DOI: 10.15514/ISPRAS-2026-38(3)-27

Implementation of a numerical model of fish swimming in OpenFOAM

A.N. Nuriev, ORCID: 0000-0003-1561-557X <Artem.Nuriev@kpfu.ru>
D.A. Ismagilov, ORCID: 0009-0000-3635-5485 <DaAIsmagilov@kpfu.ru>
N.N. Gumerov, ORCID: 0009-0003-2921-7430 <NaNGumerov@kpfu.ru>

Kazan (Volga region) Federal University,
18, Kremlevskaya st., Kazan, 420008, Russia.

Abstract. This paper examines a two-dimensional numerical model for analyzing the hydrodynamics of fish propulsion using the OpenFOAM software suite. The study proposes an oscillation method based on the deformation of a mesh rigidly attached to the body. Auxiliary modules for OpenFOAM were developed to automatically determine cruising speed and the power expended on propulsion. The proposed model is verified by comparison with known data.

Keywords: propulsive motion; Navier-Stokes equation; cruising speed; numerical simulation; OpenFOAM.

For citation: Nuriev A.N., Ismagilov D.A., Gumerov N.N. Implementation of a numerical model of fish swimming in OpenFOAM. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 3, part 2, 2026, pp. 161-174. DOI: 10.15514/ISPRAS-2026-38(3)-27.

Acknowledgements. The study was supported by a grant of the Russian Science Foundation (Project No. 22-79-10033-П).

Реализация численной модели плавания рыбы в OpenFOAM

А.Н. Нуриев, ORCID: 0000-0003-1561-557X <Artem.Nuriev@kpfu.ru>
Д.А. Исмагилов, ORCID: 0009-0000-3635-5485 <DaAIsmagilov@kpfu.ru>
Н.Н. Гумеров, ORCID: 0009-0003-2921-7430 <NaNGumerov@kpfu.ru>

Казанский (Приволжский) федеральный университет,
Россия, 420008, г. Казань, ул. Кремлевская, д.18.

Аннотация. Исследование посвящено моделированию гидродинамики рыб. В программном комплексе OpenFOAM реализуется численная модель для описания пропульсивного волнообразного движения рыбоподобно пловца в вязкой несжимаемой жидкости. В исследовании предлагается метод совершения колебаний, основанный на деформации сетки, жестко связанной с телом. Для автоматического определения крейсерской скорости и мощности, затрачиваемой на движение, разработаны вспомогательные модули для OpenFOAM. Проводится верификация предложенной в работе модели путем сравнения с известными данными.

Ключевые слова: пропульсивное движение; уравнение Навье-Стокса; крейсерская скорость; численное моделирование; OpenFOAM.

Для цитирования: Нуриев А.Н., Исмагилов Д.А., Гумеров Н.Н. Реализация численной модели плавания рыбы в OpenFOAM. Труды ИСП РАН, том 38, вып. 3, часть 2, 2026 г., стр. 161–174 (на английском языке). DOI: 10.15514/ISPRAS–2026–38(3)–27.

Благодарности. Исследование было поддержано грантом Российского научного фонда (проект № 22-79-10033-П).

1. Introduction

One of the most promising development directions in the aerohydrodynamics of unmanned devices is currently the creation of propulsive biomimetic systems. The primary challenge facing developers and researchers today is for biomimetic devices to achieve the same outstanding key aerohydrodynamic characteristics as natural propulsors. Although active research into the mechanics of bird and fish locomotion began as early as the first half of the 20th century [1-12], many aspects of this field remain incompletely explored to this day (see, for example, the review works by Wu 2020 [13], Zhang 2022 [14], Coe 2024 [15], Nuriev 2025 [16]). Currently, direct numerical simulation offers vast opportunities for studying the aerohydrodynamics of biomimetic systems. This work focuses on building a numerical model of fish swimming and developing a toolkit for analyzing its motion characteristics within the open-source software complex OpenFOAM.

The OpenFOAM software complex is deservedly considered one of the most tested, universal, and powerful open-source packages for solving aerohydrodynamic problems. However, its toolkit for simulating the propulsive motion of biomimetic systems is very limited: standard libraries and utilities do not fully enable solving such tasks. At the same time, the package's open object-oriented source code allows for the implementation and integration of any additional software modules.

The work [17] presented the Ika-Flow software library for OpenFOAM, designed for simulating fish swimming using the overset mesh method. Based on the standard solidBodyMotionSolver class, the fishBodyMotionSolver class was created in that work, which allows describing complex body deformations.

This study explores a different approach for simulating the undulatory motion of a fish, based on the deformation of body-fitted meshes. Based on the standard waveDisplacement class, the fishWaveDisplacement class is created in this work to describe the undulating motion of the fish body. Additionally, auxiliary modules UpdateBC and Power are implemented to determine the cruising speed and the power consumption of the fish. Verification of the developed numerical model is conducted by comparing the results with known data from Dong 2007[18].

