Supplement Information

Tree migration in the dynamic, global vegetation model LPJ-GM 1.1: Efficient uncertainty assessment and improved dispersal kernels of European trees

Deborah Zani[1,2], Veiko Lehsten[1,2], and Heike Lischke[1]

[1]Swiss Federal Institute of Forest, Snow and Landscape Research WSL
[2]University of Lund, Sweden

This Notebook reproduces the analyses of the paper Zani et al. Geosci. Model Dev., including the generation of figures and supplement tables. It also allows to inspect the data in more details.

Abstract

The prediction of species geographic redistribution under climate change (i.e. range shifts) has been addressed by both experimental and modelling approaches and can be used to inform efficient policy measures on the functioning and services of future ecosystems. Dynamic Global Vegetation Models (DGVMs) are considered state-of-the art tools to understand and quantify the spatio-temporal dynamics of ecosystems at large scales and their response to changing environments. They can explicitly include local vegetation dynamics relevant to migration (establishment, growth, seed (propagule) production), species-specific dispersal abilities and the competitive interactions with other species in the new environment. However, the inclusion of more detailed mechanistic formulations of range shift processes may also widen the overall uncertainty of the model. Thus, a quantification of these uncertainties is needed to evaluate and improve our confidence in the model predictions. In this study, we present an efficient assessment of parameter and model uncertainties combining low-cost analyses in successive steps: local sensitivity analysis, exploration of the performance landscape at extreme parameter values, and inclusion of relevant ecological processes in the model structure. This approach was tested on the newly-implemented migration module of the state-of-the-art DGVM LPJ-GM. Estimates of post-glacial migration rates obtained from pollen and macrofossil records of dominant European tree taxa were used to test the model performance. The results indicate higher sensitivity of migration rates to parameters associated with the dispersal kernel (dispersal distances and kernel shape) compared to plant traits (germination rate and maximum fecundity) and highlight the importance of representing rare long-distance dispersal events via fat-tailed kernels. Overall, the successful parametrization and model selection of LPJ-GM will allow simulating plant migration with a more mechanistic approach at larger spatial and temporal scales, thus improving our efforts to understand past vegetation dynamics and predict future range shifts in a context of global change.

1 How to run this notebook

After downloading the Supplement folder containing this notebook here, activate the environment `environment.yml` and start the notebook from the command line:

```
(base)%pwd >conda activate LPJGM_uncertainty
```

```
(LPJGM_uncertainty)%pwd >jupyter-notebook
```

Input data are included in the folder at the address `%pwd`.

[1]:
```
datapath = %pwd
```

2 Libraries

[2]:
```
import numpy as np
from numpy import nanmean
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import statistics
from scipy import stats
from scipy.stats import mannwhitneyu
from statsmodels.sandbox.stats.multicomp import multipletests
from statsmodels.stats.multicomp import pairwise_tukeyhsd
import copy
import math
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
from scipy.special import gamma
import itertools
```

3 Observational data: estimates of past migration rates

We used estimates of migration rates from paleo-records of European forest expansion after the Last Glacial Maximum (LGM) for the parameter optimization of 17 major European tree species implemented in LPJ-GM. Upper and lower boundaries for the range values of migration rates were derived from different studies based on the method employed for their estimation.

Table 1. Estimates of maximum post-glacial migration rates in meters year^{-1}, dispersal syndromes, and expected competition during post-glacial expansion for 17 major European tree species. Lower and upper boundaries of migration rates correspond to `OBS_MIN` and `OBS_MAX`, respectively, in `Input_EVA.csv` and `Input_KA.csv`.

Species (LPJ-GM notation)	Migration rates	Dispersal syndrome(e)	Competitor(f)
Abies alba (Abi_alb)	250(a) – 300(b)	Wi	Bet_pen & C3G
Betula pendula (Bet_pen)	540(c) – 800(a)	Wi, (Wa)	Bet_pub & C3G
Betula pubescens (Bet_pub)	540(c) – 800(a)	Wi, (Wa, LA)	Bet_pen & C3G
Carpinus betulus (Car_bet)	500(a) – 1000(b)	W, (Wa, B, SA)	Bet_pen & C3G
Corylus avellana (Cor_ave)	1000(a) – 1500(b)	(Wa), B, SA, LA	Bet_pen & C3G
Fagus sylvatica (Fag_syl)	200(b) – 300(b)	(Wa), B, SA, LA	Bet_pen & C3G OR Que_rob & C3G
Fraxinus excelsior (Fra_exc)	200(b) – 500(b)	Wi, (Wa, LA)	Bet_pen & C3G
Picea abies (Pic_abi)	150(d) – 500(b)	Wi	Bet_pen & C3G

Species (LPJ-GM notation)	Migration rates	Dispersal syndrome(e)	Competitor(f)
			OR Pin_syl & C3G
Picea sitchensis (Pic_sit)	150(d) – 500(b)	Wi	Bet_pen & C3G
Pinus halepensis (Pin_hal)	600(a) – 1500(b)	Wi	Bet_pen & C3G
Pinus sylvestris (Pin_syl)	600(a) – 1500(b)	Wi, (Wa)	Bet_pen & C3G
Quercus coccifera (Que_coc)	300(d) – 500(b)	B, SA, LA	Bet_pen & C3G
Quercus ilex (Que_ile)	300(d) – 500(b)	B, SA, LA	Bet_pen & C3G
Quercus pubescens (Que_pub)	300(d) – 500(b)	B, SA, LA	Bet_pen & C3G
Quercus robur (Que_rob)	300(d) – 500(b)	(Wa), B, SA, LA	Bet_pen & C3G
Tilia cordata (Til_cor)	150(b) – 500(b)	Wi, (Wa)	Bet_pen & C3G
			OR Que_rob & C3G
			OR Pin_syl & C3G
Ulmus glabra (Ulm_gla)	550(d) – 1000(b)	Wi, (Wa)	Bet_pen & C3G

(a) Estimates of maximum migration rates by Giesecke and Brewer (2018) with pollen analysis corrected by phylogeographic studies

(b) Estimates of maximum migration rates by Huntley and Birks (1983) with pollen analysis

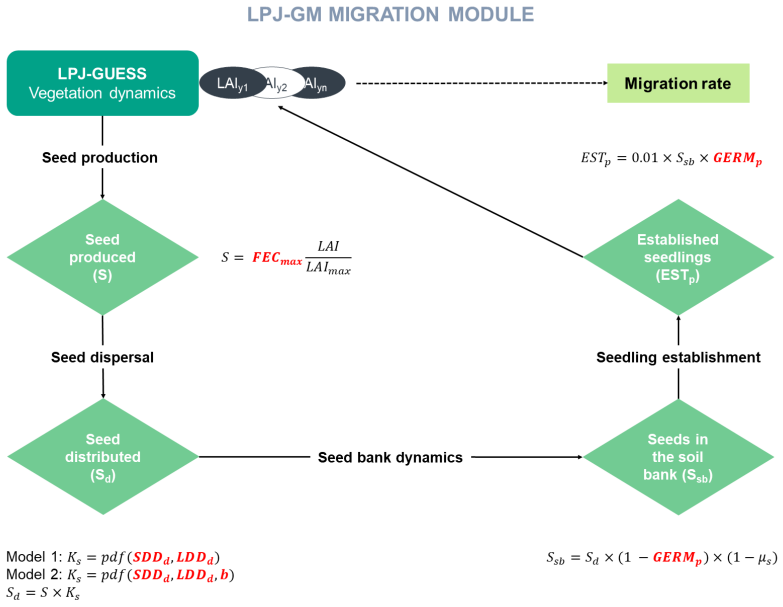
(c) Estimates of maximum migration rates by Feurdean et al. (2013) with fossil records, assuming spread from southern refugia (40-45°N latitude)

(d) Estimates of overall migration rates by Giesecke et al. (2017) derived from the increase in area of presence from interpolated pollen maps and threshold values for pollen presence

(e) Dispersal syndromes (with secondary mechanisms in parenthesis) as reported by the TRY Database (Kattge et al., 2011) and supporting literature (see Table B1 in the manuscript): Wi = wind; Wa = water; B = bird; LA = large mammal (deer, badger, cattle); SA = small animal (e.g. hoarding by rodents)

(f) Competitors in model simulations: Bet_pen / Bet_pub = *Betula pendula* / *Betula pubescens*; C3G = boreal/temperate grasses (C3 photosynthesis pathway); OR = alternative competitors (Que_rob = *Quercus robur*, Pin_syl = *Pinus sylvestris*; Brian Huntley, personal communication, January, 2022).

4 Model: LPJ-GM



LPJ-GM (Lehsten et al., 2019) couples a dynamic migration module to the widely-used DGVM, LPJ-GUESS (where LPJ-GM is short for LPJ-GUESS-MIGRATION). LPJ-GUESS employs a gap model approach for the simulation of ecophysiological processes and the structural dynamics of forests, including species composition and vertical and horizontal heterogeneity (Smith et al. 2001). Thus, the migration module of LPJ-GM simulates local vegetation dynamics and allows species to disperse simultaneously while interacting with each other. Following the TreeMig model implementation (Lischke and Löffler (2006)), LPJ-GM simulates migration at an yearly time-step through four main processes: - **seed production**. The number of seeds produced S is the product of maximum fecundity (FEC_{max}) and the proportion of current leaf area (LAI_{ind}) to the maximum (LAI_{max}):

$$S = FEC_{max} \times \frac{LAI_{ind}}{LAI_{max}}$$

- **seed dispersal**. The seeds $S(x', y')$ produced at a location (x', y') then are distributed according to a probability density function (pdf), i.e. the seed dispersal kernel k_s, so that the seed input $S_d\ (x, y)$ in location x, y is obtained by integrating over all other locations x', y'. The dispersal kernel k_s is a linear combination of two pdfs for short- (SDD) and long-distance dispersal (LDD), where LDD_p is the proportion of LDD, SDD_d and LDD_d are the average distances for SDD and LDD, respectively. In the default setting of LPJ-GM, the pdfs for both SDD and LDD components are negative exponentials.

$$S_d(x, y) = \int S(x', y') \times k_s(x - x', y - y')dx'dy'$$

$$k_s = (1 - LDD_p) \times pdf(z, SDD_d) + LDD_p \times pdf(z, LDD_d)$$

- **seed bank dynamics**. Seed bank dynamics are defined by the yearly change of dormant seeds in the soil S_{sb} that can germinate in the following years. S_{sb} increases by the seed input S_d and decreases by the number of germinated seeds (where $GERM_p$ is the rate of germination), or by loss of seeds ($_s = 0.8$ as annual rate):

$$S_{sb,t+1} = S_{sb,t+1} \times (1 - GERM_p) \times (1 - {}_s)$$
$$S_{sb,t+1} = S_{sb,t} + S_{d,t+1}$$

- **seedling establishment**. The probability of seedling establishment in a certain year EST_p depends on the number of available seeds for germination (S_{sb}) and on the germination rate. The established seedlings grow, compete, and die according to the LPJ-GUESS approach. After reaching maturity, established saplings start producing seeds according the S formula.

$$EST_p = 0.01 \times S_{sb} \times GERM_p$$

At the end of the simulation, the species-specific migration rate (meters year^{-1}) is calculated as the migration distance divided by migration time, i.e. simulation time elapsed since when trees are allowed to perform seed dispersal. Migration distance is obtained by the direct output of LPJ-GM, leaf area index (LAI), as the distance between the starting point of migration and the 95^{th} percentile farthest point in the terrain where LAI exceeds 0.5.

