TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Mechanical Science and Engineering Institute of Fluid Mechanics, Chair of Fluid Mechanics

Project Paper

Simulation Components for Evaluating a Framework for Multidisciplinary Analyzes

Simon Ehrmanntraut

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Mechanical Science and Engineering Institute of Fluid Mechanics, Chair of Fluid Mechanics

Project Paper

Simulation Components for Evaluating a Framework for Multidisciplinary Analyzes

Simon Ehrmanntraut

Born on: January 16, 1999 in Würzburg

Referee

Prof. Dr.-Ing. habil. J. Fröhlich

Supervisors

PD Dr.-Ing. J. Stiller

Dr.-Ing. A. Stück (DLR)

Dipl.-Inf. S. Gottfried (DLR)

Submitted on: October 31, 2021

Ein besonderer Dank gilt Dr. Arthur Stück sowie Herrn Sebastian Gottfried des Deutschen Zentrums für Luft- und Raumfahrt für ihre fachliche und organisatorische Unterstützung bei dieser Arbeit.

TECHNISCHE UNIVERSITÄT DRESDEN

B – 2308

Fakultät Maschinenwesen, Institut für Strömungsmechanik, Professur für Strömungsmechanik

Aufgabenstellung für die Projektarbeit zum Forschungspraktikum

Name: **Simon Ehrmanntraut** **Matr.-Nr.: 4713418**
Studienrichtung Luft- und Raumfahrttechnik

Thema: **Simulationskomponenten zur Evaluation eines Frameworks für multidisziplinäre Analysen**
(Simulation components for evaluating a framework for multidisciplinary analyzes)

Zielsetzung:

Am DLR wird ein Framework für skalierbare, multidisziplinäre Analysen und Optimierungen (MDA/O) von Flugzeugen entwickelt, das auf der DLR/AIRBUS-entwickelten HPC-Infrastruktur FlowSimulator und der NASA-Software OpenMDAO basiert. Neben gradientenbasierten Optimierungsverfahren bietet die Framework-Infrastruktur verschiedene Algorithmen zur Lösung sehr großer, nicht-linearer, multidisziplinär gekoppelter Probleme. Das Framework unterstützt eine MPI-parallele Ausführung. Verschiedene disziplinäre Simulationskomponenten werden modular über Python-Interfaces in das Framework integriert.
Im Rahmen der Studienarbeit sollen mit MPI parallel ausführbare Simulationskomponenten konzipiert und implementiert werden, die zur systematischen Evaluation (Verifikation und Benchmarking) des Frameworks verwendet werden sollen. Als Simulationskomponenten sollen nicht-lineare PDGL/GDGL-basierte Modelle verwendet werden, die mit etablierten Finite-Elemente-Methoden oder Finite-Volumen-/Finite-Differenzen-Methoden diskretisiert werden. Für die einzelnen Simulationskomponenten sollen dem MDA/O-Framework über vorgegebene Python-Interfaces unter anderem disziplinäre Lösungsoperationen, Residuen, Linearisierungen der Residuen bereitgestellt werden, z.B. mit Algorithmischer Differentiation, Finiten Differenzen oder der Complex-Step Methode. Als Teil der Arbeit soll die algorithmische und parallele Skalierbarkeit des Frameworks unter Verwendung von Newton-Krylov-Methoden evaluiert werden, indem aus den zu entwickelnden Simulationskomponenten gekoppelte Probleme zusammengesetzt werden.

Betreuer: Dipl.-Inf. Sebastian Gottfried (Inst. f. Softwaremethoden zur Produkt-Virtualisierung, DLR)
PD Dr.-Ing. Jörg Stiller (ISM, TU Dresden)

Ausgehändigt am: 01.05.2021
Einzureichen am: 30.10.2021

Prof. Dr.-Ing. habil. J. Fröhlich
Betreuender Hochschullehrer

Abstract

When monolithic approaches to solve multidisciplinary problems are not feasible, single-disciplinary solvers must be coupled. The NASA-developed *Python* framework *OpenMDAO*, which manages the communication between the single-disciplinary solvers, is used to implement various coupling methods (some of which support MPI-parallelism). The feasibility of using *OpenMDAO* on large-scale industrial problems is assessed, based on the experiences made with a small academic problem – stationary natural convection of a fluid in a square cavity. Two deliberately simple single-disciplinary solvers – one for the convection-diffusion equation and one for the NAVIER-STOKES equations – are implemented in *Python*. The solutions returned from *OpenMDAO* match the reference, and the iterative convergence behaviors of the tested coupling methods match their theoretical expectations – so there is no considerable drawback on convergence when using *OpenMDAO*. *OpenMDAO* allows to quickly test various coupling methods, but it should only be used if the single-disciplinary solvers are fixed and runtime performance is not a major issue.

Zusammenfassung

Wenn monolithische Ansätze zur Lösung multidisziplinärer Probleme nicht durchführbar sind, müssen einzeldisziplinäre Solver gekoppelt werden. Das NASA-entwickelte *Python*-Framework *OpenMDAO*, welches die Kommunikation zwischen den einzeldisziplinären Solvern verwaltet, wird zur Implementierung verschiedener Kopplungsmethoden verwendet (von denen einige MPI-Parallelität unterstützen). Die Einsetzbarkeit von *OpenMDAO* bei großen industriellen Problemen wird auf der Grundlage der Erfahrungen mit einem kleinen akademischen Problem - der stationären natürlichen Konvektion eines Fluids in einem quadratischen Hohlraum - bewertet. Zwei bewusst einfach gehaltene einzeldisziplinäre Solver – einer für die Konvektions-Diffusionsgleichung und einer für die NAVIER-STOKES-Gleichungen – werden in *Python* implementiert. Die von *OpenMDAO* ausgegebenen Lösungen stimmen mit der Referenz überein, und die iterative Konvergenzverhalten der getesteten Kopplungsmethoden entsprechen deren theoretischen Erwartungen, sodass es bei der Verwendung von *OpenMDAO* keine nennenswerten Nachteile hinsichtlich der Konvergenz gibt. *OpenMDAO* erlaubt es verschiedene Kopplungsmethoden schnell zu testen, sollte aber nur verwendet werden wenn die einzeldisziplinären Solver feststehen und die Laufzeitleistung keine große Rolle spielt.

Contents

List of Symbols

Latin Symbols

$\check{a}$	$m^2\,s^{-1}$	thermal diffusivity
$\boldsymbol{b}$		right-hand-side vector
$\boldsymbol{C}$		convection matrix
$\mathfrak{C}$		element convection array
DOF		total amount of degrees of freedom
$\mathfrak{f}$		solver of the $\mathfrak{F}$-problem
$\mathfrak{F}$		differential operator
$\boldsymbol{F}$		product matrix
$\mathfrak{g}$		solver of the $\mathfrak{G}$-problem
$\check{g}$	$m\,s^{-2}$	gravitational acceleration
$\mathfrak{G}$		differential operator
$\boldsymbol{G}$		gradient matrix
$\mathfrak{G}$		element gradient array
Gr		GRASHOF number, see (3.3)
$\boldsymbol{H}$		iteration matrix
$\boldsymbol{I}$		identity matrix
$\boldsymbol{J}$		JACOBI matrix
$\boldsymbol{K}$		stiffness matrix
$\mathfrak{K}$		element stiffness array
ℓ		LAGRANGE interpolation polynomial
L		dimensionless length, $L := \check{L}/\check{L}_{\mathrm{ref}}$
$\check{L}$	m	length
m		number of preconditioner steps
$mtol$		tolerance on mean square root residual
$\boldsymbol{M}$		mass matrix
$\mathfrak{M}$		element mass array
n		number of linear solver steps
N		amount of grid points, $N := (N_{ex}P + 1) \cdot (N_{ey}P + 1)$
N_{ex}		amount of elements in x-direction
N_{ey}		amount of elements in y-direction
p		dimensionless pressure, $p := \check{p}\check{L}_{\mathrm{ref}}/(\check{\nu}\check{\rho}_0\check{U}_{\mathrm{ref}})$
$\boldsymbol{p}$		discretization of p

$\check{p}$	Pa	pressure
P		polynomial order, LEGENDRE polynomial
$Pé$		PÉCLET number, see (3.3)
Pr		PRANDTL number, see (3.3)
$\boldsymbol{r}$		residual functions
$\Delta\boldsymbol{r}$		prescribed residual difference
Ra		RAYLEIGH number, $Ra = Gr \cdot Pr$
Re		REYNOLDS number, see (3.3)
$\boldsymbol{S}$		SCHUR complement
T		dimensionless temperature perturbation, $T := \check{T}/\Delta\check{T}$
$\boldsymbol{T}$		discretization of T
$\check{T}$	K	temperature perturbation
$\Delta\check{T}$	K	temperature difference
u		dimensionless velocity in x-direction, $u := \check{u}/\check{U}_{\text{ref}}$
$\boldsymbol{u}$		discretization of u
$\check{u}$	$\mathrm{m\,s^{-1}}$	velocity in x-direction
$\check{u}_{\max}$	$\mathrm{m\,s^{-1}}$	maximum velocity in x-direction on the vertical mid-plane
v		dimensionless velocity in y-direction, $v := \check{v}/\check{U}_{\text{ref}}$
$\boldsymbol{v}$		discretization of v
$\check{v}$	$\mathrm{m\,s^{-1}}$	velocity in y-direction
$\check{v}_{\max}$	$\mathrm{m\,s^{-1}}$	maximum velocity in y-direction on the horizontal mid-plane
w		quadrature weight
x		first spatial coordinate; function
$\boldsymbol{x}$		discretization of x
y		second spatial coordinate; function
$\boldsymbol{y}$		discretization of y

Greek Symbols

$\check{\beta}$	$\mathrm{K^{-1}}$	thermal expansion coefficient
δ_{ij}		KRONECKER delta
η		standard variable in y direction, $\eta \in [-1, 1]$
η^n		n-th y-transformation, see (B.2b)
θ		relaxation parameter
μ		asymptotic rate of linear convergence, $\mu := \limsup_{k\to\infty} \sqrt[k]{\frac{\left\|\boldsymbol{x}^{(k)}-\boldsymbol{x}^*\right\|}{\left\|\boldsymbol{x}^{(0)}-\boldsymbol{x}^*\right\|}}$
$\check{\nu}$	$\mathrm{m^2\,s^{-1}}$	kinematic viscosity
ξ		standard variable in x direction, $\xi \in [-1, 1]$
ξ^m		m-th x-transformation, see (B.2a)
$\check{\rho}$	$\mathrm{kg\,m^{-3}}$	density
φ^m		basis function in Ω_x^m
φ^n		basis function in Ω_y^n
φ^{mn}		basis function in Ω_x^{mn}

ω	test function
Ω	domain, $\Omega := [0, L_x] \times [0, L_y]$
Ω_x^m	m-th partition of $[0, L_x]$
Ω_y^n	n-th partition of $[0, L_y]$
Ω^{mn}	m-n-th element of Ω, $\Omega^{mn} := \Omega_x^m \times \Omega_y^n$
$\partial\Omega$	boundary of Ω

Operators

$\frac{\partial \cdot}{\partial n}$	normal derivative
$\kappa(\boldsymbol{A})$	condition number of $\boldsymbol{A}$, $\kappa(\boldsymbol{A}) = \left\|A^{-1}\right\|\|A\|$
$\rho(\boldsymbol{A})$	spectral radius of $\boldsymbol{A}$
∇	nabla operator, $\nabla = \left[\partial/\partial_x \; \partial/\partial_y\right]^\mathsf{T}$
$\left\|\cdot\right\|$	a vector norm; its induced matrix norm

Indices

cont	continuity
lin	linear
nonlin	nonlinear
ref	reference
s	standard
S	SCHUR
T	w.r.t. variable T
u	w.r.t. variable u
v	w.r.t. variable v
velo	velocities

Abbreviations

COO	coordinate format
CSR	compressed spare row format
DLR	Deutsches Zentrum für Luft- und Raumfahrt (German Aerospace Center)
GLL	GAUß-LOBATTO-LEGENDRE
GMRES	generalized minimal residual
GS	nonlinear block-GAUß-SEIDEL
JNK	block-JACOBI preconditioned NEWTON-KRYLOV
LGMRES	loose generalized minimal residual
MDAO	multidisciplinary design, analysis, and optimization
MPI	message parsing interface
NASA	National Aeronautics and Space Administration
NJ	(one-step) NEWTON-block-JACOBI
PDE	partial differential equation
SEM	spectral element method

1 Background

1.1 Motivation

Single- and Multidisciplinary Problems

For many single-disciplinary problems in engineering, like heat transfer, structural deformation or fluid flow, there are highly sophisticated numerical solvers returning a discretized solution. By considering the mathematical structure of the discipline, single-disciplinary solvers achieve a high degree of efficiency. Example: given a fixed load distribution on a wing and bearing conditions, its unknown deflection can efficiently be solved for; likewise, given a fixed wing geometry and free-stream conditions, the unknown aerodynamic forces can efficiently be solved for. Yet many problems of interest are multidisciplinary, such that the explicit givens of one problem are the implicit unknowns of the other. Continuing the example: for wings under real conditions, only the bearing and free-stream conditions are given, and neither the geometry nor the forces are fixed; the aerodynamic forces deform the geometry of the wing, which in turn changes the flow and consequently the aerodynamic forces which again in turn change the geometry and so on, leading possibly to an equilibrium. Solving multidisciplinary problems by a single monolithic solver is especially difficult, not only because the interaction between the unknowns must be considered, but also because theoretical and practical know-how from the single-disciplinary solvers can usually no longer be applied. When developing a new solver for a multidisciplinary problem is not feasible, single-disciplinary solvers must be coupled.

Coupling

Considering each single-disciplinary solver as black-box, returning a solution to its unknowns (outputs) for its explicit givens (inputs), there is a simple and extensively used coupling method: supply the first solver with a guess for its inputs; let it solve for its outputs, providing a guess for the inputs of the second solver; and let it solve for its outputs, providing a new guess for the inputs of the first solver. One can show that this method (nonlinear block-GAUß-SEIDEL or sequential staggered method) converges only under restrictive requirements on the problem and linear at best; though the black-box character of the single-disciplinary solvers is kept. Various authors, e.g. [21, 5], propose approximate variants of NEWTON's method where the JACOBIans are approximated from past iterations. This reduces convergence rates, but the single-disciplinary solvers are

still treated as interchangeable black-boxes. If the single-disciplinary solvers internally provide subroutines concerning their JACOBIans, NEWTON's method can be performed correctly by looking one level deeper into the black-boxes and using these subroutines. While there are now restrictive requirements for the implementation of the single-disciplinary solvers being coupled, the method converges quadratic at least and cubic at best. Also, for tightly coupled problems of many single-disciplinary solvers, the method is hopefully more robust.

OpenMDAO

OpenMDAO [13, 18] is a NASA-developed *Python* framework, allowing the user to decompose multidisciplinary problems in several problems being solved by single-disciplinary solvers. It provides top-level solvers for the coupled problem and manages the calls and data transfers to the single-disciplinary solvers. The modularity of the top-level solvers lets the user easily combine them to implement various coupling and solving algorithms. The user only has to implement the *OpenMDAO*-components – connections between the framework and the solvers, providing access to the above mentioned internal routines of the solvers. The single-disciplinary solvers themselves, when accessible via *Python* interfaces, need not be written in *Python*. Further, the components together with their respective solvers can run in their own process, allowing the computational work to be split among parallel running resources/machines.

1.2 Scope and Outline

Scope

The present work is a feasibility study testing the coupling capabilities of *OpenMDAO* for nonlinear partial differential equations. The feasibility of using *OpenMDAO* on actual large-scale industrial problems is assessed, based on the experiences made with *OpenMDAO* applied to a small academic problem. From the field of fluid dynamics and heat transfer, two single-disciplinary two-dimensional stationary solvers, one for the nonlinear NAVIER-STOKES equations and one for the linear convection-diffusion equation, are constructed. Both of them use the continuous spectral/hp element GALERKIN method [14, 12, 10] – selected because of its extension capabilities – for discretization, and both of them have deliberately implemented all subroutines necessary for coupling. They are acting as placeholders for more sophisticated solvers, and are therefore neither optimized for speed nor memory usage, but for readability and seamless integration into the *OpenMDAO* framework. For each single-disciplinary solver, an *OpenMDAO*-component wrapper, necessary for communication between the solvers and *OpenMDAO*, is implemented, calling the subroutines and providing them to the top-level solvers. As a benchmark problem, both single-disciplinary solvers are coupled by *OpenMDAO* to solve the stationary natural convection of a fluid in a square cavity [8] – a standard multidisciplinary problem

in physics and engineering. Various coupling methods, some of which support MPI-parallel execution, are implemented, tested, and compared against each other and their theoretical expectation. A special aim is to check the MPI-parallelization features of *OpenMDAO*, independently of whether or not an actual speedup is expected.

Outline

In Chapter 2, the theoretical foundation and the implementation of coupling methods are explained with abstract single-disciplinary solvers: in Sec. 2.1, the subroutines of the single-disciplinary solvers are introduced, in Sec. 2.2, the coupling methods and their performance estimates are presented, and in Sec. 2.3, the implementation in *OpenMDAO* is sketched. In Chapter 3, the benchmark problem is defined: in Sec. 3.1, the governing equations are presented, in Sec. 3.2 and Sec. 3.3, the solvers for both the convection-diffusion equation and the NAVIER-STOKES equations are outlined, and in Sec. 3.4, the corresponding *OpenMDAO*-Components are implemented. In Chapter 4, the performance of the coupling methods for the benchmark problem is evaluated: in Sec. 4.2, the solutions are verified against a reference, and in Sec. 4.3, the iterative convergence of the methods is examined. Finally in Chapter 5, the work is concluded: in Sec. 5.1, the results are summarized, and in Sec. 5.2, future work is proposed.