2. Mathematical Model of Fish Motion in a Viscous Fluid

Consider a two-dimensional fish model represented by a generalized Zhukovsky airfoil, approximated in shape to the NACA0012 airfoil. The fish swims with an average velocity U along the Ox axis of the Cartesian coordinate system. The kinematics of the fish motion are defined by the law of motion:

$$y(x,t) = \alpha A(x/L)sin\big(2\pi(x/(\lambda L) - ft)\big), \qquad 0 \le x \le L\,, \tag{1}$$

where $A(x)$ is the relative wave amplitude ($A(1) = 1$), λ is the wavelength, f is the oscillation frequency, and α is the oscillation amplitude of the tail tip. The oscillation profile is defined as:

$$A(x) = 1 + (x-1)c_1 + (x^2-1)c_2.$$

By varying the constants c_1 and c_2, we consider two of the most common swimming modes: carangiform ($c_1 = -0.825, c_2 = 1.625$) and anguilliform ($c_1 = 0.323, c_2 = 0.310$).

The hydrodynamics are described by the system of Navier-Stokes equations. After non-dimensionalizing velocity by the characteristic oscillation velocity of the tail tip $U_{osc} = 2\pi\alpha f$, spatial coordinates by L, time by L/U_{osc}, and transitioning to the Cartesian coordinate system, the governing system of equations is written as:

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u}\cdot\nabla\boldsymbol{u} = -\nabla p + \frac{1}{KC\,\beta}\Delta\boldsymbol{u}, \qquad \nabla\cdot\boldsymbol{u} = 0, \tag{2}$$

where $\vec{u}$ is the dimensionless velocity, t is the dimensionless time, and p is the dimensionless pressure. The dimensionless parameters KC and β – the dimensionless oscillation amplitude and dimensionless oscillation frequency, respectively – are defined by the following expressions:

$$KC = \frac{2\pi\,\alpha}{L}, \qquad \beta = \frac{L^2 f}{\nu},$$

where ν is the kinematic viscosity.

Equations (2) are supplemented with boundary conditions specifying the velocity on the fish surface S according to (1) and the fluid velocity at infinity:

$$\boldsymbol{u}_\infty = (-U_{st}, 0), \tag{3}$$

where $U_{st} = U/U_{osc}$ is the dimensionless swimming speed of the fish. Hereafter, U_{st} will be considered either as a constant or determined from the condition of zero mean force over the oscillation period, i.e., as the cruising speed.

In this work, the standard parameters used are $KC = 0.2\pi, \lambda = 1$.

The stationary Reynolds number is defined by the following expression:

$$Re = \frac{U_{st}L}{\nu}.$$

Define the force vector $\boldsymbol{F} = p\boldsymbol{n} - \boldsymbol{T}\cdot\boldsymbol{n}$ (where $\boldsymbol{T}$ is the surface stress tensor) at each point on the fish surface S. The thrust force coefficient C_T and the power coefficient C_P are calculated as:

$$C_T = \frac{\int_S F_x\,ds}{\frac{1}{2}\rho U_{osc}^2 LH}, \qquad C_P = \frac{\int_S F_y u_y\,ds}{\frac{1}{2}\rho U_{osc}^3 LH},$$

where F_x and F_y are the x-component and y-component of the force $\boldsymbol{F}$, respectively, and ρ is the fluid density (after non-dimensionalizat ion, $\rho = 1$). Note that the problem under consideration is planar, so the characteristic height H of the fish can be taken as unity.

3. Numerical Model

To solve the problem (1-3), a numerical finite-volume model is implemented in OpenFOAM.

3.1 Computational Mesh

The initial computational mesh is constructed around a stationary fish. The mesh is generated by conformal mapping of a rectangular region $\theta \in [0, 2\pi],\ r \in [r_0, r_\infty]$:

$$x(\theta) = \frac{m_\infty}{2}(e^r\cos\theta - m)\left(1 + \frac{(1-m)^2}{e^{2r} + m(m - 2e^r\cos\theta)}\right) - x_0,$$

$$y(\theta) = \frac{m_\infty}{2}e^r\sin\theta\left(1 - \frac{(1-m)^2}{e^{2r} + m(m - 2e^r\cos\theta}\right),$$

$$\theta \in [0,\pi], \qquad m_\infty = \frac{1+m}{2}, \qquad x_0 = \frac{m_\infty}{2}(e^{r_0}\cos\pi - m)\left(1 + \frac{(1-m)^2}{e^{2r_0} + m(m - 2e^{r_0}\cos\pi)}\right),$$

