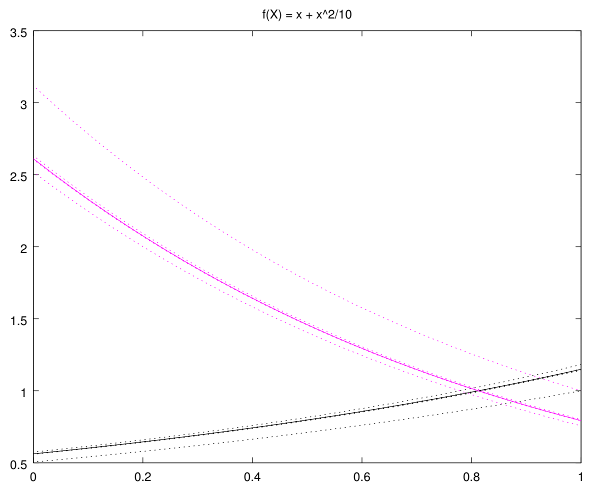
$$\begin{array}{rclcccc} U\left(\alpha\right) &=& -\frac{3}{2}\alpha^{\frac{-2}{3}} & & & & \\ U'(\alpha_n) &=& \frac{d}{d\alpha_n}\left(-\frac{3}{2}\alpha_n^{\frac{-2}{3}}\right) &=& \alpha_n^{\frac{-5}{3}} &=& \lambda_{n+1} \\ & & & & \left(\alpha_n^{\frac{-5}{3}}\right)^{\frac{-3}{5}} = \alpha_n &=& \left(\lambda_{n+1}\right)^{\frac{-3}{5}} \end{array}$$

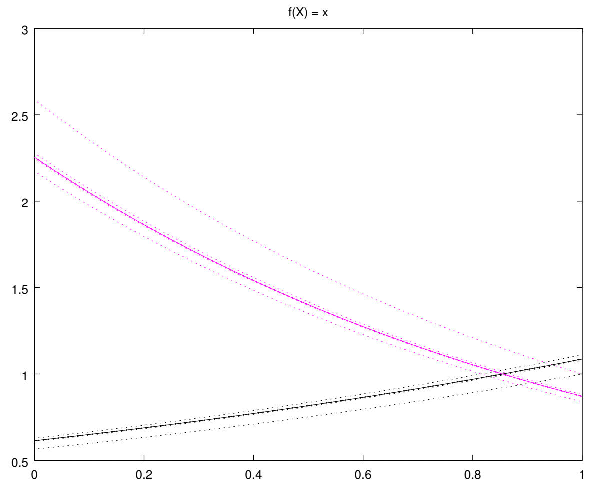
The numerical system is larger, with one equation for each time step. We find $X(t)$, $\lambda(t)$, $\alpha(t)$ as arrays of numbers through iterations.

## Results and analysis

Results are presented in these graphs. $\alpha(t)$ is represented by black plots $\lambda(t)$ is represented by magenta plots. Dotted lines are iterations, the solid line is the final result after a tolerance of $10^{-4}$ is achieved.







## Interpretation of the lagrange multipliers

What is the meaning of the lagrange multipliers in this case? Because they multiply reinvestments/savings, they represent the significance of reinvestments at time $t$ with respect to long-term consumption. Further, we see from the lagrangian system: $U' = \lambda_{n+1}$, which can be interpreted directly as marginal utility of consumption.

$\lambda(t)$ might be interpreted as the constraint on consumption at time $t$ necessary for maximum total consumption, over time. We might also interpret it as the magnitude of "the force" that restricts consumption and increases saving: analogous to the function of interest rates.