2 Coupling and Solving in *OpenMDAO*

2.1 Single-disciplinary Solvers

Consider a determining nonlinear partial differential equation NEWTON-KRYLOV

$$\mathcal{F}(x, y) \overset{!}{=} 0 \tag{2.1}$$

for an unknown function x given the function y. Discretization gives a determining system of nonlinear algebraic equations[2]

$$r_x(x; y) \overset{!}{=} 0 \tag{2.2}$$

for unknown coefficients vector $\boldsymbol{x}$ (output) and given coefficients vector $\boldsymbol{y}$ (input). Strictly mathematically speaking, the (possibly different) bases of these vectors are sets of ansatz functions depending on the method of discretization, such that the functions which these vectors represent are linear combinations of said ansatz functions – though usually, the entries are just the function values on discrete grid-points. To solve such systems, often (though not always) solvers employ the NEWTON-KRYLOV method. By NEWTON's method, given a guess for $\boldsymbol{x}$, the system is linearized, giving

$$\left.\frac{\partial \boldsymbol{r}_x}{\partial \boldsymbol{x}}\right|_{x;y} \Delta \boldsymbol{x} \overset{!}{=} -\boldsymbol{r}_x(\boldsymbol{x}; \boldsymbol{y}), \tag{2.3}$$

which is to be solved for updates $\Delta \boldsymbol{x}$ such that $\boldsymbol{x} \leftarrow \boldsymbol{x} + \Delta \boldsymbol{x}$. Depending on the method of discretization, the linear system (2.3) is usually big but sparse, such that KRYLOV methods, e.g. the matrix-free restarted generalized minimal residual (GMRES) method [20], are often employed. The method is especially popular because the JACOBI matrix $\partial \boldsymbol{r}_x / \partial \boldsymbol{x}$ only needs to be provided as operator, as KRYLOV methods only require matrix-vector products with said matrix.

[2] The subscript x shall denote that this system is solvable only for the variable $\boldsymbol{x}$, respectively that only the JACOBIan $\partial \boldsymbol{r}_x / \partial \boldsymbol{x}$ is invertible.

Subroutines

In single-disciplinary solvers, the steps are usually split into different auxiliary subroutines: for discretization

- `get_vec(`y`)`: returning coefficients $\boldsymbol{y}$ of a user-given function y,
- `get_eval(`$\boldsymbol{x}$`,`P`)`: returning the evaluation $x|_P$ of the $\boldsymbol{x}$-based solution x at user-given point(s) P,

and for solving using the NEWTON-KRYLOV method

- `get_residual(`$\boldsymbol{x}$`;`$\boldsymbol{y}$`)`: returning $r_x(\boldsymbol{x};\boldsymbol{y})$,
- `calc_jacobians(`$\boldsymbol{x}$`;`$\boldsymbol{y}$`)`: pre-calculating $\left.\frac{\partial r_x}{\partial \boldsymbol{x}}\right|_{x;y}, \left.\frac{\partial r_x}{\partial \boldsymbol{y}}\right|_{x;y}$,
- `get_dresidual(`$\mathrm{d}\boldsymbol{x}$`;`$\mathrm{d}\boldsymbol{y}$`)`: returning $\mathrm{d}r_x = \frac{\partial r_x}{\partial \boldsymbol{x}}\mathrm{d}\boldsymbol{x} + \frac{\partial r_x}{\partial \boldsymbol{y}}\mathrm{d}\boldsymbol{y}$,
- `get_update(`$\boldsymbol{b}$`)`: returning the solution to `get_dresidual(`$\Delta\boldsymbol{x}$`;`0`)` $\overset{!}{=} \boldsymbol{b}$.

The NEWTON-KRYLOV method can then be implemented by Algorithm 1. The single-disciplinary solver providing `get_solution` will, at first glance, appear as black-box: taking $\boldsymbol{y}$ and returning $\boldsymbol{x}$. By looking one level inside, the subroutines allow us to construct more efficient/robust multidisciplinary solvers coupling single-disciplinary solvers. Requiring each of them to have implemented Algorithm 1 and its subroutines is a rather restrictive requirement – especially the linearization with respect to $\boldsymbol{y}$ is usually not provided, but may be evaluated by (complex step) finite-differences. Nonetheless this will be deliberately imposed in this feasibility study.

Algorithm 1 NEWTON-KRYLOV method in single-disciplinary solvers

```
function get_solution(y)
    while termination condition not met do
        r_x ← get_residuals(x;y)
        calc_jacobians(x;y)
        Δx ← get_update(−r)
        x ← x + Δx
    end while
    return x
end function
```

2.2 Coupled Solving

Coupled problem

Consider now a general determining system of two nonlinear partial differential equations

$$\mathcal{F}(x,y) \overset{!}{=} 0, \quad \mathcal{G}(y,x) \overset{!}{=} 0, \tag{2.4}$$

for unknown functions x and y. Note that, in $\mathcal{G}$ the roles of x and y are switched, where y would be treated as unknown and x as known. Instead of making up

a monolithic discretization and linearization for the whole problem (2.4), two single-disciplinary solvers from Sec. 2.1 are coupled, from which only the auxiliary subroutines are accessible. Hence, the following equations are given in operator form, but for theoretical examination also in matrix form – though the matrices are never given explicitly. With `f` and `g` being the single-disciplinary solvers for $\mathcal{F}$ and $\mathcal{G}$, the discretization of (2.4) reads

$$r_x(x;y) = \texttt{f.get\_residuals}(x;y) \overset{!}{=} 0, \tag{2.5a}$$

$$r_y(y;x) = \texttt{g.get\_residuals}(y;x) \overset{!}{=} 0, \tag{2.5b}$$

for unknown coefficients x and y. Note that, in `g`, now y acts as output and x as input. It is, for now, assumed that the vectors are represented in the same basis, e.g. by the same grid. The nonlinear problem (2.5) can iteratively be solved either by gradient-based methods, such as NEWTON, or by generalization of linear splitting methods to nonlinear problems, such as nonlinear block-GAUß-SEIDEL.

NEWTON's Method

The application of NEWTON's method on the coupled problem (2.5) requires

$$\frac{\partial r_x}{\partial x}\Delta x + \frac{\partial r_x}{\partial y}\Delta y \;= \texttt{f.get\_dresidual}(\Delta x;\Delta y) \overset{!}{=} -r_x, \tag{2.6a}$$

$$\frac{\partial r_y}{\partial x}\Delta x + \frac{\partial r_y}{\partial y}\Delta y \;= \texttt{g.get\_dresidual}(\Delta y;\Delta x) \overset{!}{=} -r_y, \tag{2.6b}$$

to be solved for updates Δx and Δy such that $x \leftarrow x + \Delta x$ and $y \leftarrow y + \Delta y$. System (2.6) could be solved by iterative methods, such as splitting or KRYLOV methods, but not having any further information on the linear system (e.g. symmetry, definiteness, diagonal dominance) makes it difficult to choose an efficient solver. With the single-disciplinary solvers, there is still access to the update-solvers `get_update`, respectively to the approximate left-hand side multiplication of the inverses $\partial r_x/\partial x^{-1}$ and $\partial r_y/\partial y^{-1}$. Employing these in the solution process of (2.6) leads to two suggestions:

- *NEWTON-block-JACOBI*: ignoring the 'off-diagonal' terms $\partial r_x/\partial y$ and $\partial r_y/\partial x$ in (2.6) gives the explicit expression

$$\Delta x \approx -\frac{\partial r_x}{\partial x}^{-1} r_x = \texttt{f.get\_update}(-r_x) \tag{2.7a}$$

$$\Delta y \approx -\frac{\partial r_y}{\partial y}^{-1} r_y = \texttt{g.get\_update}(-r_y) \tag{2.7b}$$

  such that no additional solver for the linear system is required and `get_update` must only be applied once per NEWTON iteration. The modification from (2.6) to (2.7) can be interpreted as solving (2.6) with one step of the zero-initialized linear block-JACOBI method [see 19, Eq. 7.4.24]. By [see 19, Th. 10.3.1], the spectral radius $\rho(\cdot)$, and the condition num-

ber $\kappa(\cdot)$, NEWTON's method converges, locally in the neighborhood of the solution (x^*, y^*), linear at best, if the asymptotic rate of convergence μ is less than unity, where

$$\mu = \max\left\{ \sqrt{\rho\left(\frac{\partial r_y}{\partial y}^{-1} \frac{\partial r_y}{\partial x} \frac{\partial r_x}{\partial x}^{-1} \frac{\partial r_x}{\partial y} \right)\Big|_{x^*,y^*}}, \sqrt{\rho\left(\frac{\partial r_x}{\partial x}^{-1} \frac{\partial r_x}{\partial y} \frac{\partial r_y}{\partial y}^{-1} \frac{\partial r_y}{\partial x} \right)\Big|_{x^*,y^*}} \right\}$$
$$\leq \sqrt{\kappa\left(\frac{\partial r_y}{\partial y} \right) \frac{\left\| \frac{\partial r_y}{\partial x} \right\|}{\left\| \frac{\partial r_y}{\partial y} \right\|} \cdot \kappa\left(\frac{\partial r_x}{\partial x} \right) \frac{\left\| \frac{\partial r_x}{\partial y} \right\|}{\left\| \frac{\partial r_x}{\partial x} \right\|}\Bigg|_{x^*,y^*}}, \tag{2.8}$$

meaning the norms of the 'off-diagonal' terms in system (2.6) have to be sufficiently smaller than of the 'diagonal' terms.[1] Applying more than one step, e.g. n steps, of the block-JACOBI method is not considered here: for small n, the amount of `get_update` calls per nonlinear iteration is proportional to n, but, by [see 19, Th. 10.3.1], the asymptotic rate of convergence is just μ^n, meaning the amount of nonlinear iterations is just inverse proportional to n; hence there is no difference for the total amount of `get_update` calls. Also not considered is applying as many steps as necessary to solve system (2.6) to a given tolerance, making NEWTON's method converge quadratic: that would require the right-hand side of (2.8) to be less than unity for all encountered (x, y), not just for (x^*, y^*), which can not safely be ensured. Hence, from now on, by the NEWTON-block-JACOBI method, the *one*-step variant is meant. The linear block-GAUß-SEIDEL method is likewise not considered as solver here because it can not run in parallel – though, the single-disciplinary solvers may internally run in a parallel fashion.

- *Block-JACOBI preconditioned NEWTON-KRYLOV*: multiplying (2.6) with the inverse 'diagonal' terms gives the equivalent system

$$\Delta x + \frac{\partial r_x}{\partial x}^{-1} \frac{\partial r_x}{\partial y} \Delta y \overset{!}{=} -\frac{\partial r_x}{\partial x}^{-1} r_x, \tag{2.9aa}$$
$$\frac{\partial r_y}{\partial y}^{-1} \frac{\partial r_y}{\partial x} \Delta x + \Delta y \overset{!}{=} -\frac{\partial r_y}{\partial y}^{-1} r_y, \tag{2.9ab}$$

respectively

$$\texttt{f.get\_update((f.get\_dresidual}(\Delta x; \Delta y)) \overset{!}{=} \texttt{f.get\_update}(-r_x), \tag{2.9ba}$$
$$\texttt{g.get\_update((g.get\_dresidual}(\Delta y; \Delta x)) \overset{!}{=} \texttt{g.get\_update}(-r_y). \tag{2.9bb}$$

The modification from (2.6) to (2.9a) can be interpreted as left-preconditioning with one step of the zero-initialized linear block-JACOBI method. Sys-

[1] The identity $\rho\left(\begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}\right) = \max\left(\sqrt{\rho(AB)}, \sqrt{\rho(BA)}\right)$ was used.

tem (2.9), which has the same solution as (2.6), *still* has to be solved for updates, but *may* be better *conditioned* than system (2.6), *even* if the norms of the 'off-diagonal' terms in system (2.6) are larger than of the 'diagonal' terms – one can easily show that the success of the linear block-JACOBI method is not necessary for the success of *one*-step block-JACOBI preconditioning. Applying more than one step for preconditioning, say m, may possibly further improve the condition, but could also possibly worsen the condition. Still not having any further information on the linear system (2.9), matrix-free restarted GMRES is chosen as solver. Note that, this top-level GMRES solver considers the full system (2.9) without any omittings. As a rule, an improved condition of the system (2.9) greatly reduces the amount of GMRES iterations. For each NEWTON iteration, solving the preconditioned system requires several GMRES iterations, in which the preconditioner must apply `get_update` m times. Again, linear block-GAUß-SEIDEL is not considered for preconditioning as it can not run in parallel. Provided the return values of the operators in (2.9b) are sufficiently accurate, solving (2.9b) sufficiently accurate makes NEWTON's method converge locally quadratic at least and qubic at most; a detailed analysis of how the accuracy determines the convergence is given in [9].

GAUß-SEIDEL Method

With the nonlinear block-GAUß-SEIDEL method [see 19, Eq. 7.4.26], first $\boldsymbol{x}$ is updated from a guess on $\boldsymbol{y}$

$$\boldsymbol{x} \leftarrow \texttt{f.get\_solution}(\boldsymbol{y}), \tag{2.3a}$$

which is then used to update $\boldsymbol{y}$

$$\boldsymbol{y} \leftarrow \texttt{g.get\_solution}(\boldsymbol{x}). \tag{2.3b}$$

If system (2.6) were linear, the method would reduce to the classical linear block-GAUß-SEIDEL; hence the name. Again, by [see 19, Th. 10.3.5], the method converges, locally in the neighborhood of the solution $(\boldsymbol{x}^*, \boldsymbol{y}^*)$, linear at best if the asymptotic rate of convergence μ is less than unity, where

$$\mu = \rho\left(\frac{\partial \boldsymbol{r}_y}{\partial \boldsymbol{y}}^{-1} \frac{\partial \boldsymbol{r}_y}{\partial \boldsymbol{x}} \frac{\partial \boldsymbol{r}_x}{\partial \boldsymbol{x}}^{-1} \frac{\partial \boldsymbol{r}_x}{\partial \boldsymbol{y}}\right)\Bigg|_{\boldsymbol{x}^*,\boldsymbol{y}^*} \leq \kappa\left(\frac{\partial \boldsymbol{r}_y}{\partial \boldsymbol{y}}\right) \frac{\left\|\frac{\partial \boldsymbol{r}_y}{\partial \boldsymbol{x}}\right\|}{\left\|\frac{\partial \boldsymbol{r}_y}{\partial \boldsymbol{y}}\right\|} \cdot \kappa\left(\frac{\partial \boldsymbol{r}_x}{\partial \boldsymbol{x}}\right) \frac{\left\|\frac{\partial \boldsymbol{r}_x}{\partial \boldsymbol{y}}\right\|}{\left\|\frac{\partial \boldsymbol{r}_x}{\partial \boldsymbol{x}}\right\|}\Bigg|_{\boldsymbol{x}^*,\boldsymbol{y}^*}, \tag{2.4}$$

meaning the norms of the 'off-diagonal' terms in system (2.6) still have to be sufficiently smaller than of the 'diagonal' terms.[1] By (2.8), the convergence rate is better in comparison to NEWTON-block-JACOBI, making nonlinear block-GAUß-SEIDEL asymptotically require only half the amount of nonlinear iterations.

[1] The identity $\rho\left(\begin{bmatrix} 0 & A \\ 0 & B \end{bmatrix}\right) = \rho(\boldsymbol{B})$ was used.

Yet each call of `get_solution`, by Alg. 1, requires several internal calls of `get_update`. The nonlinear block-JACOBI method is not considered: while its convergence rate is, by [see 19, Th. 10.3.5, applied to JACOBI], identical to one-step NEWTON-block-JACOBI, calling `get_solution` every iteration is far costlier than calling `get_update` every iteration.

Performance Estimates

Considering the amount of nonlinear iterations, by rate and order of convergence, block-JACOBI preconditioned NEWTON-KRYLOV (JNK) surely requires the least iterations, NEWTON-block-JACOBI (NJ) the most and nonlinear block-GAUß-SEIDEL (GS) likely something in between. For the latter two, with increasing norms of the coupling terms, by (2.8) and (2.4), this amount is expected to grow drastically until convergence is no longer possible. For NJ, the limit of eligible coupling norms can artificially be extended by employing backtracking line search schemes. Considering now the amount of `get_update` calls *per* nonlinear iteration, JNK most likely requires the most calls, NJ the least and GS something in between. Under these considerations, taking the *total* amount of `get_update` calls as the metric of the computational work for coupling, break-even points with respect to the tightness of the coupling, i.e. the norms of the 'off-diagonal' terms, are predicted: for small norms GS should perform best, for moderate norms NJ should perform best and for big norms JNK should perform best.

2.3 Implementation

Top-Level Solvers

OpenMDAO [13, 18] provides pre-written solvers in modular form, which the user can combine to implement various coupling methods. This way, all methods from Sec. 2.2 can be implemented easily. The top-level NEWTON solver `NewtonSolver` handles the evaluation of the nonlinear residuals and initializes the linear system of operators. By setting `rtol=0` and `atol=`$mtol\sqrt{DOF}$, the solver will iterate until the absolute root mean square residual of the nonlinear system (2.5) with respect to all degrees of freedom *DOF* is below a tolerance *mtol*.[1] Further, setting `solve_subsystems=True` and `max_sub_solves=0` the initial (0-th) guess is improved by applying one single iteration of the nonlinear block-GAUß-SEIDEL method.

