

Appendix A

TestCycle: Matlab Code

The following MATLAB R2010a code solves the Poisson equation $\Delta u = F(x, y)$ with Dirichlet boundary conditions $u = G(x, y)$ on a rectangle by applying $V(\nu_1, \nu_2)$ cycles to a random initial guess. Place all files in the same directory and run the MATLAB command `TestCycle.run` (see §1.5).

A.1 addflops.m

```
function addflops(fl)
%ADDFLOPS Increment the global flopcount variable. ADDFLOPS(fl) is
%equivalent to FLOPS(FLOPS+FL), but more efficient.
global flopcount;
if ~isempty(flopcount)
    flopcount = flopcount + fl;
end
```

A.2 BilinearInterpolation.m

```
classdef BilinearInterpolator
    %BILINEARINTERPOLATOR Bi-linear interpolation of corrections.
    % Executes a second-order interpolation from level L to L+1.

    %===== METHODS =====
    methods
        function u = interpolate(obj, coarseLevel, fineLevel, uc)
            % Interpolate the coarse-level function UC at level
            % COARSELEVEL to the function U at level FINELEVEL.

            % Interpolate along one dimension at a time
            u1 = obj.interpInX(coarseLevel, fineLevel, uc);
            u = obj.interpInY(coarseLevel, fineLevel, u1);
```

```

        end
    end

%===== PRIVATE METHODS =====
methods (Access = private)
    function u = interpInX(obj, coarseLevel, fineLevel, uc)
        % Linear interpolation in x

        % Aliases, allocate output array
        nf = fineLevel.n;
        nc = coarseLevel.n;
        u = zeros(nf(1),nc(2));

        % Inject coarse points into the respective fine points
        u(1:2:nf(1),:) = uc;

        % Linearly interpolate into in-between fine-level points
        for i1 = 1:nc(1)-1
            u(2*i1,:) = 0.5*(uc(i1,:) + uc(i1+1,:));
        end
    end

    function u = interpInY(obj, coarseLevel, fineLevel, uc)
        % Linear interpolation in y

        % Aliases, allocate output array
        nf = fineLevel.n;
        nc = coarseLevel.n;
        u = zeros(nf);

        % Inject coarse points into the respective fine points
        u(:,1:2:nf(2)) = uc;

        % Linearly interpolate into in-between fine-level points
        for i2 = 1:nc(2)-1
            u(:,2*i2) = 0.5*(uc(:,i2) + uc(:,i2+1));
        end
    end

end

end

end

```

A.3 Cycle.m

```

classdef (Sealed) Cycle < handle
    %CYCLE Multigrid cycle.
    % This class holds the entire multi-level data structure
    % and executes multigrid cycles. A cycling strategy with an

```

```

% integer index is implemented (gamma=1: V-cycle; gamma=2:
% W-cycle).

%===== MEMBERS =====
properties (GetAccess = private, SetAccess = private)
    levels          % Level list (1=finest, end=coarsest)
    options          % Contains cycle parameters
    finestRelaxWork  % Finest-level relaxation sweep cost
end

%===== CONSTRUCTORS =====
methods
    function obj = Cycle(options, levels)
        % Create a cycle executor with options OPTIONS, to
        % act on the level list LEVELS.
        obj.options      = options;
        obj.levels       = levels;
        obj.finestRelaxWork = prod(levels(end).n-1);
    end
end

%===== METHODS =====
methods
    function u = cycle(obj, finest, u)
        % The main call that executes a cycle at level FINEST.
        obj.printErrorNormHeader();
        u = obj.cycleAtLevel(finest, finest, u);
    end
end

%===== PRIVATE METHODS =====
methods (Access = private)
    function u = cycleAtLevel(obj, l, finest, u)
        % Execute a cycle at level L. FINEST is the index of
        % the finest level in the cycle. Recursively calls
        % itself with the next-coarser level until NUM_LEVELS
        % is reached.

        obj.printErrorNorm(l, 'Initial', u);
        if (l == max(1, finest-obj.options.maxCycleLevels+1))
            % Coarsest level
            u = obj.relax(l, obj.options.numCoarsestSweeps,...
                u, false);
        else
            %--- Pre-relaxation ---
            u = obj.relax(l, obj.options.numPreSweeps, u,...
                true);

            %--- Coarse-grid correction ---