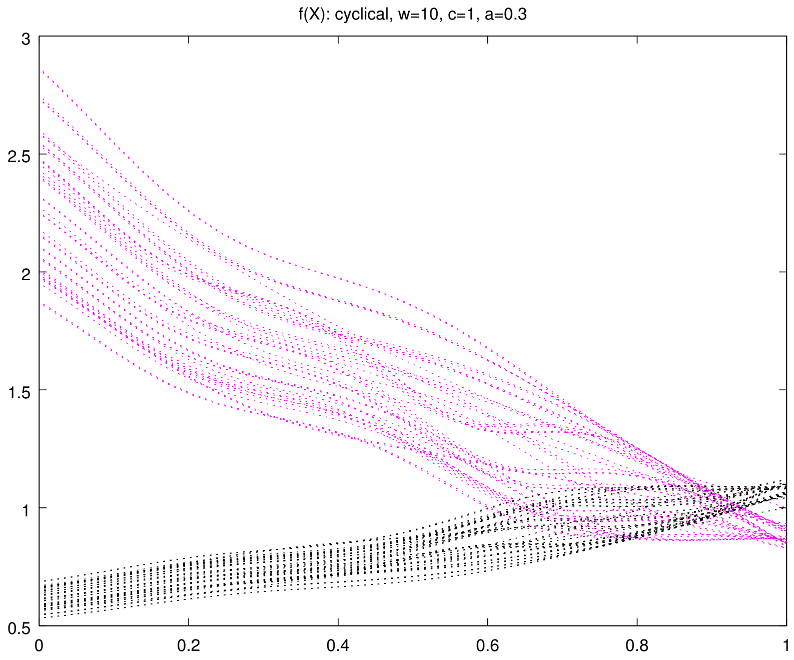
This is supported by the graphs, which show that the multipliers start high and converge.

## Attempts with cyclical growth

As an experiment, we test a production function $f(X)$ with a periodic term:

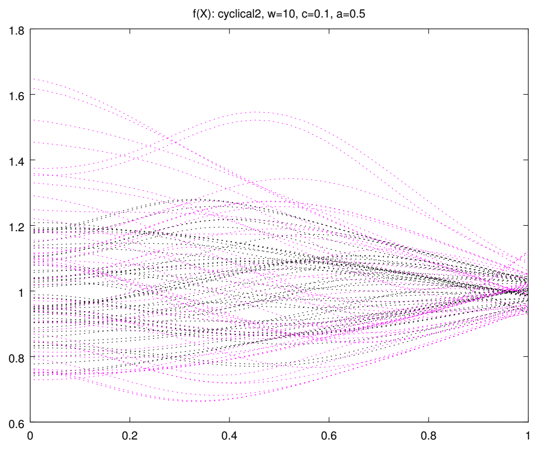
$$f(X) = x^{c+a\sin(\pi\omega x)} \qquad\qquad \frac{df}{dx} = f(X)\left(a\pi\omega\cos(\pi\omega x)\ln x + \frac{c+a\sin(\pi\omega x)}{x}\right)$$

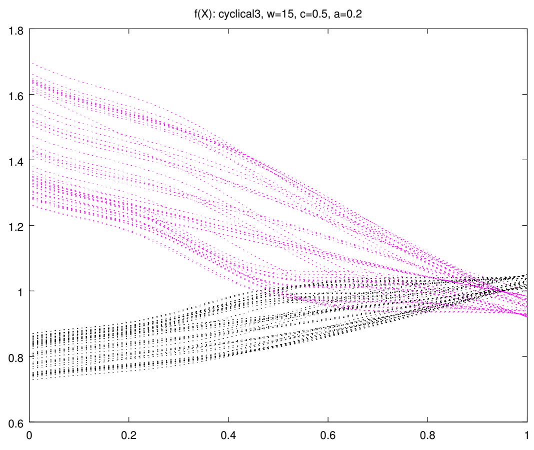
We're looking for patterns and convergence; and keeping in mind that $\lambda(t)$ is vaguely analogous to interest rates in economies.



We see that solutions stay within bounds around the related linear solution, with no clear convergence within the bounds.

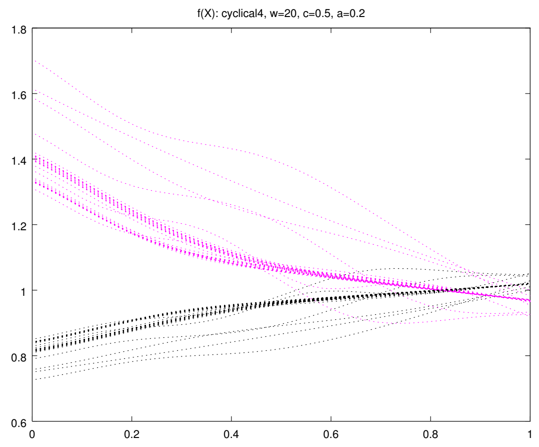
*cyclical2:* With a small growth constant c and strong imprint from the periodic term via the amplitude, the bounds are wide and we see no convergence within the bounds. No constraint seems to be able to create a convergent solution. Overall, the solutions still converge.