- For NEWTON-block-JACOBI, the evaluation of the updates is handled by the top-level block-JACOBI solver `LinearBlockJac`[2]. With `maxiter=1` set, only one step of the block-JACOBI method is applied. The backtracking line searcher `ArmijoGoldsteinLS` handles the application of the NEWTON-

[1] From an engineering point of view, *not* the norm of the residual vector is of interest, but the *mean* residual *per* degree of freedom.

[2] The standard code of `openmdao` does not support zero-initialization. This is why, in `linear_block_jac.py`, the iteration is modified, such that `b_vec` is set just to `self._rhs_vec` if `self._iter_count == 0`.

updates, ensuring the the ARMIJO-GOLDSTEIN condition [2]. With `maxiter`, `rho` and `c`, the maximum number of contractions, the contraction factor and the slope parameter, are set.

- For block-JACOBI preconditioned NEWTON-KRYLOV, the top-level KRYLOV solver `PETScKrylov` handles the solution of the preconditioned linear system of operators. Again, by setting `rtol=0` and `atol=`$mtol_{\text{lin}}\sqrt{DOF}$, the solver will iterate until the absolute root mean square residual of the linear system (2.6) with respect to all degrees of freedom *DOF* is below a tolerance $mtol_{\text{lin}}$. With `ksp_type='gmres'`, `restart=20` and `precon_side='left'` the back-end *PETSc* solver [4, 7] is set to restarted GMRES(20) and the preconditioning is set to left-hand sided. Preconditioning is again handled by the top-level block-JACOBI solver `LinearBlockJac` with `maxiter=`*m* set, so to make no more than *m* preconditioning iterations.

The top-level nonlinear block-GAUß-SEIDEL solver is given by `NonlinearBlockGS`, where again, `rtol=0` and `atol=`$mtol\sqrt{DOF}$ are set. For a more detailed description of the parameters, refer to [18].

`ImplicitComponent`

The communication between *OpenMDAO* and the single-disciplinary solvers is handled through instances of subclasses of `openmdao`'s `ImplicitComponent` class. Figure 2.1 shows a simplified sequence diagram with a subclass in action. Each subclass has to provide the following routines:

- `apply_nonlinear(inputs, outputs, residuals)`: Computing residuals `residuals` from given `inputs` and `outputs`,
- `linearize(inputs, outputs)`: Pre-calculating the linearization at given `inputs` and `outputs`,
- `apply_linear(d_inputs, d_outputs, d_residuals)`: Computing differentials `d_residuals` from given differentials `d_inputs` and `d_outputs`,
- `solve_linear(d_outputs, d_residuals)`: Computing the updates `d_outputs` from given differentials `d_residuals`,
- `solve_nonlinear(inputs, outputs)`: Computing the nonlinear solutions `outputs` from given inputs `inputs`.

By requirement of Sec. 2.1, these routines will just have to call the subroutines concerning NEWTON's method (Alg. 1) from the single-disciplinary solvers. This way, the single-disciplinary solvers can be written in any language as long as translations of arrays to and from *NumPy* arrays are possible. However, the discretization of the variables, i.e. the basis of the vectors, respectively the ansatz functions, may differ for the single-disciplinary solvers: the `f`-component, with $\boldsymbol{x}$ as output and $\boldsymbol{y}$ as input, will return $\boldsymbol{x}$ in its basis and receive $\boldsymbol{y}$ in the basis of

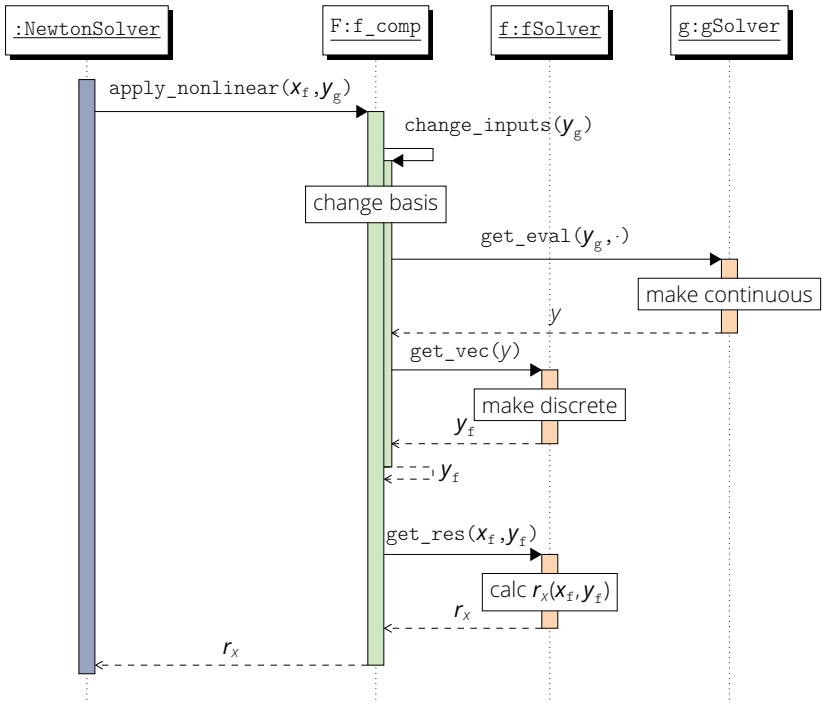

Figure 2.1: Sequence diagram of the top-level NEWTON solver calling the current residual r_x. The `f`-component `F` receives the current values x_f in its basis and y_g in the basis of `g`. For change of basis, y_g is handed over to the function `change_inputs`: from `g`, a `callable` y is requested, which allows `f` to return y_f in its basis. With both x_f and y_f now in the basis of `f`, the variables are handed over to the single-disciplinary solver `f`, calculating the residuals r_x. The residual is then returned to the top-level NEWTON solver for further processing. Not shown is the `g`-component `G` – which would, in parallel, do the same with switched roles of the variables – and the possibly necessary translations between *NumPy* and the array formats of `f` and `g`.

`g`; in turn, the `g`-component, will return y in its basis and receive x in the basis of `f`. Hence, each component additionally requires a function `change_inputs` to change the basis of its input (and their differentials) to the basis of its single-disciplinary solver.[1] If the methods of discretization are not known or not easy to implement, the function can again be implemented by calling the subroutines concerning the discretization from *both* of the single-disciplinary solvers. Having a component communicate with more than one single-disciplinary solver naturally reduces exchangeability and modularity but could not practicably be avoided: one alternative, each solver interpolating to, respectively discretizing

[1] Users of *OpenMDAO* might consider performing the change of basis in subclasses of `ExplicitComponent`; this is *not* recommended when zero-initialized block-JACOBI is used. *OpenMDAO* reformulates explicit relationships into implicit ones and appends these to the linearized system. One can show, by examination of the iteration matrix, that the originally explicit variables and the originally implicit variables are updated alternatingly, such that twice the amount of block-JACOBI iterations are necessary.

from, a common grid would require the grid to be incredibly fine to avoid additional discretization error entering in each transfer of variables.

Top-Level Solver Code

With the components implemented, writing the top-level *Python* script running *OpenMDAO* parallelized by MPI is remarkably easy. First, `openmdao` and `mpi4py` [6, 17] are imported, and for each process instances of the single-disciplinary solver are initialized. All MPI-communication shall be handled by *OpenMDAO*, so, for the components to be able to perform change of basis, each MPI-process requires instances of all solvers.

```
import openmdao.api as om
from mpi4py import MPI
rank = MPI.COMM_WORLD.Get_rank() # rank of the process
...
if rank == 0:
 f = fSolver(...) # create f solver for solving and for change of basis
 g = gSolver(...) # create g solver for change of basis
if rank == 1:
 f = fSolver(...) # create f solver for change of basis
 g = gSolver(...) # create g solver for solving and for change of basis
```

The respective components are added to a parallel group where *OpenMDAO* assigns the first component to the first MPI-process (`rank 0`) and the second component to the second MPI-process (`rank 1`).[1] Note that, this order also defines the order in which nonlinear block-GAUß-SEIDEL runs. By connecting the variables, the cross-MPI-process distribution of the variables and their differentials is handled by *OpenMDAO*.

```
prob = om.Problem() # initialize problem
pg = prob.model.add_subsystem('PG', om.ParallelGroup()) # add parallel group
pg.add_subsystem('G', g_comp(solver_g=g, solver_f=f)) # add g-component (rank=1)
pg.add_subsystem('F', f_comp(solver_f=f, solver_g=g)) # add f-component (rank=0)
pg.connect('G.x_g', 'F.x_g') # plug x into F
pg.connect('F.y_f', 'G.y_f') # plug y into G
```

The NEWTON-block-JACOBI method would then be implemented by

```
pg.nonlinear_solver = om.NewtonSolver(...) # top-level NEWTON solver
pg.linear_solver = om.LinearBlockJac(...) # top-level linear block-JACOBI solver
```

the block-JACOBI preconditioned NEWTON-KRYLOV method by

```
pg.nonlinear_solver = om.NewtonSolver(...) # top-level NEWTON solver
pg.nonlinear_solver.linesearch = om.ArmijoGoldsteinLS(...) # backtracking
pg.linear_solver = om.PETScKrylov(...) # top-level KRYLOV solver
pg.linear_solver.precon = om.LinearBlockJac(...) # linear block-JACOBI precon
```

and the nonlinear block-GAUß-SEIDEL method by

[1] For cases where the amount of MPI-processes is not equal to the amount of components, the distribution is different; the algorithm is defined in `default_allocator.py`.

```
pg.nonlinear_solver = om.NonlinearBlockGS(...) # top-level nonlin
 ↪ block-GAUSS-SEIDEL solver
```

Of course, for the latter, the components can no longer be run in parallel, though still in different MPI-processes.[1] The model can then be run and the variables can be extracted and gathered for post-processing.

```
prob.setup()
prob.run_model() # run the model
if rank == 0:
 result = pg.get_val('F.x_f') # get x in f discretization
if rank == 1:
 result = pg.get_val('G.y_g') # get y in g discretization
result = MPI.COMM_WORLD.gather(result, root=0) # 0th process gathers results
if rank == 0:
 print(result[0], result[1]) # post-processing in 0th process
```

The above top-level script is run with two MPI-processes by

```
$ mpirun -n 2 python MyTopLevelScript.py
```

[1] Acknowledging this, `nonlinear_block_gs.py` is modified so that the exception when trying to use nonlinear block-GAUß-SEIDEL on parallel groups is no longer raised.

3 Benchmark Problem

3.1 Natural Convection

Problem

To test the considerations of Sec. 2, a simple multidisciplinary nonlinear problem is implemented: stationary natural convection of a fluid in a rectangular cavity is chosen, because of its popularity as benchmark problem. The problem can be implemented rather easily: The domain for all variables is identical and rectangular, so spatial discretization is simple, the boundary conditions are fixed and homogeneous, and the problem is of parabolic nature (i. e. diffusive). Nonstationarity is not considered as to make no error from temporal discretization and as to keep verification simple. As in Fig. 3.1, the left-hand wall is kept isothermal at $\Delta\check{T}/2$ above ambient temperature, and the right-hand wall is kept at $\Delta\check{T}/2$ below ambient temperature, with the floor and ceiling being adiabatic. All sides are considered solid boundaries to the fluid, so there is neither normal nor tangential flow at the sides (no-slip condition). A fluid element at the hot wall will increase in temperature, decrease in density, and rise to the ceiling; meanwhile at the cold wall it will decrease in temperature, increase in density, and fall to the floor. The resulting clockwise circulation will both transport the heat and be driven by it.

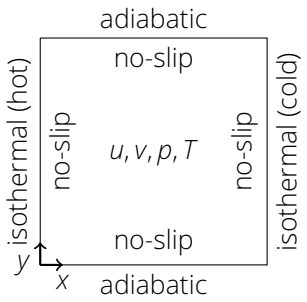

Figure 3.1: Schematic cavity for buoyancy-driven recirculation.

Continuous Equations

Approximately describing this problem of natural convection are the BOUSSINESQ equations. From [see 11, Sec. 1.6], in steady-state dimensionless form they read: For all $(x, y) \in [0, L_x] \times [0, L_y] =: \Omega$

$$Pé \begin{bmatrix} u \\ v \end{bmatrix} \circ \nabla T \overset{!}{=} \nabla^2 T, \tag{3.1a}$$

$$Re \left(\begin{bmatrix} u \\ v \end{bmatrix} \circ \nabla \right) \begin{bmatrix} u \\ v \end{bmatrix} \overset{!}{=} -\nabla p + \nabla^2 \begin{bmatrix} u \\ v \end{bmatrix} + \frac{Gr}{Re} T \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \tag{3.1b}$$

$$\nabla \circ \begin{bmatrix} u \\ v \end{bmatrix} \overset{!}{=} 0. \tag{3.1c}$$

With boundary conditions

$$u \overset{!}{=} 0, \qquad v \overset{!}{=} 0 \qquad \forall\,(x,y) \in \partial\Omega \qquad \text{(no slip)}, \tag{3.2a}$$

$$T|_{x=0} \overset{!}{=} \frac{1}{2}, \qquad T|_{x=L_x} \overset{!}{=} \frac{-1}{2} \qquad \forall\, y \in [0, L_y] \qquad \text{(isothermal)}, \tag{3.2b}$$

$$\left.\frac{\partial T}{\partial y}\right|_{y=0} \overset{!}{=} 0, \qquad \left.\frac{\partial T}{\partial y}\right|_{y=L_y} \overset{!}{=} 0 \qquad \forall\, x \in [0, L_x] \qquad \text{(adiabatic)}. \tag{3.2c}$$

Here L_x and L_y are the dimensionless width and height (scaled by $\check{L}_{\text{ref}}$) of the cavity, u and v are the dimensionless velocities (scaled by $\check{U}_{\text{ref}}$), T is the dimensionless temperature perturbation against ambient temperature (scaled by $\Delta\check{T}$), and p is the dimensionless pressure (scaled by $\check{\nu}\check{\rho}_0\check{U}_{\text{ref}}/\check{L}_{\text{ref}}$). Further,

$$Re = \frac{\check{U}_{\text{ref}}\check{L}_{\text{ref}}}{\check{\nu}}, \quad Pé = \frac{\check{U}_{\text{ref}}\check{L}_{\text{ref}}}{\check{a}} = Pr\,Re, \quad Pr = \frac{\check{\nu}}{\check{a}} := 0.71, \quad Gr = \frac{\check{g}\check{\beta}\Delta\check{T}\check{L}_{\text{ref}}^3}{\check{\nu}^2} \tag{3.3}$$

are the REYNOLDS number, the PÉCLET number, the PRANDTL number (of air) and the GRASHOF number. Note that the form of the momentum equation (3.1b) differs from literature, as the diffusive terms, and not the convective terms, are free of dimensionless numbers. This is done because the problems considered are mainly dominated by diffusion rather than by convection. With the GRASHOF number given implicitly by the RAYLEIGH number $Ra = Gr\,Pr$ of the problem, the REYNOLDS number can be chosen arbitrarily: from an analytical point of view, a high value increases nonlinearity in (3.1b), increases the coupling term in (3.1a), and decreases the coupling term in (3.1b).

Single-Disciplinary Solvers

To test the coupling capabilities, two single-disciplinary solvers are implemented: one for the convection-diffusion equation (3.1a) with boundary conditions (3.2b)-(3.2c) and one for the NAVIER-STOKES equations (3.1b)-(3.1c) with boundary condition (3.2a). For testing purposes, these solvers are deliberately implemented from scratch in *Python* according to the requirements from Sec. 2.1, as to allow for seamless integration into *OpenMDAO*. To keep matters simple, both solvers use the same method of discretization: continuous spectral/hp element GALERKIN method with nodal LAGRANGE polynomial basis and GAUß-LOBATTO-LEGENDRE quadrature. Appendix A and B describe the basic formulation of the applied method; for a rigorous description see for example [14, 12, 10]. The method was chosen because of its generality and its extension capabilities: the basic formulation from this work can easily be extended to complex geometries, different bases and quadrature rules, and can even be made discontinuous respectively conservative.

3.2 Convection-Diffusion Solver

Discretization

With the convection velocities u and v given, and the temperature perturbation T unknown, the convection-diffusion equation (3.1a) with boundary conditions (3.2b)-(3.2c) acts as the determining partial differential equation for T. Neglecting the DIRICHLET boundary condition (3.2b) for now, discretizing (3.1a) together with (3.2c) in the same fashion as in Appendix B gives a determining system of algebraic equations

$$r_T := Pé\left(uC^x + vC^y\right)T + KT \overset{!}{=} 0 \tag{3.4}$$

with differentials

$$\mathrm{d}r_T = Pé\left(uC^x + vC^y\right)\mathrm{d}T + K\mathrm{d}T + Pé\left(C^xT\right)\mathrm{d}u + Pé\left(C^yT\right)\mathrm{d}v. \tag{3.5}$$

To apply the DIRICHLET boundary conditions (3.2b), the residuals and hence their differentials are modified

$$(r_T)_p := T_p - \frac{1}{2}\ \forall\, p \in \{p\,|\,x_p = 0\}, \quad (r_T)_p := T_p - \frac{-1}{2}\ \forall\, p \in \{p\,|\,x_p = L_x\}, \tag{3.6a}$$

$$\mathrm{d}(r_T)_p = \mathrm{d}T_p \quad \forall\, p \in \{p\,|\,x_p = L_x \vee x_p = 0\}. \tag{3.6b}$$

Solving

To solve the linearized system for updates ΔT, the system

$$\frac{\partial r_T}{\partial T}\Delta T \overset{!}{=} \Delta r_T \tag{3.7}$$

is solved with given right-hand side Δr_T using the matrix-free restarted loose generalized minimal residual (LGMRES) method [3]. The required matrix-vector products of $\partial r_T/\partial T$ with arbitrary vectors are easily computed according to (3.5) and (3.6b).