where r, θ are the polar radius and angle respectively, m is the thickness parameter, r_0, r_∞ are the minimum and maximum values of the polar radius. Mesh generation is performed using the MATLAB toolbox utility [19], which provides a programming interface for creating meshes in the OpenFOAM polyMesh format.

```
function generateProfileMesh()
        % Generation of a structured hexahedral mesh around a Zhukovsky airfoil
        % Uses MATLAB toolbox for OpenFOAM
        % https://github.com/roenby/blockMesh

        %% INITIALIZATION OF PARAMETERS
        caseDir = pwd;
        toolboxDir = ['..' filesep 'meshingTools'];

        % Create OpenFOAM case directory structure
        meshDir = makeCaseDir(caseDir,toolboxDir);
        copyGeneratingCode(meshDir,toolboxDir,mfilename('fullpath'));

        % Export parameters
        compress = 0;        % 1 for compressing output files
        writePrec = 12;      % Writing precision in files
        prec = 1e-6;         % Tolerance for point identity determination

        %% GEOMETRIC PARAMETERS AND DISCRETIZATION
        l = 1;               % Cross-section side length
        nr = 250;            % Number of cells in radial coordinate
        nth = 250;           % Number of cells in azimuthal coordinate

        %% CREATE BASE BLOCK (HALF GEOMETRY)
        b = unitBlock(nr, nth/2, 1);

        % Coordinate parametrization
        th = pi * b.points(:,2);      % Azimuthal angle [0, pi]
        Z = l * (b.points(:,3) - 0.5); % Vertical coordinate [-l/2, l/2]

        %% PROFILE COORDINATE TRANSFORMATION
        % Profile transformation parameters
        r0 = 0.028;          % Radius at the body
        r_inf = 3.5;         % Radius at outer boundary
```

```
m = 0.074;           % Airfoil shape parameter
m_inf = (1 + m)/2;   % Normalization coefficient

% Calculate radial coordinate
rr = r0 + (r_inf-r0) * (b.points(:,1)) - 1 * (b.points(1,1));

% Calculate shift for symmetry
X00 = m_inf/2 * ((exp(r0) * cos(pi)) - m) * ...
    (1+(1-m)^2 * (exp(2*r0) + m*(m - 2*exp(r0)*cos(pi)))^(-1));

% X coordinate transformation (conformal mapping)
X = m_inf/2 * ((exp(rr) .* cos(th)) - m) .* ...
    (1+(1-m)^2 * (exp(2*rr) + m*(m - 2*exp(rr).*cos(th))).^(-1)) - X00;

% Y coordinate transformation (conformal mapping)
Y = m_inf/2 * (exp(rr) .* sin(th)) .* ...
    (1-(1-m)^2 * (exp(2*rr) + m*(m - 2*exp(rr).*cos(th))).^(-1));

% Apply transformation to base block
b.points(:,1) = X;
b.points(:,2) = Y;
b.points(:,3) = Z;

%% MAPPING UPPER HALF-PLANE TO LOWER HALF-PLANE
% Create block copy
b2 = b;

% Rotation to create symmetric half
b2 = rotBlock(b2, [pi 0 0]);    % Rotate around X-axis by 180°

% Merge two halves into complete geometry
b = mergeBlocks(b, b2, prec);

%% BOUNDARY SURFACE PROCESSING
% Merge upper surface patches
ind = patchesInPlane(b, [0 0 l/2], [0 0 1], prec);
b = mergePatches(b, ind, 'top', 'patch');
% Merge lower surface patches
ind = patchesInPlane(b, [0 0 -l/2], [0 0 -1], prec);
b = mergePatches(b, ind, 'bottom', 'patch');

%% EXPORT TO OPENFOAM POLYMESH FORMAT
writePolyMesh(b, meshDir, writePrec, compress);

fprintf(Mesh successfully generated in directory: %s\n', meshDir);
```

Thus, a computational domain with an approximately cylindrical shape of radius about 9 is obtained, discretized into 6.25×10^4 hexahedral cells (see Fig. 1).

To simulate the undulatory motion of the fish body, a dynamic mesh method is used. The deformation of the internal volume mesh is calculated by solving the Laplace equation for the displacement field. To minimize cell distortion, the diffusion coefficient is chosen inversely proportional to the square of the distance to the moving boundary, which localizes deformations in its vicinity. The displacement of boundary points is implemented in the fishWaveDisplacement module, described in Section 4. The surface velocity of the body is automatically determined from the computed mesh displacements and passed to the hydrodynamic solver.

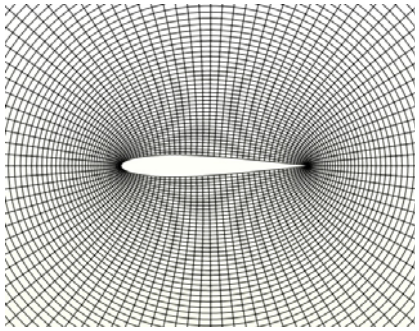

Fig. 1. Characteristic structure of the initial computational grid.