```

```

        c                = l-1;
        fineLevel        = obj.levels{1};
        coarseLevel      = obj.levels{c};

        % Transfer fine-level residuals
        r                = fineLevel.residual(u);
        coarseLevel.f    = fineLevel.restrict(r);

        % Solve residual equation at coarse level
        % Correction scheme: start from vc=0
        vc = zeros(coarseLevel.n);
        for i = 1:obj.options.cycleIndex
            vc = obj.cycleAtLevel(c, finest, vc);
        end

        % Interpolate coarse-level correction and add it
        v                = fineLevel.interpolate(vc);
        u                = u + v;
        obj.printErrorNorm(1, 'Coarse-grid correction', u);

        %--- Post-relaxation ---
        u = obj.relax(1, obj.options.numPostSweeps, u, true);
    end
end

function u = relax(obj, l, nu, u, printEverySweep)
    % Perform NU relaxation sweeps on U at level LEVEL. If
    % PRINTEVERYSWEEP is true, prints printouts after every
    % sweep; otherwise, only after the last sweep.
    for i = 1:nu
        u = obj.levels{1}.relax(u);
        if (printEverySweep)
            obj.printErrorNorm(1, ...
                sprintf('Relaxation sweep %d', i), u);
        end
    end
    if (~printEverySweep)
        obj.printErrorNorm(1, ...
            sprintf('Relaxation sweep %d', i), u);
    end
end

function printErrorNormHeader(obj)
    % Print a header line for cycle debugging printouts.
    if (obj.options.logLevel >= 1)
        fprintf('%-5s %-25s %-13s %-9s\n', 'LEVEL', ...
            'ACTION', 'ERROR NORM', 'WORK');
    end
end
```

```

function u = printErrorNorm(obj, l, action, u)
    % A debugging printout of the error norm at level L
    % after a certain action has been applied. The work per
    % finest-level relaxation sweep is also printed.
    if (obj.options.logLevel >= 1)
        fprintf('%-5d %-25s %.3e %6.2f\n', l, action, ...
            errornorm(obj.levels{l}, u), ...
            flops/obj.finestRelaxWork);
    end
end
end
end
end

```

A.4 errornorm.m

```

function e = errornorm(level, u)
%ERROR_NORM Error norm at a certain coarsening level.
% E = ERROR_NORM(LEVEL,U) computes the grid-scale L2 residual norm
% |F-L(U)|_2, where F and L are stored in the LEVEL structure.
r = level.residual(u);
e = norm(r(:))/sqrt(numel(r));

```

A.5 flops.m

```

function f = flops(fl)
%FLOPS Get or set the global flopcount variable.
% FLOPS returns the current flopcount. FLOPS(F) sets flopcount to F.

global flopcount;
if nargin == 1
    flopcount = fl;
    if nargin == 1 f = fl; end
else
    f = flopcount;
end
end

```

A.6 FwLinearRestrictor.m

```

classdef FwLinearRestrictor
    %FULLWEIGHTINGRESTRICTOR 2nd-order full weighting of residuals.
    % Executes a second-order full weighting from level L+1 to L.

    %===== METHODS =====
    methods
        function fc = restrict(obj, coarseLevel, fineLevel, f)
            % Restrict the fine-level function F at level FINELEVEL
            % to level COARSELEVEL.
    end
end

```

```

        % Interpolate along one dimension at a time
        f1 = obj.restrictInX(coarseLevel, fineLevel, f);
        fc = obj.restrictInY(coarseLevel, fineLevel, f1);
    end
end

%===== PRIVATE METHODS =====
methods (Access = private)
    function fc = restrictInX(obj, coarseLevel, fineLevel, f)
        % Full-weighting in x
        % Aliases, allocate output array
        nf = fineLevel.n;
        nc = coarseLevel.n;
        fc = zeros(nc(1),nf(2));

        % Full-weighting of boundary residuals
        fc([1 nc(1)],:) = f([1 nf(1)],:);

        % Full-weighting of interior residuals
        for i1 = 2:nc(1)-1
            fc(i1,:) = 0.25*(f(2*i1-2,:) + ...
                2*f(2*i1-1,:) + f(2*i1,:));
        end
    end
    function fc = restrictInY(obj, coarseLevel, fineLevel, f)
        % Full-weighting in y

        % Aliases, allocate output array
        nf = fineLevel.n;
        nc = coarseLevel.n;
        fc = zeros(nc);

        % Full-weighting of boundary residuals
        fc(:,[1 nc(2)]) = f(:,[1 nf(2)]);

        % Full-weighting of interior residuals
        for i2 = 2:nc(2)-1
            fc(:,i2) = 0.25*(f(:,2*i2-2) + ...
                2*f(:,2*i2-1) + f(:,2*i2));
        end
    end
end
end
end

