Iterative methods - 1D elliptic problem

Taken from R.J.LeVeque: Finite Difference Methods for Ordinary and Partial Differential Equations - Steady-State and Time-Dependent Problems, SIAM, Philadephia 2007, chapter 4.6

Preliminaries

Clear all symbols from previous evaluations to avoid conflicts

```
In[1]:= Clear["Global`*"]
```

Problem

Differential equation

We would like to solve numerically the differential equation

$$\frac{d^2 u(x)}{dx^2} = f(x) \tag{1}$$

with Dirichlet boundary conditions

$$\begin{aligned} u(a) &= u_a \\ u(b) &= u_b \end{aligned} \tag{2}$$

Particular problem

As the right-hand-side we will take

$$f(x) = -20 + c\,\phi''(x)\cos(\phi(x)) - c(\phi'(x))^2 \sin(\phi(x)) \tag{3}$$

where

$c = 1/2$

$\phi(x) = 20\,\pi\,x^3$

and boundary conditions are

$$\begin{aligned} u(0) &= 1 \\ u(1) &= 3 \end{aligned} \tag{4}$$

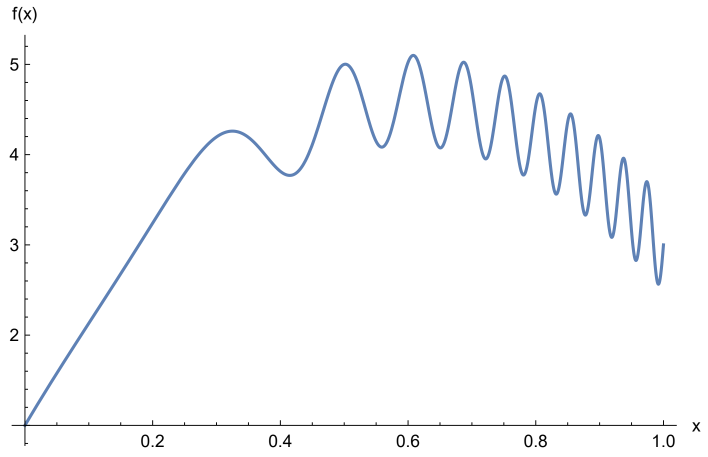
The analytic solution

This problem can be solved in the closed form

```
In[2]:= a = 0; b = 1; u_a = 1; u_b = 3; c = 1 / 2;
        ϕ[x_] = 20 π x^3;
        f[x_] = -20 + c ∂_{x,x}ϕ[x] Cos[ϕ[x]] - c (∂_x ϕ[x])^2 Sin[ϕ[x]];
        (* Delete[...,0] deletes outer {} of a list *)
        sol = Delete[DSolve[{∂_{x,x}U[x] == f[x], U[a] == u_a, U[b] == u_b}, U[x], x], 0];
        u[x_] = Expand[U[x] /. sol]
        Plot[u[x], {x, a, b}, AxesLabel → {"x", "f(x)"}]
```

Out[6]= $1 + 12\,x - 10\,x^2 + \frac{1}{2}\,\text{Sin}\left[20\,\pi\,x^3\right]$

Out[7]=



Direct numerical solution

We discretize the equation (1) using a standard finite difference formula on the equidistant grid

$x_j = j\,h$ where $h = 1/(n+1)$, $j = 0, \ldots, n+1$

i.e. we have to solve the system of n equations

$$\frac{d^2 u(x_j)}{dx^2} \simeq \frac{u(x_{j+1}) - 2\,u(x_j) + u(x_{j-1})}{h^2} = f(x_j) \;\text{ for } j = 1, \ldots, n \tag{5}$$

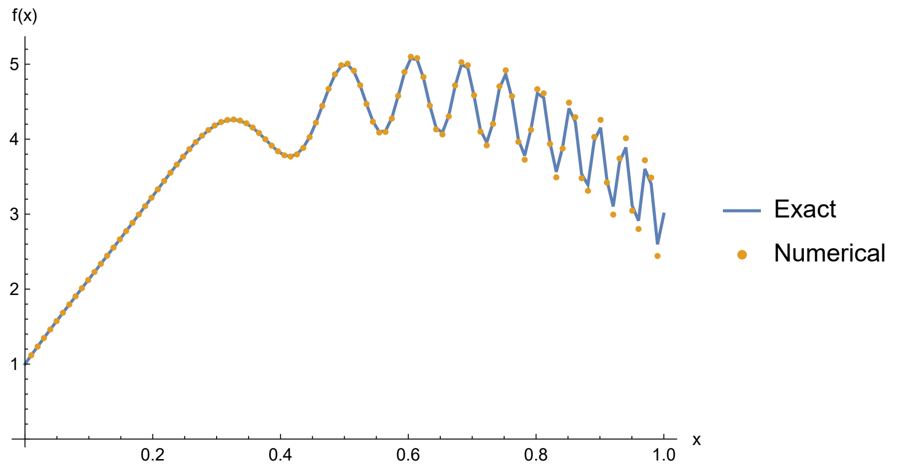
with boundary conditions

$u(x_0 = 0) = 1$, $u(x_{N+1} = 1) = 3$

We can use *Mathematica* to solve this problem directly. For later convenience in gradient methods we actually solve

$$-\frac{d^2 u(x_j)}{dx^2} \simeq \frac{-u(x_{j+1}) + 2\,u(x_j) - u(x_{j-1})}{h^2} = -f(x_j) \;\text{ for } j = 1, \ldots, n$$

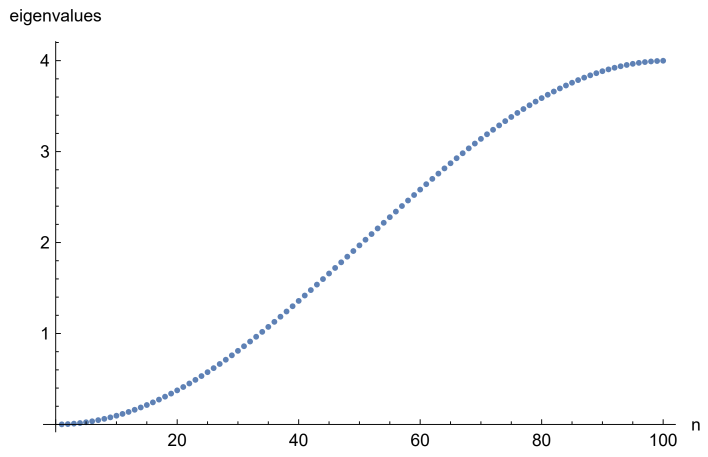
```
In[8]:= n = 100; h = (b - a) / (n + 1);
        X = Range[a, b, h]; Xin = X⟦2 ;; n + 1⟧;
        T = SparseArray[{{i_, i_} → 2.0, {i_, j_} /; Abs[i - j] == 1 → -1.0}, {n, n}];
        rhs = -N[h^2 f[X⟦2 ;; n + 1⟧], 16];
        rhs⟦1⟧ = rhs⟦1⟧ + u_a; (* boundary condtions *)
        rhs⟦n⟧ = rhs⟦n⟧ + u_b;
        xDirect = LinearSolve[T, rhs];
        Print["MaxError = ", Max[Abs[u[X⟦2 ;; n + 1⟧] - xDirect]]]
        exactSol = Transpose[{X, u[X]}]; (* for later use in plots *)
        ListPlot[{exactSol, Transpose[{Xin, xDirect}]}, Joined → {True, False},
         PlotLegends → {"Exact", "Numerical"}, AxesLabel → {"x", "f(x)"}]

        MaxError = 0.155090745751

Out[17]=
```