3.2 Numerical Scheme

The solution to problem (1-3) is conducted in OpenFOAM in integral form. The discretization of terms in the system of motion equations is performed as follows: the unsteady term is approximated at the centers of the computational mesh cells as the product of the average value of the integrand function and the cell volume; temporal derivative discretization is implemented using the second-order Crank-Nicolson scheme with parameter 0.9; for computing other volume integrals in the control volume system of equations, the Gauss-Ostrogradsky theorem is applied, and the resulting surface integrals are represented as a sum of integrals over cell faces and approximately computed using the midpoint rule. The characteristic Courant number did not exceed the value 0.1.

Function values and normal gradients on cell surfaces for internal cells of the computational domain are interpolated from function values at the centers of adjacent cells.

Pressure on cell surfaces is computed using linear interpolation (Gauss linear). For variable interpolation in convective terms, the limited NVD scheme GammaV with a parameter of 0.5 is used. In diffusion terms, normal velocity gradients on cell surfaces are computed from velocity values at the centers of adjacent cells using a second-order symmetric scheme with non-orthogonality correction (Gauss linear corrected).

The solution of the discretized hydrodynamic problem in the domain is performed using the PIMPLE method, which combines the PISO and SIMPLE algorithms. The solution of the pressure equation system is carried out using the preconditioned stabilized bi-conjugate gradient method (PBICGStab) with a geometric-algebraic multigrid preconditioner (GAMG) and a combined smoother (DICGaussSeidel), where DIC smoothing is followed by Gauss-Seidel smoothing. For solving the velocity equation system, the bi-conjugate gradient method (PBICG) with a preconditioner based on incomplete LU factorization (DILU) is used. The PIMPLE algorithm employs three outer corrector iterations, and one iteration each for pressure field correction and non-orthogonality correction. Calculations are performed distributedly using MPI technology with domain decomposition.

4. Implementation of Additional Software Modules

As part of the numerical model implementation, the following additional OpenFOAM modules were developed: fishWaveDisplacement – for simulating undulatory boundary motion (fish body), UpdateBC – for determining cruising speed, and Power – for calculating power consumption.

4.1 fishWaveDisplacement Module

The fishWaveDisplacement module was based on the source files of the standard waveDisplacement module, with subsequent modifications made to them.

The main changes in the header file include renaming the class to fishWaveDisplacementPointPatchVectorField and adding new private fields required for simulating fish biomechanical motion, specifically:

```
    private:
        vector amplitude_;    // wave amplitude
        scalar omega_;        // angular oscillation frequency
        vector waveNumber_;   // wave vector
        scalar c1_;          // envelope coefficient c1
        scalar c2_;          // envelope coefficient c2
```

In the implementation file "fishWaveDisplacementPointPatchVectorField.C", the constructor with dictionary was extended to read new parameters:

```
fishWaveDisplacementPointPatchVectorField
(
   const pointPatch& p,
   const DimensionedField<vector, pointMesh>& iF,
   const dictionary& dict
)
:
   fixedValuePointPatchField<vector>(p, iF, dict),
   amplitude_(dict.lookup("amplitude")),
   omega_(readScalar(dict.lookup("omega"))),
   waveNumber_(dict.lookupOrDefault<vector>("waveNumber", Zero)),
   c1_(readScalar(dict.lookup("c1"))),      // reading coefficient c1
   c2_(readScalar(dict.lookup("c2")))       // reading coefficient c2
{
   if (!dict.found("value"))
   {
       updateCoeffs();
   }
}
```

The core displacement calculation algorithm is implemented in the updateCoeffs() method. First, mesh and time data are obtained:

```
void Foam::fishWaveDisplacementPointPatchVectorField::updateCoeffs()
{
   if (this->updated()) return;

   const polyMesh& mesh = this->internalField().mesh()();
   const Time& t = mesh.time();
   const scalarField points(waveNumber_ & patch().localPoints());
   scalarField xCoord = patch().localPoints().component(vector::X);
```

The core algorithm is as follows:

```
const double timestart = 0.05;

   // Envelope calculation
   scalar shapeFactor=(1.0+(xCoord/1.0-1.0)*c1_+c2_*(xCoord*xCoord/1.0/1.0-1));

   if (t.value() < timestart)
   {
     // Smooth start with linear amplitude increase
     Field<vector>::operator=
     (
      (t.value()/timestart)*amplitude_*shapeFactor*sin(points-omega_*timestart)
     );
   }
   else
   {
     // Main operation mode
     Field<vector>::operator=
```



```
     (
      amplitude_*shapeFactor*sin(points-omega_*t.value())
     );
   }

   fixedValuePointPatchField<vector>::updateCoeffs();
}
```

At the end, the new displacement value is set in the boundary conditions through the assignment operator Field<vector>::operator=.