Implementation

The auxiliary subroutines from Sec. 2.1 are implemented accordingly:

- `get_residual(`T`;`u`,`v`)`: returning r_T from (3.4) and (3.6a),
- `calc_jacobians(`T`;`u`,`v`)`: pre-calculating the matrices in (3.5),
- `get_dresidual(`$\mathrm{d}T$`;`$\mathrm{d}u$`,`$\mathrm{d}v$`)`: returning $\mathrm{d}r_T$ from (3.5) and (3.6b),
- `get_update(`Δr_T`)`: returning ΔT from (3.7).

Both `get_vec` and `get_eval` are implemented according to Appendix B. The *Python* implementation for the solver is given in `ConvectionDiffusion_Solver.py`[1].

[1] `https://github.com/SEhrm/sim-comp/blob/main/Solvers/ConvectionDiffusion_Solver.py`

3.3 NAVIER-STOKES Solver

Discretization

With the temperature perturbation T given, and the pressure p and velocities u and v unknown, the NAVIER-STOKES equations (3.1b)-(3.1c) with boundary condition (3.2a) act as the determining partial differential equation for u, v and p. To make the problem well-posed, a reference DIRICHLET condition

$$p|_{x=L_x/2, y=L_y/2} \overset{!}{=} 0 \tag{3.8}$$

and an artificial homogeneous NEUMANN boundary condition

$$\frac{\partial p}{\partial n} \overset{!}{=} 0 \; \forall (x, y) \in \Omega \tag{3.9}$$

are imposed for the pressure. Neglecting the DIRICHLET boundary condition (3.2a) and (3.8) for now, discretizing (3.1b)-(3.1c) in the same fashion as in section B gives

$$r_u := Re\left(\boldsymbol{u}\boldsymbol{C}^x + \boldsymbol{v}\boldsymbol{C}^y\right)\boldsymbol{u} + \boldsymbol{K}\boldsymbol{u} + \boldsymbol{G}^x\boldsymbol{p} \overset{!}{=} \boldsymbol{0}, \tag{3.10a}$$

$$r_v := Re\left(\boldsymbol{u}\boldsymbol{C}^x + \boldsymbol{v}\boldsymbol{C}^y\right)\boldsymbol{v} + \boldsymbol{K}\boldsymbol{v} + \boldsymbol{G}^y\boldsymbol{p} - \frac{Gr}{Re}\boldsymbol{M}\boldsymbol{T} \overset{!}{=} \boldsymbol{0}, \tag{3.10b}$$

$$r_{\text{cont}} := \boldsymbol{G}^x\boldsymbol{u} + \boldsymbol{G}^x\boldsymbol{v} \overset{!}{=} \boldsymbol{0}, \tag{3.10c}$$

with differentials

$$\mathrm{d}r_u = Re\left(\boldsymbol{u}\boldsymbol{C}^x + \boldsymbol{v}\boldsymbol{C}^y + \boldsymbol{C}^x\boldsymbol{u}\right)\mathrm{d}\boldsymbol{u} + \boldsymbol{K}\mathrm{d}\boldsymbol{u} + Re\left(\boldsymbol{C}^y\boldsymbol{u}\right)\mathrm{d}\boldsymbol{v} + \boldsymbol{G}^x\mathrm{d}\boldsymbol{p}, \tag{3.11a}$$

$$\mathrm{d}r_v = Re\left(\boldsymbol{u}\boldsymbol{C}^x + \boldsymbol{v}\boldsymbol{C}^y + \boldsymbol{C}^y\boldsymbol{v}\right)\mathrm{d}\boldsymbol{v} + \boldsymbol{K}\mathrm{d}\boldsymbol{v} + Re\left(\boldsymbol{C}^x\boldsymbol{v}\right)\mathrm{d}\boldsymbol{u} + \boldsymbol{G}^y\mathrm{d}\boldsymbol{p} - \frac{Gr}{Re}\boldsymbol{M}\mathrm{d}\boldsymbol{T}, \tag{3.11b}$$

$$\mathrm{d}r_{\text{cont}} = \boldsymbol{G}^x\mathrm{d}\boldsymbol{u} + \boldsymbol{G}^y\mathrm{d}\boldsymbol{v}. \tag{3.11c}$$

To apply the boundary condition (3.2a) and (3.8)-(3.9), the residuals and their differentials are modified: For all $p \in \{p \,|\, (x_p, y_p) \in \partial\Omega\}$

$$(r_u)_p := u_p, \quad (r_v)_p := v_p, \quad (r_{\text{cont}})_p := \sum_{q=0}^{N-1} K_{pq} p_q, \quad (r_{\text{cont}})_{\lfloor N/2 \rfloor} \overset{!}{=} p_{\lfloor N/2 \rfloor}, \tag{3.12a}$$

$$\mathrm{d}(r_u)_p = \mathrm{d}u_p, \; \mathrm{d}(r_v)_p = \mathrm{d}v_p, \quad \mathrm{d}(r_{\text{cont}})_p = \sum_{q=0}^{N-1} K_{pq}\mathrm{d}p_q, \; \mathrm{d}(r_{\text{cont}})_{\lfloor N/2 \rfloor} = \mathrm{d}p_{\lfloor N/2 \rfloor}. \tag{3.12b}$$

Solving

To solve the linearized system for an update $(\Delta\boldsymbol{u}, \Delta\boldsymbol{v}, \Delta\boldsymbol{p})$, the system

$$\begin{bmatrix} \frac{\partial r_u}{\partial \boldsymbol{u}} & \frac{\partial r_u}{\partial \boldsymbol{v}} & \frac{\partial r_u}{\partial \boldsymbol{p}} \\ \frac{\partial r_v}{\partial \boldsymbol{u}} & \frac{\partial r_v}{\partial \boldsymbol{v}} & \frac{\partial r_v}{\partial \boldsymbol{p}} \\ \frac{\partial r_{\text{cont}}}{\partial \boldsymbol{u}} & \frac{\partial r_{\text{cont}}}{\partial \boldsymbol{v}} & \frac{\partial r_{\text{cont}}}{\partial \boldsymbol{p}} \end{bmatrix} \begin{bmatrix} \Delta\boldsymbol{u} \\ \Delta\boldsymbol{v} \\ \Delta\boldsymbol{p} \end{bmatrix} \overset{!}{=} \begin{bmatrix} \Delta r_u \\ \Delta r_v \\ \Delta r_{\text{cont}} \end{bmatrix} \tag{3.13}$$

is solved with given right-hand side $(\Delta r_u, \Delta r_v, \Delta r_{\text{cont}})$. As the principal submatrix

$$\begin{bmatrix} \frac{\partial r_u}{\partial u} & \frac{\partial r_u}{\partial v} \\ \frac{\partial r_v}{\partial u} & \frac{\partial r_v}{\partial v} \end{bmatrix} =: J_{\text{velo}}$$

is invertible, block-inversion by the SCHUR complement is possible, and the system can be sequentially solved by two smaller systems,

$$\underbrace{\left(\frac{\partial r_{\text{cont}}}{\partial p} - \begin{bmatrix} \frac{\partial r_{\text{cont}}}{\partial u} & \frac{\partial r_{\text{cont}}}{\partial v} \end{bmatrix} J_{\text{velo}}^{-1} \begin{bmatrix} \frac{\partial r_u}{\partial p} \\ \frac{\partial r_v}{\partial p} \end{bmatrix} \right)}_{=:S} \Delta p \overset{!}{=} \underbrace{\Delta r_{\text{cont}} - \begin{bmatrix} \frac{\partial r_{\text{cont}}}{\partial u} & \frac{\partial r_{\text{cont}}}{\partial v} \end{bmatrix} J_{\text{velo}}^{-1} \begin{bmatrix} \Delta r_u \\ \Delta r_v \end{bmatrix}}_{=:b_S}, \tag{3.14a}$$

$$\begin{bmatrix} \Delta u \\ \Delta v \end{bmatrix} \overset{!}{=} J_{\text{velo}}^{-1} \underbrace{\left(\begin{bmatrix} \Delta r_u \\ \Delta r_v \end{bmatrix} - \begin{bmatrix} \frac{\partial r_u}{\partial p} \\ \frac{\partial r_v}{\partial p} \end{bmatrix} \Delta p \right)}_{=:b_{\text{velo}}}. \tag{3.14b}$$

System (3.14a) is solved using the matrix-free preconditioned restarted LGMRES method. The matrix-vector products of the inverse J_{velo}^{-1} with an arbitrary vectors d is computed by solving the system

$$J_{\text{velo}} w \overset{!}{=} d \tag{3.15}$$

for w. Because (3.15) has to be solved several times per LGMRES iteration, J_{velo} is decomposed once by sparse lower-upper decomposition. The fill-in of non-zero entries is expected to be of factor $10 \ldots 100$, but solving for w is then cheaply done by forward/backward substitution. With J_{velo}^{-1} handled, the vectors b_S and b_{velo}, and the matrix-vector products of the SCHUR complement S with arbitrary vectors, are all easily computed according to (3.11) and (3.12b). As preconditioner for the LGMRES solver, S is approximated by the mass matrix M, which can be inverted easily because it is diagonal. The reason for this choice follows heuristically by considering S for the case $Re := 0$ with the boundary conditions neglected [see 14, p. 405]:

$$S \approx -\begin{bmatrix} G^x & G^y \end{bmatrix} \begin{bmatrix} K & \\ & K \end{bmatrix} \begin{bmatrix} G^x \\ G^y \end{bmatrix} \mathrel{\widehat{=}} \nabla^{\mathsf{T}} \operatorname{inv}\left(\nabla^2\right) \nabla \approx 1 \mathrel{\widehat{=}} M.$$

The continuous analogue of S in this case is the identity function whose discrete analogue in turn is M.

Implementation

The auxiliary subroutines from Sec. 2.1 are implemented accordingly:

- `get_residual(`$u, v, p; T$`)`: returning r_u, r_v, r_{cont} from (3.10) and (3.12a),
- `calc_jacobians(`$u, v, p; T$`)`: pre-calculating the matrices in (3.11),
- `get_dresidual(`$\mathrm{d}u, \mathrm{d}v, \mathrm{d}p; \mathrm{d}T$`)` returning $\mathrm{d}r_u$, $\mathrm{d}r_v$, $\mathrm{d}r_{\text{cont}}$ from (3.11) and (3.12b),

- `get_update`($\Delta r_u, \Delta r_v, \Delta r_{\text{cont}}$): returning Δu, Δv, Δp from (3.13).

Both `get_vec` and `get_eval` are implemented according to Appendix B. The *Python* implementation for the solver following from the above considerations is given in `NavierStokes_Solver.py`[1].

3.4 Components

Solver Components

With the convection-diffusion solver and the NAVIER-STOKES solvers implemented, the corresponding subclasses of the components can easily be implemented to the requirements of Sec. 2.3. For the convection-diffusion solver – solving T for given u and v – the implementation of the corresponding component is given below: first, `openmdao` and `numpy` are imported, and the convection-diffusion component is declared as a subclass of `om.ImplicitComponent`.

```
1 import numpy as np
2 import openmdao.api as om
3 class ConvectionDiffusion_Component(om.ImplicitComponent):
```

At initialization, an instance of a convection-diffusion solver and a NAVIER-STOKES solver is required as parameters. When *OpenMDAO* is set up, the solvers get stored and both the inputs and outputs are declared to *OpenMDAO* as coefficient vectors of appropriate size.

```
4  def initialize(self):
5   # declare parameters
6   self.options.declare('solver_CD', desc='convection-diffusion solver object')
7   self.options.declare('solver_NS', desc='NAVIER-STOKES solver object')
8
9  def setup(self):
10  self.cd = self.options['solver_CD']
11  self.ns = self.options['solver_NS']
12  # declare variables
13  self.add_output('T_cd', val=np.zeros(self.cd.N), desc='T as CD global vector')
14  self.add_input('u_ns', val=np.zeros(self.ns.N), desc='u as NS global vector')
15  self.add_input('v_ns', val=np.zeros(self.ns.N), desc='v as NS global vector')
```

The required function for change of basis for the input variables is implemented with the subroutines concerning the discretization. With `get_eval` from the NAVIER-STOKES solver, callables `u_call` and `v_call` are defined which can be evaluated at points in space. They are then passed to `get_vector` from the NAVIER-STOKES solver, requiring a callable which can be evaluated at certain (usually unknown to the user) points in space, e.g. at grid-points.

[1] https://github.com/SEhrm/sim-comp/blob/main/Solvers/NavierStokes_Solver.py

```
16 def change_inputs(self, u_ns, v_ns):
17  u_call = lambda x,y: self.ns._get_eval(vec=inputs['u_ns'], point=(x,y))
18  v_call = lambda x,y: self.ns._get_eval(vec=inputs['v_ns'], point=(x,y))
19  u_cd = self.cd._get_vector(u_call)
20  v_cd = self.cd._get_vector(v_call)
21  return u_cd, v_cd
```

The required routines for solving are again easily implemented with the auxiliary subroutines concerning the solution process. Note that, in `apply_linear`, the change of basis for the differentials is too done by `change_inputs`; this is possible because interpolation and discretization are, in the very most cases, linear maps.

```
22 def apply_nonlinear(self, inputs, outputs, residuals, *args):
23  T = outputs['T_cd']
24  u, v = self.change_inputs(inputs['u_ns'], inputs['v_ns'])
25  residuals['T_cd'] = self.cd._get_residuals(T, u, v)
26
27 def linearize(self, inputs, outputs, *args): # pre-calculations for apply_linear
28  T = outputs['T_cd']
29  self.cd._calc_jacobians(T)
30
31 def apply_linear(self, d_inputs, d_outputs, d_residuals, *args):
32  dT = d_outputs['T_cd'] if 'T_cd' in d_outputs else np.zeros(self.cd.N) # catch
     ↪ None
33  du, dv = self.change_inputs(d_inputs['u_ns'], d_inputs['v_ns'])
34  d_residuals['T_cd'] = self.cd._get_dresiduals(dT, du, dv)
35
36 def solve_linear(self, d_outputs, d_residuals):
37  dr = d_residuals['T_cd']
38  d_outputs['T_cd'] = self.cd._get_update(dr, dT0=d_outputs['T_cd'])
39
40 def solve_nonlinear(self, inputs, outputs):
41  u, v = self.change_inputs(inputs['u_ns'], inputs['v_ns'])
42  outputs['T_cd'] = self.cd._get_solution(u, v, T0=outputs['T_cd'])
```

The implementation of the convection-diffusion component is given in `ConvectionDiffusion_Component.py`[1]. The NAVIER-STOKES component is implemented in the same fashion and is given in `NavierStokes_Component.py`[2].

[1] https://github.com/SEhrm/sim-comp/blob/main/OpenMDAO/ConvectionDiffusion_Component.py

[2] https://github.com/SEhrm/sim-comp/blob/main/OpenMDAO/NavierStokes_Component.py

4 Performance Analysis

4.1 Setup

Single-Disciplinary Solvers Setup

To solve the benchmark problem of Sec. 3, the convection-diffusion solver from Sec. 3.2 and the NAVIER-STOKES solver from Sec. 3.3 are set up for a square domain (so $L_x = L_y$). The REYNOLDS number is set to $1 \cdot 10^3$, so its order of magnitude is roughly the same as of the encountered RAYLEIGH numbers. Even though, from a physical perspective, the REYNOLDS number can be chosen arbitrarily, different values give slightly different stability and convergence behaviors from the single-disciplinary solvers. In both single-disciplinary solvers, as described in App. B, the solutions are approximated by piecewise polynomials of order $P := 4$, where the domain is partitioned in square elements of equal size – in the latter with $N_{ex}\, N_{ey}$ elements, and in the former with only $N_{ex}/2\, N_{ey}/2$ elements. The internal linear solvers (i.e. `get_update`), as well as the internal NEWTON solvers (i.e. `get_solution`), will iterate until the absolute root mean square residuals are below $1 \cdot 10^{-13}$. Tighter tolerances are possible at the cost of higher memory usage by LGMRES.