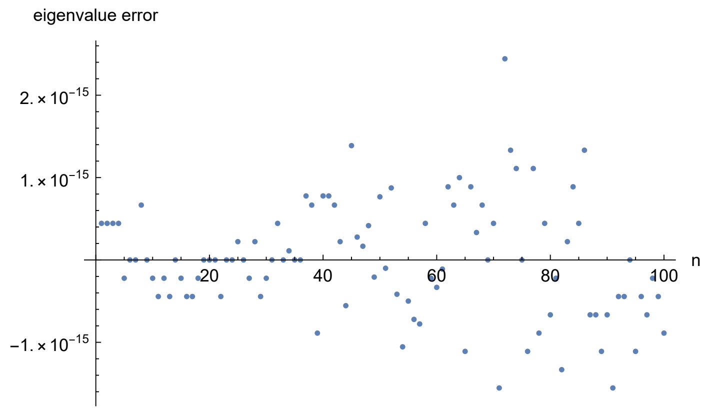
The matrix is symmetric positive definite, all eigenvalues are positive, all methods should converge

In[18]:=
```
eval = Sort[Eigenvalues[T]];
ListPlot[eval, AxesLabel → {"n", "eigenvalues"}]
ListPlot[{eval - (2 - 2 Cos[π Range[n] / (n + 1)])}, AxesLabel → {"n", "eigenvalue error"}]
```

Out[19]=



Out[20]=



Basic iterative methods

In general, we solve a system of linear equations

$$A x = b \tag{6}$$

and use a standard decomposition

$$A = D - \tilde{L} - \tilde{U} = D(I - L - U) \tag{7}$$

where D is the diagonal of A,

$-\tilde{L}$ and $-\tilde{U}$ are the strictly lower and upper triangular parts respectively and $L = D^{-1}\tilde{L}$, $U = D^{-1}\tilde{U}$,

More generally will write $A = M - K$.

One step of the general iterative method can be written as

$$x_{m+1} = R x_m + c \text{ where } R = M^{-1} K,\ c = M^{-1} b \tag{8}$$

and the method is convergent if and only if the spectral radius of the matrix R

$\rho(R) = \max \mid \lambda_i \mid$ where λ_i are eigenvalues of R (9)

satisfies

$\rho(R) < 1.$ (10)

For our particular problem (1)-(4) we use the same discretization (5) on the equidistant grid as in the direct method and compare iterative solutions with the solution obtained by the direct method. The standard decomposition is (note that we have $A = T$ and $b = rhs$ and we set $D = Diag$):

```
In[21]:= Diag = DiagonalMatrix[Diagonal[T]];
         D1 = Inverse[Diag];
         L = -D1.LowerTriangularize[T, -1];
         U = -D1.UpperTriangularize[T, 1];
         Id = IdentityMatrix[n];
         (* MatrixForm[U] *)
```

An initial guess and the maximum number of iterations will be the same for all methods. As the initial guess we will take the linear function satisfying the boundary conditions (4).

```
In[26]:= u0[x_] = InterpolatingPolynomial[{{a, ua}, {b, ub}}, x]
         x0 = N[u0[X[[2 ;; n + 1]]], 16];
Out[26]=
         1 + 2 x
```

And here is a common part of all basic iterative algorithms. It is supposed that n_{iter}, x_0 and x_{direct} are already assigned.

```
In[28]:= basicIterativeMethod[matrixR_, vectorC_, niter_] :=
          Module[{x, e},
           x = x0;
           e = ConstantArray[1.0 × 10^-20, niter + 1];
           e[[1]] = Max[Abs[x - xDirect]];
           Do[
            x = matrixR.x + vectorC;
            e[[i + 1]] = Max[Abs[x - xDirect]],
            {i, 1, niter}
           ];
           {x, e}
          ]
```

Jacobi method

One step of the Jacobi method is

$$x_{m+1,j} = \frac{1}{a_{jj}}\left(b_j - \sum_{k\neq j} a_{jk}\, x_{m,k}\right), j = 1, \ldots, n \tag{11}$$

or in a matrix form

$$x_{m+1} = R_{\text{Jac}}\, x_m + c_{\text{Jac}} \text{ where } R_{\text{Jac}} = D^{-1}(\tilde{L} + \tilde{U}) = L + U \text{ and } c_{\text{Jac}} = D^{-1}\, b \tag{12}$$

Properties of R_{Jac} for the problem (1)-(4):

In[29]:=
```
RJac = L + U;
Print["Part 5x5 of RJac:"]
MatrixForm[RJac[[1 ;; 5, 1 ;; 5]]]
cJac = D1.rhs;
(* MatrixForm[R] *)
eigenJac = Eigenvalues[RJac];
ρJac = Max[Abs[eigenJac]];
Print["Spectral radius ρ(RJac) = ", ρJac]
ListPlot[Sort[eigenJac],
 PlotLegends → {"Eigenvalues of RJac"}, AxesLabel → {"n", "eigenvalues"}]
```

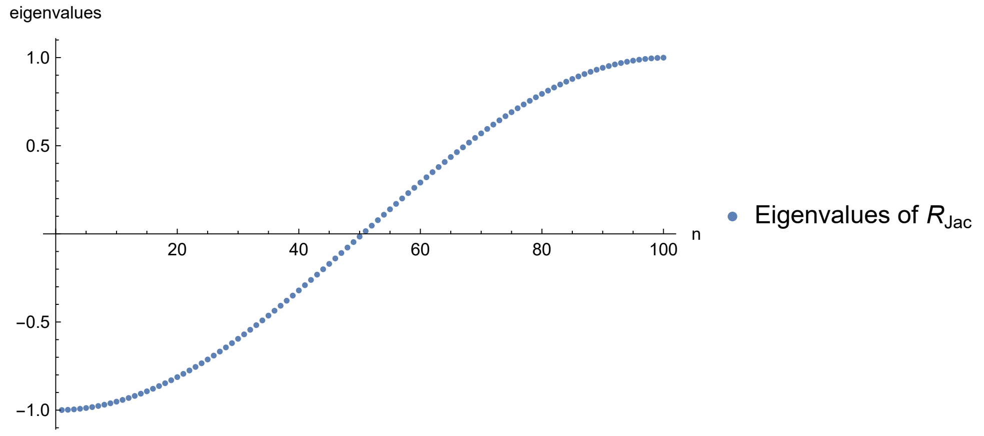
```
Part 5x5 of RJac:
```

Out[31]//MatrixForm=

$$\begin{pmatrix} 0. & 0.5 & 0. & 0. & 0. \\ 0.5 & 0. & 0.5 & 0. & 0. \\ 0. & 0.5 & 0. & 0.5 & 0. \\ 0. & 0. & 0.5 & 0. & 0.5 \\ 0. & 0. & 0. & 0.5 & 0. \end{pmatrix}$$

```
Spectral radius ρ(RJac) = 0.999516282292
```