The write method was also extended to output new parameters:

```
void Foam::fishWaveDisplacementPointPatchVectorField::write(Ostream& os) const
{
   pointPatchField<vector>::write(os);
   os.writeKeyword("amplitude")<<amplitude_<<token::END_STATEMENT<<nl;
   os.writeKeyword("omega")<<omega_<<token::END_STATEMENT<<nl;
   os.writeKeyword("waveNumber")<<waveNumber_<<token::END_STATEMENT<<nl;
   os.writeKeyword("c1")<<c1_<<token::END_STATEMENT<<nl; // linear coefficient
   os.writeKeyword("c2")<<c2_<<token::END_STATEMENT<<nl; // quadratic coeff.
   writeEntry("value", os);
}
```

Note that the algorithm implements a smooth motion start mechanism to avoid abrupt initial mesh restructuring in the computational domain. The timeStart parameter defines the time over which the amplitude reaches its steady-state value.

4.2 UpdateBC Module

The UpdateBC module was developed based on the source files of the standard forcesCoeffs module, with subsequent modifications made to them.

The main changes in the header file include renaming the class to UpdateBC and adding new private fields required for the cruising speed search algorithm, namely algorithm parameters:

```
scalar omega_; // oscillation cyclic frequency
scalar v_st_; // initial velocity value
scalar dv_start_; // initial velocity correction step
scalar per_max_; // minimum number of periods between corrections
scalar per_first_; // number of periods before first velocity correction
scalar corr_time_; // time required for correction
scalar epsilon_; // convergence criterion for averaged force
word inletPatchName_; // inlet patch name
word outletPatchName_; // outlet patch name
```

And algorithm state variables:

```
scalar v_st1; // current velocity value
scalar v_st0; // velocity value from previous correction
scalar F_av; // current averaged force value
scalar F_av_last; // averaged force value from previous period
scalar F_av_old; // averaged force value from previous correction
scalar timer1; // smooth correction timer
scalar per; // period counter
scalar per_m; // current required number of periods between corrections
scalar da; // velocity increment
```

In the implementation file "UpdateBC.C", the read method was extended to read new parameters:

```
bool UpdateBC::read(const dictionary& dict)
{
   forces::read(dict);
   dict.readEntry("inletPatchName", inletPatchName_);
   dict.readEntry("outletPatchName", outletPatchName_);
   dict.readEntry("omega", omega_);
   dict.readEntry("v_st", v_st_);
```

```
    dict.readEntry("dv_start", dv_start_);
    dict.readEntry("per_max", per_max_);
    dict.readEntry("corr_time", corr_time_);
    dict.readEntry("epsilon", epsilon_);
    return true;
}
```

The main algorithm for searching the cruising speed is implemented in the execute method. First, the parent class method calcForcesMoments is called to compute current forces, and temporary variables are created:

```
forces::calcForcesMoments();

const auto& coordSys = coordSysPtr_();
const vector localForce(coordSys.localVector(forceEff()));
scalar Fx = localForce[0]; // force increment per time step
scalar tstep = mesh.time().deltaT().value(); // time step
scalar frequency_ = omega_/2.0/M_PI; // oscillation frequency
scalar time_p = obr_.time().value() + tstep; // time within period
time_p -= floor(time_p * frequency_) / frequency_;
```

Note: the field v_st1_ in the class constructor is initialized with value -20, which is necessary for detecting the first correction:

```
if (v_st1<=-19)
{
   v_st1 = v_st0 = v_st_;
   per_m = per_first_;
   per=0;
}
```

The core algorithm is as follows:

```
if (time_p < tstep)
{
   F_av *= tstep * frequency_;
   if (fabs(F_av) > epsilon_ && per > per_m &&
       fabs(F_av - F_av_last) < epsilon_/50)
   {
       timer1 = corr_time_;
       scalar dv;
       if (per_m == per_first_)
       {
         dv = dv_start_;
         per_m = per_max_;
       }
       else
         dv = -(F_av * (v_st1 - v_st0) / (F_av - F_av_old));
       da = dv * frequency_ / timer1;
       per = 0;
       F_av_old = F_av;
       v_st0 = v_st1;
   }
   if (timer1 > 0.99999)
       timer1 -= 1.0;
   else
       da = 0.0;
   F_av_last = F_av;
   F_av = 0;
   per++;
}
```

Then force incrementation and smooth velocity change are performed:

```
F_av = F_av + Fx;
v_st1 = v_st1 + da * tstep;
```


Finally, the new velocity value is set in the boundary conditions by bypassing constness:

```
const fvMesh& mesh = refCast<const fvMesh>(obr_);
label inletPatchId = mesh.boundaryMesh().findPatchID(inletPatchName_);
const volVectorField& U = mesh.lookupObject<volVectorField>(UName_);
const inletOutletFvPatchVectorField& constVv1
    refCast<const inletOutletFvPatchVectorField>
        (U.boundaryField()[inletPatchId]);
inletOutletFvPatchVectorField& vv1 =
    const_cast<inletOutletFvPatchVectorField&>(constVv1);
vectorField& rVvv1 = vv1.refValue();

forAll(vv1, iFace)
   rVvv1[iFace] = vector(v_st1,0,0);
   vv1.updateCoeffs();
```

A similar procedure is performed for the outlet patch.