4.1 Simulation settings

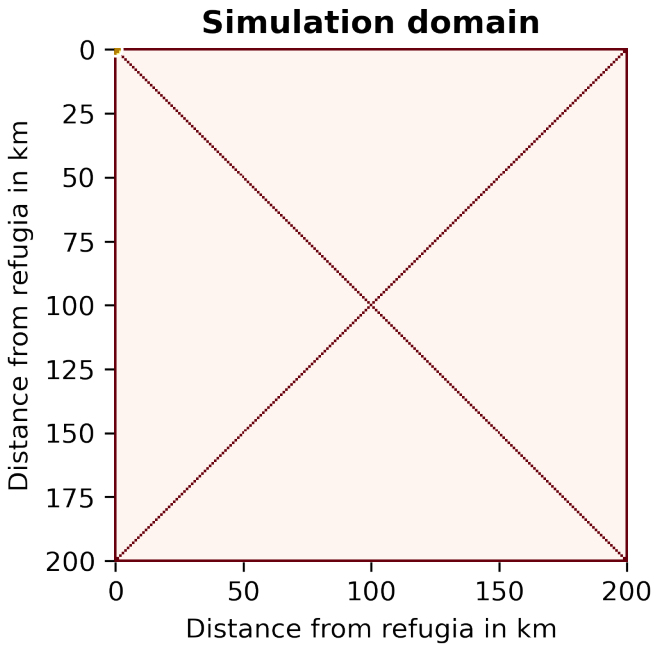
Simulations were performed for a total of 500 years and over an area of 201 x 201 cells with corridors each 200 km (for a total of 1,197 grid cells), where tree species were allowed to establish freely in the upper-left corner of the simulation landscape (the starting point of migration). We applied a static suitable climate for all species and an entirely permeable terrain to all grid cells and across all simulation years in order to reproduce optimal environmental conditions for each species' spread. We used the Fast Fourier transform method (FFTM) to enhance the computational efficiency of seed dispersal, as described in Lehsten et al., 2019.

[3]:
```
## Simulation domain
f, ax = plt.subplots()
f.set_figheight(3.54)
f.set_figwidth(3.54)
f.dpi = 300

# Initialize the spatial domain (201x2001 grid cells)
sim_domain = np.zeros((201, 201), int)

# Add corridors as major diagonals and perimeter
np.fill_diagonal(sim_domain, 1)
np.fill_diagonal(np.fliplr(sim_domain), 1)
sim_domain[0,:] = 1
sim_domain[:,0] = 1
sim_domain[200,:] = 1
sim_domain[:,200] = 1
```

Parameter	Value	Description
vegmode	“cohort”	simulation mode, either “cohort”, “individual” or “population”
nyear_spinup	53	number of years to spin up the simulation for
ifcalcsla	1	whether to calculate SLA from leaf longevity
ifcalccton	1	whether to calculate leaf C:N min from leaf longevity
firemodel	“NOFIRE”	whether to implement fire (BLAZE, GLOBFIRM) or not (NOFIRE)
weathergenerator	“GWGEN”	whether to generate climate at a smaller scale or interpolate it
npatch	1	number of replicate patches to simulate
npatch_secondarystand	1	number of replicate patches to simulate in secondary stands
reduce_all_stands	0	whether to reduce equal percentage of all stands of a stand type at land cover change
age_limit_reduce	5	Minimum age of stands to reduce at land cover change
patcharea	1000	patch area (m2)
estinterval	5	years between establishment events in cohort mode
ifdisturb	1	whether generic patch-destroying disturbances enabled
distinterval	400	average return time for generic patch-destroying disturbances
ifbgestab	1	whether background establishment enabled
ifsme	1	whether spatial mass effect enabled
ifstochestab	0	whether establishment stochastic
ifstochmort	0	whether mortality stochastic
ifcdebt	1	whether to allow vegetation C storage (1) or not (0)
wateruptake	“rootdist”	“wcont”, “rootdist”, “smart” or “speciesspecific”
rootdistribution	“jackson”	how to parameterise root distribution. Alternatives are “fixed” or “jackson”
textured_soil	1	whether to use silt/sand fractions specific to soiltype
ifcentury	1	whether to use CENTURY SOM dynamics (mandatory for N cycling)
ifnlim	1	whether plant growth limited by available N
freenyears	100	number of years to spin up without N limitation (needed to build up a N pool)
nfix_a	0.102	first term in N fixation eqn (Conservative 0.102, Central 0.234, Upper 0.367)
nfix_b	0.524	second term in N fixation eqn (Conservative 0.524, Central -0.172, Upper -0.754)
nrelocfrac	0.5	fraction of N retranslocated prior to leaf and root shedding
ifsmoothgreffmort	1	whether to vary mort_greff smoothly with growth efficiency (1) or to use the standard step-function (0)
ifdroughtlimitedestab	0	whether establishment is limited by growing season drought
ifrainonwetdaysonly	1	whether to rain on wet days only (1), or to rain a bit every day (0)
ifbvoc	0	whether to include BVOC calculations (1) or not (0)
searchradius_soil	0.01	search for soil data in up to X degrees around the EMDI coordinates

Parameter	Value	Description
years_total	500	total number of simulation years
domain	11, 49, 0.01, 0.01	coordindates of north-west corner and longitude, latitude increments
subdomains	201, 201, 1, 1	size of the domain and number of subdomains in lat-/long-itudinal direction
dispersal_patchsize	0.99	patch size for dispersal calculations (km2)
dispersal	“fft”	whether to use the Fast Fourier transform method for seed dispersal
stochastic_seed_est_scaler	0.01	scaler for stochastic seedling establishment from dispersed seed
log_stochastic_seed_est_scaler	0	if larger than 0 no log stochastic distribution for stochastic seedling establishment from dispersed seeds
output_interval	1	number of years between each output-year should be larger or equal to 1
EDGE_CELLS	40	number of gridcells to fold into the central array of dispersed seeds to avoid a torus effect

```
# Plot
ax.imshow(sim_domain, cmap='Reds', interpolation='nearest')

# Add refugium
ax.scatter(0,0, s=25, edgecolor='white', facecolor=(1, 1, 0, 0.5))

ax.set_frame_on(False)
ax.set_ylabel('Distance from refugia in km')
ax.set_xlabel('Distance from refugia in km')
ax.set_title('Simulation domain',fontweight='bold')

# Export figure
plt.savefig('Figure_S1.png', bbox_inches='tight', dpi=300)
plt.show()
```



5 Migration parameters

Since the underlying framework, LPJ-GUESS, has been already extensively validated for different metrics of vegetation composition and structure (e.g. Pappas et al. 2013), we focused our calibration effort on the newly-added migration parameters: FEC_{max}, $GERM_p$, SDD_d and LDD_d. Range values of each parameter were compiled by reviewing research articles and public databases as indicated in **Table 2**. We identified the minimum and maximum values per parameter as the lower and upper bounds, respectively, and calculated the mean along with the 25^{th} and 75^{th} percentiles, assuming a normal distribution (see `Table B1` in the main text and `SA_input` in this notebook for species-specific values: `Default`,`Min`,`25th`,`Mean`,`75th`,`Max`, where default parameters correspond to the values reported by Lischke and Löffler (2006)). `var%` indicates the normalized variability of species-specific parameters, in percentage: $\frac{x_{max}-x_{min}}{x_{max}} \times 100$.

Table 2. Description of the migration parameters and data source: a = Lischke and Löffler (2006); b = TRY database (Kattge et al., 2011); c = Royal Botanic Gardens Kew Seed Information Database (SID); d = Vittoz and Engler (2007); e = Tamme et al. (2014).

Parameter	Notation	Unit	References
Maximum fecundity per tree and year	FEC_max	no.seeds (in 100)	a, b, c
Seed germination rate	GERM_p	%	a, b, c
Average short dispersal distance	SDD_d	meters	a, b, d
Average long dispersal distance (1%)	LDD_d	meters	a, b, d, e

6 Helper functions

[4]:
```
# Scale values from 0 to 1
def scaler_0_1(df):
    return (df - np.min(df)) / (np.max(df) - np.min(df))

# Mean normalization
# i.e. centers around 0 while keeping positive and negative values
def mean_norm(data):
    return (data - np.mean(data)) / (max(data) - min(data))

# Calculate RMSE
def rmse_perc(sim,obs,obs_mean):
    MSE = np.square(np.subtract(sim,obs)).mean()
    RMSE = np.sqrt(MSE)
    return (100 / obs_mean) * RMSE

# Calculate standard error
f_ste = lambda x : x.std() / np.sqrt( x.count() )
f_ste.__name__ = 'ste'

# Find minimum residuals
def min_residual(group) :
    residuals = group.residuals_abs
    output = group[residuals == residuals.min()]
    if (output.shape[0] > 1) :
        output = output.head(1)
    return output
```

7 Load data

[5]:
```
# Load data (\\ for Windows)
SA_input = pd.read_table(datapath+'\\Input_SA.csv',header=0,delimiter=';')
EVA_input = pd.read_table(datapath+'\\Input_EVA.csv',header=0,delimiter=';')
KA_input = pd.read_table(datapath+'\\Input_KA.csv',header=0,delimiter=';')
SA_fatkernels_input = pd.read_table(datapath+'\\Input_SA_model2.
 ↪csv',header=0,delimiter=';')
```

7.1 Generate Table B2

[6]:
```
SA_input.iloc[:,0:10].to_csv('Table_B2.csv', sep=';', index=False)
```

8 Evaluation of parameter uncertainty

8.1 Local Sensitivity Analysis (LSA)

As a first step, we applied a species-specific local sensitivity analysis (LSA). We followed the approach by Downing et al. (1985) and applied a 5-points one-at-a-time sensitivity analysis, where one parameter is adjusted to its minimum, mean, maximum, 25^{th} and 75^{th} percentile values, while the others are kept at their default values (`Table S1`). We quantified the response of the model to each parameter in terms of directionality, linearity and magnitude by four summary statistics: the Sensitivity Index (SI), two Importance Indexes (II1 and II2), and the Linearity Index (LI) (Downing et al. (1985); Hamby, 1995):

$$SI = \frac{\Delta y}{\Delta x_i}$$

where Δy is the average of the differences of the output values for the 5-points, and Δx_i is the corresponding 25% value change for each input parameter x_i.

The first Importance Index $II_{1,i}$ of parameter i is the product between the uncertainty of parameter i (UI_i) and its effect on the model ouput SI_i:

$$II_{1,i} = UI_i \times SI_i$$

$$UI = \frac{x_{max} - x_{min}}{x_{max}}$$

where the uncertainty UI_i of parameter i is its scaled range from its minimum to its maximum.

The second Importance Index $II_{2,i}$ of parameter i is calculated as the percentage difference of the output y (i.e. migration rate) when varying the input parameter x_i from its minimum to its maximum:

$$II_{2,i} = \frac{y_{max} - y_{min}}{y_{max}}$$

The Linearity Index LI_i indicates whether the relationship between each input parameter and the model output approach linearity:

$$LI_i = \frac{y_{max} - y_{min}}{\sqrt{s^2}}$$

where s^2 is the sample variance of the model output and an exact linear relationship between model output and parameter corresponds to $LI_i = 1$.