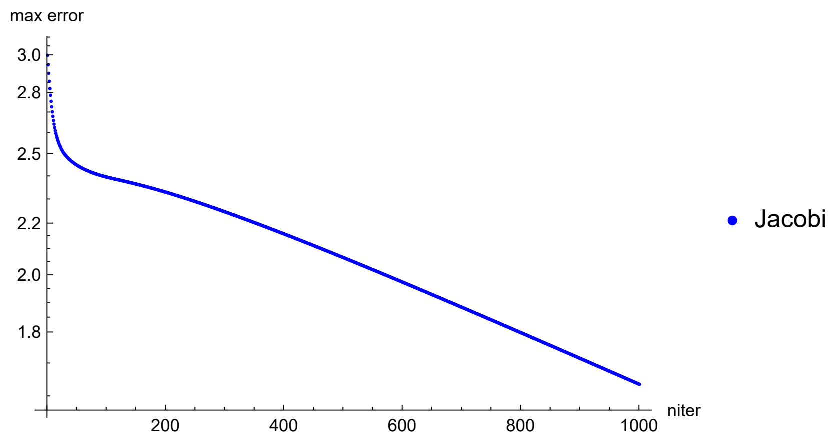
Out[36]=



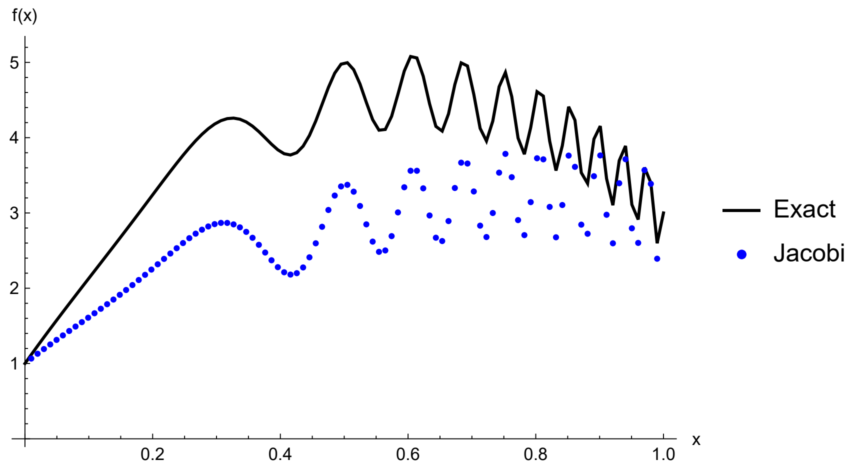
The main algorithm of the Jacobi method for our problem (1)-(4). In each step we save the maximum error of the solution.

In[37]:=
```
niter = 1000;
{xIter, errorJac} = basicIterativeMethod[RJac, cJac, niter];
xJac = Transpose[{Xin, xIter}]; (* for later use in plots *)
(* Plot results *)
ListLogPlot[{errorJac}, PlotRange → All, PlotStyle → {Blue},
 PlotLegends → {"Jacobi"}, AxesLabel → {"niter", "max error"}]
ListPlot[{exactSol, xJac}, Joined → {True, False}, PlotStyle → {Black, Blue},
 PlotLegends → {"Exact", "Jacobi"}, AxesLabel → {"x", "f(x)"}]
```

Out[40]=



Out[41]=



Gauss-Seidel method

One step of the Gauss-Seidel method is

$$x_{m+1,j} = \frac{1}{a_{jj}}\left(b_j - \sum_{k=1}^{j-1} a_{jk}\, x_{m+1,k} - \sum_{k=j+1}^{n} a_{jk}\, x_{m,k}\right), j = 1, \ldots, n \qquad (13)$$

or in a matrix form

$$x_{m+1} = R_{GS}\, x_m + c_{GS} \text{ where } R_{GS} = \left(D - \tilde{L}\right)^{-1} \tilde{U} = (I - L)^{-1} U \text{ and } c_{GS} = \left(D - \tilde{L}\right)^{-1} b = (I - L)^{-1} D^{-1} b \qquad (14)$$

Properties of R_{GS} for the problem (1)-(4):

```
In[42]:= ImL1 = Inverse[Id - L];
         RGS = ImL1.U;
         Print["Part 5x5 of RGS:"]
         MatrixForm[RGS[[1 ;; 5, 1 ;; 5]]]
         cGS = ImL1.D1.rhs;
         (*MatrixForm[R]*)
         eigenGS = Eigenvalues[RGS];
         ρGS = Max[Abs[eigenGS]];
         Print["Spectral radius ρ(RGS) = ", ρGS]
         ListPlot[Sort[Re[eigenGS]],
          PlotLegends → {"Eigenvalues of RGS"}, AxesLabel → {"n", "eigenvalues"}]
```

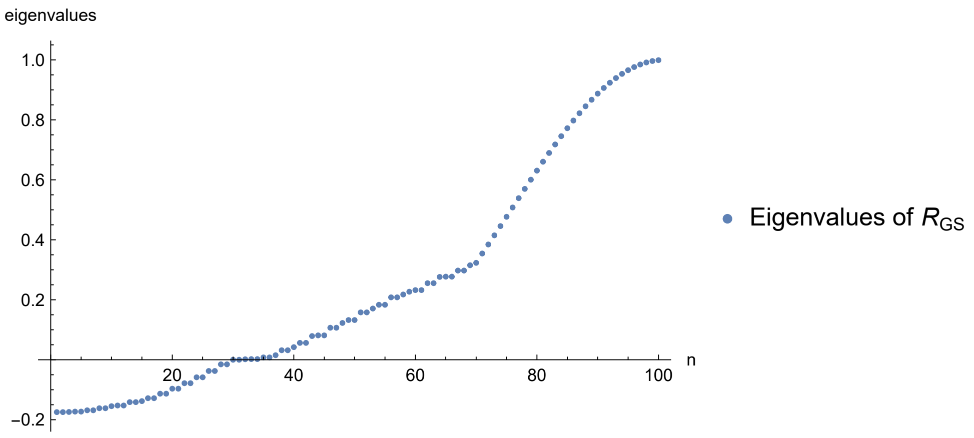
```
Part 5x5 of RGS:
```

Out[45]//MatrixForm=

$$\begin{pmatrix} 0. & 0.5 & 0. & 0. & 0. \\ 0. & 0.25 & 0.5 & 0. & 0. \\ 0. & 0.125 & 0.25 & 0.5 & 0. \\ 0. & 0.0625 & 0.125 & 0.25 & 0.5 \\ 0. & 0.03125 & 0.0625 & 0.125 & 0.25 \end{pmatrix}$$

```
Spectral radius ρ(RGS) = 0.999032798567
```