OpenMDAO Setup

For coupling, the respective components of the single-disciplinary solvers from Sec. 3.4 and a top-level solver script, as described in Sec. 2.3, are implemented. The convection-diffusion solver/component, being the non-homogeneous

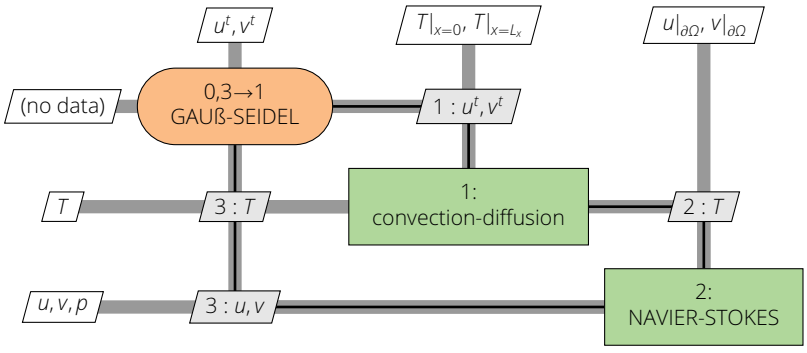

Figure 4.1: Extended design structure matrix [16] for the nonlinear block-GAUß-SEIDEL method.

one, is set to run first in block-GAUß-SEIDEL iterations. Figure 4.1 shows the extended design structure matrix, as established by [16], of the coupling by the nonlinear block-GAUß-SEIDEL method. The ARMIJO-GOLDSTEIN parameters are set to `maxiter` := 8, `rho` := 0.8, `c` := 0.2. The maximum contraction power(`rho`, `maxiter`) should not be chosen too small, as to avoid stalling, and `rho` should not be chosen too small either, as to retain precision. The top-level block-JACOBI preconditioner makes only one iteration, so $m := 1$. The tolerance for the top-level GMRES solver is set to $mtol_{\text{lin}} := 1 \cdot 10^{-13}$, and for the top-level NEWTON/GAUß-SEIDEL solver is set to $mtol := 1 \cdot 10^{-10}$. Note that, by setting the tolerance for the absolute root mean square residual, the convergence results are comparable across all methods and all amounts of degrees of freedom. In order for block-JACOBI preconditioned NEWTON-KRYLOV to converge across all coupling strengths, it appears that *mtol* must be set a few orders of magnitude apart from $mtol_{\text{lin}}$. The top-level script is given in `Boussinesq_ParallelCoupler.py`[1].

4.2 Verification

Reference

To verify the coupling algorithms from Sec. 2.2 for the benchmark problem of Sec. 3, the solutions for various RAYLEIGH numbers and grids, and the solution from a reference, are compared. As reference, [8] is chosen: these solutions, a the stream-function-vorticity formulation solved by the finite difference method, are commonly used and are widely believed to be of highest available accuracy.

Equivalence

When running the different coupling algorithms with the same grid and the same tolerance on the nonlinear residual, they are all expected to converge to the same (if not the only) solution. Table 4.1 shows the maximum horizontal velocity over the vertical mid-plane $\check{u}_{\max} := \max \check{u}|_{x=L_x/2}$ and the maximum vertical

Table 4.1: Maximum horizontal/vertical velocities over the vertical/horizontal mid-plane and their locations from NEWTON-block-JACOBI (NJ), block-JACOBI preconditioned NEWTON-KRYLOV (JNK) and nonlinear block-GAUß-SEIDEL (GS) for various RAYLEIGH numbers. $N_{ex} := N_{ey} := 8$.

method	Ra	$\frac{\check{u}_{\max}}{\check{\alpha}/L_{\text{ref}}}$	at y	$\frac{\check{v}_{\max}}{\check{\alpha}/L_{\text{ref}}}$	at y
JNK	$1 \cdot 10^3$	3.64558	0.8133	3.69527	0.1784
NJ	$1 \cdot 10^3$	3.64558	0.8133	3.69527	0.1784
GS	$1 \cdot 10^3$	3.64558	0.8133	3.69527	0.1784
JNK	$1 \cdot 10^4$	16.17167	0.8238	19.61868	0.1190
NJ	$1 \cdot 10^4$	16.17167	0.8238	19.61868	0.1190
GS	$1 \cdot 10^4$	16.17167	0.8238	19.61868	0.1190
JNK	$1 \cdot 10^5$	35.18334	0.8559	69.27017	0.0662

[1] `https://github.com/SEhrm/sim-comp/blob/main/OpenMDAO/Boussinesq_ParallelCoupler.py`

velocity over the horizontal mid-plane $\check{v}_{max} := \max \check{v}|_{x=L_y/2}$, as well as their locations for various coupling methods and RAYLEIGH numbers, but for fixed grid. As expected, the differences of the solutions are neglectable – the solutions are *independent* of the chosen method.

Accuracy

For the grids, it is expected that for finer grid, i. e. for rising number of degrees of freedom, the solution gets increasingly accurate (grid convergence). Usually, this is achieved by h-refinement, i. e. by setting a higher number of elements, rather than by p-refinement, i. e. by setting a higher polynomial order. Figure 4.2 shows the maximum horizontal velocity over the vertical mid-plane and its location against the number of (equal width) elements $N_{ex} = N_{ey}$. Note that, for all amounts of degrees of freedom, the solver iterates until the absolute nonlinear root mean square residual, rather than the norm of the (increasingly larger) residual vector, is below the same tolerance *mtol*. As expected, the values converge for rising number of elements, with their limits surprisingly being almost identical to the reference solution. The latter had not been expected from the non-conservative nature of the continuous spectral element method, respectively from the formulation of the problem in convective form with primitive variables.

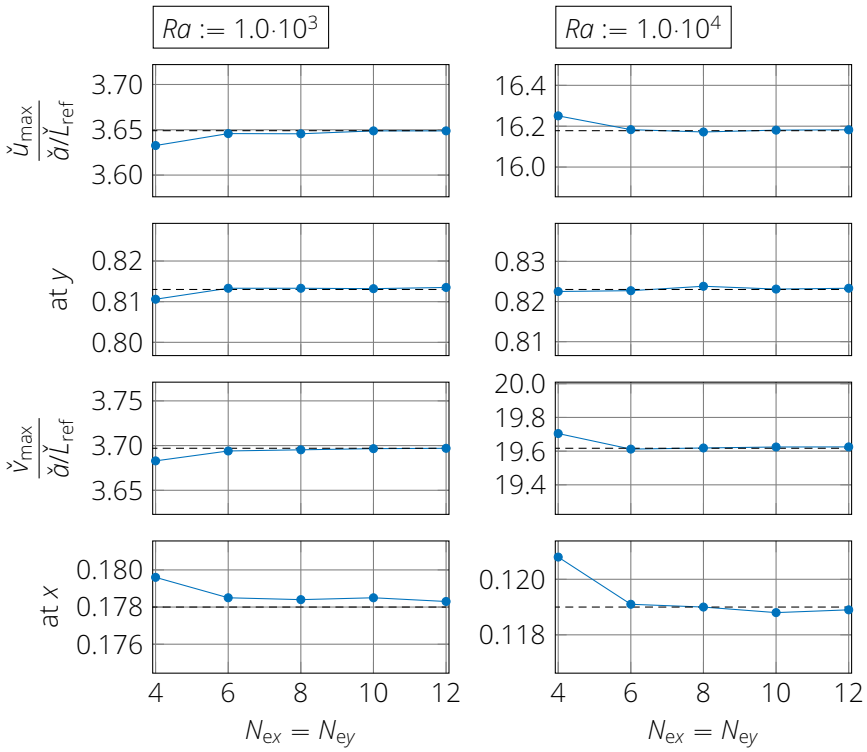

Figure 4.2: Maximum horizontal velocity over the vertical mid-plane and its locations against number of elements $N_{ex} = N_{ey}$ for various RAYLEIGH numbers. The dashed lines show the values from the reference solution [8, Tab. V]. The ordinates range from –2 % to 2 %.

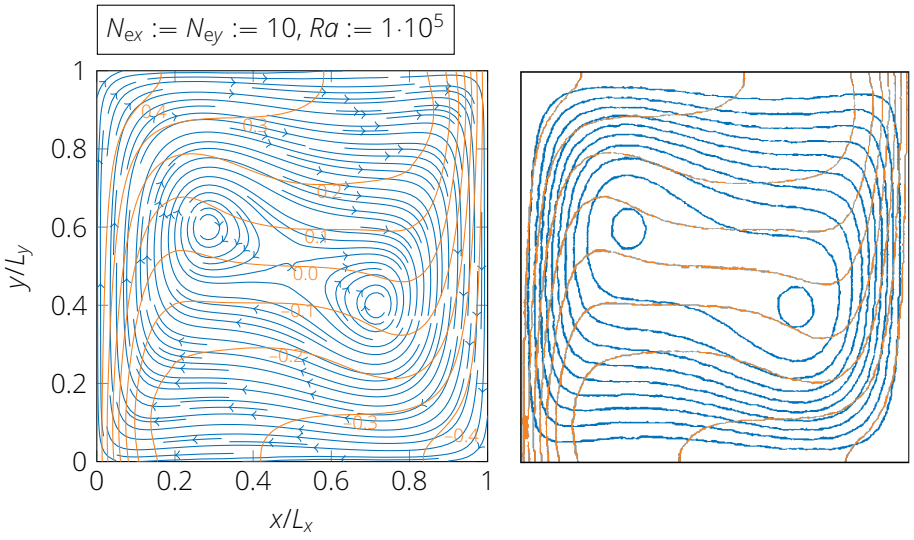

Figure 4.3: Streamlines and isotherms in the cavity for $Ra := 1 \cdot 10^5$ from the solver (left) and from the reference [see 8, Fig. 3(c), Fig. 4(c)] (right).

Qualitatively

In Fig. 4.3 the streamlines and the isotherms in the cavity for $Ra := 1 \cdot 10^5$ are shown. As expected, the solution from the solver and the reference solution agree qualitatively. The two vortexes and the horizontal segments of the isotherms were produced, and both the streamlines and the isotherms match the prescribed boundary conditions.

4.3 Iterative Convergence Behavior

Coupling Terms

From (3.1) it follows heuristically that increasing RAYLEIGH numbers make the problem increasingly stronger coupled. For a more precise estimate on the coupling terms, assume that for RAYLEIGH numbers changing in orders of magnitude, the solutions (T^*, u^*, v^*, p^*) do not change in orders of magnitude. Then, by (3.5) and (3.11), this yields, with respect to the RAYLEIGH number Ra,

$$\left.\frac{\partial r_T}{\partial (u, v, p)}\right|_{T^*, u^*, v^*, p^*} = \text{const.}, \quad \left.\frac{\partial (r_u, r_v, r_{\text{cont}})}{\partial T}\right|_{T^*, u^*, v^*, p^*} \propto Ra, \tag{4.1a}$$

for the 'off-diagonal'/coupling terms, and

$$\left.\frac{\partial r_T}{\partial T}\right|_{T^*, u^*, v^*, p^*} = \text{const.}, \quad \left.\frac{\partial (r_u, r_v, r_{\text{cont}})}{\partial (u, v, p)}\right|_{T^*, u^*, v^*, p^*} = \text{const.}, \tag{4.1b}$$

for the 'diagonal' terms. Hence, to test the iterative convergence behavior of the coupling algorithms from Sec. 2.2, the benchmark problem of Sec. 3 is solved

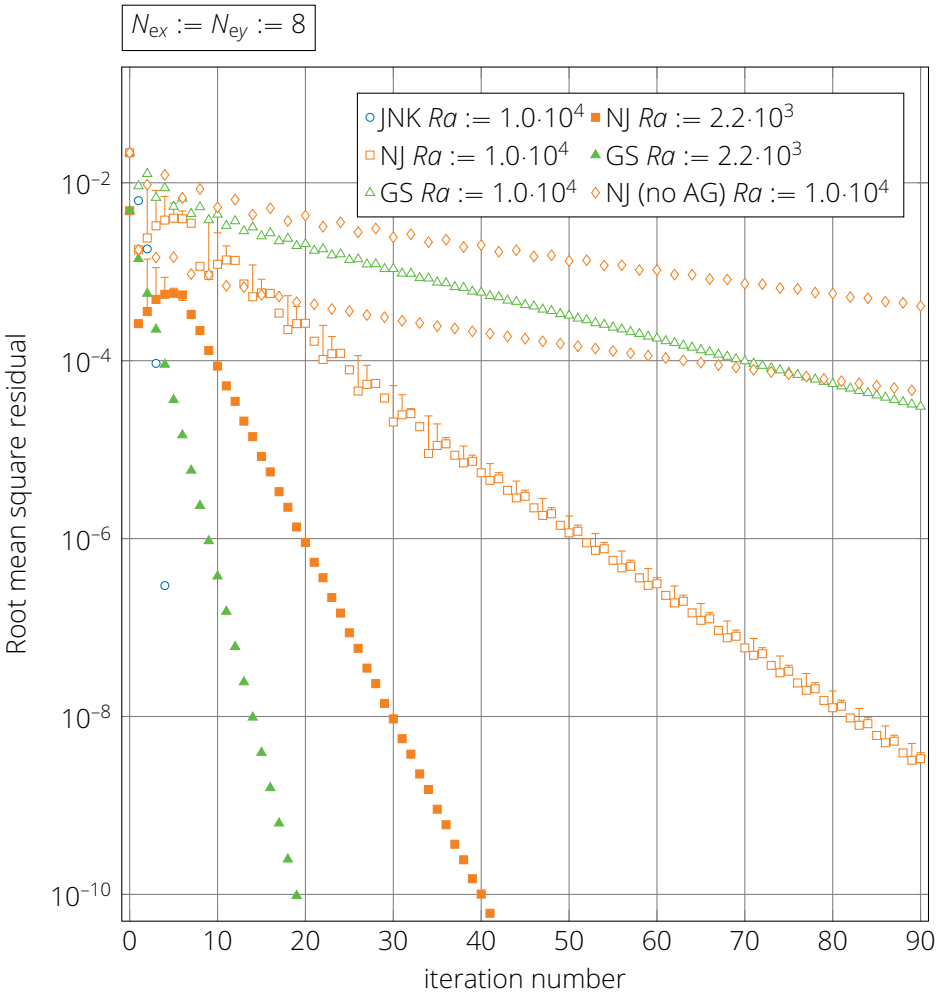

Figure 4.4: Root mean square residual against nonlinear iteration number from block-JACOBI preconditioned NEWTON-KRYLOV (JNK), NEWTON-block-JACOBI (NJ) and nonlinear block-GAUß-SEIDEL (GS) for various REYNOLDS numbers. The whiskers extend to the residual before ARMIJO-GOLDSTEIN backtracking.

with increasingly larger RAYLEIGH numbers and various amounts of degrees of freedom.

Order

Figure 4.4 shows the absolute nonlinear root mean square residual of each nonlinear iteration number from NEWTON-block-JACOBI (NJ) and nonlinear block-GAUß-SEIDEL (GS) for various RAYLEIGH numbers. As expected, the asymptotic order of convergence is linear for both methods. Figure 4.5 shows the absolute nonlinear root mean square residual of the current NEWTON iteration against the residual of the previous one from block-JACOBI preconditioned NEWTON-KRYLOV (JNK) for various RAYLEIGH numbers. As expected, the asymptotic order of convergence is quadratic, i. e. the residual of the current nonlinear iteration is proportional to the square residual of the previous one. For high RAYLEIGH

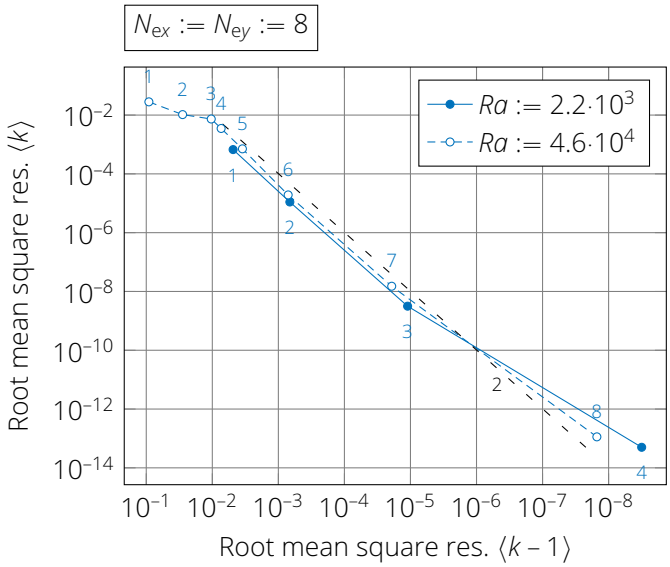

Figure 4.5: Root mean square residual of current nonlinear iteration against previous from block-JACOBI preconditioned NEWTON-KRYLOV for various RAYLEIGH numbers.

numbers, the initial guess is too far away from the solution and the order of convergence reduces to sub-quadratic, until the guess gets close enough to the solution. Figure 4.7 shows the absolute root mean square residual of the linear system against each GMRES iteration number for all NEWTON iterations from JNK for various RAYLEIGH numbers. As expected from [see 20, Th. 5], the GMRES method in JNK converges linear.

Rate

From (2.8) and (2.4), from (4.1) and from the homogeneity of the spectral radius, it follows immediately that the asymptotic rate of convergence for NJ is proportional to $\sqrt{Ra}$, and for GS is proportional to Ra. Figure 4.6 shows the mean rate of linear convergence of these methods for various RAYLEIGH numbers. As expected, for low to moderate RAYLEIGH numbers, the mean rate of convergence is, for NJ, about proportional to $\sqrt{Ra}$, and for GS, about proportional to Ra – their orders of magnitude differ by about a factor of two. For high RAYLEIGH numbers, ARMIJO-GOLDSTEIN backtracking successfully improves the overall rate of convergence (see also Fig. 4.4) of NJ, such that its rate of convergence is better than of GS. From Figure 4.7, it appears that the rate of convergence of the GMRES method in JNK worsens with rising RAYLEIGH numbers.