Note that the secant method is implemented to find the cruising speed ($F_x(U) = 0$), and the velocity changes smoothly to avoid disturbances in the calculation. Additionally, for correct algorithm operation, a check is performed for flow establishment ($|F_x - F_x^{last}| < \varepsilon/50$), where F_x^{last} is the averaged force from the previous period.

4.3 Power Module

Like the previous module, the Power module was based on the source files of the standard forcesCoeffs module, which were then modified to compute motion power.

The main changes in the header file include renaming the class to Power and adding new private fields necessary for power computation, namely:

```
private:
    vector power_;        // power coefficient vector
    scalar magUInf_;      // freestream velocity magnitude
    scalar lRef_;         // characteristic length [m]
    scalar Aref_;         // characteristic area [m²]
    autoPtr<OFstream> coeffFilePtr_;  // pointer to output file
```

In the implementation file "Power.C", the read method was extended to read new parameters:

```
bool Foam::functionObjects::Power::read(const dictionary& dict)
{
   if (!forces::read(dict))
   {
       return false;
   }

   dict.readEntry("magUInf", magUInf_);

   if (rhoName_ != "rhoInf")
   {
     dict.readEntry("rhoInf", rhoRef_);
   }

   dict.readEntry("lRef", lRef_);
   dict.readEntry("Aref", Aref_);

   return true;
}
```

The main algorithm for power computation is implemented in the execute method. First, the parent class method is called to prepare data and temporary variables are created:

```
bool Foam::functionObjects::Power::execute()
{
   createFiles();
```

```
    const volScalarField& p = lookupObject<volScalarField>(pName_);
    const volVectorField& U = lookupObject<volVectorField>(UName_);
    const surfaceVectorField::Boundary& Sfb = mesh_.Sf().boundaryField();
    tmp<volTensorField> tgradU = fvc::grad(U);
    const volTensorField& gradU = tgradU();
    const auto& gradUb = gradU.boundaryField();
    scalar pRef = pRef_/rho(p);
    power_ = Zero;
}
```

The core algorithm is as follows:

```
    for (const label patchi : patchIDs_)
    {
      const vectorField& Uc = U.boundaryField()[patchi];

      vectorField fT
      (
        Sfb[patchi] & devRhoReff(gradUb[patchi], patchi)
      );

      vectorField fN
       (
         rho(p)*Sfb[patchi]*(p.boundaryField()[patchi] - pRef)
       );

      vectorField f = fT + fN;

      power_[0] -= gSum(Uc.component(0) * f.component(0));
      power_[1] -= gSum(Uc.component(1) * f.component(1));
      power_[2] -= gSum(Uc.component(2) * f.component(2));
    }
```

Then the power coefficient is normalized:

```
  const scalar coeff = 1.0/(0.5*magUInf_*magUInf_*magUInf_*Aref_);

    power_[0] *= coeff;
    power_[1] *= coeff;
    power_[2] *= coeff;
```

Finally, the results are written to a file and output to the log:

```
    if (writeToFile())
    {
      writeCurrentTime(coeffFilePtr_());
      coeffFilePtr_()
           << tab << power_.x()
           << tab << power_.y()
           << tab << power_.z()
           << endl;
    }

    Log << "        Cp       : " << power_ << nl;

    return true;
}
```

The writeIntegratedHeader method was also modified to output the power file header:

```
  void Foam::functionObjects::Power::writeIntegratedHeader
  (
    const word& header,
    Ostream& os
  ) const
  {
    writeHeader(os, "Power coefficients");
    writeCommented(os, "Time");
    writeTabbed(os, "Px");
```



```
    writeTabbed(os, "Py");
    writeTabbed(os, "Pz");
    os  << endl;
}
```

The module allows computing motion power by components (P_x, P_y, P_z) and outputting results both to a file and to the console, providing convenient monitoring of the propulsor's energy characteristics during the calculation.

5. Testing and Simulation Results

Fig. 2 shows images of deformed meshes for carangiform and anguilliform swimming styles obtained during testing of the fishWaveDisplacement module.