```

A.7 GaussSeidelSmoother.m

```

classdef (Sealed) GaussSeidelSmoother < handle
    %GAUSSSEIDELSMOOTHER Gauss-Seidel relaxation scheme.
    % This class executes Gauss-Seidel relaxation sweeps in

```

```

% lexicographic order. It can be applied at any level.
%===== MEMBERS =====
properties (GetAccess = private, SetAccess = private)
    numColors      % Number of colors (1=LEX, 2=RB)
end

%===== CONSTRUCTORS =====
methods
    function obj = GaussSeidelSmoother(numColors)
        obj.numColors = numColors;
    end
end

%===== METHODS =====
methods
    function u = relax(obj, level, u)
        % Gauss-Seidel successive displacement in lexico-
        % graphic ordering. Because MATLAB passes array
        % parameters by value, this does not override the
        % original U array.

        %Useful aliases
        h2 = level.h^2;
        f = level.f;

        % Impose B.C.
        i1 = [1 level.n(1)];      u(i1,:) = f(i1,:);
        i2 = [1 level.n(2)];      u(:,i2) = f(:,i2);

        % Relax in the internal domain
        for c = 0:obj.numColors-1
            for i1 = 2:level.n(1)-1
                for i2 = 2:level.n(2)-1
                    if (mod(i1+i2, obj.numColors) == c)
                        u(i1,i2) = 0.25*(h2*f(i1,i2) ...
                            + u(i1 ,i2-1) + u(i1 ,i2+1) ...
                            + u(i1-1,i2 ) + u(i1+1,i2 ));
                    end
                end
            end
        end

        % A relaxation sweep is counted as one flop per
        % internal gridpoint
        addflops(prod(level.n-1));
    end
end
end
end

```

A.8 Level.m

```

classdef (Sealed) Level < handle
    %LEVEL A single level in the multi-level cycle.
    % This class holds all data and operations pertinent to a
    % single level in the multi-level cycle: right-hand-side,
    % residual computation and single-level processes such as
    % relaxation.
    %===== MEMBERS =====
    properties (GetAccess = public, SetAccess = public)
        f                % RHS of both the interior equations & B.C.
    end

    properties (GetAccess = public, SetAccess = private)
        domainSize      % Size of domain
        h                % Mesh-size (same in all directions)
        n                % Grid array size vector
    end

    properties (GetAccess = private, SetAccess = private)
        coarseLevel      % Next-coarser level
        operator          % Computes discrete operator @ this level
        smoother          % Relaxation scheme
        interpolator       % Interpolates corrections from fineLevel
        restrictor         % Restricts residuals to fineLevel
    end

    %===== CONSTRUCTORS =====
    methods (Access = private)
        function obj = Level(domainSize, n, operator, smoother,...
            coarseLevel, interpolator, restrictor)
            % Initialize a level.

            obj.domainSize = domainSize;
            obj.n           = n+1;
            hVector         = domainSize./n;
            if (std(hVector) > eps)
                error('Incompatible domain size [%f,%f] ...
                    and #intervals [%d,%d]: meshsize must be the ...
                    same in all directions', domainSize(1), ...
                    domainSize(2), n(1), n(2));
            end
            obj.h           = hVector(1);
            obj.f           = zeros(obj.n);
            obj.operator     = operator(obj);
            obj.smoother     = smoother;
            obj.coarseLevel  = coarseLevel;
            obj.interpolator = interpolator;
            obj.restrictor   = restrictor;
        end
    end
end

