Exercise 1. The conjugate gradient method for solving a symmetric positive definite linear system $Ax = b$ can be written as below:

```
Given $x_0$, compute $r_0 = b - Ax_0$, and set $p_0 = r_0$.
For $k = 1, 2, \ldots$,
    Compute $Ap_{k-1}$.
    Set $x_k = x_{k-1} + a_{k-1}p_{k-1}$, where $a_{k-1} = \frac{\langle r_{k-1}, r_{k-1} \rangle}{\langle p_{k-1}, Ap_{k-1} \rangle}$.
    Compute $r_k = r_{k-1} - a_{k-1}Ap_{k-1}$.
    Set $p_k = r_k + b_{k-1}p_{k-1}$, where $b_{k-1} = \frac{\langle r_k, r_k \rangle}{\langle r_{k-1}, r_{k-1} \rangle}$.
Endfor
```

(a) Show that the residual vectors $r_0, \ldots, r_k$ are orthogonal to each other ($\langle r_i, r_j \rangle = 0$ if $i \neq j$) and that the direction vectors $p_0, \ldots, p_k$ are A-orthogonal ($\langle p_i, Ap_j \rangle = 0$ if $i \neq j$). [Hint: First show that $\langle r_1, r_0 \rangle = \langle p_1, Ap_0 \rangle = 0$ and then use induction on k.]

(b) If A is the $N \times N$ matrix of the 5-point operator for Poisson's equation on a square, count the number of operations (additions, subtractions, multiplications, and divisions) performed in each iteration. (Show how you arrive at your result.)

(c) Compare your operation count in (b) to that of a Gauss-Seidel iteration applied to the same N by N 5-point matrix A. Also compare to the operation count for a multigrid V-cycle, using one Gauss-Seidel iteration on each visit to each grid level. (Again, show how you derive your operation counts.)

Solution 1. (a) We'll begin with the base case. Starting with

$$\begin{aligned}(r_1, r_0) &= (r_0, r_0) - a_0(Ap_0, r_0)\\ &= (r_0, r_0) - \left(\frac{(r_0, r_0)}{(p_0, Ap_0)}\right)(Ap_0, r_0)\\ &= 0,\end{aligned}$$

where in the last line we've used that $p_0 = r_0$. Next, we have that

$$\begin{aligned}(p_1, Ap_0) &= (r_1, Ap_0) + b_0(p_0, Ap_0)\\ &= (r_1, a_0^{-1}(r_0 - r_1)) + b_0(r_0, a_0^{-1}(r_0 - r_1))\\ &= -(r_1, a_0^{-1}r_1) + a_0^{-1}b_0(r_0, r_0)\\ &= 0,\end{aligned}$$

where we've used that $Ap_0 = a_0^{-1}(r_0 - r_1)$, $(r_1, r_0) = 0$, and the definition of b_0. We'll now assume that the following is true for all $i < k$,

$$(r_k, r_i) = (p_k, Ap_i) = 0.$$

We can then write

$$\begin{aligned}(p_k, Ap_k) &= (r_k, Ap_k) + b_{k-1}(p_{k-1}, Ap_k)\\ &= (r_k, Ap_k) + b_{k-1}(Ap_{k-1}, p_k)\\ &= (r_k, Ap_k)\end{aligned}$$

since A is real symmetric. This allows us to show

$$\begin{aligned}(r_{k+1}, r_k) &= (r_k, r_k) - a_k(Ap_k, r_k)\\ &= (r_k, r_k) - \left(\frac{(Ap_k, r_k)}{(Ap_k, p_k)}\right)(r_k, r_k) = 0.\end{aligned}$$

Additionally, we have that

$$\begin{aligned}(r_k, p_k) &= (r_k, r_k) + b_{k-1}(r_k, p_{k-1})\\ &= (r_k, r_k) + b_{k-1}(r_k, r_{k-1}) + b_{k-1}b_{k-2}(r_k, p_{k-2}) + \cdots\\ &= (r_k, r_k),\end{aligned}$$

as we continue this pattern of reducing $p_k = r_k + b_{k-1}p_{k-1}$, we will get terms of the form $(r_k, r_j) = 0$ and (r_k, p_j) until we reach $(r_k, p_0) = (r_k, r_0) = 0$. We'll pair this with the following

$$(r_{k+1}, p_k) = (r_k, p_k) - a_k(Ap_k, p_k) = 0.$$

These two equalities allow us to show that

$$\begin{aligned}(Ap_k, p_{k+1}) &= (Ap_k, r_{k+1}) + \frac{(r_{k+1}, r_{k+1})}{(r_k, r_k)}(Ap_k, p_k)\\ &= (a_k^{-1}(r_k - r_{k+1}), r_{k+1}) + \frac{(r_{k+1}, r_{k+1})}{(r_k, r_k)}\left[(a_k^{-1}(r_k - r_{k+1}), p_k)\right]\\ &= -a_k^{-1}(r_{k+1}, r_{k+1}) + a_k^{-1}(r_{k+1}, r_{k+1}) = 0.\end{aligned}$$

We've therefore shown

$$(Ap_k, p_{k+1}) = (r_{k+1}, r_k) = 0.$$

We then have that

$$\begin{aligned}(r_{k+1}, r_j) &= (r_k, r_k) - a_k(Ap_k, r_j)\\ &= -a_k(p_k, A[p_j - b_{j-1}p_{j-1}]) = 0\end{aligned}$$

Lastly,

$$\begin{aligned}(p_{k+1}, Ap_j) &= (r_{k+1}, Ap_j) + b_k(p_k, Ap_j)\\ &= (r_{k+1}, a_j^{-1}(r_j - r_{j+1}) = 0,\end{aligned}$$

which completes the proof.

(b) We'll start with some baseline groundwork. Each inner product computed (assuming we add any zeros anyway) will typically result in N multiplications (for each index) and $N-1$ additions. Starting Ap_{k-1}, computing the product in each row requires 5 multiplications and 4 additions for each of the N rows. Therefore, Ap_{k-1} requires $5N$ multplications and $4N$ additions.

$$+ : 4(N-1), \quad \times : 5N$$

We then must compute a_{k-1} which requires computing two inner products of known values and one division

$$+ : 4(N-1) + 2(N-1) = 6(N-1), \quad \times : 5N + 2N = 7N, \quad \div : 1$$

Multiplying p_{k-1} by a_{k-1} is scalar multiplication and requires N multiplications

$$+ : 6(N-1), \quad \times : 7N + N = 8N, \quad \div : 1$$

Adding $x_{k+1} + a_{k-1}p_{k-1}$ requires N additions

$$+ : 6(N-1) + N = 7N - 6, \quad \times : 8N, \quad \div : 1$$

To compute r_k, we'll need N subtractions and N multiplications

$$+ : 7N - 6, \quad \times : 8N + N = 9N, \quad \div : 1, \quad - : N.$$

For b_{k-1}, we compute two inner products and divide once

$$+ : 7N - 6 + 2(N-1) = 9N - 7, \quad \times : 9N + 2N = 11N, \quad \div : 1 + 1 = 2, \quad - : N.$$

To compute p_k, we need N additions and N multiplications

$$+ : 9N - 7 + N = 10N - 7, \quad \times : 11N + N = 12N, \quad \div : 1 + 1 = 2, \quad - : N.$$

This gives $\approx 23N$ operations for each loop of conjugate gradient.
(c) We'll now do the operator count for each iteration of Gauss-Seidel. First, we set M to be the lower triangle of the matrix. We also define $N = U$, the strict upper triangle of A. The rest of the loop is given by

$$Mu^{(k+1)} = Nu^{(k)} + f,$$

where we solve for u using forward substitution, so that

$$u_i^{(k+1)} = \frac{1}{a_{ii}} \left(f_i - \sum_{j=1}^{i-1} a_{ij} u_i^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} u_j^{(k)} \right).$$

As the matrix A has only 5 non-zero elements per row, we have that there are 5 multiplications, 5 subtractions, and 1 division per row

$$+ : 0, \quad \times : 5N, \quad \div : N, \quad - : 5N.$$

For a V cycle using one Gauss-Seidel iteration, we have that if there are n layers for which the size decreases by 1/2 per layer. And smoothing occurs once at each visit (of which there are two to each layer), we have that total operations from follow the GS step alone

$$\mathrm{Op}_G(N) = 2 \cdot \sum_{k=0}^{n} \left(11 \frac{N}{2^k} \right) \approx 44N,$$

where N is the size of the largest grid. We use an approximation large n to get an estimate of $44N$. At each step, we must also project or interpolate the error which requires (at the highest level) N additions and N divisions for interpolation as we take the average of neighboring points and 0 mathematical operations for projection (we simply take every other element of the finer grid, so there's no need for addition, subtraction, multiplication or division). We then have that as there $n-1$ interpolations on the up cycle to get back to the full grid

$$\mathrm{Op}_{IC}(N) = \sum_{k=1}^{n} \left(2 \frac{N}{2^k} \right) \approx 2N.$$

Lastly, we have that we must add the errors accumulated to the final solution which has N additions. This gives a final operations count as

$$\mathrm{Op}(N) \approx 44N + 2N + N = 47N$$

in the large n limit.