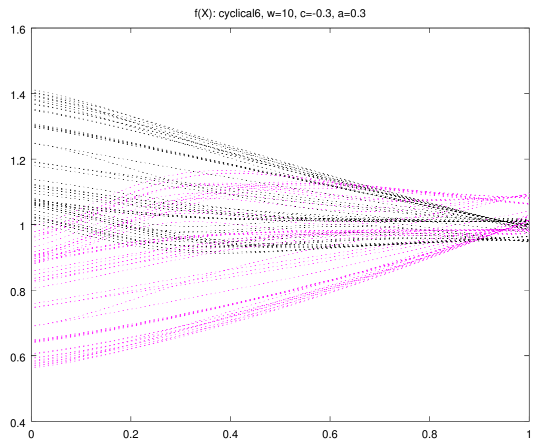




Many attempts and variations showed similar patterns. It suggests that under more realistic circumstances the problem is less deterministic. We see that $\lambda(t)$ seems to converge to more linear shapes; suggesting that over-corrections lead to less optimal solutions.

*cyclical4:* Increasing frequency changes the shape, and sometimes allows for clear convergence:





*cyclical6:* Negative growth gives an interesting example; starting with high decreasing consumption and low investment, and eventually inverting. We see many possible solutions with early invertions, but solutions with later invertions are more clustered:

## 5 Error sources

### Assumptions about physical conditions

To what degree does Ramsey's mathematical model correspond to physical reality? Ramsey makes assumptions about production as function of capital (disregarding the role of human creative input), and enjoyment/utility as a function of production (disregarding misallocations), and labour as "disutility" (disregarding work a meaningful). Applying a model like Ramsey's requires careful understanding of the assumptions.

In other applications, it might be less obvious how the mathematical model misrepresents physical reality, even if calculated without errors. A book (Peter Pohl) gives the example "a point-mass, pendulating with infinitely small movements on a massless string".

**Input data** In our example, input data was picked from Ramsey's original paper, and likely has little connection to present economic conditions.

**Truncation** Truncation is the error resulting from the interrupted limit. For example; in euler and trapezoid methods, since $\Delta t$ is fixed an does not approach zero (as in the analytical solutions) each iteration will accumulate an error. This will happen regardless of convergence characteristics of the algorithm (both forward and implicit euler).

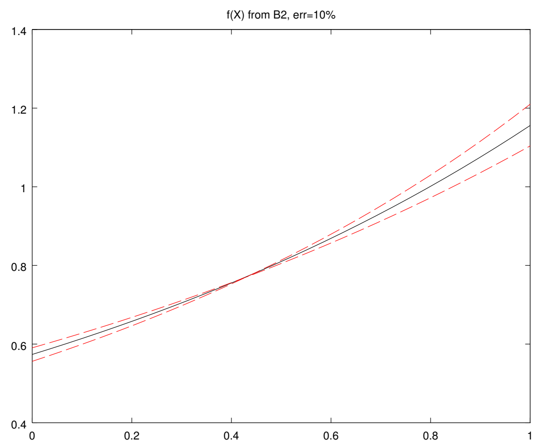
**Euler and trapezoid** Algorithms such as euler approximation cause mathematical errors relative to analytical solutions. Forward euler, for example, will diverge. While this would not happen if somehow implemented numerically with true infinitesimals and limits, the truncation error is particularily worse in certain algorithms. The integration approximation in problem 4, has error $O\left(\frac{1}{n}\right)$. Trapezoid would have an error on the order of $O\left(\frac{1}{n^2}\right)$. These can be derived through the taylor series.

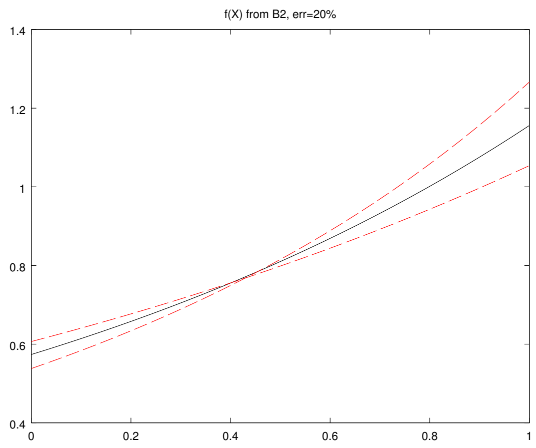
Truncation errors can be minimized by reducing stepsizes; however, precision errors grow with smaller stepsizes; and as number of iterations grow, all errors can compound.