Computational Work

Figure 4.6 shows both the total amount of nonlinear iterations and the total amount of calls on `get_update` for various coupling methods and RAYLEIGH numbers, but for fixed grid. As expected, for low RAYLEIGH numbers, the amount of nonlinear iterations from GS is about half as much as from NJ. As predicted by Sec. 2.2, for increasing RAYLEIGH numbers, with respect to the total amount of calls on `get_update`, first GS performs best, then NJ, and then

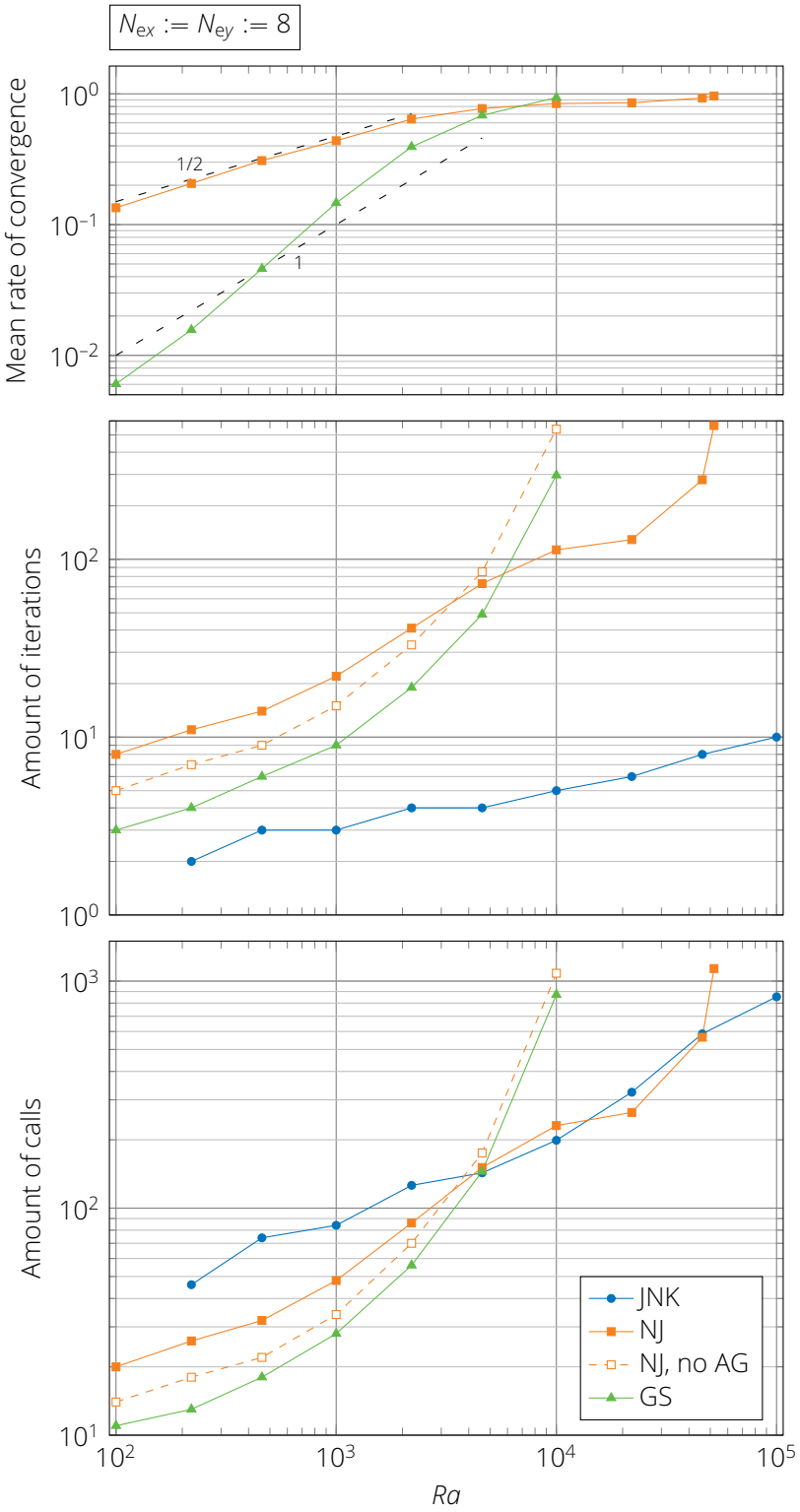

Figure 4.6: Mean rate of linear convergence (top), amount of nonlinear iterations (middle) and total amount of calls on `get_update` (bottom) against RAYLEIGH number from NEWTON-block-JACOBI (NJ), block-JACOBI preconditioned NEWTON-KRYLOV (JNK) and nonlinear GAUß-SEIDEL (GS) for various REYNOLDS numbers.

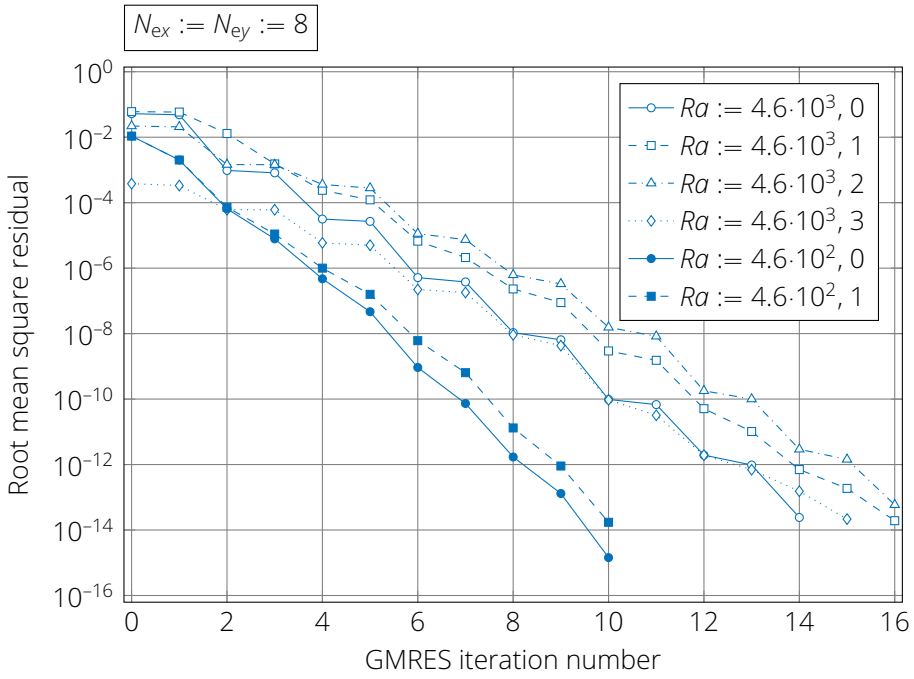

Figure 4.7: Root mean square residual against GMRES iteration number for NEWTON iteration from block-JACOBI preconditioned NEWTON-KRYLOV.

JNK. For the latter, the amount of calls grows moderately, while for the former two, it grows, as expected, dramatically when approaching the limit of eligible RAYLEIGH numbers. While GS has its limit for eligible RAYLEIGH numbers just above $1{\cdot}10^4$, ARMIJO-GOLDSTEIN backtracking successfully pushed the limit for NJ to just above $5.2{\cdot}10^5$.

Scalability

As a rule, for increasingly finer grids, single-disciplinary solvers require increasingly more work to solve the problem, not only because the amount of degrees of freedom is larger, but also because the linear systems they solve get increasingly worse conditioned. By (2.8) and (2.4), when the systems of the single-disciplinary solvers, i. e. the 'diagonal' terms, get increasingly worse conditioned, the coupling methods may also need more nonlinear iterations. As a rough estimate for the increase in the condition numbers, a lower bound for $\kappa\left(\partial \boldsymbol{r}_T/\partial \boldsymbol{T}|_{\dot{Pe}:=0}\right)$ from (3.5) and (3.6b) is evaluated. By the inequalities of matrix norms and the spectral radius, this lower bound is given by the quotient of the absolute largest eigenvalue and the absolute smallest eigenvalue. Again, grid refinement will be made by increasing the number of (equal width) elements $N_{ex} = N_{ey}$. Figure 4.8 shows said lower bound, the amount of nonlinear iterations, the total amount of calls on `get_update` and the amount of calls per nonlinear iteration against increasingly more elements for various coupling methods and grids. As expected from [see

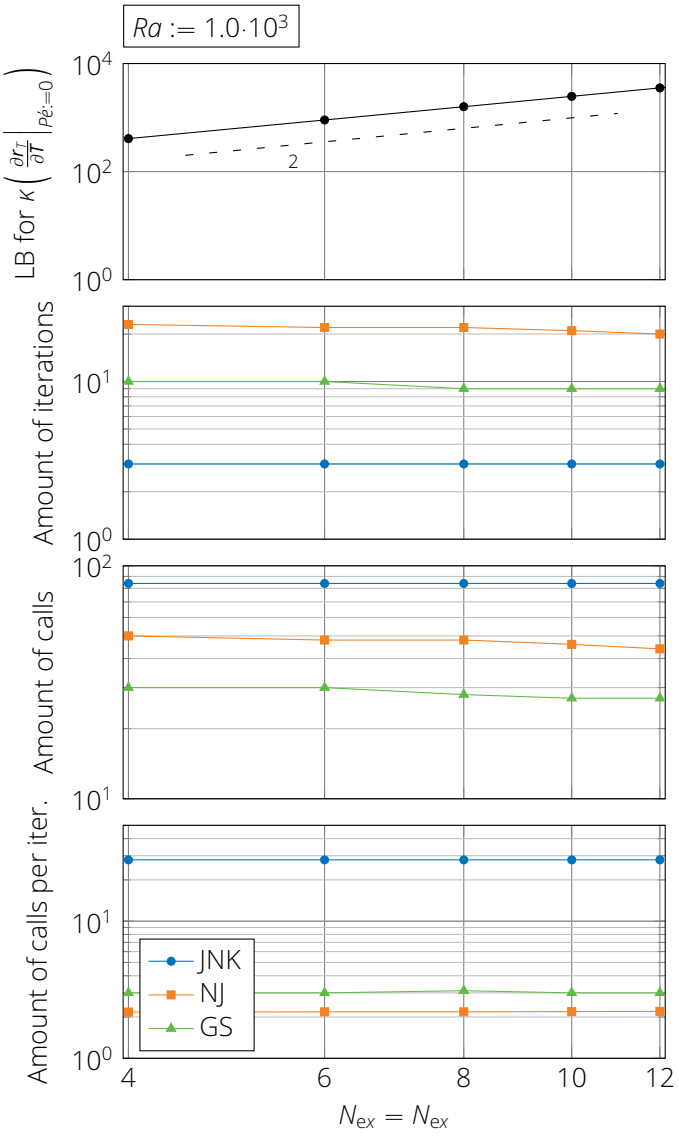

Figure 4.8: Lower bound on condition number, amount of nonlinear iterations, total amount of calls on `get_update` and amount of calls per nonlinear iteration (from top to bottom) against number of elements from NEWTON-block-JACOBI (NJ), block-JACOBI preconditioned NEWTON-KRYLOV (JNK) and nonlinear GAUß-SEIDEL (GS) for various REYNOLDS numbers.

10, pp. 78-79][1], the quotient of eigenvalues increases quadratic with more elements. Unexpectedly, even with the condition number rising in orders of magnitude, the computational work hardly changes and even decreases with more elements. Note that, the actual amount of float point operations increases nonetheless, simply because the system is larger. The inequalities in (2.8) and (2.4) were most likely too rough.

[1] Note that, the spectral properties from the matrices of the one-dimensional problem are given, but in this two-dimensional special case here, they too are applicable.

5 Conclusions

5.1 Results

Coupling Capabilities

OpenMDAO allows the user to implement a wide range of coupling methods: both nonlinear block-JACOBI (parallel staggered) and nonlinear block-GAUß-SEIDEL (sequential staggered), as well as NEWTON's method; with the latter, both for solving and preconditioning of the linearized system, linear block-JACOBI, linear block-GAUß-SEIDEL and KRYLOV solvers can be employed. The methods considered in Sec. 2.2 and applied on the benchmark problem from Sec. 3.1 perform as expected. The solutions fit the reference, as shown in Sec. 4.2 – so there is no considerable source of error from *OpenMDAO*. The observed convergence behaviors mostly agree with the theoretical expectations, as shown in Sec. 4.3 – so there is no considerable drawback on convergence when using *OpenMDAO*.

Implementation

OpenMDAO's strength clearly lies in its modular structure, allowing the user to implement various coupling methods by combining its modules: as shown in Sec. 2.3, just by changing a few lines, fundamentally different coupling methods were implemented. If the single-disciplinary solvers run in a way as described in Sec. 2.1, the *OpenMDAO*-components, i.e. the connections between *OpenMDAO* and the single-disciplinary solvers, are too easily implemented, as seen in Sec. 3.4, for example. Of the auxiliary subroutines from Sec. 2.1, `get_dresidual` is the least likely to be accessible, but it can be approximated by finite-differences (complex step) at the cost of higher overhead. If the single-disciplinary solvers are not written in python but say in *Fortran*, it should still be possible to implement the components by interface generators such as *NumPy*'s *F2PY*. Computational work from the components is successfully split among different MPI-processes allowing parallel execution. Unfortunately, change of basis requires the components to be able to communicate with more than one single-disciplinary solver. Therefore, the components must be implemented with not just one single-disciplinary solver in mind, but with all involved solvers in mind, reducing exchangeability and modularity. Hence, *OpenMDAO* should be used only if the single-disciplinary solvers are fixed. Further, *OpenMDAO* should not be used if runtime or memory usage is an issue: even though much of the intensive computing work of *OpenMDAO* is outsourced to *NumPy*/*PETSc*, a considerable amount of work must

be done in 'pure' *Python*, which is incredibly slow compared to compiled high-performance languages, such as *Fortran* or *C*. Summarizing, in scenarios where the single-disciplinary solvers are decided and only the coupling method is to be determined, usage of *OpenMDAO* allows to quickly test a wide range of methods in a user-accessible language. If a suitable coupling method is then found, its implementation should be done in a compiled language for performance optimization.

5.2 Future Work

Benchmarking

The benchmark problem in Sec. 3 was deliberately chosen simple to keep the focus on the implementation of the coupling methods in *OpenMDAO*. Consequently, there arise immediate possibilities for extension of the benchmark problem which could be realized.

- *Number of Components*: To test the behavior for increasingly more components, a purely mathematically motivated nonlinear system of many coupled viscous BURGERS' equations could be solved. With M unknown functions $u_1, \dots, u_M$ and known functions $f_1, \dots, f_M$, but one spatial dimension it could read

$$\sum_{j=1}^{M} Re_{ij} u_j \frac{\partial u_i}{\partial x} \overset{!}{=} \frac{\partial^2 u_i}{\partial x^2} + f_i \; \forall \, x \in \Omega \quad \forall \, i = 1, \dots, M.$$

- *Grid*: The problem could be extended such that the domains of the single-disciplinary solvers no longer overlap and only share boundaries; e.g. the infinitesimal thin walls from Sec. 3 could be replaced by solid walls of finite thickness, in which the temperature is transferred by thermal conduction (conjugate heat transfer). As a more challenging extension, problems with changing domains could be realized, e.g. a wing twisting under aerodynamic load (fluid-structure interaction).
- *Non-stationarity*: The single-disciplinary solvers form Sec. 3.2 and Sec. 3.3 could be extended to solve for time-dependent outputs for given time-dependent inputs – i.e. the convection-diffusion solver would solve for the temperature in a future time-step based on the temperature of past time-steps and the time-dependent velocities. *OpenMDAO* could then be employed to solve the coupled system for future time-steps by carefully distributing the results from past time-steps.

Implementation

In this feasibility study, the single-disciplinary solvers were implemented in a way that was *very* favorable. Consequently, for the implementation of the components in Sec. 3.4 and the solvers in Sec. 3.2 and 3.3, some assumptions could be

dropped. Additionally, there are still some more parameters for the implementation in Sec. 2.3 of the coupling, which could be examined.

- *Commercial Solvers*: Especially commercial solvers usually do not allow direct communication, but only allow communication via text files and console commands. Such communication could be handled by instances of subclasses of the `ExternalCodeImplicitComp` class, which can handle writing files, running console commands, and reading files again.
- *Finite-Differences*: The single-disciplinary solvers do usually not provide the directional derivatives of their residuals with respect to their inputs, but only to their outputs. Consequently, in the components, the former must be calculated by finite-differences of the residuals. For the convection-diffusion component in Sec. 3.4, this could be implemented by

```
27 def linearize(self, inputs, outputs, *args): # pre-calculations for
    ↪ apply_linear
28  self.T_fd = outputs['T_cd'] # reference point for fd
29  self.u_fd, self.v_fd = self.change_inputs(inputs['u_ns'], inputs['v_ns'])
30  self.cd._calc_jacobians(self.T_fd)
31  self.r_fd = self.cd._get_residuals(self.T_fd, self.u_fd, self.v_fd) #
    ↪ reference value for fd
32
33 def apply_linear(self, d_inputs, d_outputs, d_residuals, *args):
34  dT = d_outputs['T_cd'] if 'T_cd' in d_outputs else np.zeros(self.cd.N) #
    ↪ catch None
35  du, dv = self.change_inputs(d_inputs['u_ns'], d_inputs['v_ns'])
36  d_residuals['T_cd'] = self.cd._get_dresiduals(dT) # only wrt to outputs
37  u_fd = self.u_fd + self.h*du # forward fd
38  v_fd = self.v_fd + self.h*dv
39  d_residuals['T_cd'] += (self.cd._get_residuals(T, u_fd, v_fd) -
    ↪ self.r_fd)/self.h # add total differential
```

  This can naturally be extended to the derivatives with respect to the outputs, if necessary.