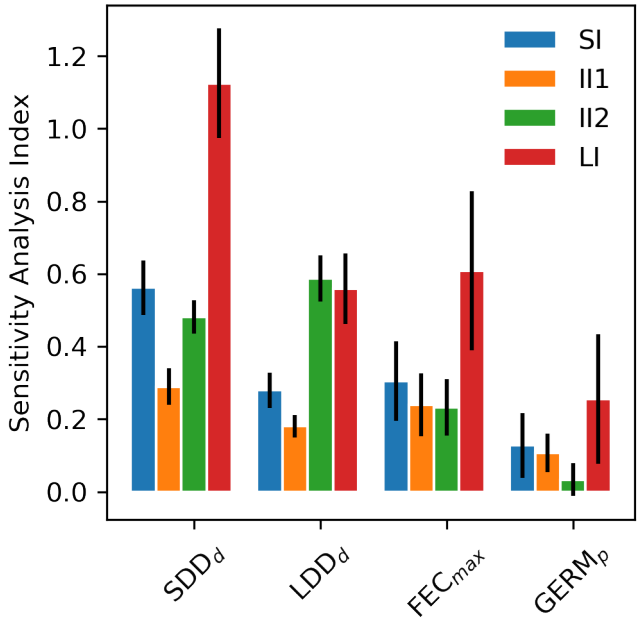
[7]:
```
# Calculate parameter uncertainty (UI)
SA_input['UI'] = SA_input['Max'] - SA_input['Min']

# Print UI statistics
print('Parameter uncertainty mean:')
```

```
print(SA_input[['Parameters','UI']].groupby('Parameters').mean())
print('Parameter uncertainty standard deviation:')
print(SA_input[['Parameters','UI']].groupby('Parameters').std())

# Calculate Sensitivity Index (SI)
SA_input['SI'] =  (SA_input['MigRate_Max'] - SA_input['MigRate_Min']) /␣
 ↪(SA_input['Max'] - SA_input['Min'])

# Calculate Importance Index (II1)
# product of sensitivity and normalized [0-1] parameter uncertainty
UI_norm = (SA_input['Max'] - SA_input['Min']) / SA_input['Max']
SA_input['II1'] = UI_norm * SA_input['SI']

# Calculate Importance Index (II2)
# output percentage difference
SA_input['II2'] = (SA_input['MigRate_Max'] - SA_input['MigRate_Min']) /␣
 ↪SA_input['MigRate_Max']

# Calculate the Linearity Index (LI)
# ratio between output range and ca. standard deviation of parameter range
SA_input['LI'] = (SA_input['MigRate_Max'] - SA_input['MigRate_Min']) /␣
 ↪((SA_input['Max'] - SA_input['Min']) / 2)

# Calculate summary statistic
SA_input['SA_total'] = SA_input['SI'] + SA_input['II1'] + SA_input['II2'] +␣
 ↪SA_input['LI']

# Export Supplement Table 1
Supplement_Table_S1 =␣
 ↪SA_input[['Species','Parameters','UI','SI','II1','II2','LI']]

# Save table
Supplement_Table_S1.to_csv('Table_S1.csv', sep=';', index=False)

# Summary statistics
Supplement_Table_S1.groupby('Parameters').describe()

# Plot Figure 1
SI_df = SA_input.groupby('Parameters').agg(['mean',f_ste]).T.loc['SI']
II1_df = SA_input.groupby('Parameters').agg(['mean',f_ste]).T.loc['II1']
II2_df = SA_input.groupby('Parameters').agg(['mean',f_ste]).T.loc['II2']
LI_df = SA_input.groupby('Parameters').agg(['mean',f_ste]).T.loc['LI']
SA_tot_df = SA_input.groupby('Parameters').agg(['mean',f_ste]).T.loc['SA_total']

SA_toPlot = pd.concat([SI_df, II1_df, II2_df, LI_df, SA_tot_df]).T
SA_toPlot.columns =␣
 ↪['SI_mean','SI_ste','II1_mean','II1_ste','II2_mean','II2_ste','LI_mean','LI_ste',
```

```
                    'SA_tot_mean','SA_tot_ste']
SA_toPlot = SA_toPlot.sort_values('SA_tot_mean',ascending=False)
labels = SA_toPlot.index
print('\nRanked parameters across species: '+str(labels.values.tolist()))
labels = ['SDD$_d$','LDD$_d$','FEC$_{max}$','GERM$_p$']

x = np.arange(len(labels)) # the label locations
width = 0.2  # the width of the bars

f, ax = plt.subplots()
f.set_figheight(3.54)
f.set_figwidth(3.54)
f.dpi = 300

rects1 = ax.bar(x - 2*width, SA_toPlot['SI_mean'], width,␣
 ↪yerr=SA_toPlot['SI_ste'], label='SI', edgecolor="white")
rects2 = ax.bar(x - width, SA_toPlot['II1_mean'], width,␣
 ↪yerr=SA_toPlot['II1_ste'], label='II1', edgecolor="white")
rects3 = ax.bar(x, SA_toPlot['II2_mean'], width, yerr=SA_toPlot['II2_ste'],␣
 ↪label='II2', edgecolor="white")
rects4 = ax.bar(x + width, SA_toPlot['LI_mean'], width,␣
 ↪yerr=SA_toPlot['LI_ste'], label='LI', edgecolor="white")

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Sensitivity Analysis Index')
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend(frameon=False)
plt.xticks(rotation=45)

# Export figure
plt.savefig('Figure_1.png', bbox_inches='tight', dpi=300)
```

```
Parameter uncertainty mean:
                    UI
Parameters
FEC_max     3470.823529
GERM_p        27.294118
LDD_d        775.352941
SDD_d         61.521765
Parameter uncertainty standard deviation:
                    UI
Parameters
FEC_max     9339.379083
GERM_p        19.639771
LDD_d       1627.992166
SDD_d         34.656158
```

```
Ranked parameters across species: ['SDD_d', 'LDD_d', 'FEC_max', 'GERM_p']
```



8.1.1 Species-specific parameter ranking

[8]:
```
# Calculate a summary index as the summation of all others
SA_input['SA_total'] = SA_input['SI'] + SA_input['II1'] + SA_input['II2'] +␣
 ↪SA_input['LI']

species = SA_input.Species.unique()
for sp in species :
    sub_sp = SA_input[SA_input['Species']==sp]
    sub_sp = sub_sp.sort_values('SA_total',ascending=False)
    labels = sub_sp.Parameters
    print('Species: '+str(sp)+' --> Ranks 1: '+labels.values[0]+
          ' | 2: '+labels.values[1]+' | 2: '+labels.values[2]+
          ' | 3: '+labels.values[3])
```

```
Species: Abi_alb --> Ranks 1: SDD_d | 2: FEC_max | 2: LDD_d | 3: GERM_p
Species: Bet_pen --> Ranks 1: GERM_p | 2: SDD_d | 2: LDD_d | 3: FEC_max
Species: Bet_pub --> Ranks 1: GERM_p | 2: SDD_d | 2: LDD_d | 3: FEC_max
Species: Car_bet --> Ranks 1: SDD_d | 2: LDD_d | 2: GERM_p | 3: FEC_max
Species: Cor_ave --> Ranks 1: FEC_max | 2: SDD_d | 2: LDD_d | 3: GERM_p
Species: Fag_syl --> Ranks 1: SDD_d | 2: FEC_max | 2: LDD_d | 3: GERM_p
Species: Fra_exc --> Ranks 1: SDD_d | 2: LDD_d | 2: GERM_p | 3: FEC_max
Species: Pic_abi --> Ranks 1: LDD_d | 2: SDD_d | 2: GERM_p | 3: FEC_max
```

```
Species: Pic_sit --> Ranks 1: SDD_d | 2: LDD_d | 2: GERM_p | 3: FEC_max
Species: Pin_hal --> Ranks 1: LDD_d | 2: FEC_max | 2: GERM_p | 3: SDD_d
Species: Pin_syl --> Ranks 1: FEC_max | 2: SDD_d | 2: LDD_d | 3: GERM_p
Species: Que_coc --> Ranks 1: LDD_d | 2: SDD_d | 2: FEC_max | 3: GERM_p
Species: Que_ile --> Ranks 1: LDD_d | 2: FEC_max | 2: SDD_d | 3: GERM_p
Species: Que_pub --> Ranks 1: FEC_max | 2: SDD_d | 2: GERM_p | 3: LDD_d
Species: Que_rob --> Ranks 1: LDD_d | 2: FEC_max | 2: SDD_d | 3: GERM_p
Species: Til_cor --> Ranks 1: SDD_d | 2: LDD_d | 2: GERM_p | 3: FEC_max
Species: Ulm_gla --> Ranks 1: SDD_d | 2: LDD_d | 2: GERM_p | 3: FEC_max
```

8.1.2 Species-specific linear regression statistics

Additionally, we conducted species-specific linear regression analyses of the type $y \sim x_i$, such that every change of one parameter x_i unit translates to a change of migration rate (y) given by the slope coefficient of the regression (i.e. a slope value of 1 should correspond to $LI_i \sim 1$).

[9]:
```
# Calculate linear regression statistics
def lm_slope(group):
    x = group['value']
    y = group['MigRates']
    slope, intercept, r, p, se = stats.linregress(x, y)
    return slope

def lm_intercept(group):
    x = group['value']
    y = group['MigRates']
    slope, intercept, r, p, se = stats.linregress(x, y)
    return intercept

def lm_r(group):
    x = group['value']
    y = group['MigRates']
    slope, intercept, r, p, se = stats.linregress(x, y)
    return r

def lm_pvalue(group):
    x = group['value']
    y = group['MigRates']
    slope, intercept, r, p, se = stats.linregress(x, y)
    return p

# Format table to longer format (from 5-points)
long_df = SA_input[['Species','Parameters','Min','25th','Mean','75th','Max']]
long_df = long_df.melt(id_vars=['Species','Parameters'])
long_df.drop('variable', axis='columns', inplace=True)
mig_rates =␣
 ↪SA_input[['Species','Parameters','MigRate_Min','MigRate_25th','MigRate_Mean','MigRate_75th','
mig_rates = mig_rates.melt(id_vars=['Species','Parameters'])
```

```
long_df['MigRates'] = mig_rates['value']

# Calculate slopes
lm_stats_df = long_df.groupby(['Parameters','Species'])[['value','MigRates']].
 ↪apply(lm_slope)
lm_stats_df = lm_stats_df.reset_index()

intercepts = long_df.groupby(['Parameters','Species'])[['value','MigRates']].
 ↪apply(lm_intercept)
intercepts = intercepts.reset_index()

r_coeffs = long_df.groupby(['Parameters','Species'])[['value','MigRates']].
 ↪apply(lm_r)
r_coeffs = r_coeffs.reset_index()

pvalues = long_df.groupby(['Parameters','Species'])[['value','MigRates']].
 ↪apply(lm_pvalue)
pvalues = pvalues.reset_index()

lm_stats_df.columns = ['Parameters','Species','slope']
lm_stats_df['intercept'] = intercepts.iloc[:,2]
lm_stats_df['r2'] = r_coeffs.iloc[:,2]**2
lm_stats_df['pvalue'] = pvalues.iloc[:,2]
lm_stats_df.groupby('Parameters').mean()

# Print results
print('Linear regression summary:')
lm_stats_df.groupby('Parameters').describe()
```

```
Linear regression summary:
```

[9]:

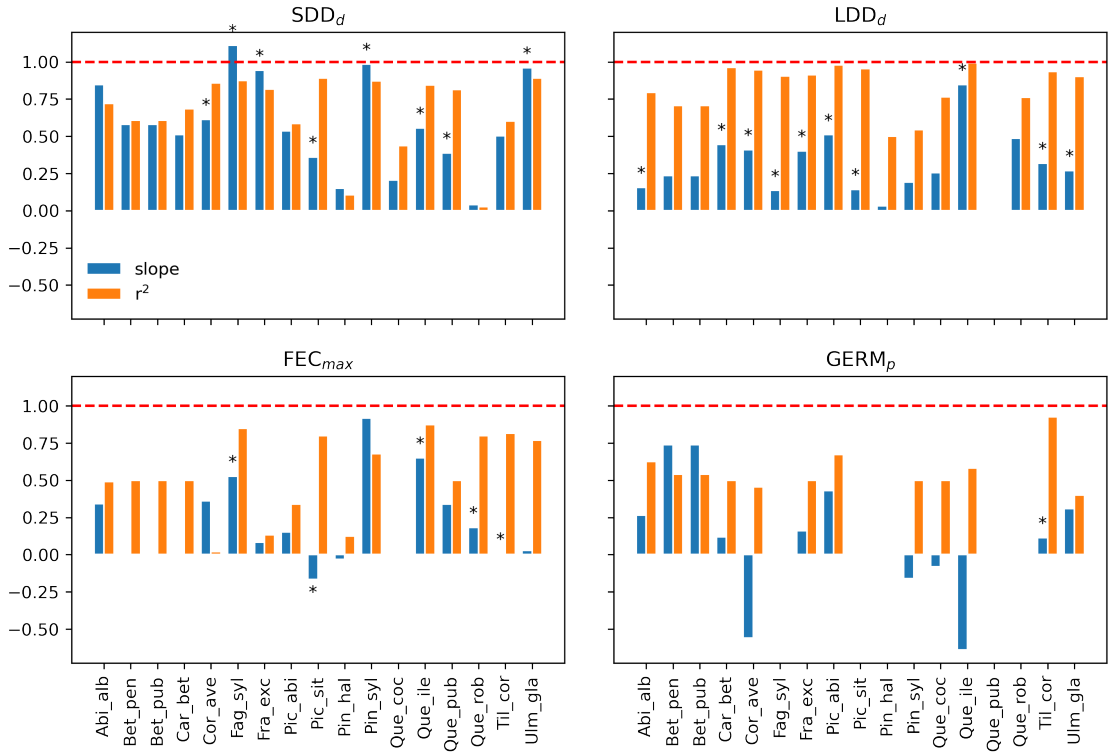
	slope							\
	count	mean	std	min	25%	50%	75%	
Parameters								
FEC_max	17.0	0.199839	0.287580	-0.168000	0.000085	0.083333	0.341935	
GERM_p	17.0	0.084665	0.365090	-0.640000	0.000000	0.000000	0.266667	
LDD_d	17.0	0.298626	0.206200	0.000000	0.156023	0.256000	0.409231	
SDD_d	17.0	0.580878	0.306906	0.041365	0.388151	0.555311	0.848000	

	intercept			...	r2		pvalue	\
	max	count	mean	...	75%	max	count	
Parameters				...				
FEC_max	0.918919	17.0	39.734360	...	0.798913	0.874379	17.0	
GERM_p	0.740000	17.0	47.836903	...	0.541535	0.925926	17.0	
LDD_d	0.848000	17.0	-14.141041	...	0.945202	0.991966	17.0	
SDD_d	1.111644	17.0	-5.000152	...	0.858112	0.892193	17.0	

```
                                                                        \
                mean       std       min       25%       50%       75%
Parameters
FEC_max     0.260878  0.294861  0.019661  0.040866  0.181690  0.303056
GERM_p      0.402671  0.400843  0.008754  0.156318  0.181690  1.000000
LDD_d       0.099438  0.238192  0.000306  0.005536  0.013064  0.074067
SDD_d       0.143831  0.216615  0.015539  0.023730  0.068449  0.122752


                max
Parameters
FEC_max     1.000000
GERM_p      1.000000
LDD_d       1.000000
SDD_d       0.789803

[4 rows x 32 columns]
```