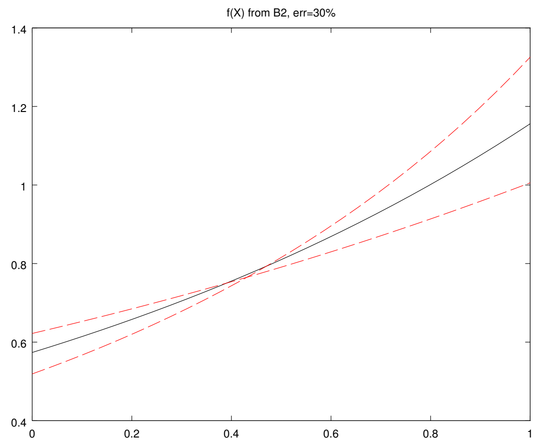
**Precision** Cancellation errors occur as one number is subtracted from another close number. The difference that is smaller than precision is the error. Since the resulting number is small, the relative error might be large.

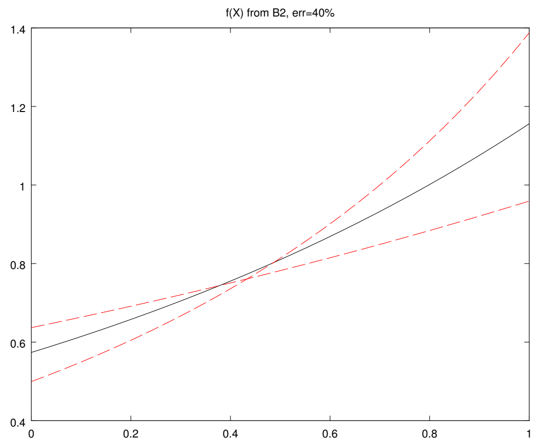
**Comparison** The biggest source of error in our lagrange system is likely truncation from large stepsizes, but precision and accuracy is probably adequate when compared to reliability of the input data and intended use of the output.

**Experimental error analysis and condition numbers** We test the optimization method from section 4, by distorting value of the production function $aX + bX^2$ with $err\%$. The red/dashed lines are the $\alpha$-plots corresponding to the test. We get a visual impression of the "condition" of the optimization method, and we see the error bounds. At worst, when $t = 1$, we see condition numbers $\left(\frac{err_{out}}{err_{in}}\right)$ close to one, but we also observe that the condition-number grows fast with time. This method would not be reliable if we extended $t$ much further beyond 1.









## Code listings

All graphs were produced by the code listed below. All code was run with octave.

## lab1_b2.m

```
function file = lab1_b2
endfunction

function [solution, residual] = computeLeastSquaresFit (varargin)
  assert(nargin > 2)

  f_values = (varargin{1})

  num_terms = nargin - 1

  y = zeros(rows(f_values), 1)
  A = zeros(rows(f_values), num_terms)

  for this_row = 1:rows(f_values)
    y(this_row, 1) = f_values(this_row, 2)
    for this_column = 2:(nargin)
      f_term = (varargin{this_column})
      f_arg = f_values(this_row, 1)
      A(this_row, this_column-1) = f_term(f_arg)
    end
  end

  solution = A \ y
  residual = y - A*solution
endfunction

function testLeastSquaresFit ()
  f_values = [40   , 55.3  ;
              45   , 71.9  ;
              50   , 92.5  ;
              55   , 118   ;
              60   , 149.4 ]

  [ coeffs, residual ] = computeLeastSquaresFit(f_values, @(x) 1, @(x) x)

  stepsize = (max(f_values(:,1) - min(f_values(:,1)))) / 50

  for i = 1:50
    u_v = i * stepsize
    x(i) = u_v
    ls_y(i) = coeffs(1) * 1 + coeffs(2) * u_v
    #p_y(i) = polyval(polynomial_coeffs, u_v)
  end

  plot(x, ls_y)
  hold on
  plot(f_values(:,1), f_values(:,2), "or", "markersize", 5)
  # plot(f_values(:,1), residual, "or", "markersize", 5)
  pause
endfunction

#testLeastSquaresFit()
```

```
function p = computePolyfit (f_values)
  x = transpose(f_values(:,1))
  y = transpose(f_values(:,2))
  p = polyfit(x, y, rows(f_values))
endfunction

function B2 ()
  f_values = [0     , 0     ;
              0.5   , 0.52  ;
              1     , 1.09  ;
              1.5   , 1.75  ;
              2     , 2.45  ;
              2.99  , 3.5   ;
              3     , 4.0]

  [ls_coeffs, ls_residual] = computeLeastSquaresFit(f_values, @(x) x, @(x) x^2)

  polynomial_coeffs = computePolyfit(f_values)

  stepsize = (max(f_values(:,1) - min(f_values(:,1)))) / 50

  for i = 1:50
    u_v = i * stepsize
    x(i) = u_v
    ls_y(i) = ls_coeffs(1) * u_v + ls_coeffs(2) * u_v^2
    p_y(i) = polyval(polynomial_coeffs, u_v)
  end

  plot(x, ls_y)
  hold on
  plot(x, p_y, "--r")
  plot(f_values(:,1), f_values(:,2), "or", "markersize", 5)
  plot(f_values(:,1), ls_residual, "ox", "markersize", 5)
  title(["ax + bx^2, a=" num2str(ls_coeffs(1)) ", b=" num2str(ls_coeffs(2))  ])

  print -depsc lab1_b2.eps
  hold off
endfunction

B2()
```