Exercise 2. Repeat the experiments on p. 103 of the text, leading to Figures 4.8 and 4.9, but use the Gauss-Seidel method and (unpreconditioned) CG instead of Jacobi iteration. That is, set up difference equations for the problem

$$u''(x) = f(x), \quad u(0) = 1, \; u(1) = 3,$$

where

$$f(x) = -20 + a\varphi''(x)\cos(\varphi(x)) - a(\varphi'(x))^2 \sin(\varphi(x)),$$

where $a = 0.5$ and $\varphi(x) = 20\pi x^3$. The true solution is

$$u(x) = 1 + 12x - 10x^2 + a\sin(\varphi(x)).$$

Starting with initial guess $u^{(0)}$ with components $1 + 2x_i$, $i = 1, \ldots, 255$, run, say, 20 Gauss-Seidel iterations and then 20 CG iterations, plotting the true solution to the linear system and the approximate solution, say, at steps 0, 5, 10, and 20, and also plotting the error (the difference between true and approximate solution). Print the size of the error (the L_2-norm or the ∞-norm) at these steps too. Based on your results, would you say that Gauss-Seidel and CG would be effective smoothers for a multigrid method?

Solution 2. I've implemented this all using julia. It appears that for the specified number of steps that Gauss-Seidel and CG both make progress by figuring out the higher frequencies first. Because of this, I think both methods would make effective smoothers for multigrid as the coarse grid operators would be useful for figuring out lower frequencies, leaving little work for GS and CG and allowing faster convergence.

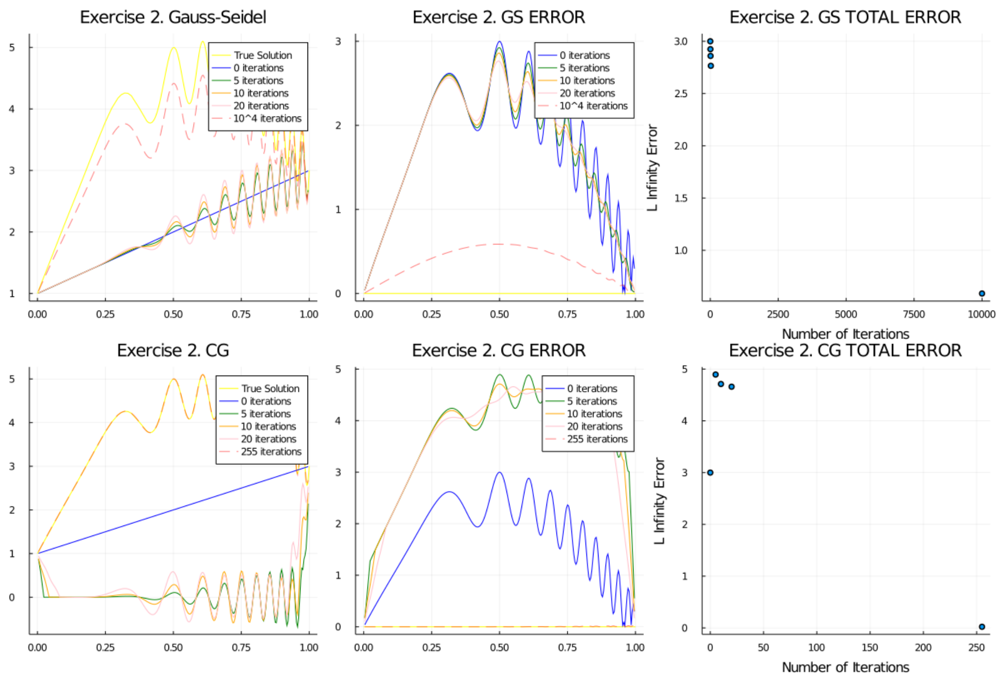


Figure 1

Exercise 3. Implement a 2-grid method for solving the 1D model problem with homogeneous Dirichlet boundary conditions:

$$u_{xx} = f(x), \quad u(0) = u(1) = 0.$$

Use linear interpolation to go from the coarse grid with spacing $2h$ to the fine grid with spacing h. Take the projection matrix I_h^{2h}, going from the fine grid to the coarse grid, to be 0.5 times the transpose of the interpolation matrix: $I_h^{2h} = \frac{1}{2}(I_{2h}^h)^T$. Use a multigrid V-cycle with 1 smoothing step on each visit to each grid level. Try weighted Jacobi and Gauss-Seidel as the smoothing step. Try several different values of the mesh spacing h and show that you achieve convergence to a fixed tolerance in a number of cycles that is independent of the mesh size.

Solution 3. I've implemented this in Julia using weighted Jacobi smoothing. Code can be found in the appendix. We see that for the fixed tolerance 0.001. I find that for various grid sizes $M = 150, \dots, 1000$ and the corresponding h it takes around 5 cycles to converge within the tolerance. For some intermediate M values this goes up to 7 but there doesn't appear to be a strong dependence on the mesh size for the number of cycles needed to reach a fixed tolerance.

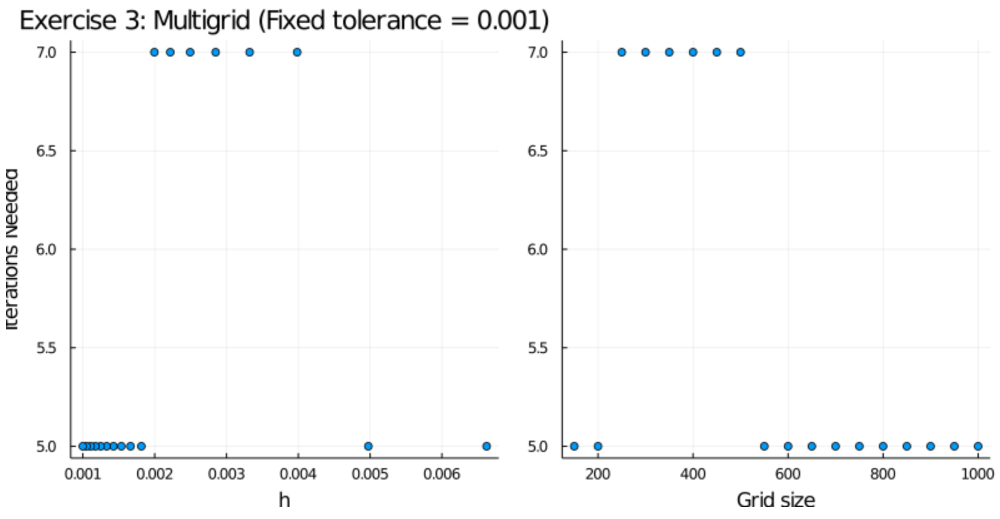


Figure 2

Exercise 4. (a) Consider an iteration of the form

$$x_k = x_{k-1} + M^{-1}(b - Ax_{k-1}),$$

for solving a nonsingular linear system $Ax = b$. Note that the error $e_k := A^{-1}b - x_k$ satisfies

$$e_k = (I - M^{-1}A)e_{k-1} = \ldots = (I - M^{-1}A)^k e_0.$$

Assume that $\|e_0\|_2 = 1$ and that $\|I - M^{-1}A\|_2 = \frac{1}{2}$. Estimate the number of iterations required to reduce the 2-norm of the error below 2^{-20}. Show how you obtain your estimate. Now suppose you know only that the spectral radius $\rho(I - M^{-1}A) = \frac{1}{2}$. Can you give an estimate of the number of iterations required to reduce the 2-norm of the error below 2^{-20}? Explain why or why not.