- *Step Count*: As shown in Sec. 2.2, when using NEWTON-block-JACOBI or NEWTON-block-GAUß-SEIDEL, the total computational work for coupling is asymptotically independent of the amount of linear solver steps. Yet, an effect on stability is possible, which could be tested.
- *Relaxation*: *OpenMDAO* also has possibilities to allow for a relaxation factor in both linear and nonlinear block-GAUß-SEIDEL. If $\tilde{x}$ and $\tilde{y}$ are the solutions of the standard block-GAUß-SEIDEL method, then the variables are updated by

$$x \leftarrow x(1-\theta) + \tilde{x}\theta, \quad y \leftarrow y(1-\theta) + \tilde{y}\theta,$$

  where $\theta \in \mathbb{R}$ is the relaxation factor.[1] Note that, this scheme is not to be

[1] If $\boldsymbol{H}$ is the iteration matrix of the standard block-GAUß-SEIDEL method, the iteration matrix of the given scheme is $(1-\theta)\boldsymbol{I} + \theta\boldsymbol{H}$.

confused with the classical block-GAUß-SEIDEL relaxation, where relaxation is applied in every subiteration. The relaxation factor θ can either be set manually or can dynamically be chosen by AITKEN acceleration as in [15].

- *Parallel Speedup*: When testing the parallelization capabilities of *OpenMDAO*, coupling methods that support parallelization were preferred. Whether an actual speedup in wall-clock time is observed, and how it is dependent on the runtimes of the single-disciplinary solvers, could be tested. Of special interest should be NEWTON-block-GAUß-SEIDEL: while it can not run in parallel, by [see 19, Th. 10.3.1] it asymptotically requires only half as much nonlinear iterations as NEWTON-block-JACOBI.

Bibliography

1. ABRAMOWITZ, M.; STEGUN, I. A. (eds.). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. 10th ed. US Government Printing Office, 1972. Available also from: `https://www.cs.bham.ac.uk/~aps/research/projects/as/project.php`.
2. ARMIJO, L. Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of Mathematics*. 1966, vol. 16, no. 1. Available also from DOI: `10.2140/pjm.1966.16.1`.
3. BAKER, A. H.; JESSUP, E. R.; MANTEUFFEL, T. A Technique for Accelerating the Convergence of Restarted GMRES. *SIAM Journal on Matrix Analysis and Applications*. 2005, vol. 26, no. 4, pp. 962–984. Available also from DOI: `10.1137/S0895479803422014`.
4. BALAY, S.; ABHYANKAR, S.; ADAMS, M. F., et al. *PETSc Web page* [online]. 2021-08-14. Available also from: `https://petsc.org/`.
5. BOGAERS, A.; KOK, S.; REDDY, B., et al. Quasi-Newton methods for implicit black-box FSI coupling. *Computer Methods in Applied Mechanics and Engineering*. 2014, vol. 279, pp. 113–132. Available also from DOI: `10.1016/j.cma.2014.06.033`.
6. DALCIN, L.; FANG, Y.-L. L. mpi4py: Status Update After 12 Years of Development. *Computing in Science & Engineering*. 2021, vol. 23, no. 4, pp. 47–54. Available also from DOI: `10.1109/MCSE.2021.3083216`.
7. DALCIN, L. D.; PAZ, R. R.; KLER, P. A., et al. Parallel distributed computing using Python. *Advances in Water Resources*. 2011, vol. 34, no. 9, pp. 1124–1139. Available also from DOI: `10.1016/j.advwatres.2011.04.013`.
8. DE VAHL DAVIS, G. Natural convection of air in a square cavity: A bench mark numerical solution. *International Journal for Numerical Methods in Fluids*. 1983, vol. 3, no. 3, pp. 249–264. Available also from DOI: `10.1002/fld.1650030305`.
9. DEMBO, R. S.; EISENSTAT, S. C.; STEIHAUG, T. Inexact Newton Methods. *SIAM Journal on Numerical Analysis*. 1982, vol. 19, no. 2, pp. 400–408. Available also from DOI: `10.1137/0719025`.

10. DEVILLE, M. O.; FISCHER, P. F. *High-Order Methods for Incompressible Fluid Flow*. Cambridge University Press, 2002. Available also from DOI: 10.1017/CBO9780511546792.

11. FERZIGER, J. H. *Computational Methods for Fluid Dynamics*. Springer, 2020. Available also from DOI: 10.1007/978-3-319-99693-6.

12. GIRALDO, F. X. *An Introduction to Element-Based Galerkin Methods on Tensor-Product Bases*. Springer, 2020. Available also from DOI: 10.1007/978-3-030-55069-1.

13. GRAY, J. S.; HWANG, J. T.; MARTINS, J. R. R. A., et al. OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization. *Structural and Multidisciplinary Optimization*. 2019, vol. 59, no. 4, pp. 1075–1104. Available also from DOI: 10.1007/s00158-019-02211-z.

14. KARNIADAKIS, G.; SHERWIN, S. *Spectral/hp element methods for computational fluid dynamics*. 2nd ed. Oxford University Press, 2005. Available also from DOI: 10.1093/acprof:oso/9780198528692.001.0001.

15. KENWAY, G. K. W.; KENNEDY, G. J.; MARTINS, J. R. R. A. Scalable Parallel Approach for High-Fidelity Steady-State Aeroelastic Analysis and Adjoint Derivative Computations. *AIAA Journal*. 2014, vol. 52, no. 5, pp. 935–951. Available also from DOI: 10.2514/1.J052255.

16. LAMBE, A. B.; MARTINS, J. R. R. A. Extensions to the Design Structure Matrix for the Description of Multidisciplinary Design, Analysis, and Optimization Processes. *Structural and Multidisciplinary Optimization*. 2012, vol. 46, pp. 273–284. Available also from DOI: 10.1007/s00158-012-0763-y.

17. *MPI for python 3.1.1 Documentation* [online]. 2021-08-14. Available also from: https://mpi4py.readthedocs.io/en/stable/index.html.

18. *OpenMDAO 3.9.2 documentation* [online]. 2021-05-14. Available also from: https://openmdao.org/twodocs/versions/3.9.2/index.html.

19. ORTEGA, J. M.; RHEINBOLDT, W. C. *Iterative Solution of Nonlinear Equations in Several Variables*. Society for Industrial and Applied Mathematics, 2000. Available also from DOI: 10.1137/1.9780898719468.

20. SAAD, Y.; SCHULTZ, M. H. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*. 1986, vol. 7, no. 3, pp. 856–869. Available also from DOI: 10.1137/0907058.

21. VIERENDEELS, J.; LANOYE, L.; DEGROOTE, J., et al. Implicit coupling of partitioned fluid structure interaction problems with reduced order models. *Computers & Structures*. 2007, vol. 85, no. 11, pp. 970–976. Available also from DOI: 10.1016/j.compstruc.2006.11.006.

A Quadrature and Interpolation

A.1 GAUß-LEGENDRE-LOBATTO Quadrature

GLL Quadrature Rule

Given a function $f(\xi)$ on $\Omega^S := [-1, 1]$, the approximate integral over Ω^S is of the form

$$\int_{-1}^{1} f \, \mathrm{d}\xi \approx \sum_{i=0}^{P} w_i f(\xi_i), \tag{A.1}$$

where $(\xi_i)_{i=0,\dots,P}$ are called the quadrature nodes and $(w_i)_{i=0,\dots,P}$ are called the quadrature weights. The GAUß-LEGENDRE-LOBATTO (GLL) quadrature rule [see 10, Eq. B.2.9] is chosen, maximizing the accuracy while also incorporating the boundaries $\xi = \pm 1$ of the interval. With $P_j(\xi)$ being the j-th LEGENDRE polynomial, the quadrature nodes are found by NEWTON's method where the relations of the LEGENDRE polynomials [see 1, entry 22.8.5 and entry 22.6.13] are used.

$$\xi_i \leftarrow \xi_i - \frac{\left.\frac{\mathrm{d}P_P}{\mathrm{d}\xi}\right|_{\xi_i}}{\left.\frac{\mathrm{d}^2 P_P}{\mathrm{d}\xi^2}\right|_{\xi_i}} = \xi_i - \frac{xP_P(\xi_i) - P_{P-1}(\xi_i)}{(P+1)P_P(\xi_i) - \frac{2\xi_i}{P}\underbrace{\left.\frac{\mathrm{d}P_P}{\mathrm{d}\xi}\right|_{\xi_i}}_{\to 0}} \quad \forall\, i = 0, \dots, P \tag{A.2}$$

with the GAUß-CHEBYSHEV-LOBATTO nodes [10, Eq. B.2.13] as the first guess. The entries of the VANDERMONDE matrix $P_j(\xi_i)$ are given by the recursive relation [1, entry 22.7.10]. With the nodes found, the weights are given by [see 10, Eq. B.2.9]. From [1, entry 25.4.32] the quadrature is exact if f is a polynomial of order less than $2P$.

Python Functions

The corresponding *Python* function `GLL.standard_nodes` returns the nodes, weights and the VANDERMONDE matrix as *NumPy* array.

A.2 LAGRANGE Interpolation

LAGRANGE Interpolation Rule

Given a function $f(\xi)$ on Ω^S, the approximate evaluation at $\xi \in \Omega^S$ is of the form

$$f(\xi) \approx \sum_{i=0}^{P} \ell_i(\xi) f(\xi_i), \tag{A.3}$$

where $(\xi_i)_{i=0,\dots,P}$ are called the interpolation nodes and $(\ell_i)_{i=0,\dots,P}$ are called the basis functions. The mentioned GLL quadrature nodes are chosen as interpolation nodes, and the corresponding LAGRANGE polynomials [1, entry 25.2.2] are chosen as basis functions. Although not maximizing the accuracy, choosing the same nodes for interpolation and quadrature greatly simplifies their application in the spectral element method. The LAGRANGE polynomials are evidently satisfying the interpolation condition

$$\ell_i(\xi_j) = \delta_{ij} := \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}. \tag{A.4}$$

Python Functions

The corresponding *Python* function `GLL.standard_evaluation_matrix` takes an array $[\xi_i^{\text{Plot}}]$ as *NumPy* array and returns the evaluation matrix $[\ell_j(\xi_i^{\text{Plot}})]_{i,j}$ as *NumPy* array.

A.3 Standard Matrices

Standard Matrices

The choice of quadrature and interpolation leads to some particularly important matrices, the *standard matrices*, which are needed in App. B on. We define the *standard mass matrix* $\boldsymbol{M}^S := [M_{ij}^S]_{i,j}$ as the approximate inner product of the i-th and j-th basis function ℓ_i and ℓ_j

$$M_{ij}^S := \int_{-1}^{1} \ell_i \ell_j \, \mathrm{d}\xi \approx \sum_{k=0}^{P} w_k \ell_i(\xi_k) \ell_j(\xi_k) = \sum_{k=0}^{P} w_k \delta_{ki} \delta_{kj} = w_i \delta_{ij} \quad \forall\, i,j = 0,\dots,P, \tag{A.5}$$

where the quadrature rule (A.1) and the interpolation condition (A.4) are used. We define the *standard differentiation matrix* $\boldsymbol{D}^S := [D_{ij}^S]_{i,j} := [\ell_j'(\xi_i)]_{i,j}$ as the derivative of the j-th basis function ℓ_j at the i-th node ξ_i [see 10, Eq. B.3.51]. With the differentiation matrix in mind, we additionally define the *standard gradient matrix* $\boldsymbol{G}^S := [G_{ij}^S]_{i,j}$, the *standard stiffness matrix* $\boldsymbol{K}^S = [K_{ij}^S]_{i,j}$, the *standard product matrix* $\boldsymbol{F}^S := [F_{ijk}^S]_{i,j,k}$ and the *standard convection matrix* $\boldsymbol{C}^S := [C_{ijk}^S]_{i,j,k}$ by

$$G_{ij}^{\mathrm{S}} := \int_{-1}^{1} \ell_i \ell_j' \,\mathrm{d}\xi = \sum_{r=0}^{P} w_r \ell_i(\xi_r)\ell_j'(\xi_r) = w_i D_{ij}^{\mathrm{S}} \qquad \forall\, i,j = 0,\ldots,P, \tag{A.6}$$

$$K_{ij}^{\mathrm{S}} := \int_{-1}^{1} \ell_i' \ell_j' \,\mathrm{d}\xi = \sum_{r=0}^{P} w_r \ell_i'(\xi_r)\ell_j'(\xi_r) = \sum_{r=0}^{P} w_r D_{ri}^{\mathrm{S}} D_{rj}^{\mathrm{S}} \qquad \forall\, i,j = 0,\ldots,P, \tag{A.7}$$

$$F_{ijk}^{\mathrm{S}} := \int_{-1}^{1} \ell_i \ell_j \ell_k \,\mathrm{d}\xi \approx \sum_{r=0}^{P} w_r \ell_i(\xi_r)\ell_j(\xi_r)\ell_k(\xi_r) = w_i \delta_{ij}\delta_{ik} \qquad \forall\, i,j,k = 0,\ldots,P, \tag{A.8}$$

$$C_{ijk}^{\mathrm{S}} := \int_{-1}^{1} \ell_i \ell_j \ell_k' \,\mathrm{d}\xi \approx \sum_{r=0}^{P} w_r \ell_i(\xi_r)\ell_j(\xi_r)\ell_k'(\xi_r) = w_i \delta_{ij} D_{ik} \qquad \forall\, i,j,k = 0,\ldots,P. \tag{A.9}$$

Note that the quadratures for the matrices and $\boldsymbol{G}^{\mathrm{S}}$ and $\boldsymbol{K}^{\mathrm{S}}$ are formally exact as the integrands are polynomials of order $2P-1$ and $2P-2$.

Python Functions

The corresponding *Python* functions `GLL.standard_mass_matrix`, `GLL.standard_differentiation_matrix`, `GLL.standard_gradient_matrix`, `GLL.standard_stiffness_matrix`, `GLL.standard_product_matrix` and `GLL.standard_convection_matrix` return the matrices as *NumPy* array.

B Discretization of Partial Differential Equations

B.1 Discretization of Space

Element Partitioning

The rectangular domain $\Omega := [0, L_x] \times [0, L_y]$ is *partitioned* by $N_{ex} \cdot N_{ey}$ rectangular elements $\Omega^{mn} := \Omega_x^m \times \Omega_y^n$ with equal width $\Delta x := L_x/N_{ex}$ and height $\Delta y := L_y/N_{ey}$, such that

$$\Omega =: \bigcup_{m=0}^{N_{ex}-1} \bigcup_{n=0}^{N_{ey}-1} \Omega^{mn}. \tag{B.1}$$

Figure B.1 shows the elements and their indexing in the rectangular domain. For App. B.2, it is necessary to define bijective transformations between Ω_x^m/Ω_y^n and Ω^s. We define them by linear functions

$$x^m : \Omega^s \leftrightarrow \Omega_x^m,\ \xi \mapsto x^m(\xi), \quad \xi^m(x) := \left(x^m\right)^{-1}(x) \tag{B.2a}$$

$$y^n : \Omega^s \leftrightarrow \Omega_y^n,\ \eta \mapsto y^n(\eta), \quad \eta^n(y) := \left(y^n\right)^{-1}(y). \tag{B.2b}$$

Note that their derivatives $\frac{\mathrm{d}x^m}{\mathrm{d}\xi} = \frac{\Delta x}{2}$ and $\frac{\mathrm{d}y^n}{\mathrm{d}\eta} = \frac{\Delta y}{2}$ are constant.

Function Discretization and Nodes

Having established the element partitioning, continuous functions are piecewisely approximated by their interpolation polynomial in every element:

$$f(x,y) \approx \sum_{k,l=0}^{P} \underbrace{f(\overbrace{x^m(\xi_k)}^{=:x_k^m}, \overbrace{y^m(\eta_l)}^{=:y_l^m})}_{=:f_{kl}^{mn}} \cdot \underbrace{\overbrace{\ell_k(\xi^m(x))}^{=:\varphi_k^m(x)} \cdot \overbrace{\ell_l(\eta^n(y))}^{=:\varphi_l^n(y)}}_{=:\varphi_{kl}^{mn}(x,y)} \quad \forall (x,y) \in \Omega^{mn}. \tag{B.3}$$

The function f is hereby discretized to its evaluations at the *element nodes* (x_k^m, y_l^n), the *element coefficients* $\mathfrak{f} := [f_{kl}^{mn}]$. Figure B.1 shows the element nodes and their indexing in an element. As the element nodes are containing duplicates, e.g. $x_P^0 = x_0^1$, all $(N_{ex}P + 1)(N_{ey}P + 1) =: N$ unique nodes define the *global nodes* vectors $(\boldsymbol{x}, \boldsymbol{y}) := ([x_p]_{p=0}^{N-1}, [y_p]_{p=0}^{N-1})$. The surjective but not injective function GlobalIndex$(\cdot,\cdot,\cdot,\cdot)$ maps an element node (m,n,i,j) to a global node.

$$\begin{aligned} \{0,\dots,N_{ex}-1\}\times\{0,\dots,N_{ey}-1\}\times\{0,\dots,P\}^2 &\xrightarrow{\text{GlobalIndex}} \{0,\dots,N-1\},\\ m,n,i,j &\mapsto \text{GlobalIndex}(m,n,i,j) := (nP+j)+(N_{ey}P+1)\cdot(mP+i) \end{aligned} \tag{B.4}$$

Figure B.1 shows the global nodes and their indexing in the domain. Note that, motivated by *NumPy*'s row-major order, and contrary to most literature, the global nodes are ordered first along the y-axis and then along the x-axis. The evaluations of f at the global nodes define the *global coefficients* vector $\boldsymbol{f} := [f_p]_{p=0}^{N-1} := [f(x_p, y_p)]_{p=0}^{N-1}$.