## lab1_b3.m

```
function file = lab1_b3
endfunction

function y = TOL
  y = 10^(-6)
endfunction

function [solution, iter] = computeGaussNewtonFit (_compute_F, _compute_J, ck)
  iter = 0
  delta_f = 1

  while (delta_f > TOL) && (iter++ < 100)
    c_prev = ck

    f = _compute_F(ck)
    jacobian = _compute_J(ck)

    delta_c = jacobian \ f

    ck = ck - transpose(delta_c)

    delta_f = norm(ck - c_prev)
  endwhile

  solution = ck
endfunction

function f_matrix = computeF (F, x_values, c_args)
  f_matrix = zeros(1, rows(x_values))
  for i = 1:rows(x_values)
    args = num2cell([x_values(i,:), c_args(1,:)])
    f_matrix(i) = F(args{:})
  end
  f_matrix = transpose(f_matrix)
endfunction

function jacobian = computeJacobian (partial_derivatives, x_values, c_args)
  jacobian = zeros(rows(x_values), columns(partial_derivatives))
  for j = 1:columns(partial_derivatives)
    f_partial = (partial_derivatives{j})
    for i = 1:rows(x_values)
      args = num2cell([x_values(i,:), c_args(1,:)])
      jacobian(i, j) = f_partial(args{:})
    end
  end
endfunction

function testGaussNewtonFit1 ()
  f_values = [0.0 , 7.4 ;
              0.1 , 6.6 ;
              0.2 , 6.0 ;
              0.3 , 5.6 ;
              0.4 , 5.2 ;
              0.5 , 4.9 ;
              0.6 , 4.7 ;
              0.7 , 4.4 ;
              0.8 , 4.2 ;
              0.9 , 4.1 ;
              1.0 , 4.0 ]
  x_values = f_values(:,1)
  y_values = f_values(:,2)

  F    = @(x, c1, c2, c3) c1 + c2*e^(-c3*x)
  F_c1 = @(x, c1, c2, c3) 1
  F_c2 = @(x, c1, c2, c3)         e^(-c3*x)
  F_c3 = @(x, c1, c2, c3)   - x*c2*e^(-c3*x)

  F_partials = {F_c1, F_c2, F_c3}

  _compute_F = @(ck) computeF(F, x_values, ck) - y_values
  _compute_J = @(ck) computeJacobian(F_partials, x_values, ck)

  start_c = [3.5, 3.9, 2]
  iter = 0

  [solution, iter] = computeGaussNewtonFit(_compute_F, _compute_J, start_c)
  residual = y_values - F([x_values, solution]{:})
endfunction

#testGaussNewtonFit1()

function testGaussNewtonFit2 ()
  f_values = [0.2 , 3.16 ;
              0.3 , 2.38 ;
              0.4 , 1.75 ;
              0.5 , 1.34 ;
              0.6 , 1.00 ]

  x_values = f_values(:,1)
  y_values = f_values(:,2)

  F   = @(x, a, b)     a*e^(-b*x)
  F_a = @(x, a, b)       e^(-b*x)
  F_b = @(x, a, b) -x*a*e^(-b*x)

  F_partials = {F_a, F_b}

  _compute_F = @(ck) computeF(F, x_values, ck) - y_values
  _compute_J = @(ck) computeJacobian(F_partials, x_values, ck)

  start_c = [5.618, 2.88]

  [solution, iter] = computeGaussNewtonFit(_compute_F, _compute_J, start_c)

  # expect = [5.6310 , 2.8872]
endfunction

#testGaussNewtonFit2()

function testGaussNewtonFit3 ()
  f_values = [0.5 , 0.3 ;
              0.8 , 0.3 ;
              1.0 , 0.5 ;
              1.2 , 0.9 ;
              1.5 , 1.4 ;
              1.8 , 1.1 ;
              2.0 , 0.5 ;
              2.4 , 0.3 ]
  x_values = f_values(:,1)
  y_values = f_values(:,2)

  F    = @(x, a, b, w, x0) a          + b * sin(w*(x - x0))
  F_a  = @(x, a, b, w, x0) 1
  F_b  = @(x, a, b, w, x0)                sin(w*(x - x0))