(b) Consider the GMRES algorithm applied to an n by n matrix A with the sparsity pattern pictured below:

$$\begin{bmatrix} * & * & \cdots & * & * \\ * & * & \cdots & * & 0 \\ 0 & * & \cdots & * & 0 \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & \cdots & \cdots & * & 0 \end{bmatrix},$$

where the $*$'s represent nonzero entries. Show that if the initial residual is the nth unit vector $(0, \ldots, 0, 1)^T$, then the algorithm makes no progress until step n. Show that a matrix with this sparsity pattern can have *any* eigenvalues. Conclude that eigenvalue information alone cannot be enough to ensure fast convergence of the GMRES algorithm.

Solution 4. We will bound e_k using bounds on $I - M^{-1}A$. We know that

$$||e_k||_2 \leq \left|\left|I - M^{-1}A\right|\right|_2 ||e_{k-1}||_2 \leq \left|\left|I - M^{-1}A\right|\right|_2^k ||e_0||_2 = 2^{-k}.$$

With this estimate, we estimate that the it would take at most 20 iterations to reach 2^{-20} accuracy. If we only know the spectral radius, the rate of convergence will additionally depend on the condition number of the eigenvalue matrix. We can see this as follows. Assume that we can eigen-decompose $I - M^{-1}A$ as follows

$$(I - M^{-1}A) = R^{-1}\Lambda R^{-1}$$

Then we would have that

$$||e_k||_2 \leq \left|\left|\Lambda^k\right|\right|_2 ||R||_2 \left|\left|R^{-1}\right|\right|_2 ||e_0||_2$$

As Λ is diagonal, we have that

$$||e_k||_2 \leq \rho^k ||R||_2 \left|\left|R^{-1}\right|\right|_2 ||e_0||_2 = 2^{-k} ||R||_2 \left|\left|R^{-1}\right|\right|_2,$$

so that ρ alone is not enough enough to make an estimate unless we know the condition number of R i.e. $\kappa(R) = ||R||_2 \, ||R^{-1}||_2$.

(b) If $r_0 = q_1 = e_n$, then $Aq_1 = a_{1,n}e_1$ where e_i is the i unit vector. It then follows that $(v, q_1) = h_{11} = 0$ and $q_2 = e_1$. We then have that $v = Aq_2 = a_{1,1}e_1 + a_{1,2}e_2$. We see that for the previous q_2, $(v, q_2) = h_{22} = a_{11}$ and $(v, q_1) = h_{21} = 0$, so that $q_3 = e_2$. This process continues with $q_n = e_{n-1}$ as q_n must be orthogonal to the previous q_k. This gives the matrix Q_k which essentially just a permutation matrix

$$Q_k = [e_n, e_1, \dots, e_{k-1}]$$

and the Hessenberg matrix H_k is also just a swapping of the first $k-1$ columns and the last column of A. We then have that

$$r_k - r_0 = -Q_{k+1}H_k y_k.$$

As the first row of the Hesenberg matrix H_k is 0 until $k = n$, the least squares solution will return the vector

$$y_k = \min ||\eta - H_k y||_2$$

which is 0 when η is defined as 4.72 since the valid solution is all vectors of form αe_1 and 0 is the least squares among these.
To see this matrix has non-zero eigenvalues, we simply pick constants λ_i and place them on the open diagonal spot. We then put 1 in the upper right hand corner. This matrix is upper triangular and will have its eigenvalues be the chosen λ_i. This shows that independent of its eigenvalues a matrix with this sparsity pattern cannot have fast convergence of GMRES, so eigenvalues alone are not alone a determinant of convergence rate.

HW-6-Code-Figgins

March 10, 2021

[1]:
```
using Plots, LinearAlgebra, IterativeSolvers
```

0.1 Exercise 2

[329]:
```
function make_A(M)
   # Step size
   h = 1/(M+1)

    # Xjs of interest
    xjs = [j*h for j in 1:M]

    # Defining diagonal
    diagA = [ -2 for xj in xjs]
    diagB = [ 1 for xj in xjs[1:M-1]]

    ##Building Matrix
    A = Array(Tridiagonal(diagB, diagA, diagB))
    #A[end, [M, M-1, M-2]] .= 3*h/2, -2*h, h/2
    A = A ./ h^2

    return A
end

# Generate MxM approximation
M = 255
A = make_A(M)
```

[329]:
```
255×255 Array{Float64,2}:
 -131072.0    65536.0         0.0  …       0.0        0.0        0.0
   65536.0  -131072.0     65536.0          0.0        0.0        0.0
       0.0    65536.0  -131072.0           0.0        0.0        0.0
       0.0        0.0     65536.0          0.0        0.0        0.0
       0.0        0.0         0.0          0.0        0.0        0.0
       0.0        0.0         0.0  …       0.0        0.0        0.0
       0.0        0.0         0.0          0.0        0.0        0.0
       0.0        0.0         0.0          0.0        0.0        0.0
       0.0        0.0         0.0          0.0        0.0        0.0
       0.0       0.0       0.0             0.0       0.0       0.0
       0.0       0.0       0.0   …         0.0       0.0       0.0
       0.0       0.0       0.0             0.0       0.0       0.0
       0.0       0.0       0.0             0.0       0.0       0.0

       0.0       0.0       0.0             0.0       0.0       0.0
       0.0       0.0       0.0             0.0       0.0       0.0
       0.0       0.0       0.0   …         0.0       0.0       0.0
       0.0       0.0       0.0             0.0       0.0       0.0
       0.0       0.0       0.0             0.0       0.0       0.0
       0.0       0.0       0.0             0.0       0.0       0.0
       0.0       0.0       0.0             0.0       0.0       0.0
       0.0       0.0       0.0   …         0.0       0.0       0.0
       0.0       0.0       0.0         65536.0       0.0       0.0
       0.0       0.0       0.0       -131072.0   65536.0       0.0
       0.0       0.0       0.0         65536.0 -131072.0   65536.0
       0.0       0.0       0.0             0.0   65536.0 -131072.0
```

[330]:
```
function make_F(M, f,   = 0,   = 0)
    # Step size
    h = 1/(M+1)

    # Xjs of interest
    xjs = [j*h for j in 1:M]

    F = [ f(xj) for xj in xjs]
    F[1] -=   / h^2
    F[end] -=   / h^2
    return F
end

 (x) = 20 *x^3
 _p(x) = 60 *x^2
 _pp(x) = 120 *x
a = 0.5

f(x) = -20 + a* _pp(x)*cos( (x)) - a*( _p(x))^2*sin( (x))
F = make_F(M, f, 1, 3)
```

[330]:
```
255-element Array{Float64,1}:
 -65555.26368922183
     -18.527378446273538
     -17.791067710615692
     -17.0547572256601
     -16.31844772269389
     -15.582141112779148
     -14.845841562428523
```