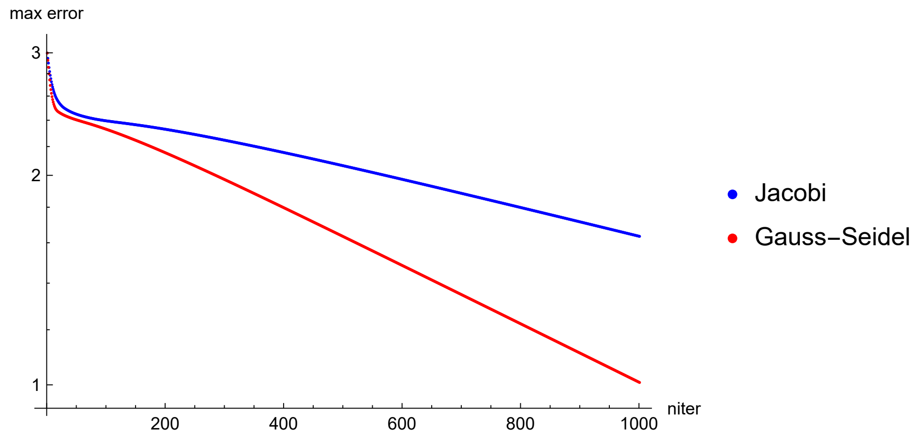
Out[50]=



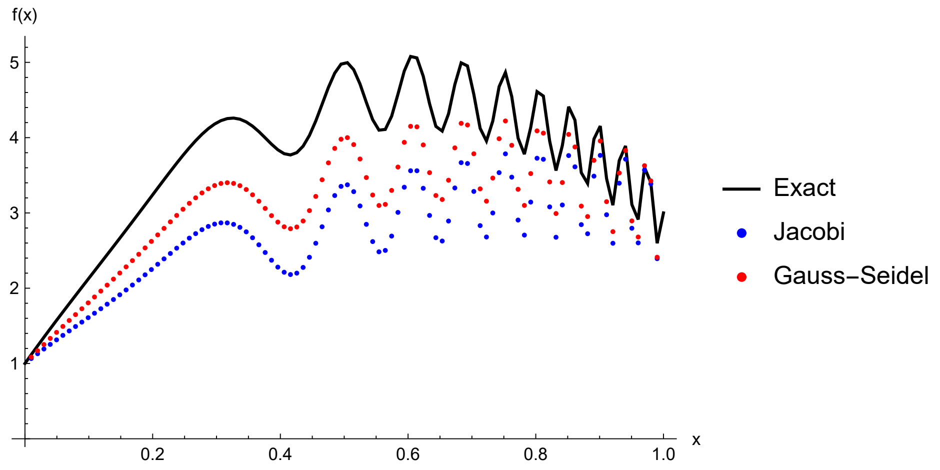
The main algorithm of the Jacobi method for our problem (1)-(4). In each step we save the maximum error of the solution.

```
In[51]:= {xIter, errorGS} = basicIterativeMethod[RGS, cGS, niter];
         xGS = Transpose[{Xin, xIter}]; (* for later use in plots *)
         (* Plot results *)
         ListLogPlot[{errorJac, errorGS}, PlotRange → All, PlotStyle → {Blue, Red},
          PlotLegends → {"Jacobi", "Gauss-Seidel"}, AxesLabel → {"niter", "max error"}]
         ListPlot[{exactSol, xJac, xGS},
          Joined → {True, False, False}, PlotStyle → {Black, Blue, Red},
          PlotLegends → {"Exact", "Jacobi", "Gauss-Seidel"}, AxesLabel → {"x", "f(x)"}]
```

Out[53]=



Out[54]=



Successive overrelaxation method

One step of the SOR(ω) method is

$$x_{m+1,j} = (1-\omega)\, x_{m,j} + \frac{\omega}{a_{jj}}\left(b_j - \sum_{k=1}^{j-1} a_{jk}\, x_{m+1,k} - \sum_{k=j+1}^{n} a_{jk}\, x_{m,k}\right), j = 1, \ldots, n \tag{15}$$

or in a matrix form

$$x_{m+1} = R_{\mathrm{SOR}}\, x_m + c_{\mathrm{SOR}} \quad \text{where } R_{\mathrm{SOR}} = \left(D - \omega\,\tilde{L}\right)^{-1}\left[(1-\omega)\, D + \omega\,\tilde{U}\right] = (I - \omega\, L)^{-1}\left[(1-\omega)\, I + \omega\, U\right]$$
$$\text{and } c_{\mathrm{GS}} = \omega\left(D - \omega\,\tilde{L}\right)^{-1} b = \omega(I - \omega\, L)^{-1}\, D^{-1}\, b \tag{16}$$

Properties of R_{SOR} for the problem (1)-(4) for an optimal certain ω:

```
In[55]:= ω = 2 / (1 + Sqrt[1 - ρJac²]);
         Print["ω = ", ω]
         ImL1 = Inverse[Id - ω L];
         RSOR = ImL1.((1 - ω) Id + ω U);
         Print["Part 5x5 of R_SOR:"]
         MatrixForm[RSOR⟦1 ;; 5, 1 ;; 5⟧]
         cSOR = ω ImL1.D1.rhs;
         (*MatrixForm[R]*)
         eigenSOR = Eigenvalues[RSOR];
         ρSOR = Max[Abs[eigenSOR]];
         Print["Spectral radius ρ(R_SOR) = ", ρSOR]
         ListPlot[Sort[Re[eigenSOR]], PlotLegends → {"Eigenvalues of SOR"}]
```

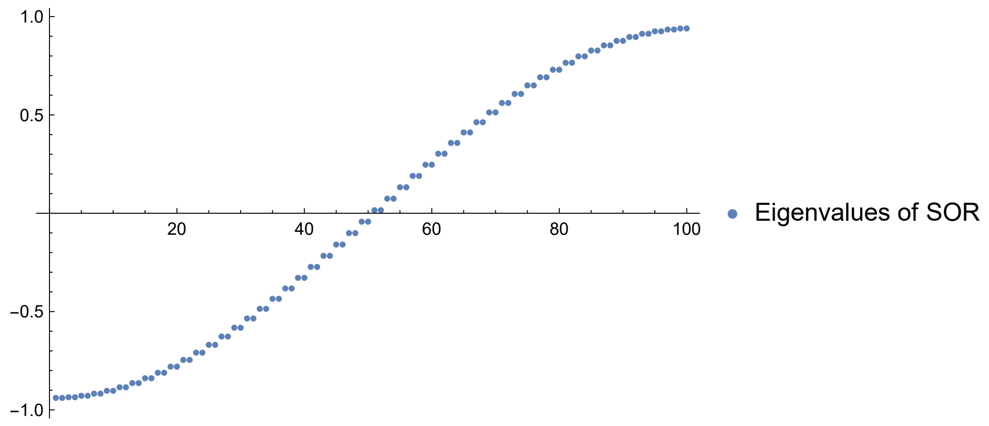
```
ω = 1.93967633319

Part 5x5 of R_SOR:
```