[10]:
```
# Plot relationships migration rate vs. parameter values at the species level
f, axes = plt.subplots(2, 2, sharex=True, sharey=True)
(ax1, ax2), (ax3, ax4) = axes
f.set_figheight(7.25*1.5/1.6)
f.set_figwidth(7.25*1.5)
f.dpi = 300

headers = ['SDD_d','LDD_d','FEC_max','GERM_p']

for header,ax in zip(headers,axes.flatten()):
    lm_sub = lm_stats_df[lm_stats_df['Parameters']==header]

    labels = lm_sub.Species.unique()
    x = np.arange(len(labels))
    width = 0.375

    rects1 = ax.bar(x - width/2, lm_sub.slope, width, label='slope',␣
 ↪edgecolor="white")
    rects2 = ax.bar(x + width/2, lm_sub.r2, width, label='r$^2$',␣
 ↪edgecolor="white")

    # Add asterisks for significance
    sig_label = ['*' if (e < 0.05) else ('**' if (e < 0.01) else '') for e in␣
 ↪lm_sub.pvalue]
    ax.bar_label(rects1, labels=sig_label, padding=3)
    ax.axhline(y=1, color='r', linestyle='--')

    # Add headers
    if header == 'SDD_d' :
```

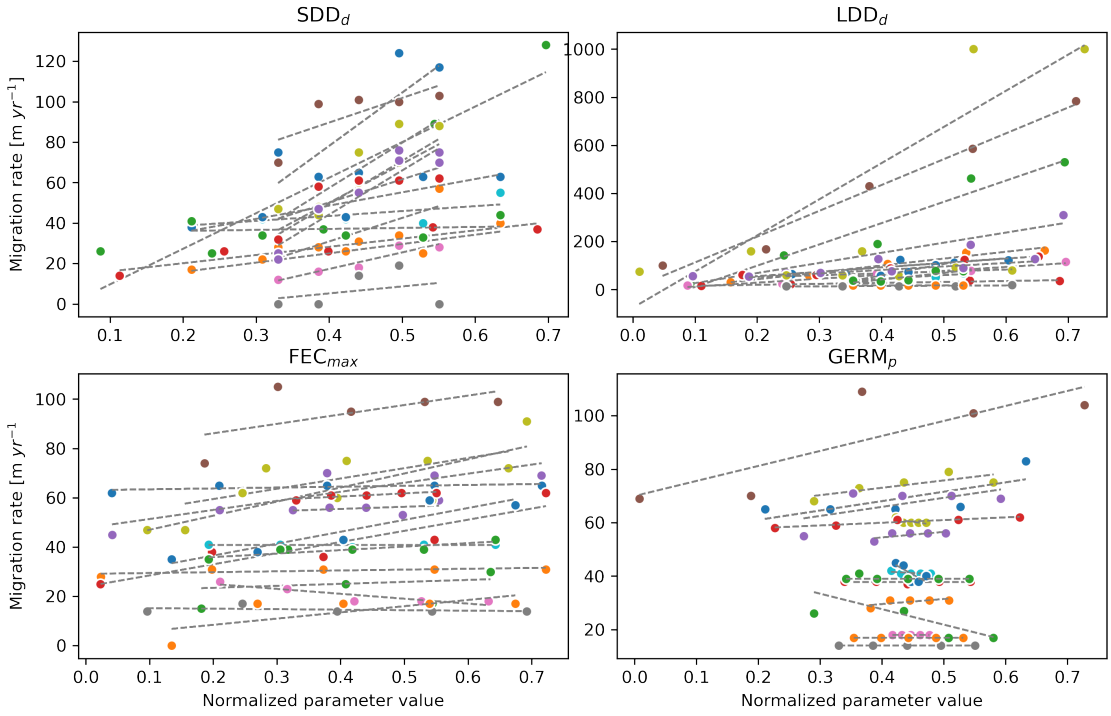
```
        ax.set_title('SDD$_d$')
        ax.legend(frameon=False)
    if header == 'LDD_d' :
        ax.set_title('LDD$_d$')
    if header == 'FEC_max' :
        ax.set_title('FEC$_{max}$')
        ax.set_xticks(x)
        ax.set_xticklabels(labels)
        ax.set_xticklabels(labels, rotation=90)
    if header == 'GERM_p' :
        ax.set_title('GERM$_p$')
        ax.set_xticks(x)
        ax.set_xticklabels(labels)
        ax.set_xticklabels(labels, rotation=90)
f.subplots_adjust(wspace=0.1)

plt.savefig('Figure_S2.png', bbox_inches='tight', dpi=300)
```



8.1.3 Species-specific relationship of migration rate vs. parameter values

[11]:
```
def scatter_plot_par(par_df, sub_plot):
    species = par_df['Species']
    for sp in species :
        # Normalize parameter values
        sp_sub = par_sub[par_sub['Species']==sp]
        norm_value = preprocessing.normalize([sp_sub['value']])
        x, y = norm_value[0], sp_sub['MigRates']
        sub_plot.plot(x, y, marker="o", linestyle="", markeredgecolor='white')
        slope, intercept = np.polyfit(x, y, 1)
        sub_plot.plot(x, slope*x+intercept, '--', color='grey', linewidth=1)

        # Add headers
        if header == 'SDD_d' :
            sub_plot.set_title('SDD$_d$')
            sub_plot.set_ylabel('Migration rate [m $yr^{-1}$]')
        if header == 'LDD_d' :
            sub_plot.set_title('LDD$_d$')
        if header == 'FEC_max' :
            sub_plot.set_title('FEC$_{max}$')
            sub_plot.set_xlabel('Normalized parameter value')
            sub_plot.set_ylabel('Migration rate [m $yr^{-1}$')
        if header == 'GERM_p' :
            sub_plot.set_title('GERM$_p$')
            sub_plot.set_xlabel('Normalized parameter value')

f, ax = plt.subplots()
f.set_figheight(3.54*2.)
f.set_figwidth(3.54*2.5)
f.set_figheight(7.25*1.5/1.6)
f.set_figwidth(7.25*1.5)
f.dpi = 300

# Place the plots in the plane
plot1 = plt.subplot2grid((2, 2), (0, 0))
plot2 = plt.subplot2grid((2, 2), (0, 1))
plot3 = plt.subplot2grid((2, 2), (1, 0))
plot4 = plt.subplot2grid((2, 2), (1, 1))

headers = ['SDD_d','LDD_d','FEC_max','GERM_p']

# SDD
par_sub = long_df[long_df['Parameters']==headers[0]]
header = headers[0]
scatter_plot_par(par_sub, plot1)
```

```
# LDD
par_sub = long_df[long_df['Parameters']==headers[1]]
header = headers[1]
scatter_plot_par(par_sub, plot2)

# FEC_max
par_sub = long_df[long_df['Parameters']==headers[2]]
header = headers[2]
scatter_plot_par(par_sub, plot3)

# GERM_p
par_sub = long_df[long_df['Parameters']==headers[3]]
header = headers[3]
scatter_plot_par(par_sub, plot4)

f.subplots_adjust(wspace=0.1)

plt.savefig('Figure_S3.png', bbox_inches='tight', dpi=300)
```



8.2 Extreme Value Analysis (EVA)

We fixed all influential parameters at their minimum (*all_MIN*) and maximum (*all_MAX*) values and calculated the corresponding errors with respect to observational data for each species. Species-specific performance was calculated with residuals (*res*) with respect to the maximum of

observational values:

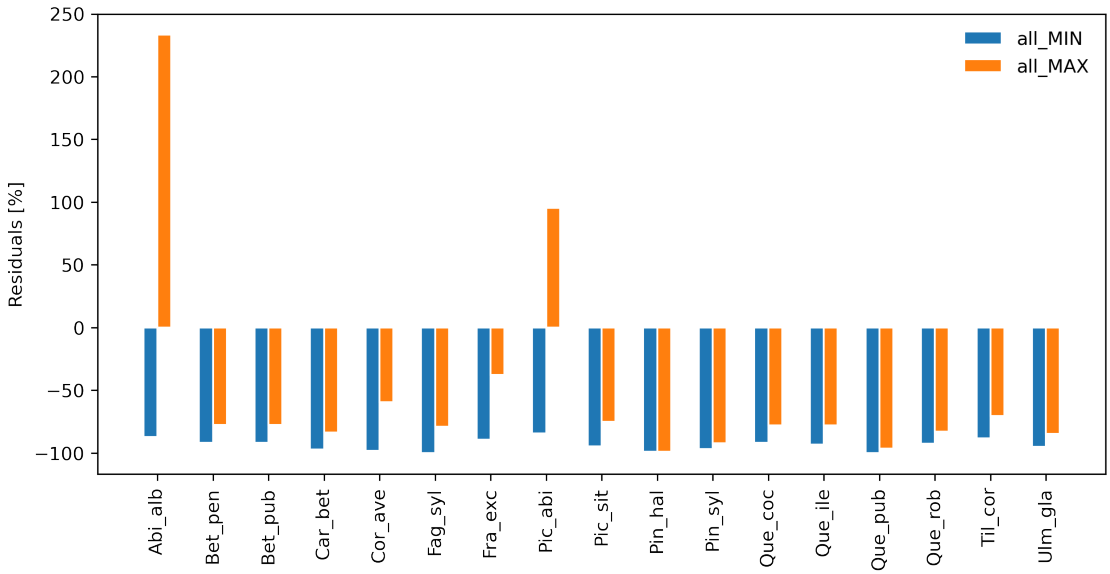
$$res = sim - obs_{max}$$

where obs_{max} indicate the maximum values of migration rates estimates (Table 1).