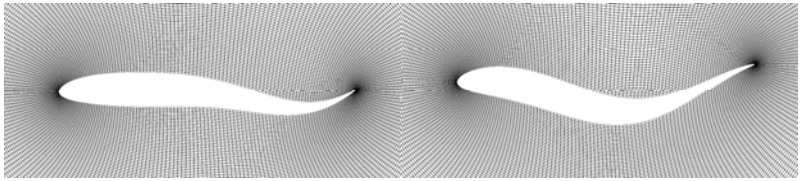

Fig. 2. Images of grids in a deformed state for carangiform (left) and anguilliform (right) swimming styles.

The results show that despite relatively large mesh deformations, its key metrics including maximum skewness and non-orthogonality remain within acceptable ranges (see Fig. 3), ensuring the required calculation accuracy.

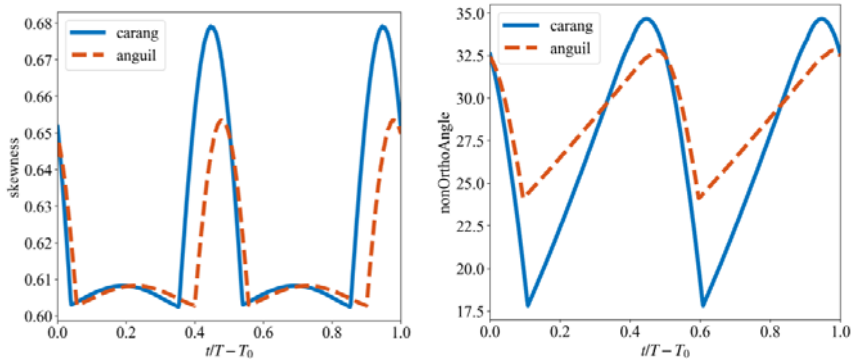

Fig. 3. Graphs of the maximum skewness and non-orthogonality of the computational grid over the period.

The results of the cruising speed search module are presented in Table 1. They show that as the β parameter increases, the simulated biomimetic swimmer reaches a cruising speed approximately equal to the oscillation amplitude of its tail tip velocity, which agrees well with known data for living fish [20-21].

Table. 1. Calculation results for finding the cruising speed.

Carangiform style		Anguilliform style	
β	U_{st}	β	U_{st}
2500	0.7725	2500	0.7566
5000	0.9011	5000	0.9069
10000	1.0328	10000	1.0465

Model verification based on integral characteristics was performed for the carangiform swimming style. Period-averaged power coefficients were compared with data from the article Dong 2007[18] at different values of dimensionless oscillation frequency at the Reynolds number $Re = 5000$. The comparison of results, presented in Fig. 4, indicates that the developed model accurately describes the characteristics of the biomimetic propulsor.

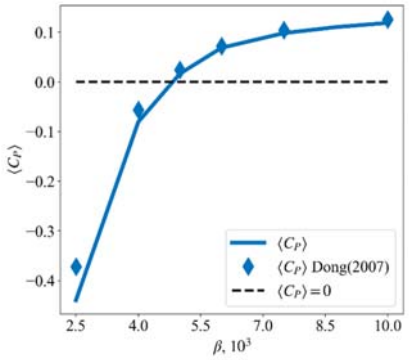

Fig. 4. Graph of the period-averaged power coefficient $\langle C_P \rangle$ versus the dimensionless oscillation frequency β, compared with data from the article Dong 2007[18] for carangiform swimming at the Re = 5000.

6. Conclusion

This work developed a numerical model for simulating undulatory fish motion in the OpenFOAM package. The presented results confirm the correct operation of auxiliary modules for simulating undulatory motion, determining cruising speed, and calculating power consumption. The developed toolkit provides an effective method for future analysis of fish propulsion hydrodynamics.

References

[1]. Prandtl L. Über die Entstehung von Wirbeln in der idealen Flüssigkeit, mit Anwendung auf die Tragflügeltheorie und andere Aufgaben. Vorträge aus dem Gebiete der Hydround Aerodynamik (Innsbruck 1922), 1924, pp. 18-33.
[2]. Birnbaum W. Der schlagflügelpropeller und die kleinen schwingungen elastisch befestigter tragflügel. Z. Flugtech. Motorluftschiffahrt, 1924, bd. 15, pp. 128-134.
[3]. Kármán T. V., Burgers J. M. Aerodynamic Theory: General aerodynamic theory: Perfect fluids. Number v. 2. J. Springer, 1935.
[4]. Theodorsen T., NACA R-496: General Theory of Aerodynamic Instability and the Mechanism of Flutter. National Advisory Committee for Aeronautics, 1935.
[5]. Keldysh M. V., Lavrentiev M. A. On the theory of the oscillating wing. Tekh. Zametki TsaGI, 1935, iss. 45, pp. 48-52. (in Russian).
[6]. Siekmann J. Theoretical studies of sea animal locomotion, part 1. Ingeneur-Archiv, 1962, vol. 31, no. 3, pp. 214-227.
[7]. Wu T. Y. T. Swimming of a waving plate. Journal of Fluid Mechanics, 1961, vol. 10, no. 3, pp. 321-344.
[8]. Wu T. Y. T. Hydromechanics of swimming propulsion. part 1. swimming of a two-dimensional flexible plate at variable forward speeds in an inviscid fluid. Journal of Fluid Mechanics, 1971, vol. 46, no. 2, pp. 337-355.
[9]. Lighthill M. J. Note on the swimming of slender fish. Journal of Fluid Mechanics, 1960, vol. 9, no. 2, pp. 305-317.
[10]. Lighthill M. J. Hydromechanics of aquatic animal propulsion. Annual Review of Fluid Mechanics, 1969, vol. 1, no. 1, pp. 413-446.
[11]. Lighthill M. J. Aquatic animal propulsion of high hydromechanical efficiency. Journal of Fluid Mechanics, 1970, vol. 44, no. 2, pp. 265-301.