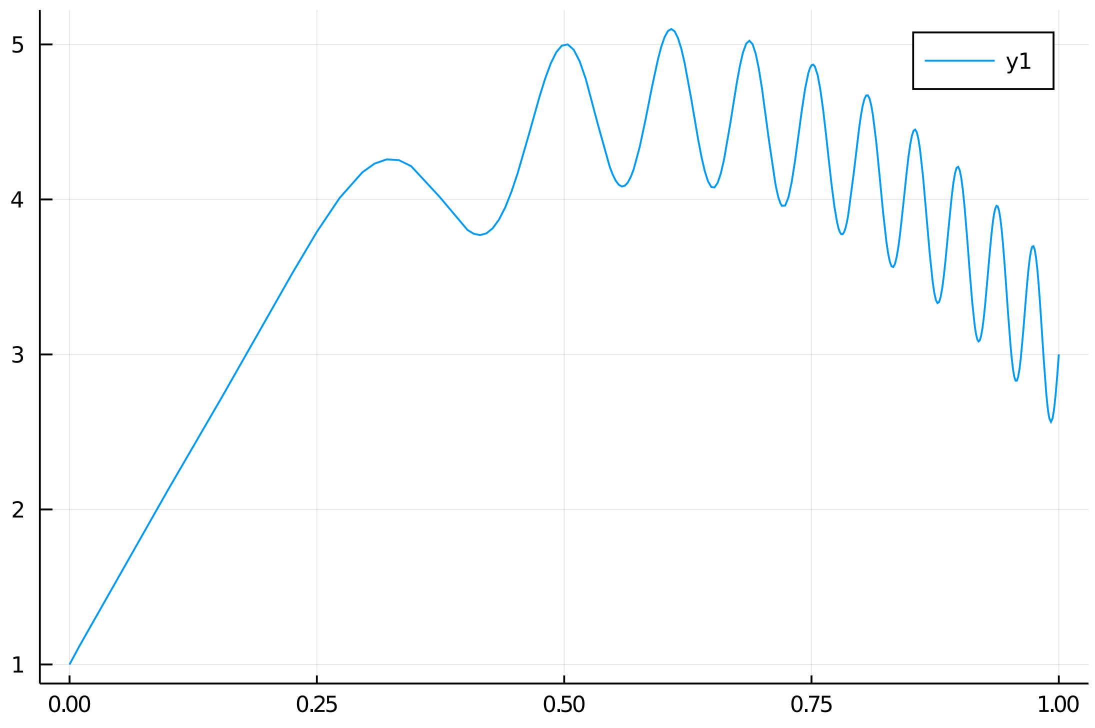
```
    -14.109557089764055
    -13.373301785243724
    -12.637098761020129
    -11.900983932953629
    -11.165010739222906
    -10.429255899335724

  12193.68059167691
  14829.756818078598
  10931.344580565343
   1902.3557347421106
  -8374.04615570515
 -15145.663490619392
 -14990.391565451891
  -7598.550065824561
   3799.4018909742267
  13740.686498107276
  17113.666305882147
-184780.6754116434
```

[331]:
```
true_u(x) = 1 + 12*x - 10*x^2 + a*sin( (x))
```

[331]:
```
true_u (generic function with 1 method)
```

[332]:
```
plot(x -> true_u(x), 0, 1)
```

[332]:



[333]:
```
function get_error(u, uhat, xjs)
    error = (u.(xjs) .- uhat)
    return abs.(error)
end

function get_L2_error(u, uhat, xjs)
    error = get_error(u, uhat, xjs)
    h = 1/ (1 + length(xjs))

    error = h*(error .^2)
    error = sum(error)
    return (error)
end
```

[333]:
```
get_L2_error (generic function with 2 methods)
```

[334]:
```
function get_error(u, uhat)
    h = 1/ (1 + length(uhat))
    M = length(uhat)

    error = [u.(h*i) .- uhat[i] for i in 1:M]
    return abs.(error)
end

function get_L2_error(u, uhat)
    error = get_error(u, uhat)
    h = 1/ (1 + length(uhat))

    error = h*(error .^2)
    error = sum(error)
    return (error)
end
```

[334]:
```
get_L2_error (generic function with 2 methods)
```

[335]:
```
function GS_iter(A, b, u0, max_iter)
    # Defining M and N
    M = LowerTriangular(A)
    N = UpperTriangular(-A)
    N[diagind(N)] .= 0.0

    # Initializing storage for u
    u_iter = Vector{Vector{Float64}}(undef, max_iter+1)
    u_iter[1] = u0

    c = M \ b

    for iter in 2:(max_iter+1)
        u_iter[iter] = M \ (N*u_iter[iter-1]) + c
    end
   return u_iter
end
```

[335]:
```
GS_iter (generic function with 1 method)
```

[336]:
```
h = 1/(M+1)
xjs = [j*h for j in 1:M]
u0 = [ 1 + 2*xj for xj in xjs]
```

[336]:
```
255-element Array{Float64,1}:
 1.0078125
 1.015625
 1.0234375
 1.03125
 1.0390625
 1.046875
 1.0546875
 1.0625
 1.0703125
 1.078125
 1.0859375
 1.09375
 1.1015625

 2.90625
 2.9140625
 2.921875
 2.9296875
 2.9375
 2.9453125
 2.953125
 2.9609375
 2.96875
 2.9765625
 2.984375
 2.9921875
```

[337]:
```
GS = GS_iter(A, F, u0, 10^4)
GS_0 = GS[1]
GS_5 = GS[6]
GS_10 = GS[11]
GS_20 = GS[20]
```

```
GS_big = GS[end];
```

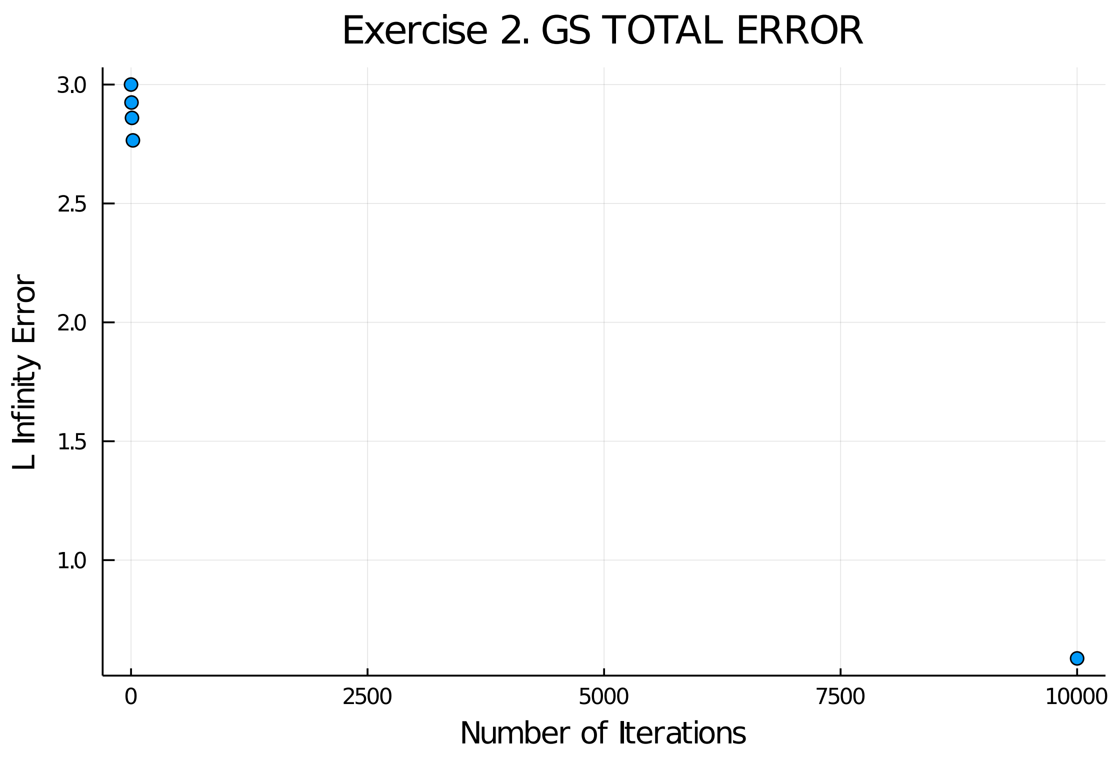
[338]:
```
exer_2_GS_sol = plot(x -> true_u(x), 0, 1,
    title = "Exercise 2. Gauss-Seidel",
    label = "True Solution",
    color = "yellow")
exer_2_GS_sol = plot!(xjs, GS_0,
    label = "0 iterations",
    color = "blue")
exer_2_GS_sol = plot!(xjs, GS_5,
    label = "5 iterations",
    color = "green")
exer_2_GS_sol = plot!(xjs, GS_10,
    label = "10 iterations",
    color = "orange")
exer_2_GS_sol = plot!(xjs, GS_20,
    label = "20 iterations",
    color = "pink")
exer_2_GS_sol = plot!(xjs, GS_big,
    label = "10^4 iterations",
    color = "red",
    alpha = 0.5,
    linestyle = :dash);
```