[12]:
```
# Function to calculate species-specific residuals of observations vs. simulated migration values
def f_residuals(df_input) :

    # Calculate species-specific residuals
    Species = df_input['Species'].unique()
    Parameter_combo = df_input['Parameter_combo'].unique()
    if (Parameter_combo[2] == 'all_MAX_opt'):
        Parameter_combo = ['all_MAX','all_MIN']

    # Initialize dataframe to store results
    residuals_df = pd.DataFrame()

    # Loop through species
    for sp in Species :
        df_sp = df_input[df_input['Species']==sp]
        column_names = df_sp.columns.values.tolist()

        # Get the maximum observed value
        max_obs = float(df_sp['OBS_MAX'].unique())
        min_obs = float(df_sp['OBS_MIN'].unique())

        # Loop through parameters
        for par in Parameter_combo :
            par_df = df_sp.loc[df_sp['Parameter_combo']==par]

            # Get simulation value
            sim = par_df['SIM']
            check_na = pd.isna(sim)
            if (len(sim) != 1):
                sim = sim.reset_index()
                sim = sim['SIM'][0]
            if (check_na.any()) :
                sim = sim
            else :
                sim = int(sim)

            # Get residual and percentage
            res = float((sim - max_obs))
            res_perc = float(100/max_obs * res)
            res_min = float((sim - min_obs))
```

```
            res_min_perc = float(100/min_obs * res_min)
            residuals_sp = pd.
 ↪DataFrame([[sp,par,sim,max_obs,res,res_perc,res_min,res_min_perc]],
                                                 ␣
 ↪columns=['Species','Parameter_combo','sim','obs','residuals',
                                                          ␣
 ↪'residuals_perc','min_residuals','min_residuals_perc'])
            residuals_df = pd.concat([residuals_df,residuals_sp])
    return residuals_df

# Function to make a violinplot of species-specific residuals
# input dataframe should be formatted with f_rmse_residuals()
def make_plot(residuals_df) :

    # Format dataframe to plot
    res_toPlot = residuals_df[["Species","Parameter_combo","residuals_perc"]]
    res_toPlot = res_toPlot.
 ↪pivot(index="Species",columns="Parameter_combo",values="residuals_perc")

    # Set dimensions
    f, ax = plt.subplots()
    f.set_figheight(4.5)
    f.set_figwidth(8.5)
    f.dpi = 300

    labels = residuals_df.Species.unique()
    x = np.arange(len(labels))
    width = 0.25
    if (len(residuals_df['Parameter_combo'].unique())==2):
        rects1 = ax.bar(x - width/2, res_toPlot['all_MIN'], width,␣
 ↪label='all_MIN', edgecolor='white')
        rects2 = ax.bar(x + width/2, res_toPlot['all_MAX'], width,␣
 ↪label='all_MAX', edgecolor='white')
    else:
        width = 0.15
        rects1 = ax.bar(x - width*2, res_toPlot['ExpPow'], width,␣
 ↪label='ExpPow', edgecolor='white')
        rects2 = ax.bar(x - width, res_toPlot['Weibull'], width,␣
 ↪label='Weibull', edgecolor='white')
        rects3 = ax.bar(x, res_toPlot['twoDt'], width, label='twoDt',␣
 ↪edgecolor='white')
        rects4 = ax.bar(x + width, res_toPlot['Logistic'], width,␣
 ↪label='Logistic', edgecolor='white')
        rects5 = ax.bar(x + width*2, res_toPlot['LogSec'], width,␣
 ↪label='LogSec', edgecolor='white')
```

```
    # Add some text for labels, title and custom x-axis tick labels, etc.
    ax.set_ylabel('Residuals [%]')
    ax.set_xticks(x)
    ax.set_xticklabels(labels)
    ax.legend(frameon=False)
    plt.xticks(rotation=90)
    fig_out = f.get_figure()
    plt.tight_layout()
    return fig_out

# Make dataframe with RMSE and species-specific residuals (mean and standard␣
 ↪deviation)
residuals_df = f_residuals(EVA_input)

# Export to CSV
residuals_df.to_csv('Table_S2.csv', sep=';', index=False)

# Make figure
fig_out = make_plot(residuals_df)

# Export figure
fig_out.savefig('Figure_2.png', dpi=300, format='png')
```



8.2.1 Species-specific all_MIN vs. all_MAX

[13]:
```
print("all_MIN vs. all_MAX performance:")

# Check significance for the difference all_MAX vs. all_MIN per species
# Mann-Whitney U Test
species = EVA_input.Species.unique().tolist()
best_EVA_df = pd.DataFrame()

# Store p-values for Bonferroni corrections
p_values = []
for sp in species :
    best_EVA_sp = pd.DataFrame()
    best_EVA_sp['Species'] = sp
    sp_sub = residuals_df[residuals_df['Species']==sp]
    all_min = sp_sub[sp_sub['Parameter_combo']=='all_MIN']
    all_max = sp_sub[sp_sub['Parameter_combo']=='all_MAX']

    # Get p-values from Mann-Whitney U Test
    all_min_residuals = range(int(all_min.residuals),int(all_min.min_residuals))
    all_max_residuals = range(int(all_max.residuals),int(all_max.min_residuals))
    _, p = mannwhitneyu(all_min_residuals, all_max_residuals)
    p_values.append(p)

# Apply Bonferroni corrections for multiple comparison
p_adjusted = multipletests(p_values, method='bonferroni')[1]

# all_MIN vs. all_MAX based on corrected p-values
pos = 0
for sp in species :
    best_EVA_sp = pd.DataFrame()
    best_EVA_sp['Species'] = sp
    sp_sub = residuals_df[residuals_df['Species']==sp]
    all_min = sp_sub[sp_sub['Parameter_combo']=='all_MIN']
    all_max = sp_sub[sp_sub['Parameter_combo']=='all_MAX']
    max_opt = EVA_input.loc[(EVA_input['Parameter_combo']=='all_MAX_opt') &␣
 ↪(EVA_input['Species']==sp)]

    # Compare all_MIN and all_MAX
    all_max_res = abs(np.mean(all_max_residuals))
    all_min_res = abs(np.mean(all_min_residuals))
    if (all_max_res <= all_min_res) :
        best_MAX_sp = all_max
    else :
        best_MAX_sp = all_min

    # Check significance with low threshold
```

```
    p = p_adjusted[pos]
    pos = pos+1
    alpha = 1e-3
    if p > alpha:
        print(sp+ ' --> p-value: '+str(round(p,5))+' --> NOT-significant␣
↪difference')
    else:
        print(sp+ ' --> p-value: '+str(round(p,5))+' --> significant difference')

    # Select optimized set of parameters when possible
    if (len(max_opt) > 0):
        max_opt = max_opt.copy()
        max_opt['res_MAX'] = abs(max_opt.OBS_MAX - max_opt.SIM)
        best_MAX_sp = max_opt.loc[max_opt.res_MAX == min(max_opt.res_MAX)]

    # Select best parameter setting and corresponding simulated values
    best_EVA_sp['Parameter_combo'] = best_MAX_sp.Parameter_combo.unique()
    if (len(max_opt) > 0):
        best_EVA_sp['sim'] = best_MAX_sp.SIM.unique()
        best_EVA_sp['obs_MAX'] = best_MAX_sp.OBS_MAX.unique()
    else :
        best_EVA_sp['sim'] = best_MAX_sp.sim
        best_EVA_sp['obs_MAX'] = max(best_MAX_sp.obs)
    best_EVA_df = pd.concat([best_EVA_df,best_EVA_sp])
best_EVA_df['Species'] = species
```

```
all_MIN vs. all_MAX performance:
Abi_alb --> p-value: 0.0 --> significant difference
Bet_pen --> p-value: 0.0 --> significant difference
Bet_pub --> p-value: 0.0 --> significant difference
Car_bet --> p-value: 0.0 --> significant difference
Cor_ave --> p-value: 0.0 --> significant difference
Fag_syl --> p-value: 0.0 --> significant difference
Fra_exc --> p-value: 0.0 --> significant difference
Pic_abi --> p-value: 0.0 --> significant difference
Pic_sit --> p-value: 0.0 --> significant difference
Pin_hal --> p-value: 1.0 --> NOT-significant difference
Pin_syl --> p-value: 0.0 --> significant difference
Que_coc --> p-value: 0.0 --> significant difference
Que_ile --> p-value: 0.0 --> significant difference
Que_pub --> p-value: 0.04141 --> NOT-significant difference
Que_rob --> p-value: 0.0 --> significant difference
Til_cor --> p-value: 0.0 --> significant difference
Ulm_gla --> p-value: 0.0 --> significant difference
```

8.2.2 Overall error of all_MIN and all_MAX

The root-mean-square-error (RMSE) was used to quantify the model performance across species:

$$RMSE = \frac{100}{\overline{obs_{max}}} \times \sqrt{\frac{\sum_{i=1}^{n}(sim_i - obs_{max,i})^2}{n}}$$

where n is the number of species, obs_{max} is the maximum value in the confidence range of observational values, and $\overline{obs_{max}}$ is the average across species.

[14]:
```
allMIN_df = EVA_input[EVA_input['Parameter_combo']=='all_MIN']
allMIN_RMSE_perc = rmse_perc(allMIN_df.SIM,allMIN_df.OBS_MAX,allMIN_df.OBS_MAX.
↪mean())

allMAX_df = EVA_input[EVA_input['Parameter_combo']=='all_MAX']
allMAX_RMSE_perc = rmse_perc(allMAX_df.SIM,allMAX_df.OBS_MAX,allMAX_df.OBS_MAX.
↪mean())

print("Observation values are the MAXIMUM values")
print("Root Mean Square Error for all_MIN: "+str(round(allMIN_RMSE_perc,2)))
print("Root Mean Square Error for all_MAX: "+str(round(allMAX_RMSE_perc,2)))
```

```
Observation values are the MAXIMUM values
Root Mean Square Error for all_MIN: 109.0
Root Mean Square Error for all_MAX: 96.79
```

8.2.3 Species-specific over/under/good estimations

Good estimations correspond to residuals within 15% from the observed migration speed (maximum value), over- and under-estimations correspond to positive and negative residuals outside of the 15% range, respectively.

Species-specific performance was calculated as residuals (res) with respect to the maximum value of observed migration rates, obs_{max} (where obs_{max} corresponds to the maxium value of `Migration rates`, Table 1):

$$res = \frac{100}{obs_{max}} \times (sim - obs_{max})$$

[15]:
```
def estimatesGoodness(df, good_margin=25.):
    target = df.residuals_perc
    par_val = "Parameter_combo"

    good_est = df.loc[(target < good_margin) & (target > -good_margin)]
    bad_est = df.loc[(target > good_margin) | (target < -good_margin)]
    over_est = bad_est.loc[bad_est.residuals_perc > 0.]
    under_est = bad_est.loc[bad_est.residuals_perc < 0.]

    good_sp = good_est.Species.unique()
    for sp in good_sp :
        sp_sub = good_est[good_est['Species'] == sp]
        print('Good estimates: '+sp+' for '+str(sp_sub[par_val].values.tolist()))
```

```
    over_sp = over_est.Species.unique()
    for sp in over_sp :
        sp_sub = over_est[over_est['Species'] == sp]
        print('Over-estimations: '+sp+' for '+str(sp_sub[par_val].values.
↪tolist()))

    under_sp = under_est.Species.unique()
    for sp in under_sp :
        sp_sub = under_est[under_est['Species'] == sp]
        print('Under-estimations: '+sp+' for '+str(sp_sub[par_val].values.
↪tolist()))
```

[16]:
```
print("Observation values are the MAXIMUM values")
estimatesGoodness(residuals_df, good_margin=15.)
```

```
Observation values are the MAXIMUM values
Over-estimations: Abi_alb for ['all_MAX']
Over-estimations: Pic_abi for ['all_MAX']
Under-estimations: Abi_alb for ['all_MIN']
Under-estimations: Bet_pen for ['all_MAX', 'all_MIN']
Under-estimations: Bet_pub for ['all_MAX', 'all_MIN']
Under-estimations: Car_bet for ['all_MAX', 'all_MIN']
Under-estimations: Cor_ave for ['all_MAX', 'all_MIN']
Under-estimations: Fag_syl for ['all_MAX', 'all_MIN']
Under-estimations: Fra_exc for ['all_MAX', 'all_MIN']
Under-estimations: Pic_abi for ['all_MIN']
Under-estimations: Pic_sit for ['all_MAX', 'all_MIN']
Under-estimations: Pin_hal for ['all_MAX', 'all_MIN']
Under-estimations: Pin_syl for ['all_MAX', 'all_MIN']
Under-estimations: Que_coc for ['all_MAX', 'all_MIN']
Under-estimations: Que_ile for ['all_MAX', 'all_MIN']
Under-estimations: Que_pub for ['all_MAX', 'all_MIN']
Under-estimations: Que_rob for ['all_MAX', 'all_MIN']
Under-estimations: Til_cor for ['all_MAX', 'all_MIN']
Under-estimations: Ulm_gla for ['all_MAX', 'all_MIN']
```

8.2.4 Best-performing EVA setting per species

[17]:
```
# Calculate RMSE%
best_EVA_df['rmse_MAX'] = best_EVA_df.apply(lambda row: rmse_perc(row['sim'],␣
↪row['obs_MAX'], row['obs_MAX']), axis = 1)

# Add migration parameters for all_MAX settings
best_EVA_df['FEC_max'] = EVA_input.loc[EVA_input['Parameter_combo'] ==␣
↪'all_MAX','FEC_max'].to_list()
```

```
best_EVA_df['GERM_p'] = EVA_input.loc[EVA_input['Parameter_combo'] ==␣
 ↪'all_MAX','GERM_p'].to_list()
best_EVA_df['SDD_d'] = EVA_input.loc[EVA_input['Parameter_combo'] ==␣
 ↪'all_MAX','SDD_d'].to_list()
best_EVA_df['LDD_d'] = EVA_input.loc[EVA_input['Parameter_combo'] ==␣
 ↪'all_MAX','LDD_d'].to_list()

# Add migration parameters for all_MAX_opt species
sp_opt = best_EVA_df.loc[best_EVA_df['Parameter_combo'] ==␣
 ↪'all_MAX_opt','Species'].to_list()
for sp in sp_opt:
    best_sim = best_EVA_df.loc[best_EVA_df['Species'] == sp].sim
    best_pars = EVA_input.loc[(EVA_input['Parameter_combo'] == 'all_MAX_opt') &␣
 ↪(EVA_input['Species'] == sp)
                              & (EVA_input['SIM'] == int(best_sim))]
    best_EVA_df.loc[best_EVA_df['Species'] == sp,'FEC_max'] = int(best_pars.
 ↪FEC_max)
    best_EVA_df.loc[best_EVA_df['Species'] == sp,'GERM_p'] = int(best_pars.
 ↪GERM_p)
    best_EVA_df.loc[best_EVA_df['Species'] == sp,'SDD_d'] = int(best_pars.SDD_d)
    best_EVA_df.loc[best_EVA_df['Species'] == sp,'LDD_d'] = int(best_pars.LDD_d)
best_EVA_df
```

