Multilevel Methods for HPC Programming Task

Jan Van lent

BICS, University of Bath, UK

Wednesday 28 November 2007

Introduction

- implement multigrid for 1D and 2D Poisson
- on unit interval/unit square
- with Dirichlet boundary conditions
- implementation in Python using Numpy

Grid Data Structure 1D

- n subintervals
- grid functions are represented by `numpy` arrays
- size: $n + 1$, e.g. (`u = zeros(n+1)`)
- boundary points are included
- e.g. function $u(x) = x^2$ on interval $[0, 1]$

```
from numpy import *
n = 8
dx = 1.0 / n
x = arange(n+1) * dx
u = x**2
print u[0], u[-1]
```

Grid Data Structure 2D

- `nx`, `ny` subintervals in each direction
- grid functions are represented by `numpy` arrays
- size: `(nx+1,ny+1)`, e.g. `u = zeros((nx+1, ny+1))`
- boundary points are included
- e.g. function $u(x,y) = x^2 + y^2$ on interval $[0,1]^2$

```
from numpy import *
nx, ny = 8, 8
dx, dy = 1.0 / nx, 1.0 / ny
ix, iy = mgrid[:nx+1,:ny+1]
x, y = dx * ix, dy * iy
u = x**2 + y**2
print u[:,0], u[-1,:], u[:,-1], u[0,:]
```

Operator Data Structures

- represent the operator L by
  - the stepsize Δx in 1D
  - a tuple of the stepsizes $(\Delta x, \Delta y)$ in 2D
- this is sufficient to implement the operators needed for the Poisson equation
- represent the hierarchy of operators by a list of operator representations
- levels are represented by integer
- level 0 is the coarsest level
- level 1 is the next finer level, etc.

Task 1D

implement the following functions

- `li_1d(uc)`
- `fwr_1d(r)`
- `residual_1d(v, L, f)`
- `rb_gs_1d(v, L, f)`
- `lex_gs_1d(v, L, f)`

Test each function individually!

You can of course create other functions as you see fit.

Linear Interpolation 1D

- specification

$$e_{2i} = \bar{u}_i$$
$$e_{2i+1} = \frac{\bar{u}_i + \bar{u}_{i+1}}{2}$$

- the boundary values can be included in these equations

```
def li_1d(uc):
    # uc is a grid function on a coarse grid
    # e is a grid function on finer grid
    # create e
    return e
```

Full Weighting Restriction 1D

- specification

$$\bar{f}_i = \frac{r_{2i-1} + 2r_{2i} + r_{2i+1}}{4}$$

- the boundary values can be copied
- they should be 0

```
def fwr_1d(r):
    # r is a grid function on a fine grid
    # fc is a grid function on coarser grid
    # create fc
    return fc
```

Residual 1D

- specification

$$\bar{r}_i = f_i - (-u_{i-1} + 2u_i - u_{i+1})/\Delta x^2$$

- the values at the boundaries are 0

```
def res_1d(v, L, f):
    # equation L u = f
    # f is right hand side
    # L represents operator
    # v is approximation to u
    # r is residual r = f - L u
    # create r
    return r
```

Lexicographical Gauss-Seidel 1D

- specification

$$\bar{u}_i \leftarrow \left(\frac{2}{\Delta x^2}\right)^{-1} \left(f_i + \frac{u_{i-1} + u_{i+1}}{\Delta x^2}\right)$$

- the values at the boundaries should be copied

```
def lex_gs_1d(v, L, f):
    # equation L u = f
    # f is the right hand side
    # v is an approximation to u
    # L represents an operator
    # w is the new approximation
    # create w
    return w
```

Red-Black Gauss-Seidel 1D

- specification

$$\bar{u}_{2i+1} \leftarrow \left(\frac{2}{\Delta x^2}\right)^{-1} \left(f_{2i+1} + \frac{u_{2i} + u_{2i+2}}{\Delta x^2}\right)$$

$$\bar{u}_{2i} \leftarrow \left(\frac{2}{\Delta x^2}\right)^{-1} \left(f_{2i} + \frac{u_{2i-1} + u_{2i+1}}{\Delta x^2}\right)$$

- the values at the boundaries should be copied

```
def rb_gs_1d(v, L, f):
    # similar to lex_gs_1d
    # create w
    return w
```

Task Multigrid

- implement a multigrid V-cycle function
- test multigrid using 1D functions

Multigrid Function

```
def multigrid(level, v, Ls, f,
              smooth, res, restrict, prolong,
              coarse_solve, cycle):
    # level: integer
    # v: initial approximation
    # Ls: hierarchy of operators
    # f: right hand side
    # smooth: w = smooth(v, L, f)
    # res: r = res(v, L, f)
    # restrict: fc = restrict(r)
    # prolong: e = prlong(uc)
    # coarse_solve: w = coarse_solve(v, L, f)
    # cycle: ignored
    # create new approximation w
    return w
```

Task 2D

implement the following functions

- `bli_2d(uc)`
- `fwr_2d(r)`
- `residual_2d(v, L, f)`
- `rb_gs_2d(v, L, f)`
- `lex_gs_2d(v, L, f)`

Test each function individually!