```



```

end
methods (Static)
    function obj = newLevel(domainSize, n, operator, ...
        smoother, coarseLevel, interpolator, restrictor)
        % A factory method of the next-finer level over
        % COARSELEVEL, with an NxN grid of a domain of size
        % DOMAINSIZExDOMAINSIZE, discrete operator OPERATOR a
        % relaxation scheme SMOOTHER and inter-grid transfers
        % INTERPOLATOR and RESTRICTOR. The right-hand-side is
        % initialized to zero.
        obj = Level(domainSize, n, operator, smoother, ...
            coarseLevel, interpolator, restrictor);
    end

    function obj = newCoarsestLevel(domainSize, n, operator,...
        smoother)
        % A factory method of the coarsest level, with an NxN
        % grid of a domain of size DOMAINSIZExDOMAINSIZE, a
        % discrete operator OPERATOR and a relaxation scheme
        % SMOOTHER.
        obj = Level(domainSize,n,operator,smoother,[],[],[]);
    end
end

%===== METHODS =====
methods
    function r = residual(obj, u)
        % Compute the residual F-L(U) for a function U at this
        % level.
        r = obj.f - obj.L(u);
    end

    function v = relax(obj, u)
        % Perform a relaxation sweep. Delegates to the smoother
        % with a call-back to this level.
        v = obj.smoother.relax(obj, u);
    end

    function u = interpolate(obj, uc)
        % Interpolate the correction uc from the next-coarser
        % level.
        u = obj.interpolator.interpolate(obj.coarseLevel,obj, uc);
    end

    function fc = restrict(obj, f)
        % Restrict the residual FC to the next-coarser level.
        fc = obj.restrictor.restrict(obj.coarseLevel, obj, f);
    end
end

```

```

function [x, y] = location(obj, i1, i2)
    % Return gridpoint locations at indices (I1,I2).
    x = obj.h*(i1-1);
    y = obj.h*(i2-1);
end

function result = L(obj, u)
    % Apply the discrete operator L to a function U.
    result = obj.operator.L(u);
end

function handle = plot(obj, u)
    % Plot the discrete function U on the grid of this level.
    [x,y] = obj.location(1:obj.n(1), 1:obj.n(2));
    [X,Y] = ndgrid(x,y);
    handle = surf(X,Y,u);
end
end
end
end

```

A.9 MultilevelBuilder.m

```

classdef (Sealed) MultilevelBuilder < handle
    %MULTILEVELBUILDER Constructs the multi-level data structure.
    % This class builds a list of increasingly-finer levels to be
    % used in the multigrid cycle.

    %===== CONSTRUCTORS =====
    methods
        function obj = MultilevelBuilder
            % Explicit constructor is required for a handle class.
        end
    end

    %===== METHODS =====
    methods
        function levels = build(obj, options) %#ok<MANU>
            % Build the list of levels from options.
            levels = cell(options.numLevels, 1);

            % Coarsest level
            n = options.nCoarsest;
            levels{1} = Level.newCoarsestLevel(options.domainSize,...
                n, options.operator, options.smoother);

            % Increasingly-finer levels
            for l = 2:options.numLevels
                n = 2*n;
            end
        end
    end
end

```

```

        lev = Level.newLevel(options.domainSize, n, ...
            options.operator, options.smoother, ...
            levels{1-1}, options.interpolator, ...
            options.restrictor);

% Initialize finest right-hand side
if (1 == options.numLevels)
    % Interior RHS
    MultilevelBuilder.setRhsValues(...
        lev, 2:n(1)-1, 2:n(2)-1, options.f);

    % Boundary RHS
    MultilevelBuilder.setRhsValues(...
        lev, [1 n(1)], 1:n(2) , options.g);
    MultilevelBuilder.setRhsValues(...
        lev, 1:n(1) , [1 n(2)], options.g);
end

levels{1} = lev;
end
end
end

%===== PRIVATE METHODS =====
methods (Static, Access = private)
    function setRhsValues(lev, i1, i2, f)
        % Set the values of indices (i1,i2) of a level's RHS
        % vector to the function f, evaluated at the corres-
        % ponding gridpoint locations.
        [xInterior,yInterior] = lev.location(i1,i2);
        % Convert singleton x,y vectors to 2-D matrices
        [X,Y] = ndgrid(xInterior, yInterior);
        lev.f(i1,i2) = f(X,Y);
    end
end
end
end

```

A.10 Operator.m

```

classdef (Sealed) Operator < handle
    %OPERATOR Discrete operator computer.
    % This class computes the discrete operator L(U) of a function
    % U at a certain level in the multi-level algorithm.

    %===== MEMBERS =====
    properties (GetAccess = private, SetAccess = private)
        level % Holds convenient level variables
    end
end