Out[60]//MatrixForm=

$$\begin{pmatrix} -0.93967633319 & 0.969838166595 & 0. & 0. & 0. \\ -0.911333972173 & 0.000909736194359 & 0.969838166595 & 0. & 0. \\ -0.883846468728 & 0.000882296882822 & 0.000909736194359 & 0.969838166595 & 0. \\ -0.857188038783 & 0.000855685191229 & 0.000882296882822 & 0.000909736194359 & 0.969838166595 \\ -0.83133367596 & 0.000829876157043 & 0.000855685191229 & 0.000882296882822 & 0.000909736194359 \end{pmatrix}$$

```
Spectral radius ρ(R_SOR) = 0.939676335419
```

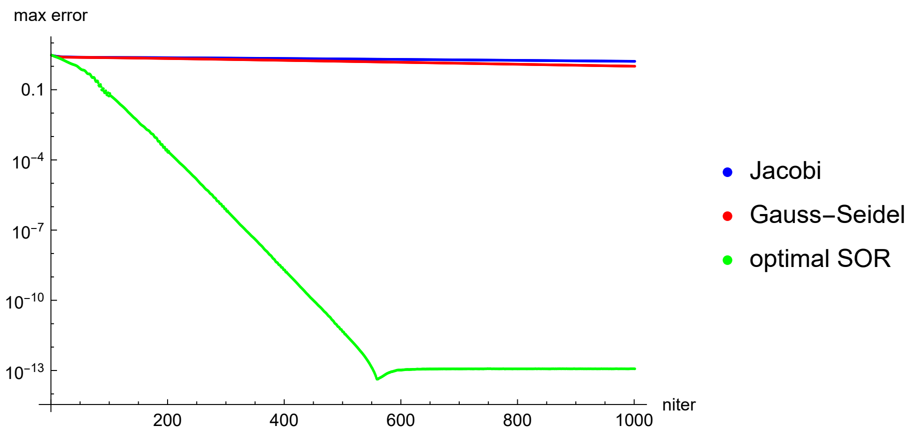
Out[65]=



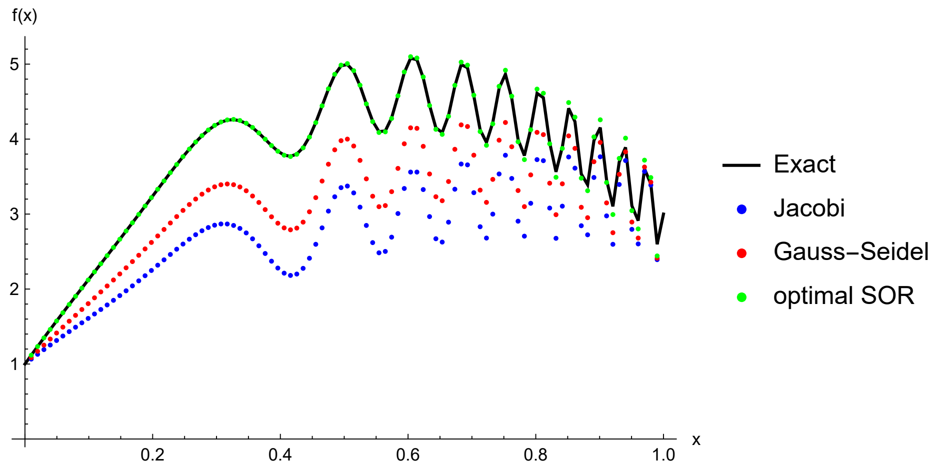
The main algorithm of the Jacobi method for our problem (1)-(4). In each step we save the maximum error of the solution.

In[66]:=
```
{xIter, errorSORopt} = basicIterativeMethod[RSOR, cSOR, niter];
xSORopt = Transpose[{Xin, xIter}]; (* for later use in plots *)
(* Plot results *)
ListLogPlot[{errorJac, errorGS, errorSORopt}, PlotRange → All,
 PlotStyle → {Blue, Red, Green}, PlotLegends → {"Jacobi", "Gauss-Seidel", "optimal SOR"},
 AxesLabel → {"niter", "max error"}]
ListPlot[{exactSol, xJac, xGS, xSORopt},
 Joined → {True, False, False, False}, PlotStyle → {Black, Blue, Red, Green},
 PlotLegends → {"Exact", "Jacobi", "Gauss-Seidel", "optimal SOR"},
 AxesLabel → {"x", "f(x)"}]
```