[339]:
```
GS_0_e = get_error(true_u, GS_0, xjs)
GS_5_e = get_error(true_u, GS_5, xjs)
GS_10_e = get_error(true_u, GS_10, xjs)
GS_20_e = get_error(true_u, GS_20, xjs)
GS_big_e = get_error(true_u, GS_big, xjs);
```

[340]:
```
exer_2_GS_e = plot(x -> 0, 0, 1,
    title = "Exercise 2. GS ERROR",
    label = false,
    color = "yellow")
exer_2_GS_e = plot!(xjs, GS_0_e,
    label = "0 iterations",
    color = "blue")
exer_2_GS_e = plot!(xjs, GS_5_e,
    label = "5 iterations",
    color = "green")
exer_2_GS_e = plot!(xjs, GS_10_e,
    label = "10 iterations",
    color = "orange")
exer_2_GS_e = plot!(xjs, GS_20_e,
    label = "20 iterations",
    color = "pink")
exer_2_GS_e = plot!(xjs, GS_big_e,
    label = "10^4 iterations",
    color = "red",
    alpha = 0.5,
    linestyle = :dash);
```

[341]:
```
GS_tot_e = [maximum(GS_0_e), maximum(GS_5_e), maximum(GS_10_e),␣
 ↪maximum(GS_20_e), maximum(GS_big_e)]
iters = [0, 5, 10, 20, 10^4]

exer_2_GS_tot_e = scatter(iters, GS_tot_e,
    title = "Exercise 2. GS TOTAL ERROR",
    label = false,
    ylabel = "L Infinity Error",
    xlabel = "Number of Iterations")
```

[341]:



[342]:
```
CG_0 = u0
CG_5 = cg(A, F, maxiter =5)
CG_10 = cg(A, F, maxiter =10)
CG_20 = cg(A, F, maxiter =20)
CG_big, hist = cg(A, log = true, F);

CG_0_e = get_error(true_u, CG_0, xjs)
CG_5_e = get_error(true_u, CG_5, xjs)
```

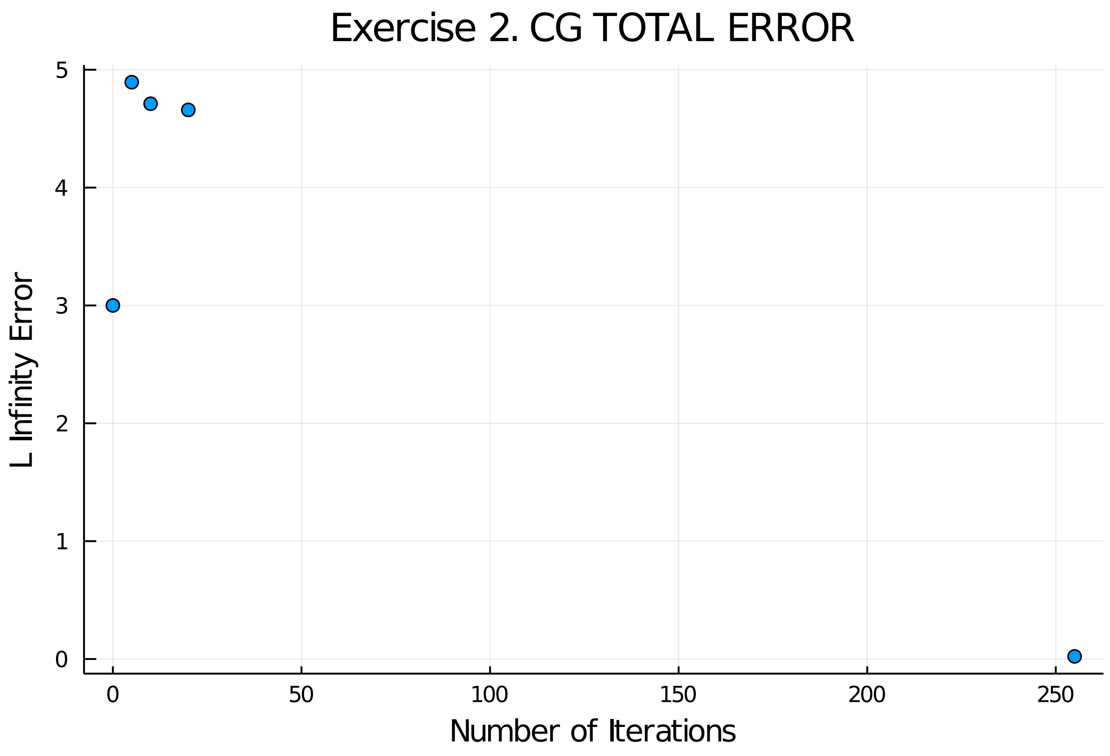
```
CG_10_e = get_error(true_u, CG_10, xjs)
CG_20_e = get_error(true_u, CG_20, xjs)
CG_big_e = get_error(true_u, CG_big, xjs);
```

[343]:
```
exer_2_CG_sol = plot(x -> true_u(x), 0, 1,
    title = "Exercise 2. CG",
    label = "True Solution",
    color = "yellow")
exer_2_CG_sol = plot!(xjs, CG_0,
    label = "0 iterations",
    color = "blue")
exer_2_CG_sol = plot!(xjs, CG_5,
    label = "5 iterations",
    color = "green")
exer_2_CG_sol = plot!(xjs, CG_10,
    label = "10 iterations",
    color = "orange")
exer_2_CG_sol = plot!(xjs, CG_20,
    label = "20 iterations",
    color = "pink")
exer_2_CG_sol = plot!(xjs, CG_big,
    label = "$(hist.iters) iterations",
    color = "red",
    alpha = 0.5,
    linestyle = :dash);
```

[344]:
```
exer_2_CG_e = plot(x -> 0, 0, 1,
    title = "Exercise 2. CG ERROR",
    label = false,
    color = "yellow")
exer_2_CG_e = plot!(xjs, CG_0_e,
    label = "0 iterations",
    color = "blue")
exer_2_CG_e = plot!(xjs, CG_5_e,
    label = "5 iterations",
    color = "green")
exer_2_CG_e = plot!(xjs, CG_10_e,
    label = "10 iterations",
    color = "orange")
exer_2_CG_e = plot!(xjs, CG_20_e,
    label = "20 iterations",
    color = "pink")
exer_2_CG_e = plot!(xjs, CG_big_e,
    label = "$(hist.iters) iterations",
    color = "red",
    alpha = 0.5,
    linestyle = :dash);
```