Bilinear Interpolation 2D

- specification
  $e_{2i,2j} = \bar{u}_{i,j}, \quad e_{2i+1,2j+1} = \frac{\bar{u}_{i,j}+\bar{u}_{i,j+1}+\bar{u}_{i+1,j}+\bar{u}_{i+1,j+1}}{4}$
  $e_{2i,2j+1} = \frac{\bar{u}_{i,j}+\bar{u}_{i,j+1}}{2}, \quad e_{2i+1,2j} = \frac{\bar{u}_{i,j}+\bar{u}_{i+1,j}}{2}$
- the boundary values can be included in these equations

```
def bli_2d(uc):
    # uc is a grid function on a coarse grid
    # e is a grid function on finer grid
    # create e
    return e
```

Full Weighting Restriction 2D

- specification

$$\bar{f}_{i,j} = \frac{1}{16} \begin{pmatrix} r_{2i-1,2j-1} + 2r_{2i-1,2j} + r_{2i-1,2j+1} + \\ 2r_{2i,2j-1} + 4r_{2i,2j} + 2r_{2i,2j+1} + \\ r_{2i+1,2j-1} + 2r_{2i+1,2j} + r_{2i+1,2j+1} \end{pmatrix}$$

- the boundary values can be copied
- they should be 0

```
def fwr_2d(r):
    # r is a grid function on a fine grid
    # fc is a grid function on coarser grid
    # create fc
    return fc
```

Residual 2D

- specification
  $\bar{r}_{i,j} = f_{i,j} - (\frac{-u_{i-1,j}+2u_{i,j}-u_{i+1,j}}{\Delta x^2} + \frac{-u_{i,j-1}+2u_{i,j}-u_{i,j+1}}{\Delta y^2})$
- the values at the boundaries are 0

```
def res_2d(v, L, f):
    # equation L u = f
    # f is right hand side
    # L represents operator
    # v is approximation to u
    # r is residual r = f - L u
    # create r
    return r
```

Lexicographical Gauss-Seidel 2D

- stencil

$$L = \begin{bmatrix} & \frac{-1}{\Delta y^2} & \\ \frac{-1}{\Delta x^2} & \frac{2}{\Delta x^2} + \frac{2}{\Delta y^2} & \frac{-1}{\Delta x^2} \\ & \frac{-1}{\Delta y^2} & \end{bmatrix} = \begin{bmatrix} & \bullet & \\ \bullet & \bullet & \bullet \\ & \bullet & \end{bmatrix}$$

- copy (boundary) values
- for $i = 1, \ldots, n_x - 1$
  - for $j = 1, \ldots, n_y - 1$

$$u_{i,j} \leftarrow \left[\bullet\right]^{-1} \left(f_{i,j} - \left(\begin{bmatrix} & \bullet & \\ \bullet & & \bullet \\ & \bullet & \end{bmatrix} u \right)_{i,j} \right)$$

Lexicographical Gauss-Seidel 2D (cont.)

```
def lex_gs_2d(v, L, f):
    # equation L u = f
    # f is the right hand side
    # v is an approximation to u
    # L represents an operator
    # w is the new approximation
    # create w
    return w
```

Red-Black Gauss-Seidel 2D

- stencil

$$L = \begin{bmatrix} & \frac{-1}{\Delta y^2} & \\ \frac{-1}{\Delta x^2} & \frac{2}{\Delta x^2} + \frac{2}{\Delta y^2} & \frac{-1}{\Delta x^2} \\ & \frac{-1}{\Delta y^2} & \end{bmatrix} = \begin{bmatrix} & \bullet & \\ \bullet & \bullet & \bullet \\ & \bullet & \end{bmatrix}$$

- copy (boundary) values
- for points with odd $i + j$

$$u_{i,j} \leftarrow \left[\bullet\right]^{-1} \left(f_{i,j} - \left(\begin{bmatrix} & \bullet & \\ \bullet & & \bullet \\ & \bullet & \end{bmatrix} u \right)_{i,j} \right)$$

- same for points with odd $i + j$

Red-Black Gauss-Seidel 2D (cont.)

```
def rb_gs_2d(v, L, f):
    # equation L u = f
    # f is the right hand side
    # v is an approximation to u
    # L represents an operator
    # w is the new approximation
    # create w
    return w
```

Multigrid 2D

- test multigrid using 2D functions

Solver

- put all your files in a directory `multigrid_task_<yourname>_<version>`
- create a file `multigrid_task.py` in it
- in this file implement the functions `solve_1d(g, f)` and `solve_2d(g, f)`
- make a tarred and gzipped file `multigrid_task_<yourname>_<version>.tar.gz` containing this directory and all files in it
- send this file to me `j.van.lent@maths.bath.ac.uk`
- use as subject `"multigrid task, <yourname>"`
- this code will be run with something like `multigrid_task_run.py`

```
# multigrid_task.py

# equation L u = f
# u = g on boundary
# grid function g
#  only boundary values are essential
#  other values may be used as
#  initial approximation
def solve_1d(g, f):
    # create u using 1D multigrid
    return u
def solve_2d(g, f):
    # create u using 2D multigrid
    return u
```

```
# multigrid_task_run.py
from numpy import *
import multigrid_task

# 1D test
n = 2**8
dx = 1.0 / n
x = arange(n+1) * dx
u_exact = x**2
f  = -2 + 0*x
g = u_exact.copy()
g[1:-1] = 0.0
u = multigrid_task.solve_1d(g, f)
print abs(u - u_exact).max()
```

```
# multigrid_task_run.py (cont.)

# 2D test
nx, ny = 2**8, 2**8
dx, dy = 1.0 / nx, 1.0 / ny
ix, iy = mgrid[:nx+1,:ny+1]
x, y = dx * ix, dy * iy
u_exact = x**2 + y**2
f  = -4 + 0*x  + 0*y
g = u_exact.copy()
g[1:-1,1:-1] = 0.0
u = multigrid_task.solve_2d(g, f)
print abs(u - u_exact).max()
```