Out[68]=

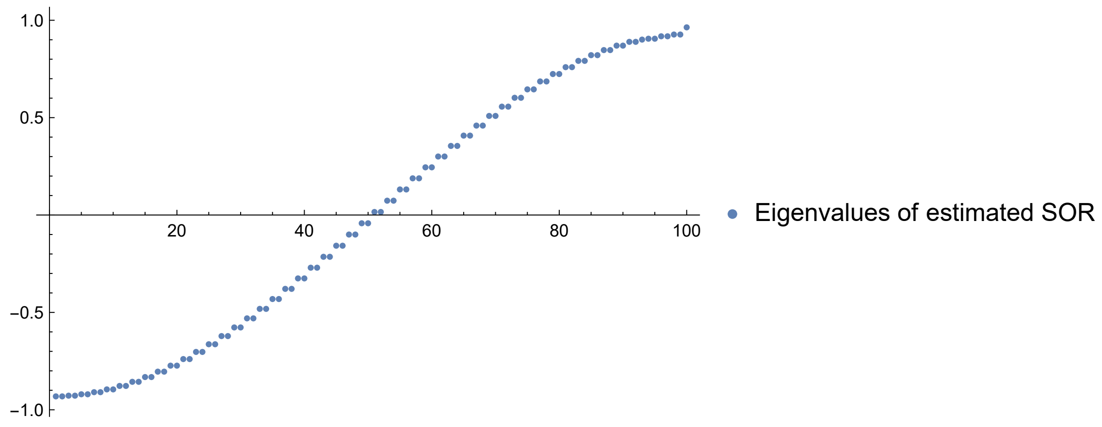


Out[69]=



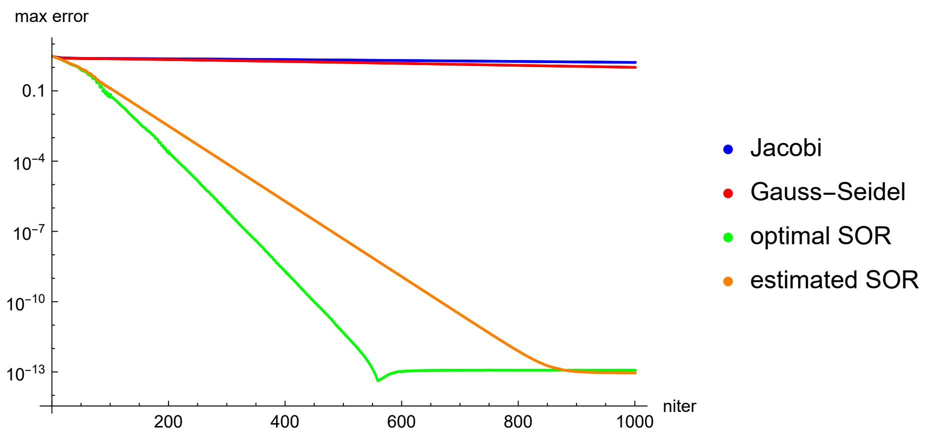
Properties of R_{SOR} for the problem (1)-(4) for an estimated ω:

```
In[70]:= Print["ρ(R_Jac) = ", ρJac]
         (* Determination of ρ(R_Jac) from the error *)
         nit1 = 50; nit2 = 60;
         ρJacEst = Exp[(Log[errorJac⟦nit2⟧] - Log[errorJac⟦nit1⟧]) / (nit2 - nit1)];
         Print["estimated ρ(R_Jac) = ", ρJacEst]
         (* Use this value in SOR algorithm *)
         ω = 2 / (1 + Sqrt[1 - ρJacEst^2]);
         Print["ω = ", ω]
         ImL1 = Inverse[Id - ω L];
         RSORest = ImL1.((1 - ω) Id + ω U);
         cSORest = ω ImL1.D1.rhs;
         (*MatrixForm[R]*)
         eigenSORest = Eigenvalues[RSORest];
         ρSORest = Max[Abs[eigenSORest]];
         Print["estimated ρ(R_SOR) = ", ρSORest, " (Optimal ρ(R_SOR) = ", ρSOR, ")"]
         ListPlot[Sort[Re[eigenSORest]], PlotLegends → {"Eigenvalues of estimated SOR"}]

         ρ(R_Jac) = 0.999516282292

         estimated ρ(R_Jac) = 0.99937385529

         ω = 1.93165390915

         estimated ρ(R_SOR) = 0.963648703027 (Optimal ρ(R_SOR) = 0.939676335419)

Out[82]=
```

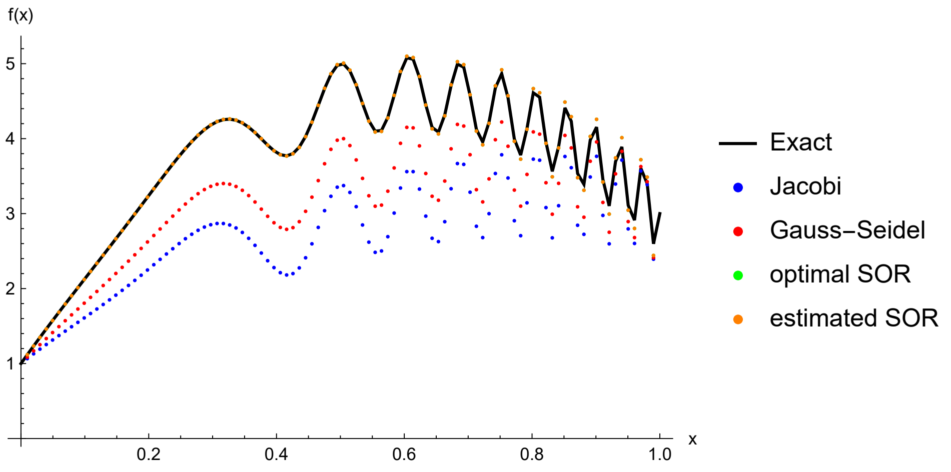


In[83]:=
```
{xIter, errorSORest} = basicIterativeMethod[RSORest, cSORest, niter];
xSORest = Transpose[{Xin, xIter}]; (* for later use in plots *)
(* Plot results *)
ListLogPlot[{errorJac, errorGS, errorSORopt, errorSORest},
 PlotRange → All, PlotStyle → {Blue, Red, Green, Orange},
 PlotLegends → {"Jacobi", "Gauss-Seidel", "optimal SOR", "estimated SOR"},
 AxesLabel → {"niter", "max error"}]
ListPlot[{exactSol, xJac, xGS, xSORopt, xSORest},
 Joined → {True, False, False, False, False}, PlotStyle → {Black, Blue, Red, Green, Orange},
 PlotLegends → {"Exact", "Jacobi", "Gauss-Seidel", "optimal SOR", "estimated SOR"},
 AxesLabel → {"x", "f(x)"}]
```

Out[85]=



Out[86]=



Gradient iterative methods

Let us solve a system of linear equations

$$A x = b \tag{17}$$

with a symmetric positive definite matrix A by searching a minimum of the function

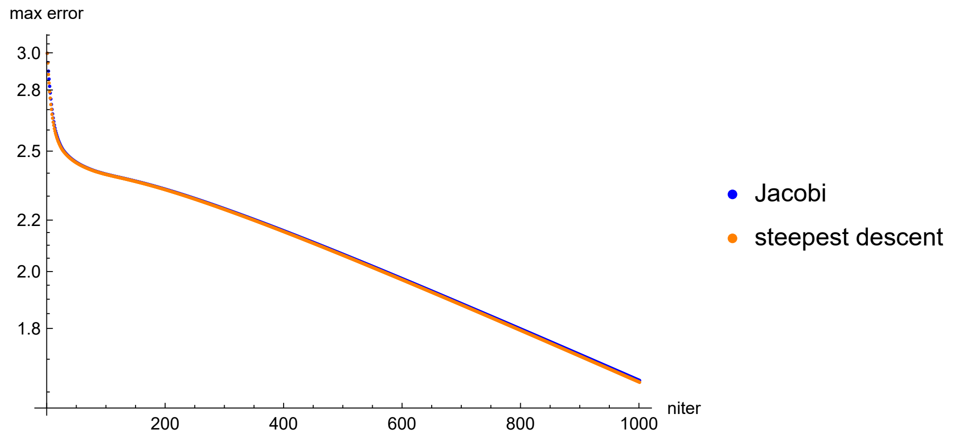
$$\phi(x) = \frac{1}{2} x^T A x - x^T b \tag{18}$$

An initial guess and the maximum number of iterations will be the same as in basic iterative method.