[17]:

	Species	Parameter_combo	sim	obs_MAX	rmse_MAX	FEC_max	GERM_p	SDD_d	\
0	Abi_alb	all_MAX_opt	305	300.0	1.666667	81	60	100	
0	Bet_pen	all_MAX	179	800.0	77.625000	30000	30	200	
0	Bet_pub	all_MAX	179	800.0	77.625000	30000	30	200	
0	Car_bet	all_MAX	162	1000.0	83.800000	705	80	100	
0	Cor_ave	all_MAX	607	1500.0	59.533333	12	60	25	
0	Fag_syl	all_MAX	63	300.0	79.000000	62	80	25	
0	Fra_exc	all_MAX	312	500.0	37.600000	50	65	100	
0	Pic_abi	all_MAX_opt	509	500.0	1.800000	163	95	100	
0	Pic_sit	all_MAX	125	500.0	75.000000	75	80	100	
0	Pin_hal	all_MAX	16	1500.0	98.933333	43	60	100	
0	Pin_syl	all_MAX	116	1500.0	92.266667	43	95	100	
0	Que_coc	all_MAX	110	500.0	78.000000	10	75	25	
0	Que_ile	all_MAX	111	500.0	77.800000	50	95	25	
0	Que_pub	all_MAX	17	500.0	96.600000	50	90	25	
0	Que_rob	all_MAX	85	500.0	83.000000	50	95	25	
0	Til_cor	all_MAX	147	500.0	70.600000	720	55	100	
0	Ulm_gla	all_MAX	153	1000.0	84.700000	949	65	100	

	LDD_d
0	710
0	475
0	475

```
0     425
0    1500
0     200
0     725
0     785
0     800
0     250
0     250
0     300
0     300
0     300
0     300
0     374
0     350
```

9 Evaluation of model uncertainty (Dispersal Kernel Analysis)

9.1 Implementation of fat-tailed seed dispersal kernels

We implemented five additional fat-tailed kernels into the dispersal sub-model of LPJ-GM and run simulations with the best set of migration parameters found by EVA, while varying the shape parameter b in a meaningful range for each kernel.

$$k_s = (1 - LDD_p) \times pdf(z, SDD_d) + LDD_p \times pdf(z, LDD_d)$$

The dispersal kernel k_s is a linear combination of two pdfs for short- (SDD) and long-distance dispersal (LDD), where LDD_p is the proportion of LDD (default = 0.99), SDD_d and LDD_d are the average distances for SDD and LDD, respectively. In the default setting of LPJ-GM, the pdfs for both SDD and LDD components are negative exponentials. In the updated setting, the pdfs correspond to the fat-tailed kernels listed in Table 3.

Table 3. Probability density functions (pdf) for the default dispersal kernel in LPJ-GM (negative exponential) and five additional kernels. d = mean distance (in meters); a = scale parameter as a function of dispersal distance (SDD_d and LDD_d for , respectively); b = shape parameter range to search for better representation of LDD events. Range boundaries are defined by the values for which pdf are mathematically significant and the corresponding tail is fat (Nathan et al., 2012). Table adapted from Bullock et al. (2017).

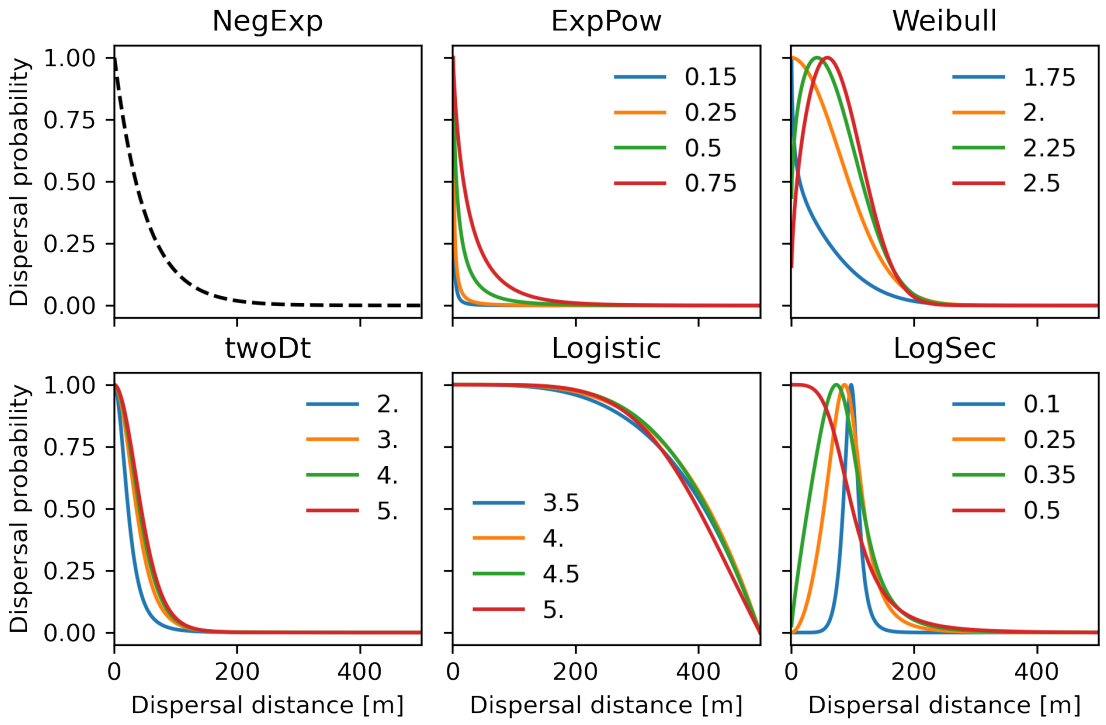
Kernel family	Probability density function	Scale parameter (a)	Shape parameter (b)	Weight of the tail
Negative exponential	$\frac{1}{2\pi a^2}\exp(-\frac{d}{a})$	$\frac{d}{2}$	-	Exponentially bounded
Exponential power	$\frac{b}{2\pi a^2\Gamma(\frac{2}{b})}\exp(-\frac{d^b}{a^b})$	$\frac{\Gamma(\frac{2}{b})}{\Gamma(\frac{3}{b})}$	$0-1$	Fat-tailed (for $b<1$) non-power law
Weibull	$\frac{b}{2\pi a^2}d^{b-2}\exp(-\frac{d^b}{a^b})$	$\frac{b}{\Gamma(\frac{1}{b})}d$	$0-2.5$	Fat-tailed non-power law
twoDt	$\frac{b-1}{\pi a^2}(1+\frac{d^2}{a^2}^{-b})$	$\frac{2}{\pi}\frac{\Gamma(b-1)}{\Gamma(b-\frac{3}{2})}d$	$1-5$	Fat-tailed power law
Logistic	$\frac{b}{2\pi a^2\Gamma(1-\frac{2}{b})}(1+\frac{d^b}{a^b})^{-1}$	$\frac{\Gamma(\frac{2}{b})\Gamma(1-\frac{2}{b})}{\Gamma(\frac{3}{b})\Gamma(1-\frac{3}{b})}d$	$2-5$	Fat-tailed power law
Log-hyperbolic secant	$\frac{1}{\pi^2 bd^2}(\frac{d}{a}^{\frac{1}{b}}+\frac{d}{a}^{-\frac{1}{b}})^{-1}$	d	$0-1$	Fat-tailed power law

[18]:
```
f, axes = plt.subplots(2, 3, sharex=True, sharey=True)
(ax1, ax2, ax3), (ax4, ax5, ax6) = axes
f.set_figheight(7.25/1.6)
f.set_figwidth(7.25)
f.dpi = 300

kernels = ['NegExp','ExpPow','Weibull','twoDt','Logistic','LogSec']
distance_range = np.linspace(1.,500.,500)
b = 1.
sdd = 100.
ldd = 500.
short_range_disp_frac = 0.99

for kernel,ax in zip(kernels,axes.flatten()):
    if kernel=='NegExp':
        def neg_exponential(dist, sdd, ldd) :
            a1 = sdd / 2
            a2 = ldd / 2
            if (a1 <= 0):
                return np.zeros(len(dist))
            dist_prob_sdd = 1 / (2 * np.pi * a1**2) * np.exp(-dist / a1)
            dist_prob_ldd = 1 / (2 * np.pi * a2**2) * np.exp(-dist / a2)
            dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
 ↪short_range_disp_frac)*dist_prob_ldd
            return scaler_0_1(dist_prob)
        x = neg_exponential(distance_range, sdd, ldd)

    if kernel=='ExpPow':
        def exp_power(dist, sdd, ldd, b) :
            a1 = (gamma(2/b) / gamma(3/b)) * sdd
            a2 = (gamma(2/b) / gamma(3/b)) * ldd
            if (a1 == 0. or b <= 0.):
                return np.zeros(len(dist))
            dist_prob_sdd = b / (2 * np.pi * a1**2 * gamma(2/b)) * np.
 ↪exp(-(dist**b / a1**b))
            dist_prob_ldd = b / (2 * np.pi * a2**2 * gamma(2/b)) * np.
 ↪exp(-(dist**b / a2**b))
            dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
 ↪short_range_disp_frac)*dist_prob_ldd
            return scaler_0_1(dist_prob)
        x1 = exp_power(distance_range, sdd, ldd, 0.15)
        x2 = exp_power(distance_range, sdd, ldd, 0.25)
        x3 = exp_power(distance_range, sdd, ldd, 0.5)
        x4 = exp_power(distance_range, sdd, ldd, 0.75)

    if kernel=='Weibull':
        def weibull(dist, sdd, ldd, b) :
```

```
            a1 = (b / gamma(1/b)) * sdd
            a2 = (b / gamma(1/b)) * ldd
            if (a1 <= 0. or b <= 0.):
                return np.zeros(len(dist))
            dist_prob_sdd = b / (2 * np.pi * a1**2) * dist**(b - 2) * np.
↪exp(-dist**b / a1**b)
            dist_prob_ldd = b / (2 * np.pi * a2**2) * dist**(b - 2) * np.
↪exp(-dist**b / a2**b)
            dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
↪short_range_disp_frac)*dist_prob_ldd
            return scaler_0_1(dist_prob)
        x1 = weibull(distance_range, sdd, ldd, 1.75)
        x2 = weibull(distance_range, sdd, ldd, 2.)
        x3 = weibull(distance_range, sdd, ldd, 2.25)
        x4 = weibull(distance_range, sdd, ldd, 2.5)

    if kernel=='twoDt':
        def twoDt(dist, sdd, ldd, b) :
            a1 = (gamma(b - 1) / gamma(b - 3/2)) * 2/np.pi * sdd
            a2 = (gamma(b - 1) / gamma(b - 3/2)) * 2/np.pi * ldd
            if (a1 <= 0. or b <= 1.) :
                return np.zeros(len(dist))
            dist_prob_sdd = (b - 1) / (np.pi * a1**2) * (1 + dist**2 /␣
↪a1**2)**(-b)
            dist_prob_ldd = (b - 1) / (np.pi * a2**2) * (1 + dist**2 /␣
↪a2**2)**(-b)
            dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
↪short_range_disp_frac)*dist_prob_ldd
            return scaler_0_1(dist_prob)
        x1 = twoDt(distance_range, sdd, ldd, 2.)
        x2 = twoDt(distance_range, sdd, ldd, 3.)
        x3 = twoDt(distance_range, sdd, ldd, 4.)
        x4 = twoDt(distance_range, sdd, ldd, 5.)

    if kernel=='Logistic':
        def logistic(dist, sdd, ldd, b) :
            a1 = (gamma(2/b) * gamma(1 - 2/b) / gamma(3/b) * gamma(1 - 3/b)) *␣
↪sdd
            a2 = (gamma(2/b) * gamma(1 - 2/b) / gamma(3/b) * gamma(1 - 3/b)) *␣
↪ldd
            if (a1 <= 0. or b <= 2.) :
                return np.zeros(len(dist))
            dist_prob_sdd = b / (2 * np.pi * a1**2 * gamma(2/b) * gamma(1 - 2/
↪b)) * (1 + dist**b / a1**b)**-1
            dist_prob_ldd = b / (2 * np.pi * a2**2 * gamma(2/b) * gamma(1 - 2/
↪b)) * (1 + dist**b / a2**b)**-1
```

```
            dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
↪short_range_disp_frac)*dist_prob_ldd
            return scaler_0_1(dist_prob)
        x1 = logistic(distance_range, sdd, ldd, 3.5)
        x2 = logistic(distance_range, sdd, ldd, 4.)
        x3 = logistic(distance_range, sdd, ldd, 4.5)
        x4 = logistic(distance_range, sdd, ldd, 5.)

    if kernel=='LogSec':
        def log_sech(dist, sdd, ldd, b) :
            a1 = sdd
            a2 = ldd
            if (a1 <= 0. or b <= 0.) :
                return np.zeros(len(dist))
            dist_prob_sdd = 1 / (np.pi**2 * b * dist**2) / ((dist / a1)**(1 / b)␣
↪+ (dist / a1)**(-1 / b))
            dist_prob_ldd = 1 / (np.pi**2 * b * dist**2) / ((dist / a2)**(1 / b)␣
↪+ (dist / a2)**(-1 / b))
            dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
↪short_range_disp_frac)*dist_prob_ldd
            return scaler_0_1(dist_prob)
        x1 = log_sech(distance_range, sdd, ldd, 0.1)
        x2 = log_sech(distance_range, sdd, ldd, 0.25)
        x3 = log_sech(distance_range, sdd, ldd, 0.35)
        x4 = log_sech(distance_range, sdd, ldd, 0.5)

    if kernel=='NegExp':
        ax.plot(distance_range,x,color='black',ls='--')
        ax.set_title(kernel)
        ax.set_xlim(-0.05,distance_range.max())
        ax.set_ylim(-0.05,1.05)
    else :
        ax.plot(distance_range,x1)
        ax.plot(distance_range,x2)
        ax.plot(distance_range,x3)
        ax.plot(distance_range,x4)
        if kernel=='ExpPow':
            ax.legend(['0.15','0.25','0.5','0.75'], frameon=False)
        if kernel=='Weibull':
            ax.legend(['1.75','2.','2.25','2.5'], frameon=False)
        if kernel=='twoDt':
            ax.legend(['2.','3.','4.','5.'], frameon=False)
        if kernel=='Logistic':
            ax.legend(['3.5','4.','4.5','5.'], frameon=False)
        if kernel=='LogSec':
            ax.legend(['0.1','0.25','0.35','0.5'], frameon=False)
```

```
    ax.grid(False)
    ax.set_xlim(-0.05,distance_range.max())
    ax.set_ylim(-0.05,1.05)
    ax.set_title(kernel)

ax4.set_xlabel('Dispersal distance [m]')
ax5.set_xlabel('Dispersal distance [m]')
ax6.set_xlabel('Dispersal distance [m]')
ax1.set_ylabel('Dispersal probability')
ax4.set_ylabel('Dispersal probability')
f.subplots_adjust(wspace=0.1)

plt.savefig('Figure_3.png', bbox_inches='tight', dpi=300)
```