  F_w  = @(x, a, b, w, x0) (x - x0) * b * cos(w*(x - x0))
  F_x0 = @(x, a, b, w, x0)       -w * b * cos(w*(x - x0))

  F_partials = {F_a, F_b, F_w, F_x0}

  _compute_F = @(ck) computeF(F, x_values, ck) - y_values
  _compute_J = @(ck) computeJacobian(F_partials, x_values, ck)

  start_c = [0.7, 0.7, pi, 1.2]

  [solution, iter] = computeGaussNewtonFit(_compute_F, _compute_J, start_c)

  # expect = [0.7761, 0.5860, 3.9225, 1.1092]
endfunction

#testGaussNewtonFit3()

function p = computePolyfit (f_values)
  x = transpose(f_values(:,1))
  y = transpose(f_values(:,2))
  p = polyfit(x, y, rows(f_values))
endfunction

function v = U_DATA ()
  v = [150  , 2 ;
       200  , 3 ;
       300  , 4 ;
       500  , 5 ;
       1000 , 6 ;
       2000 , 7 ]
endfunction

function B3_1 ()
  a_values = U_DATA() (:,1)
  u_values = U_DATA() (:,2)

  U   = @(alfa, a) (8*alfa) / (alfa + 8*a)
  U_a = @(alfa, a) (8*alfa) * (-8/(alfa + 8*a)^2)

  U_partials = {U_a}

  start_c = [56]

  _compute_U = @(ck) computeF(U, a_values, ck) - u_values
  _compute_J = @(ck) computeJacobian(U_partials, a_values, ck)

  [solution, iter] = computeGaussNewtonFit(_compute_U, _compute_J, start_c)

  residual = _compute_U(solution)
  err = norm(residual)

  U_fun = @(alfa) U(alfa, solution(1))

  for i = 1:30
    u_v=i*200;    x(i) = u_v;    y(i) = U_fun(u_v);
  end
  plot(x, y);
  title(["8*alfa / (alfa + 8a), a=" num2str(solution(1)) ", iterations: " num2str(iter) "
    ,2 norm(residual): " num2str(err) ])
  hold on
  plot(a_values, u_values, "or", "markersize", 5)
  plot(a_values, residual, "ox", "markersize", 5)
  print -depsc lab1_b3_1.eps
  hold off
endfunction

B3_1()

function B3_2 ()
  a_values = U_DATA() (:,1)
  u_values = U_DATA() (:,2)

  U     = @(alfa, a, b) 8 - a*(alfa^b)
  U_a   = @(alfa, a, b) - alfa^b
  U_b   = @(alfa, a, b) - a * (alfa^b) * log(alfa)

  U_partials = {U_a, U_b}

  _compute_U = @(ck) computeF(U, a_values, ck) - u_values
  _compute_J = @(ck) computeJacobian(U_partials, a_values, ck)

  start_c = [100, -1/2]   % Gissningar, experimentering. Känslig

  [solution, iter] = computeGaussNewtonFit(_compute_U, _compute_J, start_c)

  residual = _compute_U(solution)
  err = norm(residual)

  U_fun = @(alfa) U(alfa, solution(1), solution(2))

  for i = 1:30
    u_v= i*200;    x(i) = u_v;    y(i) = U_fun(u_v);
  end
  plot(x, y)
  title(["8 - a*(alfa^b), a=" num2str(solution(1)) ", b=" num2str(solution(2)) ",
    iterations: " num2str(iter)  ", norm(residual): " num2str(err) ])
  hold on
  plot(a_values, u_values, "or", "markersize", 5)
  plot(a_values, residual, "ox", "markersize", 5)
  print -depsc lab1_b3_2.eps
  hold off
endfunction

B3_2()

function B3_polynomial ()
  polynomial = computePolyfit(U_DATA)

  for i = 1:100
    u_v = i*20;   x(i) = u_v;    y(i) = polyval(polynomial, u_v)
  end
  plot(x, y)
  hold on
  plot(U_DATA() (:,1), U_DATA() (:,2), "or", "markersize", 5)
  print -depsc lab1_b3_polynomial.eps
  hold off
```

```
endfunction

B3_polynomial()
```