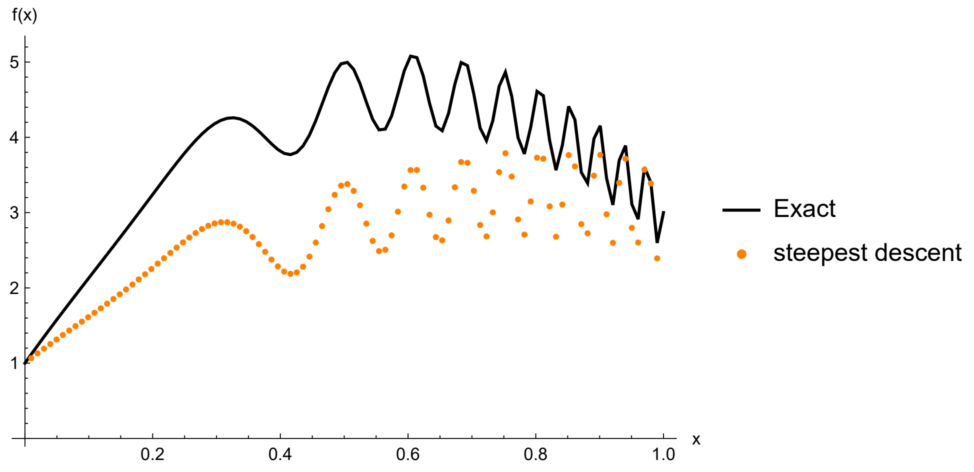
Method of the steepest descent

In[87]:=
```
xIter = x0;
r = rhs - T.x0;
errorMSD = ConstantArray[1.0 × 10^-50, niter + 1];
errorMSD[[1]] = Max[Abs[xIter - xDirect]];
Do[
 w = T.r;
 α = r.r / r.w;
 xIter = xIter + α r;
 r = r - α w;
 errorMSD[[i + 1]] = Max[Abs[xIter - xDirect]],
 {i, 1, niter}
]
xMSD = Transpose[{Xin, xIter}]; (* for later use in plots *)
(* Plot results *)
ListLogPlot[{errorJac, errorMSD}, PlotRange → All, PlotStyle → {Blue, Orange},
 PlotLegends → {"Jacobi", "steepest descent"}, AxesLabel → {"niter", "max error"}]
ListPlot[{exactSol, xMSD}, PlotStyle → {Black, Orange}, Joined → {True, False},
 PlotLegends → {"Exact", "steepest descent"}, AxesLabel → {"x", "f(x)"}]
```

Out[93]=



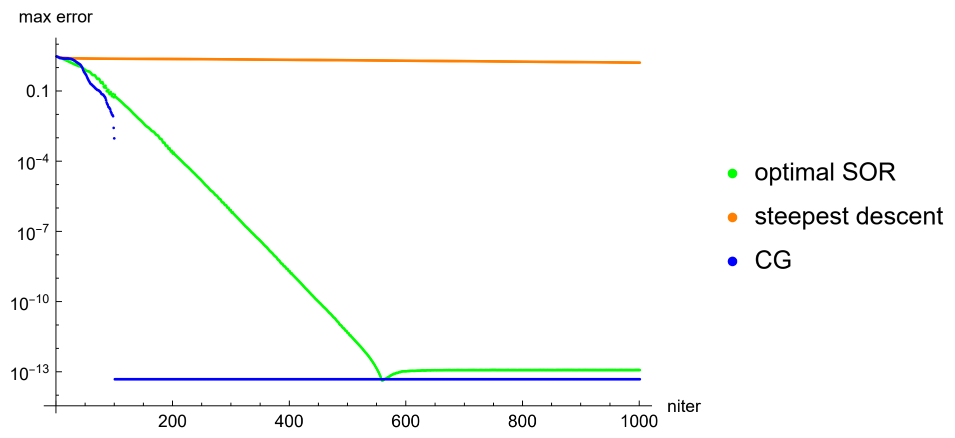
Out[94]=



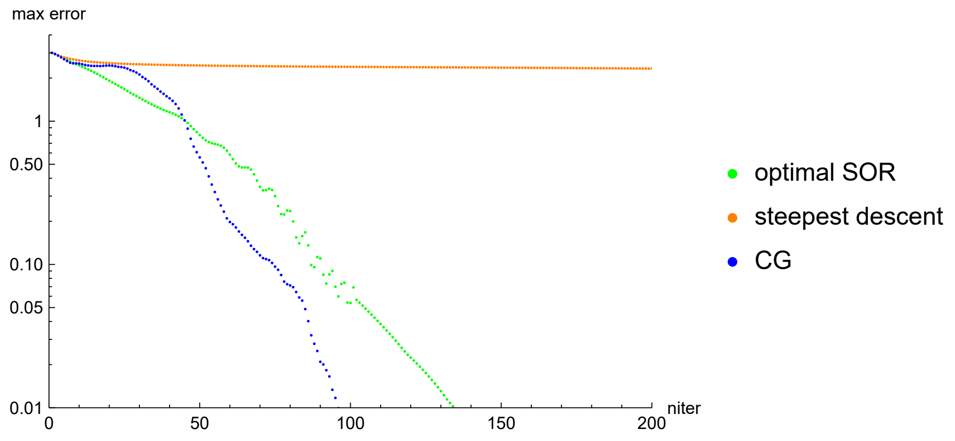
Conjugate Gradient Method

In[95]:=
```
xIter = x0;
r = rhs - T.x0;
p = r;
γ = r.r;
errorCG = ConstantArray[1.0 × 10^-50, niter + 1];
errorCG[[1]] = Max[Abs[xIter - xDirect]];
Do[
 w = T.p;
 α = γ / p.w;
 xIter = xIter + α p;
 r = r - α w;
 β = 1 / γ;
 γ = r.r;
 β = β γ;
 p = r + β p;
 errorCG[[i + 1]] = Max[Abs[xIter - xDirect]],
 {i, 1, niter}
]
xCG = Transpose[{Xin, xIter}]; (* for later use in plots *)
(* Plot results *)
ListLogPlot[{errorSORopt, errorMSD, errorCG}, PlotRange → All,
 PlotStyle → {Green, Orange, Blue}, PlotLegends → {"optimal SOR", "steepest descent", "CG"},
 AxesLabel → {"niter", "max error"}]
ListLogPlot[{errorSORopt, errorMSD, errorCG}, PlotRange → {{0, 200}, {0.01, 4}},
 PlotStyle → {Green, Orange, Blue}, PlotLegends → {"optimal SOR", "steepest descent", "CG"},
 AxesLabel → {"niter", "max error"}]
ListPlot[{exactSol, xSORopt, xMSD, xCG}, PlotStyle → {Black, Green, Orange, Blue},
 PlotLegends → {"Exact", "optimal SOR", "steepest descent", "CG"},
 Joined → {True, False, False, False}, AxesLabel → {"x", "f(x)"}]
```