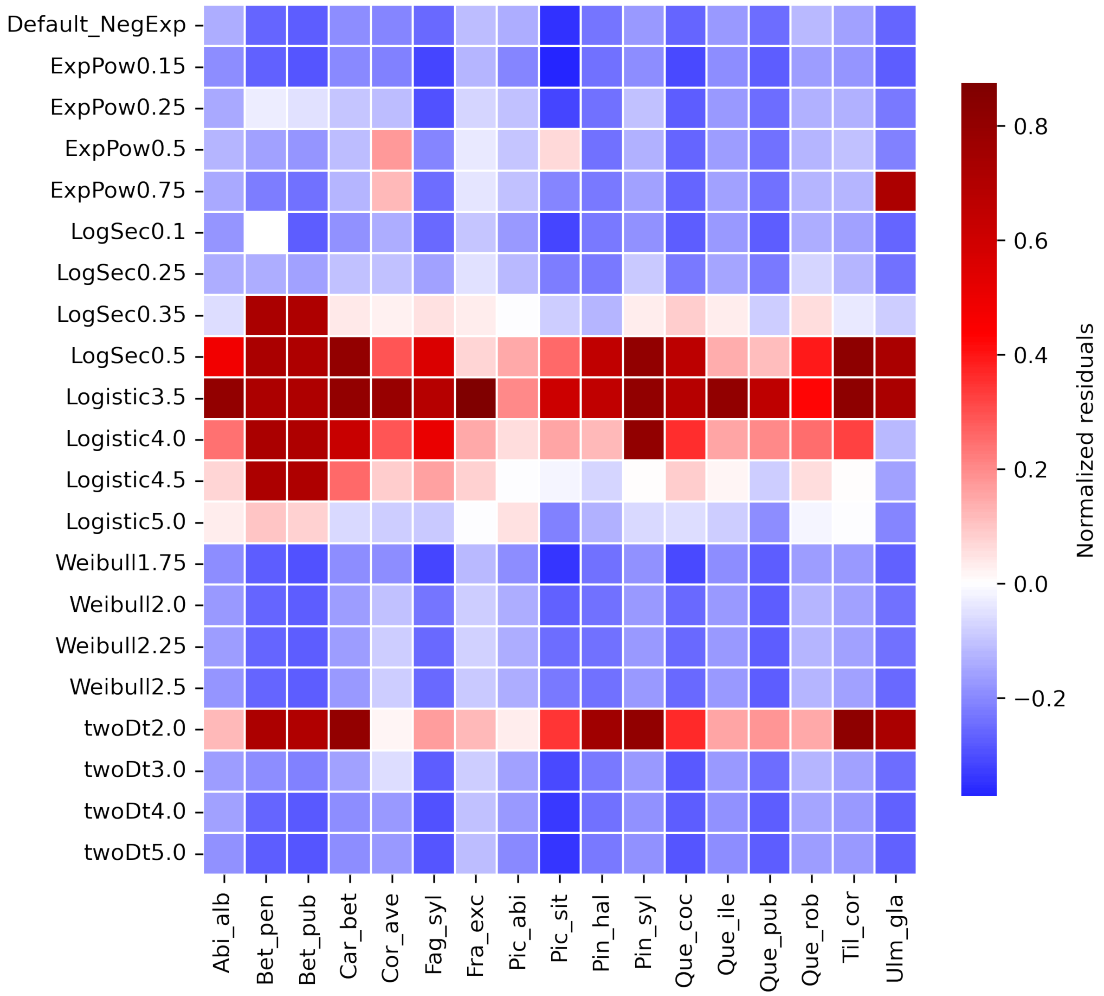
9.2 Species-specific performance for different shape parameters

We explored species- and kernel-specific parameter space of the shape parameter b by looking at the mean-normalized residuals of simulated migration rates with respect to the maximum of observed values.

[19]:
```
# Calculate residuals to the maximum observational value
KA_input['residuals_MAX'] =  KA_input.SIM - KA_input.OBS_MAX
newkernels_df = KA_input.copy()
```

```
# Scale residuals at the species-level
species = newkernels_df.Species.unique()
res_mean_norm_df = pd.DataFrame()
for sp in species :
    sp_df = newkernels_df[newkernels_df['Species'] == sp]
    res_mean_norm_MAX = mean_norm(sp_df.residuals_MAX)
    res_mean_norm = pd.DataFrame({"res_MAX":res_mean_norm_MAX})
    res_mean_norm_df = pd.concat([res_mean_norm_df,res_mean_norm])
newkernels_df = newkernels_df.copy()
newkernels_df['residuals_scaled_MAX'] = res_mean_norm_df.res_MAX

# Merge labels of kernel function and shape parameter
newkernels_df['shape_par'] = list(map(str, newkernels_df['shape_par']))
newkernels_df['kernel_combo'] = newkernels_df.kernel_fun.str.cat(newkernels_df.
 ↪shape_par)
newkernels_df.loc[newkernels_df.kernel_fun == 'NegExp','kernel_combo'] =␣
 ↪'Default_NegExp'

# Select representetive shape parameter values to plot
ExpPow_df = newkernels_df.loc[(newkernels_df.kernel_fun == 'ExpPow') &
                  (newkernels_df['shape_par'].isin(['0.15','0.25','0.5','0.
 ↪75']))]
Weibull_df = newkernels_df.loc[(newkernels_df.kernel_fun == 'Weibull') &
                  (newkernels_df['shape_par'].isin(['1.75','2.0','2.25','2.5']))]
twoDt_df = newkernels_df.loc[(newkernels_df.kernel_fun == 'twoDt') &
                  (newkernels_df['shape_par'].isin(['2.0','3.0','4.0','5.0']))]
Logistic_df = newkernels_df.loc[(newkernels_df.kernel_fun == 'Logistic') &
                  (newkernels_df['shape_par'].isin(['3.5','4.0','4.5','5.0']))]
LogSec_df = newkernels_df.loc[(newkernels_df.kernel_fun == 'LogSec') &
                  (newkernels_df['shape_par'].isin(['0.1','0.25','0.35','0.5']))]
NegExp_df = newkernels_df[newkernels_df.kernel_fun == 'NegExp']
kernel_sel_df = pd.
 ↪concat([ExpPow_df,Weibull_df,twoDt_df,Logistic_df,LogSec_df,NegExp_df])

# Make dataframe to plot (maximum observations)
heatmap_df = pd.DataFrame()
heatmap_df['Species'] = kernel_sel_df.Species.values
heatmap_df['Kernel'] = kernel_sel_df.kernel_combo.values
heatmap_df['Residuals'] = kernel_sel_df.residuals_scaled_MAX.values
heatmap_df.Residuals.astype(float)
heatmap_df = heatmap_df.pivot('Kernel', 'Species', 'Residuals')

# Plot heatmap
f, ax = plt.subplots()
f.set_figheight(3.54*2)
f.set_figwidth(3.54*2)
```

```
f.dpi = 300
heatmap = sns.heatmap(heatmap_df, cmap='seismic', center=0.00, linewidths=.5,
                      cbar_kws={"shrink": .82,'label': 'Normalized residuals'})
ax.set_ylabel('')
ax.set_xlabel('')

# Export figure
plt.savefig('Figure_4.png', bbox_inches='tight', dpi=300)
```

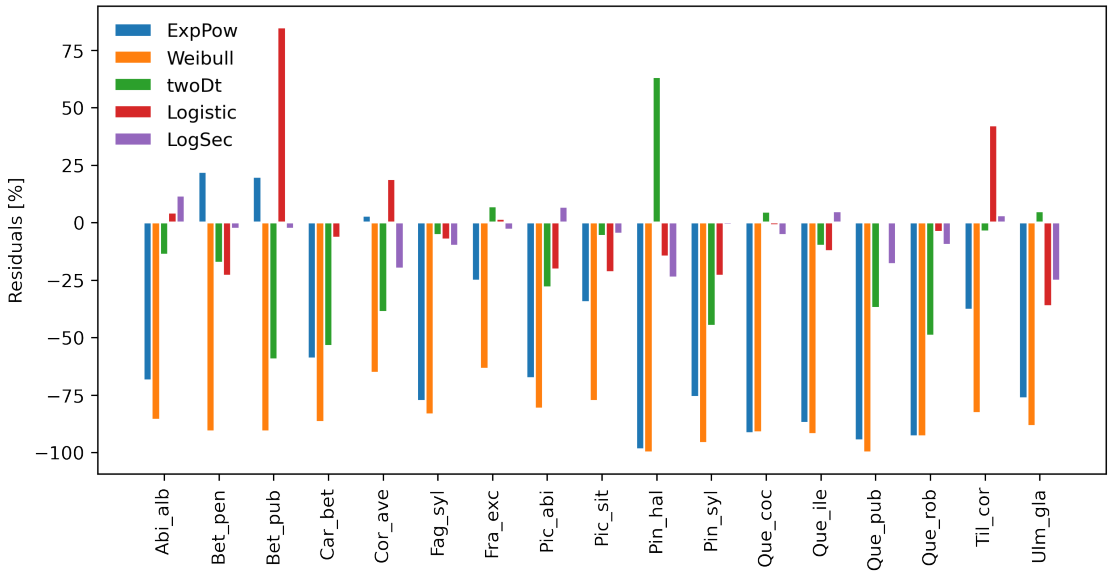


9.3 Best-fitted shape parameters per kernel

We selected the species-specific shape parameter values that significantly minimized residuals for each kernel function.