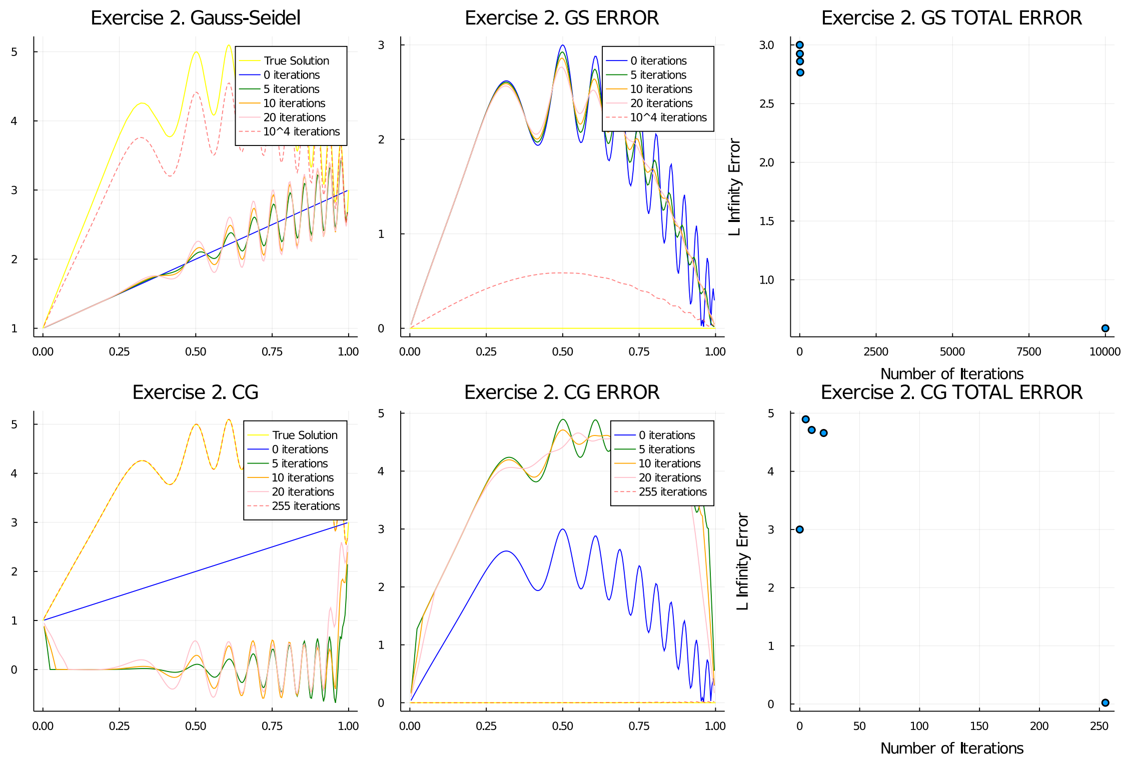
[345]:
```
CG_tot_e = [maximum(CG_0_e), maximum(CG_5_e), maximum(CG_10_e),␣
 ↪maximum(CG_20_e), maximum(CG_big_e)]
iters = [0, 5, 10, 20, hist.iters]

exer_2_CG_tot_e = scatter(iters, CG_tot_e,
    title = "Exercise 2. CG TOTAL ERROR",
    label = false,
    ylabel = "L Infinity Error",
    xlabel = "Number of Iterations")
```

[345]:



[347]:
```
exer_2 = plot( exer_2_GS_sol, exer_2_GS_e,exer_2_GS_tot_e, exer_2_CG_sol,␣
 ↪exer_2_CG_e,  exer_2_CG_tot_e, layout = (2, 3), size = (1200, 800))
```

[347]:



```
[348]: savefig(exer_2, "../hw/figs/hw-6-exer-2-gs-cg.png")
```

0.2 Exercise 3

```
[133]: A = make_A(20)
```

```
[133]: 20×20 Array{Float64,2}:
        -882.0    441.0      0.0      0.0      0.0   …     0.0      0.0      0.0      0.0
         441.0   -882.0    441.0      0.0      0.0         0.0      0.0      0.0      0.0
           0.0    441.0   -882.0    441.0      0.0         0.0      0.0      0.0      0.0
           0.0      0.0    441.0   -882.0    441.0         0.0      0.0      0.0      0.0
           0.0      0.0      0.0    441.0   -882.0         0.0      0.0      0.0      0.0
           0.0      0.0      0.0      0.0    441.0   …     0.0      0.0      0.0      0.0
           0.0      0.0      0.0      0.0      0.0         0.0      0.0      0.0      0.0
           0.0      0.0      0.0      0.0      0.0         0.0      0.0      0.0      0.0
           0.0      0.0      0.0      0.0      0.0         0.0      0.0      0.0      0.0
           0.0      0.0      0.0      0.0      0.0         0.0      0.0      0.0      0.0
           0.0      0.0      0.0      0.0      0.0   …     0.0      0.0      0.0      0.0
           0.0      0.0      0.0      0.0      0.0         0.0      0.0      0.0      0.0
           0.0      0.0      0.0      0.0      0.0         0.0      0.0      0.0      0.0
           0.0      0.0      0.0      0.0      0.0         0.0      0.0      0.0      0.0
           0.0      0.0      0.0      0.0      0.0         0.0      0.0      0.0      0.0
           0.0      0.0      0.0      0.0      0.0   …   441.0      0.0      0.0      0.0
```

```
 0.0      0.0      0.0      0.0      0.0      -882.0   441.0      0.0      0.0
 0.0      0.0      0.0      0.0      0.0       441.0  -882.0    441.0      0.0
 0.0      0.0      0.0      0.0      0.0         0.0   441.0   -882.0    441.0
 0.0      0.0      0.0      0.0      0.0         0.0     0.0    441.0   -882.0
```

[283]:
```
function interpolate_coarse(u_coarse)
M_coarse = length(u_coarse)
u_fine = zeros(2*M_coarse)

for i in 1:M_coarse
    u_fine[2*i] = u_coarse[i]
end

for i in 1:(M_coarse-1)
    u_fine[2*i + 1] = (u_coarse[i+1] + u_coarse[i])/2
end
    u_fine[1] = u_coarse[1]/2
    return u_fine
end

function restrict_fine(u_fine)
    M_fine = length(u_fine)
    M_coarse = div(M_fine, 2)
    u_coarse = zeros(M_coarse)

    for i in 1:M_coarse
       u_coarse[i] = u_fine[2*i]
    end
    return u_coarse
end
```

[283]:
```
restrict_fine (generic function with 1 method)
```

[284]:
```
M = 2^6
h = 1/(M+1)
xjs = [j*h for j in 1:M]
u0 = [ 1 + 2*xj for xj in xjs]
```

[284]:
```
64-element Array{Float64,1}:
 1.0307692307692307
 1.0615384615384615
 1.0923076923076924
 1.123076923076923
 1.1538461538461537
 1.1846153846153846
 1.2153846153846155
 1.2461538461538462
```

```
1.2769230769230768
1.3076923076923077
1.3384615384615386
1.3692307692307693
1.4

2.6307692307692307
2.661538461538462
2.6923076923076925
2.723076923076923
2.753846153846154
2.7846153846153845
2.8153846153846156
2.8461538461538463
2.876923076923077
2.907692307692308
2.9384615384615387
2.9692307692307693
```

[285]:
```
function under_jac(A, b, u0, max_iter;   = 2/3)
  # Defining M and N
    M = (1/ )*diag(A)
    #M = diag(A)
    N = Diagonal(M) - A

    # Initializing storage for u
    u_iter = Vector{Vector{Float64}}(undef, max_iter+1)
    u_iter[1] = u0

    for iter in 2:(max_iter+1)
        u_iter[iter] = (1 ./ M) .* (N*u_iter[iter-1] + b)
        #u_iter[iter] = (1 -  )*u_iter[iter-1] +  *(1 ./ M) .* (N*u_iter[iter-1]␣
 ↪+ b)
    end
   return u_iter
end
```

[285]:
```
under_jac (generic function with 2 methods)
```

[286]:
```
function full_multi_grid(v, n_levels, M, method)
    # Grid sizes
    g_sizes = [Int(M / 2^(level)) for level in 0:(n_levels-1)]

    # Down recurse

    # Make proper size A
    A = [make_A(level) for level in g_sizes]
    F = [make_F(level, f, 1, 3) for level in g_sizes]
    #A = make_A(g_sizes[1])

    u = [1 + 2*x / (g_sizes[end] + 1) for x in 1:g_sizes[end]]
    for level in reverse(1:n_levels)
        u = method(A[level], F[level], u, v)[end]

        if level > 1
            u = interpolate_coarse(u)
        end
    end
    return u
end


#u = full_multi_grid(500, 6, M, under_jac)
```