[12]. Lighthill, M. J. Large-Amplitude Elongated-Body Theory of Fish Locomotion. Proceedings of the Royal Society B, 1971, vol. 179, no. 1055, pp. 125-138.
[13]. Wu X., Zhang X., Tian X., Li X., Lu W. A review on fluid dynamics of flapping foils. Ocean Engineering, 2020, vol. 195, id. 106712.
[14]. Zhang D., Zhang J. D., Huang W. X. Physical models and vortex dynamics of swimming and flying. Acta Mechanica, 2022, vol. 233, no. 4, pp. 1-40.
[15]. Coe M., Gutschmidt S. Cost of Transport is not the whole story. Ocean Engineering, 2024, vol. 313, id. 119332.
[16]. Nuriev A.N., Zaitseva O.N., Zhuchkova O.S. On the study of oscillatory motion of bodies in fluid. Uchenye Zapiski Kazanskogo Universiteta, Seriya Fiziko-Matematicheskie Nauki, 2025, vol. 167, no. 1, pp. 54-98.
[17]. Coe M., Gutschmidt S. IKA-FLOW: A Flexible Body Overset Mesh Implementation for Fish Swimming. OpenFOAM® Journal, 2023, vol. 3, pp. 75-119.
[18]. Dong G. L., Lu X. Y. Characteristics of flow over traveling wavy foils in a side-by-side arrangement. Physics of Fluids, 2007, vol. 19, no. 5, id. 057107.
[19]. Roenby J. blockMesh. Available at: https://github.com/roenby/blockMesh, 2012, accessed 01.03.2026.
[20]. Bainbridge R. The Speed of Swimming of Fish as Related to Size and to the Frequency and Amplitude of the Tail Beat. The Journal of Experimental Biology, 1958, vol. 35, pp. 109-133.
[21]. Sánchez-Rodríguez J., Raufaste C., Argentina M. Scaling the tail beat frequency and swimming speed in underwater undulatory swimming. Nature Communications, 2023, vol. 14, no. 1, id. 5569.

Информация об авторах / Information about authors

Артем Наилевич НУРИЕВ – доктор физико-математических наук, профессор кафедры аэрогидромеханики Института математики и механики им. Н.И. Лобачевского Казанского (Приволжского) федерального университета. Сфера научных интересов: механика жидкости и газа, вычислительная гидродинамика, теория турбулентности, параллельные вычисления.

Artem Nailevich NURIEV – Dr. Sci. (Phys.-Math), Professor of the Department of Aerohydrodynamics at the N.I. Lobachevsky Institute of Mathematics and Mechanics of Kazan (Volga Region) Federal University. Field of scientific interests: fluid and gas mechanics, computational fluid dynamics, turbulence theory, parallel computing.

Дамир Альбертович ИСМАГИЛОВ – студент магистратуры Института математики и механики им Н.И. Лобачевского Казанского (Приволжского) федерального университета. Сфера научных интересов: механика жидкости и газа, численное моделирование, теория оптимизации.

Damir Albertovich ISMAGILOV – Master's student at the N.I. Lobachevsky Institute of Mathematics and Mechanics of Kazan (Volga Region) Federal University. Field of scientific interests: fluid and gas mechanics, numerical modeling, optimization theory.

Наиль Нафисович ГУМЕРОВ – студент магистратуры Института математики и механики им Н.И. Лобачевского Казанского (Приволжского) федерального университета. Сфера научных интересов: механика жидкости и газа, вычислительная гидродинамика, моделирование движения пропульсивных систем.

Nail Nafisovich GUMEROV – Master's student at the N.I. Lobachevsky Institute of Mathematics and Mechanics of Kazan (Volga Region) Federal University. Field of scientific interests: fluid and gas mechanics, computational fluid dynamics, modeling of propulsive systems.