## lab1_b4.m

```
function file = lab1_b4
endfunction

function y = TOL
  y = 10^(-4)
endfunction

function y = N_STEPS
  y = 30
endfunction

function y = f_Xb2 (x, err=0)
  a = 1.0709
  b = 0.061849
  y = a*x + b*(x^2)
  y = y*(1+err)
endfunction

function y = fp_Xb2 (x, err=0)
  a = 1.0709
  b = 0.061849
  y = a + 2*b*(x)
  y = y*(1+err)
endfunction

function y = f_X2 (x)
  y = x + ((x^2) / 10)
endfunction

function y = fp_X2 (x)
  y = 1 + (2*x / 10)
endfunction

function y = f_X (x)
  y = x
endfunction

function y = fp_X (x)
  y = 1
endfunction

function y = f_cyclical_X (x, w, c, a)
  g = c + a * sin(pi*w*x)
  y = x^(g)
endfunction

function yp = fp_cyclical_X (x, w, c, a)
  g = c + a * sin(pi*w*x)
  g_p = a * w * cos(pi*w*x)
  y = x^(g)
  yp = y * (g_p*log(x) + g / x)
endfunction

function y = f_g (X)
  y = 2 * sqrt(X)
endfunction

function y = fp_g (X)
  y = 1 / sqrt(X)
```

```
endfunction

function solveLagrangeSystem (_f_X, _fp_X, _title , _plot_iter=1, _err_analysis=0)
 T = 1
 dt = T / N_STEPS

 X = ones(N_STEPS+1, 1)
 l = zeros(N_STEPS, 1)
 alfa_values = zeros(N_STEPS, 1)

 vector_t = linspace(0, T, N_STEPS)

 tol_achieved = 0

 if 0 == _err_analysis
   hold off
   newplot()
   hold on
 end

 i = 0
 while (i++ <= 50)
   n = N_STEPS

   prev_x = X

   l(n) = fp_g(X(n))

   do
     l(n-1) = l(n) * (1 + dt * _fp_X(X(n)) )
   until (--n == 1)

   alfa = @(_n) (l(_n))^(-3/5)

   alfa_values(n) = alfa(n)
   do
     a_n = alfa_values(n+1) = alfa(n+1)
     X(n+1) = X(n) + dt * (_f_X(X(n)) - a_n)
   until (++n == N_STEPS)

   if norm(X - prev_x) < TOL
     tol_achieved = 1
     break
   end

   if 1 == _plot_iter
     plot(vector_t, l, ":m")
     plot(vector_t, alfa_values, ":k")
   end
 end

 if 1 == tol_achieved
   if 1 == _err_analysis
     plot(vector_t, alfa_values, "--r")
   else
     plot(vector_t, alfa_values, "-k")
   end

   if 1 == _plot_iter
     plot(vector_t, l, "-m")
   end
```

```
    end
    title(_title)
endfunction

function doB4()
    solveLagrangeSystem(@f_Xb2, @fp_Xb2, "f(X) from B2")
    print -depsc lab1_b4_1.eps

    solveLagrangeSystem(@f_X2, @fp_X2, "f(X) = x + x^2/10")
    print -depsc lab1_b4_2.eps

    solveLagrangeSystem(@f_X, @fp_X, "f(X) = x")
    print -depsc lab1_b4_3.eps

   w = 10; c=1; a=0.3
    solveLagrangeSystem(@(x) f_cyclical_X(x, w, c, a),
                        @(x) fp_cyclical_X(x, w, c, a),
                        "f(X): cyclical , w=10, c=1, a=0.3")
    print -depsc lab1_b4_cyclical1.eps

   w = 10; c = 0.1; a = 0.5
    solveLagrangeSystem(@(x) f_cyclical_X(x, w, c, a),
                        @(x) fp_cyclical_X(x, w, c, a),
                        "f(X): cyclical2 , w=10, c=0.1, a=0.5")