[286]:
```
full_multi_grid (generic function with 1 method)
```

[287]:
```
function multi_grid(v, n_cycles, n_levels, M, u, method)
    # Grid sizes
    g_sizes = [Int(M / 2^(level)) for level in 0:(n_levels-1)]

    # Down recurse

    # Make proper size A

    A = [make_A(level) for level in g_sizes]
    F = [make_F(level, f, 1, 3) for level in g_sizes]

    e = Vector{Vector{Float64}}(undef, n_levels)
    r = Vector{Vector{Float64}}(undef, n_levels)
    ef = Vector{Vector{Float64}}(undef, n_levels)

    for k in 1:n_cycles
        # Compute Initial Residual
        r[1] = F[1] - A[1]*u
        # Compute initial error
        e[1] = method(A[1], r[1], zeros(g_sizes[1]), v)[end]

        for level in 2:(n_levels-1)
            # Project residual to smaller grid
            ef[level] = restrict_fine(r[level-1])
            # Solve for error
            e[level] = method(A[level], ef[level], zeros(g_sizes[level]),␣
 ↪v)[end]
```

```
            # Update residual with new error
            r[level] = ef[level] - A[level]*e[level]
        end

        # Solving on coarsest grid
        ef[end] = restrict_fine(r[end-1])
        d = A[end] \ ef[end]

        for level in reverse(2:n_levels-1)
            d = interpolate_coarse(d) + e[level]
            d = method(A[level],  ef[level], d, v)[end]
        end

        d = interpolate_coarse(d) + e[1]

        u += d
        u = method(A[1], F[1], u, v)[end]
    end
    return u
end

#u = multi_grid(3, 2, 2, M, u0, GS_iter)
```

[287]:
```
multi_grid (generic function with 1 method)
```

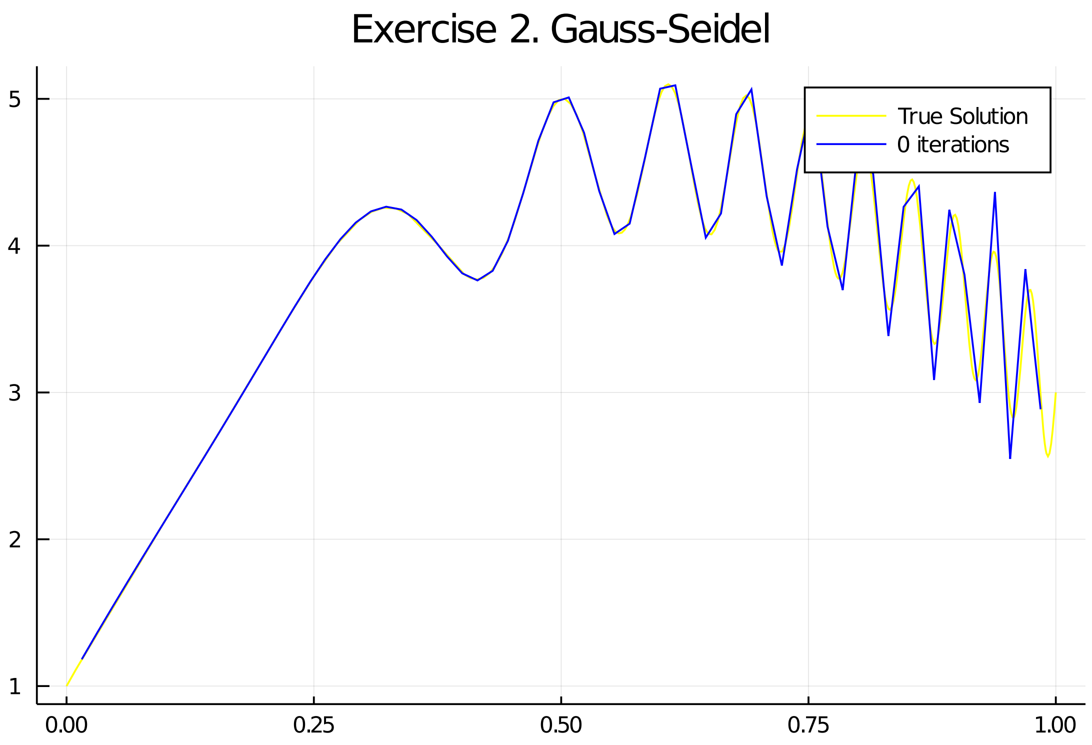
[288]:
```
function two_grid(v, n_cycles, M, u, method)
    # Grid sizes
    g_sizes = [Int(M / 2^(level)) for level in 0:1]

    # Down recurse

    # Make proper size A

    A = [make_A(level) for level in g_sizes]
    F = [make_F(level, f, 1, 3) for level in g_sizes]

    e = Vector{Vector{Float64}}(undef, 2)
    r = Vector{Vector{Float64}}(undef, 2)

    # Pre-smooth
    u = method(A[1], F[1], u, v)[end]

    for k in 1:n_cycles
        # Compute Initial Residual
        r[1] = F[1] - A[1]*u

        # Project residual to smaller grid
        r[2] = restrict_fine(r[1])

        # Solve for error on coarse grid
        e[2] = method(A[2], r[2], zeros(g_sizes[2]), v)[end]
        # e[2] = A[2] \ r[2]

        # Scale up to fine grid
        e[1] = interpolate_coarse(e[2])

        # Update approximate solution
        u = u + e[1]

        # post-smoothing
        u = method(A[1], F[1], u, v)[end]
    end
    return u
end

M = 2^6
h = 1/(M+1)
xjs = [j*h for j in 1:M]
u0 = [ 1 + 2*xj for xj in xjs]

u = two_grid(3, 500, M, u0, under_jac)
```

[288]:
```
64-element Array{Float64,1}:
 1.1823098249956907
 1.360573296255838
 1.5354777777606623
 1.7077104874097102
 1.8779577678212769
 2.046902531390069
 2.2152176261697933
 2.3835513576752727
 2.552499770593132
 2.7225582467410594
 2.8940432609209457
 3.066973715054326
 3.2409017606154307

 4.488631620391491
 3.3845972335922885
 4.264616285145567
 4.403618904148157
 3.084924304395966
 4.244338677900736
 3.8011039852564577
```

```
2.9290505436670067
4.366049212042762
2.5476065565802823
3.8393363868419095
2.885561166863964
```

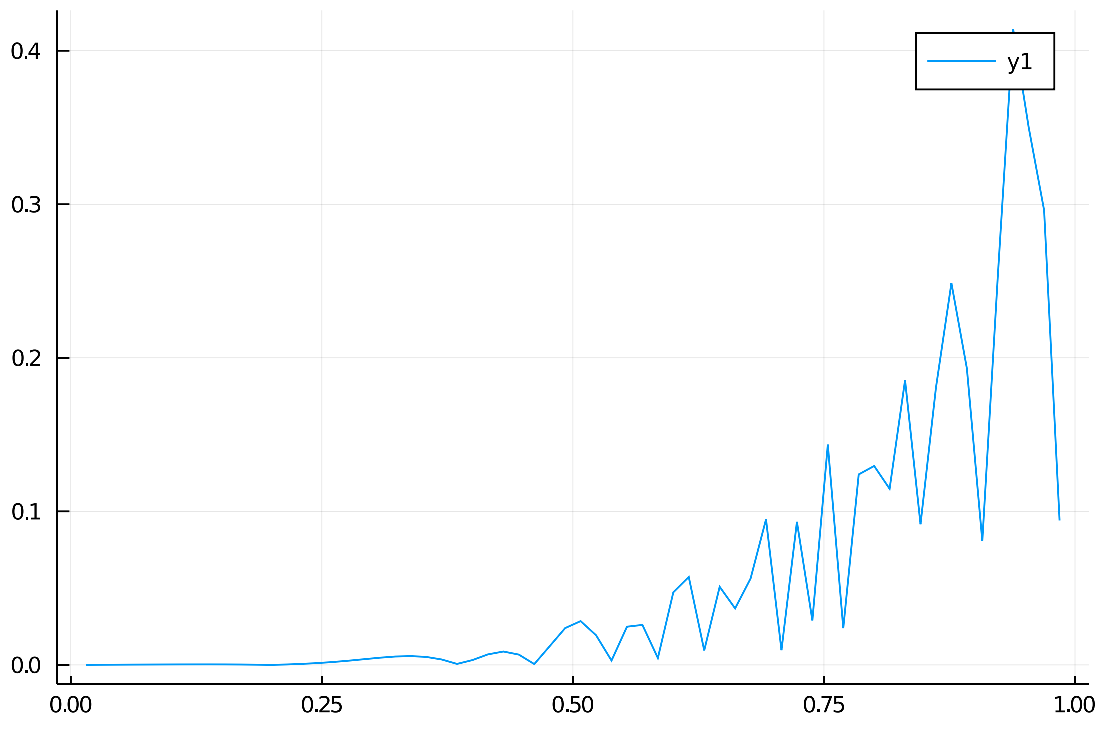
[289]:
```
plot(x -> true_u(x), 0, 1,
    title = "Exercise 2. Gauss-Seidel",
    label = "True Solution",
    color = "yellow")
exer_2_GS_sol = plot!(xjs, u,
    label = "0 iterations",
    color = "blue")
```