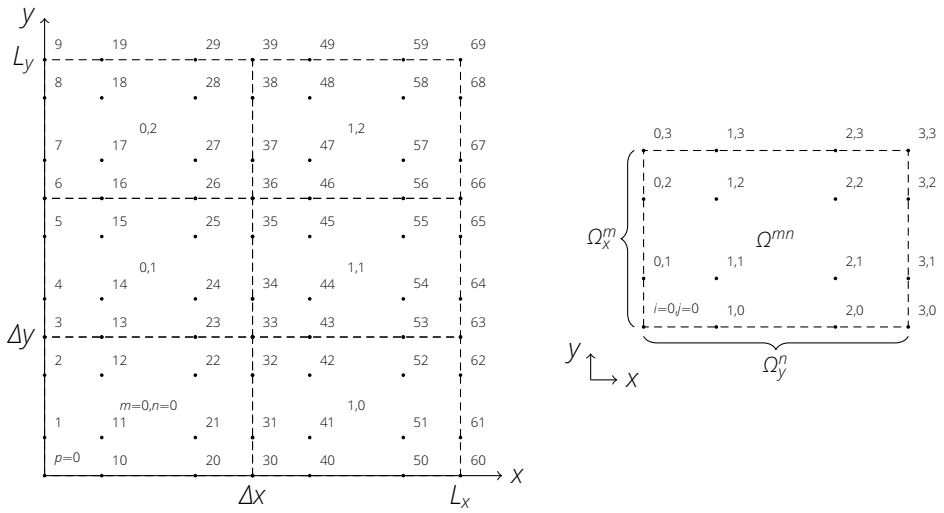

Figure B.1: Left: Element indices m, n, and global node index p of the spatial discretization for the domain $[0, L_x] \times [0, L_y]$. Right: Element node indices i, j inside an element $\Omega^{mn} = \Omega_x^m \times \Omega_y^n$. Each with $N_{ey} := 3, N_{ex} := 2, P := 3$.

Python Functions

The corresponding *Python* functions read

- `SEM.xi2x` takes m (resp. n) and ξ (resp. η), and returns $x^m(\xi)$ (resp. $y^m(\eta)$). `SEM.x2xi` takes x (resp. y), and returns m (resp. n) and ξ (resp. η),
- `SEM.element_nodes_1d` and `SEM.global_nodes_1d` return the element nodes as two-dimensional *NumPy* array and the global nodes as one-dimensional *NumPy* array in $[0, L_x]$ (resp. $[0, L_y]$),
- `SEM.element_nodes` and `SEM.global_nodes` return the element nodes as two four-dimensional *NumPy* arrays and the global nodes as two one-dimensional *NumPy* arrays in Ω,
- `SEM.global_index` takes the element node indices and returns the global index.

B.2 Discretization of Operators

Weak Form of PDE

Given an exemplary differential equation for the unknown continuously differentiable function $u(x,y)$ with continuous convection velocities $v_1(x,y)$, $v_2(x,y)$, source $f(x,y)$ and homogeneous NEUMANN boundary conditions

$$u + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} + v_1 \frac{\partial u}{\partial x} + v_2 \frac{\partial u}{\partial y} \overset{!}{=} \nabla^2 u + f \quad \forall\,(x,y) \in \Omega, \tag{B.5a}$$

$$\frac{\partial u}{\partial n} \overset{!}{=} 0 \qquad \forall\,(x,y) \in \partial\Omega. \tag{B.5b}$$

With use of GREEN's first identity on the LAPLACE operator, its weak form is given by

$$\int_\Omega u\omega \,\mathrm{d}\Omega + \int_\Omega \frac{\partial u}{\partial x}\omega \,\mathrm{d}\Omega + \int_\Omega \frac{\partial u}{\partial y}\omega \,\mathrm{d}\Omega + \int_\Omega v_1 \frac{\partial u}{\partial x}\omega \,\mathrm{d}\Omega + \int_\Omega v_2 \frac{\partial u}{\partial y}\omega \,\mathrm{d}\Omega + \int_\Omega \nabla u \circ \nabla\omega \,\mathrm{d}\Omega \overset{!}{=} \int_\Omega f\omega \,\mathrm{d}\Omega \quad \forall\,\omega, \tag{B.6}$$

where $\omega(x,y)$ is an arbitrary continuous test function on Ω such that the integrals exists and are finite. Applying the approximation to u, v_1, v_2 and f, as in (B.3), setting $\omega := \varphi_{ij}^{mn}$ (GALERKIN formulation), and considering each element Ω^{mn}, $m = 0,\dots,N_{ex}-1$, $n = 0,\dots,N_{ey}-1$ gives

$$\begin{aligned}
\sum_{k,l=0}^{P} u_{kl}^{mn} \Bigg(& \overbrace{\int_{\Omega^{mn}} \varphi_{kl}^{mn}\varphi_{ij}^{mn} \,\mathrm{d}\Omega}^{=:\,M_{ijkl}^{mn}} + \overbrace{\int_{\Omega^{mn}} \frac{\partial\varphi_{kl}^{mn}}{\partial x}\varphi_{ij}^{mn} \,\mathrm{d}\Omega}^{=:\,(G^x)_{ijkl}^{mn}} + \overbrace{\int_{\Omega^{mn}} \frac{\partial\varphi_{kl}^{mn}}{\partial y}\varphi_{ij}^{mn} \,\mathrm{d}\Omega}^{=:\,(G^y)_{ijkl}^{mn}} \\
& + \sum_{r,s=0}^{P} (v_1)_{rs}^{mn} \underbrace{\int_{\Omega^{mn}} \varphi_{rs}^{mn} \frac{\partial\varphi_{kl}^{mn}}{\partial x}\varphi_{ij}^{mn} \,\mathrm{d}\Omega}_{=:\,(C^x)_{ijrskl}^{mn}} + \sum_{r,s=0}^{P} (v_2)_{rs}^{mn} \underbrace{\int_{\Omega^{mn}} \varphi_{rs}^{mn} \frac{\partial\varphi_{kl}^{mn}}{\partial y}\varphi_{ij}^{mn} \,\mathrm{d}\Omega}_{=:\,(C^y)_{ijrskl}^{mn}} \\
& + \underbrace{\int_{\Omega^{mn}} \nabla\varphi_{kl}^{mn} \circ \nabla\varphi_{ij}^{mn} \,\mathrm{d}\Omega}_{=:\,K_{ijkl}^{mn}} \Bigg) \overset{!}{=} \sum_{k,l=0}^{P} f_{kl}^{mn} \underbrace{\int_{\Omega^{mn}} \varphi_{kl}^{mn}\varphi_{ij}^{mn} \,\mathrm{d}\Omega}_{=\,M_{ijkl}^{mn}} \quad \forall\, m,n,\ i,j = 0,\dots,P.
\end{aligned} \tag{B.7}$$

Element Arrays

We define the *element mass array* $\mathfrak{M} := [M_{ijkl}^{mn}]$ by

$$M_{ijkl}^{mn} = \underbrace{\int_{\Omega_x^m} \varphi_k^m \varphi_i^m \,\mathrm{d}x}_{=\,\frac{\Delta x}{2} M_{ik}^{s}} \cdot \underbrace{\int_{\Omega_y^n} \varphi_l^n \varphi_j^n \,\mathrm{d}y}_{=\,\frac{\Delta y}{2} M_{jl}^{s}}, \tag{B.8}$$

where we have applied integration by change of variables from $\mathrm{d}x$ to $\mathrm{d}\xi$, respectively form $\mathrm{d}y$ to $\mathrm{d}\eta$, with use of (B.2). We define the *element gradient arrays* $\mathfrak{G}^x := [(G^x)^{mn}_{ijkl}]$ and $\mathfrak{G}^y := [(G^y)^{mn}_{ijkl}]$ by

$$(G^x)^{mn}_{ijkl} = \underbrace{\int_{\Omega^m_x} \frac{\partial \varphi^m_k}{\partial x} \varphi^m_i \,\mathrm{d}x}_{= G^s_{ik}} \cdot \underbrace{\int_{\Omega^n_y} \varphi^n_l \varphi^n_j \,\mathrm{d}y}_{= \frac{\Delta y}{2} M^s_{jl}}, \quad (G^y)^{mn}_{ijkl} = \underbrace{\int_{\Omega^m_x} \varphi^m_k \varphi^m_i \,\mathrm{d}x}_{= \frac{\Delta x}{2} M^s_{ik}} \cdot \underbrace{\int_{\Omega^n_y} \frac{\partial \varphi^n_l}{\partial y} \varphi^n_j \,\mathrm{d}y}_{= G^s_{jl}}. \tag{B.9}$$

We define the *element convection arrays* $\mathfrak{C}^x := [(C^x)^{mn}_{ijrskl}]$ and $\mathfrak{C}^y := [(C^y)^{mn}_{ijrskl}]$ by

$$\begin{aligned}
(C^x)^{mn}_{ijrskl} &= \overbrace{\int_{\Omega^m_x} \varphi^m_r \frac{\partial \varphi^m_k}{\partial x} \varphi^m_i \,\mathrm{d}x}^{= C^s_{irk}} \overbrace{\int_{\Omega^n_y} \varphi^m_s \varphi^m_l \varphi^m_j \,\mathrm{d}y}^{= \frac{\Delta y}{2} F^s_{jsl}}, \\
(C^y)^{mn}_{ijrskl} &= \underbrace{\int_{\Omega^m_x} \varphi^m_r \varphi^m_k \varphi^m_i \,\mathrm{d}x}_{= \frac{\Delta x}{2} F^s_{irk}} \underbrace{\int_{\Omega^n_y} \varphi^m_s \frac{\partial \varphi^m_l}{\partial y} \varphi^m_j \,\mathrm{d}y}_{= C^s_{jsl}}.
\end{aligned} \tag{B.10}$$

We define the *element stiffness array* $\mathfrak{K} := [K^{mn}_{ijkl}]$ by

$$K^{mn}_{ijkl} = \underbrace{\int_{\Omega^m_x} \frac{\partial \varphi^m_k}{\partial x} \frac{\partial \varphi^m_i}{\partial x} \,\mathrm{d}x}_{= \frac{2}{\Delta x} K^s_{ik}} \underbrace{\int_{\Omega^n_y} \varphi^n_l \varphi^n_j \,\mathrm{d}y}_{= \frac{\Delta y}{2} M^s_{jl}} + \underbrace{\int_{\Omega^m_x} \varphi^m_k \varphi^m_i \,\mathrm{d}x}_{= \frac{\Delta x}{2} M^s_{ik}} \underbrace{\int_{\Omega^n_y} \frac{\partial \varphi^n_l}{\partial y} \frac{\partial \varphi^n_j}{\partial y} \,\mathrm{d}y}_{= \frac{2}{\Delta y} K^s_{jl}} \tag{B.11}$$

Global Assembly

As u is continuous across element boundaries, and because each element shares nodes at its boundaries, the element coefficients $\mathfrak{u}$ are containing duplicates. Equation (B.7) is therefore assembled to global form, such that it is instead solvable for the global coefficients vector $\boldsymbol{u}$.[1] Using the mapping function $\mathrm{GlobalIndex}(\cdot,\cdot,\cdot,\cdot)$ from (B.4), the global matrices are derived from the element matrices using the assembly operator $\mathrm{assemble}(\cdot)$, e.g.

$$\boldsymbol{M} := \mathrm{assemble}(\mathfrak{M}) \tag{B.12a}$$

$$\Leftrightarrow\ M_{pq} := \sum \left\{ M^{mn}_{ijkl} \,\middle|\, \begin{array}{l} \mathrm{GlobalIndex}(m,n,i,j) = p \\ \mathrm{GlobalIndex}(m,n,k,l) = q \end{array} \right\} \forall\, p,q = 0,\ldots,N-1. \tag{B.12b}$$

More detailed descriptions are given in [10, Sec. 4.5.1], [14, Sec. 2.3.1.4], [12, Sec. 12.9.1 and Alg. 12.13]. With the global coefficients vectors $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ and $\boldsymbol{f}$, the

[1] This process is often referred to as direct stiffness summation.

global form of (B.7) is given by

$$\sum_{q=0}^{N-1} \Big(M_{pq} + (G^x)_{pq} + (G^y)_{pq} + \sum_{w=0}^{N-1} (v_1)_w (C^x)_{pwq} + \sum_{w=0}^{N-1} (v_2)_w (C^y)_{pwq} + K_{pq} \Big) u_q \overset{!}{=} \sum_{q=0}^{N-1} M_{pq} f_q \quad \forall\, p = 0, \ldots, N-1 \tag{B.13a}$$

$$\iff \Big(\boldsymbol{M} + \boldsymbol{G}^x + \boldsymbol{G}^y + \boldsymbol{v}_1 \boldsymbol{C}^x + \boldsymbol{v}_2 \boldsymbol{C}^y + \boldsymbol{K} \Big) \boldsymbol{u} \overset{!}{=} \boldsymbol{M} \boldsymbol{f}, \tag{B.13b}$$

where $\boldsymbol{M}$ is the global mass matrix, $\boldsymbol{G}^x$ and $\boldsymbol{G}^y$ are the global gradient matrices, $\boldsymbol{C}^x$ and $\boldsymbol{C}^y$ are the global convection matrices, and $\boldsymbol{K}$ is the global stiffness matrix. Note that, following the definition from *NumPy*, the left-hand side multiplication of a column vector to a matrix of three indices will be defined as the sum-product over the second-to-last axis of the matrix. Likewise, the right-hand side multiplication of a column vector will be defined as the sum-product over the last axis of the matrix.

Scattering

Having solved (B.13) for $\boldsymbol{u}$, the element coefficients $\mathfrak{u}$ can be recovered using the scatter operator scatter(·):

$$\mathfrak{u} = \text{scatter}(\boldsymbol{u}) \iff u_{ij}^{mn} = u_{p=\text{GlobalIndex}(m,n,i,j)} \quad \forall\, m, n, i, j. \tag{B.14}$$

The element coefficients are mainly needed for plotting purposes. Note that scatter(·) is not the inverse of assemble(·), nor vice-versa.

Python Functions

Taking the sparsity of the global matrices into account, the corresponding *Python* functions read:

- `SEM.assemble` takes an element array as *NumPy* array and returns the assembled global matrix. A six-dimensional array (e.g. $\mathfrak{M}$) is returned as two-dimensional *SciPy* CSR-matrix (e.g. $\boldsymbol{M}$). An eight-dimensional array (e.g. $\mathfrak{C}^x$) is returned as three-dimensional `Sparse` COO-matrix (e.g. $\boldsymbol{C}^x$).
- `SEM.scatter` takes the global vector $\boldsymbol{u}$ as *NumPy* array and returns the scattered element coefficient array $\mathfrak{u}$ as four-dimensional *NumPy* array.
- `SEM.global_mass_matrix`, `SEM.global_stiffness_matrix` and `SEM.global_gradient_matrices` return the global mass, stiffness and gradient matrices as two-dimensional *SciPy* CSR-matrices.
- `SEM.global_convection_matrices` returns the global convection matrices as three-dimensional `Sparse` COO-matrices.

B.3 Boundary Conditions

Reconsider the differential equation (B.5a) but with DIRICHLET and homogeneous NEUMANN boundary conditions on the partitions $\partial\Omega_\mathrm{D}$ and $\partial\Omega_\mathrm{N}$ of the boundary $\partial\Omega$

$$\frac{\partial u}{\partial n} \overset{!}{=} 0 \quad \forall\,(x,y) \in \partial\Omega_\mathrm{N}, \tag{B.15a}$$

$$u \overset{!}{=} u_\mathrm{D} \quad \forall\,(x,y) \in \partial\Omega_\mathrm{D}. \tag{B.15b}$$

The general procedure is to neglect the DIRICHLET condition for now, and to artificially impose a homogeneous NEUMANN boundary condition on $\partial\Omega_\mathrm{D}$ instead. Together with (B.15a), the global matrix $\boldsymbol{A}$ and the right-hand-side vector $\boldsymbol{f}$ is constructed as described in subsection B.2. The DIRICHLET condition (B.15b) is then imposed on all rows corresponding to points on $\partial\Omega_\mathrm{D}$, such that

$$\sum_{q=0}^{N-1} A_{pq} u_q \overset{!}{=} f_p \qquad \forall\, p \in \{p = 0, \ldots, N-1 \,|\, (x_p, y_p) \in \partial\Omega_\mathrm{N}\}, \tag{B.16a}$$

$$u_p \overset{!}{=} u_\mathrm{D}(x_p, y_p) \quad \forall\, p \in \{p = 0, \ldots, N-1 \,|\, (x_p, y_p) \in \partial\Omega_\mathrm{D}\}. \tag{B.16b}$$

This way, for each point on the boundary, there *must* be *either* a NEUMANN or a DIRICHLET condition.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag der Professur für Strömungsmechanik eingereichte Projektarbeit zum Thema

Simulation Components for Evaluating a Framework for Multidisciplinary Analyzes

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Dresden, 31. Oktober, 2021

Simon Ehrmanntraut