    print -depsc lab1_b4_cyclical2.eps

   w = 15; c = 0.5; a = 0.2
    solveLagrangeSystem(@(x) f_cyclical_X(x, w, c, a),
                        @(x) fp_cyclical_X(x, w, c, a),
                        "f(X): cyclical3 , w=15, c=0.5, a=0.2")
    print -depsc lab1_b4_cyclical3.eps

   w = 20
    solveLagrangeSystem(@(x) f_cyclical_X(x, w, c, a),
                        @(x) (fp_cyclical_X(x, w, c, a)),
                        "f(X): cyclical4 , w=20, c=0.5, a=0.2")
    print -depsc lab1_b4_cyclical4.eps

   w = 5; c = 0.1; a = 0.5
    solveLagrangeSystem(@(x) f_cyclical_X(x, w, c, a),
                        @(x) fp_cyclical_X(x, w, c, a),
                        "f(X): cyclical5 , w=5, c=0.1, a=0.5")
    print -depsc lab1_b4_cyclical5.eps

   w = 10; c = -0.3; a = 0.3
    solveLagrangeSystem(@(x) f_cyclical_X(x, w, c, a),
                        @(x) fp_cyclical_X(x, w, c, a),
                        "f(X): cyclical6 , w=10, c=-0.3, a=0.3")
    print -depsc lab1_b4_cyclical6.eps
endfunction

doB4()

function doB4_errors()
   _f_X = @(x) f_Xb2(x)
   _fp_X = @(x) fp_Xb2(x)
    solveLagrangeSystem(_f_X, _fp_X, "f(X) from B2", 0)

    err=0.1;
   _f_X = @(x) f_Xb2(x, err)
   _fp_X = @(x) fp_Xb2(x, err)
```

```
    solveLagrangeSystem(_f_X, _fp_X, "", 0, 1)

    _f_X = @(x) f_Xb2(x, -err)
    _fp_X = @(x) fp_Xb2(x, -err)
    solveLagrangeSystem(_f_X, _fp_X, "f(X) from B2, err=10%", 0, 1)
    print -depsc lab1_b4_err1.eps

    _f_X = @(x) f_Xb2(x)
    _fp_X = @(x) fp_Xb2(x)
    solveLagrangeSystem(_f_X, _fp_X, "f(X) from B2", 0)

    err=0.2;
    _f_X = @(x) f_Xb2(x, err)
    _fp_X = @(x) fp_Xb2(x, err)
    solveLagrangeSystem(_f_X, _fp_X, "", 0, 1)

    _f_X = @(x) f_Xb2(x, -err)
    _fp_X = @(x) fp_Xb2(x, -err)
    solveLagrangeSystem(_f_X, _fp_X, "f(X) from B2, err=20%", 0, 1)
    print -depsc lab1_b4_err2.eps

    _f_X = @(x) f_Xb2(x)
    _fp_X = @(x) fp_Xb2(x)
    solveLagrangeSystem(_f_X, _fp_X, "f(X) from B2", 0)

    err=0.3;
    _f_X = @(x) f_Xb2(x, err)
    _fp_X = @(x) fp_Xb2(x, err)
    solveLagrangeSystem(_f_X, _fp_X, "", 0, 1)

    _f_X = @(x) f_Xb2(x, -err)
    _fp_X = @(x) fp_Xb2(x, -err)
    solveLagrangeSystem(_f_X, _fp_X, "f(X) from B2, err=30%", 0, 1)
    print -depsc lab1_b4_err3.eps

    _f_X = @(x) f_Xb2(x)
    _fp_X = @(x) fp_Xb2(x)
    solveLagrangeSystem(_f_X, _fp_X, "f(X) from B2", 0)

    err=0.4;
    _f_X = @(x) f_Xb2(x, err)
    _fp_X = @(x) fp_Xb2(x, err)

    solveLagrangeSystem(_f_X, _fp_X, "", 0, 1)

    _f_X = @(x) f_Xb2(x, -err)
    _fp_X = @(x) fp_Xb2(x, -err)
    solveLagrangeSystem(_f_X, _fp_X, "f(X) from B2, err=40%", 0, 1)
    print -depsc lab1_b4_err4.eps

endfunction

doB4_errors()
```