```

```

%===== CONSTRUCTORS =====
methods
    function obj = Operator(level)
        % Initializes an operator computer at level LEVEL.
        obj.level = level;
    end
end

%===== METHODS =====
methods
    function result = L(obj, u)
        % Apply the discrete operator L to a function U. This
        % is the 5-point Laplacian with Dirichlet boundary
        % conditions.

        % Allocate output array
        result = zeros(obj.level.n);

        % Set Dirichlet boundary conditions
        i1 = [1 obj.level.n(1)];      result(i1,:) = u(i1,:);
        i2 = [1 obj.level.n(2)];      result(:,i2) = u(:,i2);

        % 5-point discrete Laplacian in the interior domain
        rh2 = 1/obj.level.h^2;
        for i1 = 2:obj.level.n(1)-1
            for i2 = 2:obj.level.n(2)-1
                result(i1,i2) = rh2*(...
                    4*u(i1,i2) ...
                    -u(i1 ,i2-1) - u(i1 ,i2+1) ...
                    -u(i1-1,i2 ) - u(i1+1,i2 ));
            end
        end
    end
end

end
end
end

```

A.11 Options.m

```

classdef (Sealed) Options < handle
    %OPTIONS Multi-level algorithm options.
    % Includes both model parameters and cycle parameters. Sets
    % default values for parameters that can be overridden by the
    % user.

    %===== MEMBERS =====
    properties
        % Model parameters
    end
end

```

```

domainSize = [2.0 3.0]                % Domain size

f = @(x,y)(sin(x.^2+y)+0.5)           % Right-hand-side
g = @(x,y)(cos(2*x+y)+0.5)           % Dirichlet B.C.

% Known solution u = (2*pi^(-2))*sin(pi*x).*sin(pi*y)
%f = @(x,y)(sin(pi*x).*sin(pi*y))    % Right-hand-side
%g = @(x,y)(sin(pi*x).*sin(pi*y))    % Dirichlet B.C.

% To debug the cycle error, set f=g=0 so that u=error
%f = @(x,y)(zeros(size(x)))          % Right-hand-side
%g = @(x,y)(zeros(size(x)))          % Dirichlet B.C.

% Discretization
nCoarsest = [2 3]                    % #coarsest grid intervals
numLevels = 6                        % Total #levels
operator = @(level)(Operator(level)) % Discrete operator

% Relaxation parameters
smoother = GaussSeidelSmoother(1)    % Gauss-Seidel relaxation
                                           % (1=LEX, 2=RB)

% Inter-grid transfers
interpolator = BilinearInterpolator % Interp. of corrections
restrictor = FwLinearRestrictor;    % Residual transfer

% Cycle parameters
maxCycleLevels = 100                 % # levels in the cycle
cycleIndex = 1                       % V-cycle/W-cycle/etc.
numCoarsestSweeps = 5                % # relaxation sweeps
                                           % at coarsest level
numPreSweeps = 2                     % # pre-CGC relax sweeps
numPostSweeps = 1                    % # post-CGC relax sweeps

% Multi-grid run
numCycles = 12                       % #cycles to run

% Miscellaneous
logLevel = 1                         % Cycle logging level
end
end

```

A.12 TestCycle.m

```

classdef TestCycle
    %TESTCYCLE Test the multigrid cycle for the 2D Poisson equation.
    % This class iteratively runs multigrid V-cycles and measures
    % their convergence factor.

```

```

%
% See also: ERROR_NORM, CYCLE.

%===== METHODS =====
methods
function [u, finestLevel] = run(obj) %#ok<MANU>
    % Initialize objects
    flops(0);          % Reset flop count
    options            = Options;
    levels              = MultilevelBuilder().build(options);
    cycle              = Cycle(options, levels);
    finest              = length(levels);
    finestLevel = levels{finest};

    % Initial guess
    u                = rand(finestLevel,n);
    eNew              = errornorm(finestLevel, u);
    % Run cycles
    for numCycle = 1:options.numCycles
        % Print debugging lines only for the first few cycles
        if (numCycle <= 3)
            options.logLevel = 1;
            fprintf('##### CYCLE #%d #####\n',...
                numCycle);
        else
            options.logLevel = 0;
        end
        eOld      = eNew;
        u          = cycle.cycle(finest, u);
        eNew        = errornorm(finestLevel, u);
        fprintf('CYCLE %#2d CONVERGENCE FACTOR = %.3f\n', ...
            numCycle, eNew/eOld);
    end

end

end

end

```