[289]:



[290]:
```
plot(xjs, get_error(true_u, u, xjs))
```

[290]:



[291]:
```
get_L2_error(true_u, u, xjs)
```

[291]:
```
0.011333526512984454
```

[292]:
```
## Two Grid multi grid v cycle
```

[293]:
```
M_vec = [2^n for n in 3:10]
tol = []

## Loop over n get error
for M in M_vec
    h = 1/(M+1)
    xjs = [j*h for j in 1:M]
    u0 = [1 + 2*xj for xj in xjs]
    u = two_grid(3, 100, M, u0, GS_iter)
    push!(tol, sqrt(sum(h*(get_error(true_u, u, xjs)).^2) ))
end
tol
```

[293]:
```
8-element Array{Any,1}:
 104.49336087281534
  39.7816864331069
   9.430721803389329
```

```
0.11649345729731984
0.7761438065517052
1.5002734788792298
1.7761647167771497
1.855996734892714
```

[294]:
```
function multi_grid_fixed_tol(v, n_levels, M, u, method, tol = 0.001, maxiter =␣
↪200)
    # Grid sizes
    g_sizes = [Int(M / 2^(level)) for level in 0:(n_levels-1)]

    h = 1/(M+1)
    xjs = [j*h for j in 1:M]

    # Make proper size A

    A = [make_A(level) for level in g_sizes]
    F = [make_F(level, f, 1, 3) for level in g_sizes]

    e = Vector{Vector{Float64}}(undef, n_levels)
    r = Vector{Vector{Float64}}(undef, n_levels)
    ef = Vector{Vector{Float64}}(undef, n_levels)

    n_cycles = 0
    while n_cycles < maxiter
        # Compute Initial Residual
        r[1] = F[1] - A[1]*u
        # Compute initial error
        e[1] = method(A[1], r[1], zeros(g_sizes[1]), v)[end]

        for level in 2:(n_levels-1)
            # Project residual to smaller grid
            ef[level] = restrict_fine(r[level-1])
            # Solve for error
            e[level] = method(A[level], ef[level], zeros(g_sizes[level]),␣
↪v)[end]
            # Update residual with new error
            r[level] = ef[level] - A[level]*e[level]
        end

        # Solving on coarsest grid
        ef[end] = restrict_fine(r[end-1])
        d = A[end] \ ef[end]

        for level in reverse(2:n_levels-1)
            d = interpolate_coarse(d) + e[level]
            d = method(A[level],  ef[level], d, v)[end]
        end

        d = interpolate_coarse(d) + e[1]

        u += d
        u = method(A[1], F[1], u, v)[end]

        n_cycles += 1

        if maximum(get_error(true_u, u, xjs)) < tol
            return n_cycles
        end

    end
    return n_cycles
end


function two_grid_fixed_tol(v, M, u, method, tol = 0.001, maxiter = 200)
    # Grid sizes
    g_sizes = [Int(M / 2^(level)) for level in 0:1]

    h = 1/(M+1)
    xjs = [j*h for j in 1:M]

    # Make proper size A

    A = [make_A(level) for level in g_sizes]
    F = [make_F(level, f, 1, 3) for level in g_sizes]

    e = Vector{Vector{Float64}}(undef, 2)
    r = Vector{Vector{Float64}}(undef, 2)

    # Pre-smooth
    #u = method(A[1], F[1], u, v)[end]

    n_cycles = 0
    while n_cycles < maxiter
        # Compute Initial Residual
        r[1] = F[1] - A[1]*u

        # Project residual to smaller grid
        r[2] = restrict_fine(r[1])

        # Solve for error on coarse grid
        #e[2] = method(A[2], r[2], zeros(g_sizes[2]), v)[end]
        e[2] = A[2] \ r[2]
        # Scale up to fine grid
        e[1] = interpolate_coarse(e[2])

        # Update approximate solution
        u = u + e[1]

        # post-smoothing
        u = method(A[1], F[1], u, v)[end]

        n_cycles += 1

        #println(get_L2_error(true_u, u))
        if get_L2_error(true_u, u) < tol
            return n_cycles
        end
    end
    return n_cycles
end
```

[294]:
```
two_grid_fixed_tol (generic function with 3 methods)
```

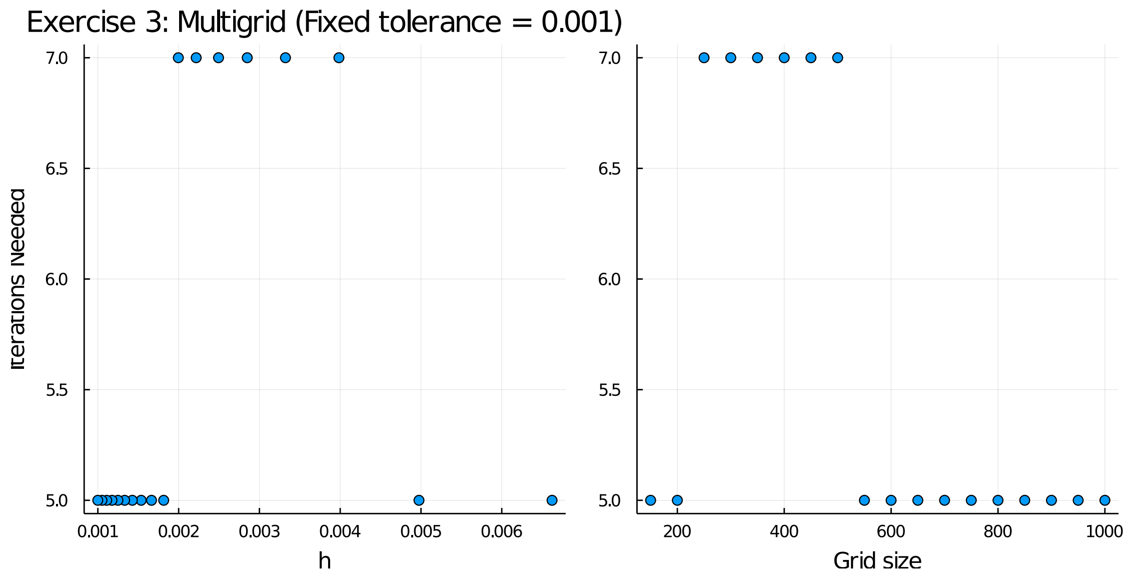
[322]:
```
M_vec = [50*n for n in 3:20]
cycles_needed = []

## Loop over n get error
for M in M_vec
    h = 1/(M+1)
    xjs = [j*h for j in 1:M]
    u0 = [1 + 2*xj for xj in xjs]

     push!(cycles_needed,
        two_grid_fixed_tol(1, M, u0, under_jac, 0.001, 20))
end
cycles_needed
```

[322]:
```
18-element Array{Any,1}:
 5
 5
 7
 7
 7
 7
 7
 7
 5
 5
```

```
5
5
5
5
5
5
5
5
```

[327]:
```
exer_3_iters_needed_1 = scatter(1 ./ (M_vec .+ 1),
    cycles_needed,
    label = false,
    ylabel = "Iterations Needed",
    xlabel = "h",
    title = "Exercise 3: Multigrid (Fixed tolerance = 0.001)")

exer_3_iters_needed_2 = scatter(M_vec,
    cycles_needed,
    label = false,
    xlabel = "Grid size")


exer_3_iters_needed = plot(exer_3_iters_needed_1, exer_3_iters_needed_2, size =␣
 ↪(800, 400))
```

[327]:



[328]:
```
savefig(exer_3_iters_needed, "../hw/figs/hw-6-exer-3-multigrid.png")
```