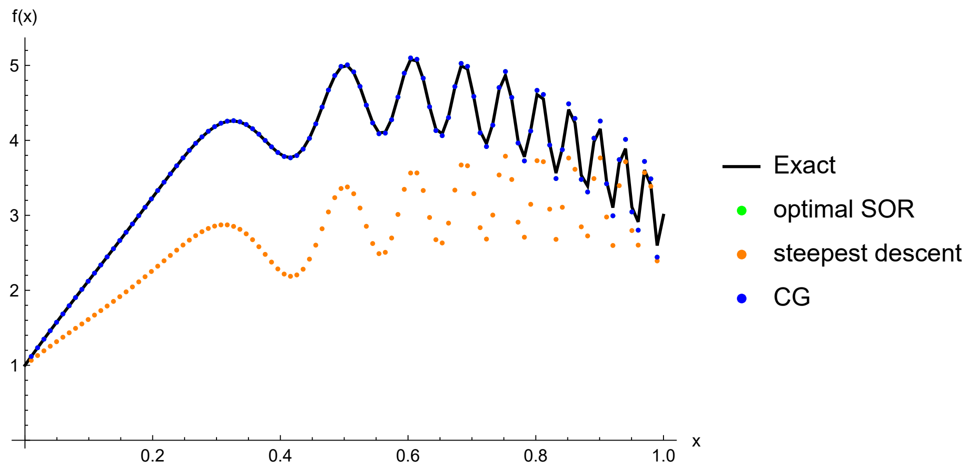
Out[103]=



Out[104]=



Out[105]=



In[106]:=

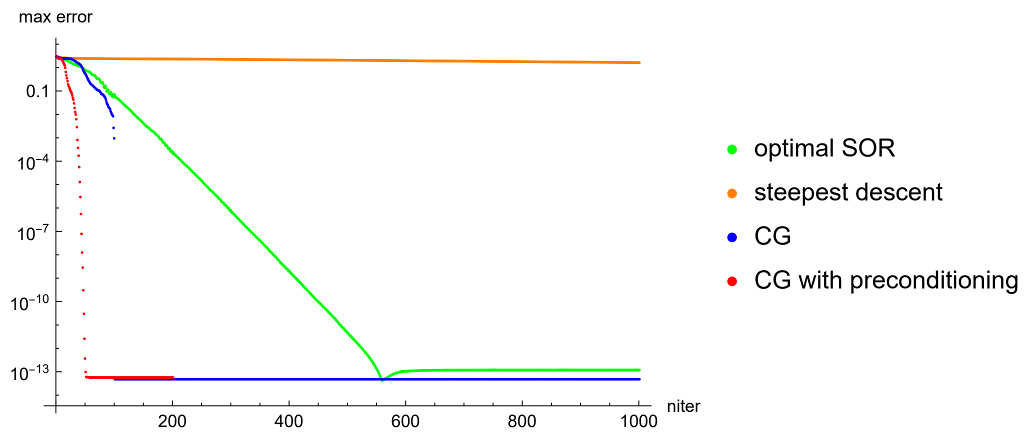
Conjugate Gradient Method with Preconditioning

In this simple case of tridiagonal matrix, it is difficult to choose some reasonable preconditioner. Therefore I only illustrate here the effect by choosing as a preconditioner the original matrix with smaller values of off-diagonal elements.

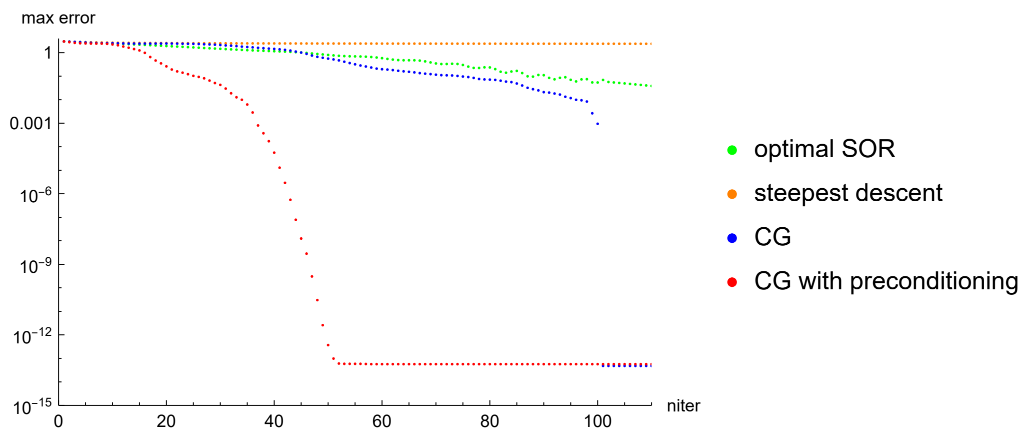
In[107]:=

```
M = SparseArray[{{i_, i_} → 2.0, {i_, j_} /; Abs[i - j] == 1 → -0.8}, {n, n}];
niter = 200;
xIter = x0;
r = rhs - T.x0;
errorCGP = ConstantArray[1.0 × 10^-50, niter + 1];
errorCGP[[1]] = Max[Abs[xIter - xDirect]];
Do[
 z = LinearSolve[M, r];
 If[i == 1,
  p = z; γ = r.z,
  β = 1 / γ; γ = r.z; β = β γ; p = z + β p
 ];
 w = T.p;
 α = γ / p.w;
 xIter = xIter + α p;
 r = r - α w;
 errorCGP[[i + 1]] = Max[Abs[xIter - xDirect]],
 {i, 1, niter}
]
xCGP = Transpose[{Xin, xIter}]; (* for later use in plots *)
(* Plot results *)
ListLogPlot[{errorSORopt, errorMSD, errorCG, errorCGP},
 PlotRange → All, PlotStyle → {Green, Orange, Blue, Red}, PlotRange → All,
 PlotLegends → {"optimal SOR", "steepest descent", "CG", "CG with preconditioning"},
 AxesLabel → {"niter", "max error"}]
ListLogPlot[{errorSORopt, errorMSD, errorCG, errorCGP},
 PlotRange → {{0, 110}, {10^-15, 4}}, PlotStyle → {Green, Orange, Blue, Red},
 PlotLegends → {"optimal SOR", "steepest descent", "CG", "CG with preconditioning"},
 AxesLabel → {"niter", "max error"}]
ListPlot[{exactSol, xMSD, xCG, xCGP}, PlotStyle → {Black, Orange, Blue, Red},
 PlotLegends → {"Exact", "steepest descent", "CG", "CG with preconditioning"},
 Joined → {True, False, False, False}, AxesLabel → {"x", "f(x)"}]
```

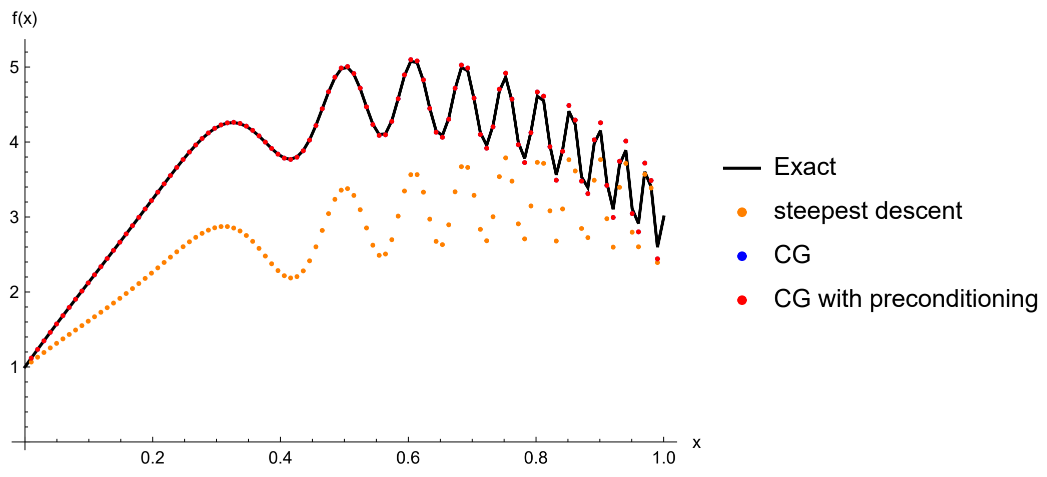
Out[115]=



Out[116]=



Out[117]=


f(x)
x
Exact
steepest descent
CG
CG with preconditioning