[20]:
```
def sel_bestshape_kernel(newkernels_df,obs_target,residuals_scaled):

    # Calculate absolute residuals
    newkernels_df = newkernels_df[newkernels_df['kernel_fun'] != 'NegExp']
    newkernels_df = newkernels_df.copy()
    newkernels_df['residuals_abs'] = abs(obs_target)

    # Find the minimum residual per species per kernel
    species = newkernels_df.Species.unique()
    bestfit_newkernels_df = newkernels_df.groupby(['Species','kernel_fun']).
 ↪apply(min_residual)

    # Format table for further analysis
    bestfit_newkernels_df.drop('Species', axis='columns', inplace=True)
    bestfit_newkernels_df.drop('kernel_fun', axis='columns', inplace=True)
    bestfit_newkernels_df = bestfit_newkernels_df.reset_index()
    bestfit_newkernels_df.drop('level_2', axis='columns', inplace=True)
    bestfit_newkernels_df.drop(residuals_scaled, axis='columns', inplace=True)
    bestfit_newkernels_df.drop('kernel_combo', axis='columns', inplace=True)
    bestfit_newkernels_df.drop('residuals_abs', axis='columns', inplace=True)
    bestfit_newkernels_df = bestfit_newkernels_df.rename(columns={'kernel_fun':␣
 ↪'Parameter_combo'})
    return bestfit_newkernels_df

bestfit_newkernels_df_MAX = sel_bestshape_kernel(newkernels_df,newkernels_df.
 ↪residuals_MAX,
                                                 'residuals_scaled_MAX')

# Add the percentage
bestfit_newkernels_df_MAX["residuals_perc"] = 100/bestfit_newkernels_df_MAX.
 ↪OBS_MAX * bestfit_newkernels_df_MAX.residuals_MAX

# Save table
bestfit_newkernels_df_MAX.to_csv('Table_S3.csv', sep=';', index=False)
```

[21]:
```
## Species-specific residuals for the best-fitted shape parameter of each␣
 ↪fat-tailed kernel
# Make figure
fig_out = make_plot(bestfit_newkernels_df_MAX)

# Export figure
fig_out.savefig('Figure_5.png', dpi=300, format='png')
```



9.4 Performance evaluation of new dispersal kernels

We evaluated the performance of the new kernels by calculating the error between simulated and observed migration rates for each species (residuals) and across species (RMSE). Additionally, we conducted a one-way ANOVA with post-hoc Tukey's HSD test among errors generated by newly-added kernels to verify whether one or more fat-tailed kernels improved the predictions across all species (RMSEs) or at the species level (residuals).

9.4.1 Overall error per kernel function

Error across species was calculated as root-mean-square-error (RMSE) was used to quantify the model performance across species:

$$RMSE = \frac{100}{\overline{obs_{max}}} \times \sqrt{\frac{\sum_{i=1}^{n}(sim_i - obs_{max,i})^2}{n}}$$

where n is the number of species, obs_{max} is the maximum value in the confidence range of observational values, and $\overline{obs_{max}}$ is the average across species.

[22]:
```
print("Observation values are the MAXIMUM values")
RMSE = rmse_perc(best_EVA_df.sim,best_EVA_df.obs_MAX,best_EVA_df.obs_MAX.mean())
print('NegExp RMSE: '+str(round(RMSE,2)))

kernels = bestfit_newkernels_df_MAX.Parameter_combo.unique()
for kernel in kernels:
    kernel_df =␣
 ↪bestfit_newkernels_df_MAX[bestfit_newkernels_df_MAX['Parameter_combo']==kernel]
    RMSE = rmse_perc(kernel_df.SIM,kernel_df.OBS_MAX,kernel_df.OBS_MAX.mean())
```

```
    print(kernel+' RMSE: '+str(round(RMSE,2)))
```

```
Observation values are the MAXIMUM values
NegExp RMSE: 92.81
ExpPow RMSE: 76.9
LogSec RMSE: 17.73
Logistic RMSE: 31.7
Weibull RMSE: 99.9
twoDt RMSE: 49.81
```

9.4.2 Test species-specific kernel performance

We conducted a one-way ANOVA with post-hoc Tukey's HSD test among residuals generated by the newly-added kernels to verify whether one or more fat-tailed kernels had a similar performance at the species level.

[23]:
```
def find_sameKernels(residuals_df):
    samePerformance_dict = {}
    for sp in species :
        sp_df = residuals_df[residuals_df['Species'] == sp]

        # Get all residuals from the minimum to the maximum observational range␣
↪(per kernel)
        res_df = pd.DataFrame()
        for par in sp_df.Parameter_combo.unique():
            par_df = sp_df.loc[sp_df.Parameter_combo==par]
            all_obs = np.arange(sp_df['OBS_MIN'].unique(),sp_df['OBS_MAX'].
↪unique(),1)
            residuals = all_obs - int(par_df.SIM.unique())
            res_sub = pd.DataFrame(index=np.arange(0,len(residuals),1))
            res_sub.insert(0,'residuals',residuals)
            res_sub.insert(1,'Parameter_combo',par)
            res_df = pd.concat([res_df,res_sub])

        # Perform Tukey's test
        tukey = pairwise_tukeyhsd(endog=res_df.residuals,
                                  groups=res_df.Parameter_combo,
                                  alpha=0.05)

        # Filter pairs with not significant difference (i.e. equally good or bad)
        kernelCombo = np.array(list(itertools.combinations(sp_df.Parameter_combo.
↪unique(), 2)))
        samePair = kernelCombo[np.where(tukey.reject==False)]
        if samePair.size!=0:
            samePair = list(samePair[0])
            samePerformance_dict[sp] = samePair
    return samePerformance_dict
```

[24]:
```
#samePerformance_dict_MAX = find_sameKernels(residuals_MAX_toPlot)
samePerformance_dict_MAX = find_sameKernels(bestfit_newkernels_df_MAX)
print("Kernels selected with observed values as MAXIMUM values. \nThe following␣
 ↪kernels have the same performance:")
samePerformance_dict_MAX
```

```
Kernels selected with observed values as MAXIMUM values.
The following kernels have the same performance:
```

[24]:
```
{'Fag_syl': ['LogSec', 'Logistic'],
 'Pic_sit': ['LogSec', 'twoDt'],
 'Pin_hal': ['ExpPow', 'Weibull'],
 'Que_coc': ['ExpPow', 'Weibull'],
 'Que_ile': ['Logistic', 'twoDt'],
 'Que_rob': ['ExpPow', 'Weibull']}
```

9.4.3 Select best-fitted kernel per species

We selected the kernel function(s) that significantly minimized RMSE% for each species:

$$RMSE = \frac{100}{obs_{max}} \times (sim - obs_{max})^2$$

where obs_{max} indicates the maximum migration rate estimates per species (Table 1).

[25]:
```
def select_bestKernel(residuals_df, samePerformance_dict):
    species = residuals_df.Species.unique()
    rmse_all_df = pd.DataFrame()
    for sp in species :
        sp_df = residuals_df[residuals_df['Species'] == sp]
        if sp in samePerformance_dict.keys():
            dict_sp = samePerformance_dict[sp]
        else:
            dict_sp = -2

        # RMSE% of each kernel
        rmse_df = pd.DataFrame()
        rmse_df['Species'] = sp
        kernels = sp_df.Parameter_combo
        rmse_df['kernel'] = kernels
        rmse_df['shape_par'] = sp_df.shape_par
        rmse_df['rmse'] = 0.
        rmse_df['Species'] = sp
        for kernel in kernels:
            kernel_df = sp_df[sp_df['Parameter_combo'] == kernel]
            RMSE = rmse_perc(kernel_df.SIM,float(max(kernel_df.
 ↪OBS_MAX)),float(max(kernel_df.OBS_MAX)))
            rmse_df.loc[rmse_df['kernel'] == kernel, 'rmse'] = RMSE
```

```
        print(sp+' - Kernel ranking [RMSE%]')

        # Rank kernel performance based on absolute residual value
        sp_df = sp_df.copy()
        sp_df['residuals_abs'] = abs(sp_df['residuals_MAX'])
        kernels_rank = sp_df.sort_values('residuals_abs',ascending=True).
↪reset_index(drop=True).Parameter_combo
        print('1: '+str(kernels_rank[0])+' '+str(rmse_df.
↪loc[rmse_df['kernel']==kernels_rank[0]].rmse.values.round(2))+
              ' | 2: '+str(kernels_rank[1])+' '+str(rmse_df.
↪loc[rmse_df['kernel']==kernels_rank[1]].rmse.values.round(2))+
              ' | 3: '+str(kernels_rank[2])+' '+str(rmse_df.
↪loc[rmse_df['kernel']==kernels_rank[2]].rmse.values.round(2))+
              ' | 4: '+str(kernels_rank[3])+' '+str(rmse_df.
↪loc[rmse_df['kernel']==kernels_rank[3]].rmse.values.round(2))+
              ' | 5: '+str(kernels_rank[4])+' '+str(rmse_df.
↪loc[rmse_df['kernel']==kernels_rank[4]].rmse.values.round(2)))

        # Select best kernel(s)
        best_kernel_sp = rmse_df.loc[rmse_df['kernel']==kernels_rank[0]]
        best_k = str(best_kernel_sp.iloc[0,1])
        sim_value = sp_df[sp_df['Parameter_combo']==kernels_rank[0]].SIM.unique()
        best_kernel_sp = best_kernel_sp.copy()
        best_kernel_sp['sim'] = float(sim_value)

        # Calculate residuals with respect to the MAXIMUM observation
        obs_values = sp_df[sp_df['Parameter_combo']==kernels_rank[0]].OBS_MAX
        best_kernel_sp['obs_MAX'] = float(max(obs_values))
        best_kernel_sp['res_MAX'] = best_kernel_sp.sim - best_kernel_sp.obs_MAX

        # If more than one optimal kernel
        if dict_sp!=-2:
            if best_k in dict_sp:
                second_best_k = dict_sp.copy()
                second_best_k.remove(best_k)
                second_best_k = str(second_best_k[0])
                second_best_kernel_sp = rmse_df.
↪loc[rmse_df['kernel']==second_best_k]
                sim_value = sp_df[sp_df['Parameter_combo']==second_best_k].SIM.
↪unique()
                obs_values = sp_df[sp_df['Parameter_combo']==second_best_k].
↪OBS_MAX
                best_kernel_sp['kernel2'] = second_best_k
                best_kernel_sp['shape_par2'] = float(second_best_kernel_sp.
↪shape_par)
```

```
                best_kernel_sp['rmse2'] = float(second_best_kernel_sp.rmse)
                best_kernel_sp['sim2'] = float(sim_value)

        rmse_all_df = pd.concat([rmse_all_df,best_kernel_sp])
    rmse_all_df = rmse_all_df.reset_index()
    rmse_all_df.drop('index', axis='columns', inplace=True)
    return rmse_all_df
```

[26]:
```
rmse_all_df_MAX = select_bestKernel(bestfit_newkernels_df_MAX,␣
↪samePerformance_dict_MAX)
```

```
Abi_alb - Kernel ranking [RMSE%]
1: Logistic [4.33] | 2: LogSec [11.67] | 3: twoDt [14.] | 4: ExpPow [68.67] | 5:
Weibull [85.67]
Bet_pen - Kernel ranking [RMSE%]
1: LogSec [2.75] | 2: twoDt [17.5] | 3: ExpPow [22.12] | 4: Logistic [23.] | 5:
Weibull [90.88]
Bet_pub - Kernel ranking [RMSE%]
1: LogSec [2.75] | 2: ExpPow [20.] | 3: twoDt [59.5] | 4: Logistic [85.] | 5:
Weibull [90.88]
Car_bet - Kernel ranking [RMSE%]
1: LogSec [0.7] | 2: Logistic [6.6] | 3: twoDt [53.7] | 4: ExpPow [59.1] | 5:
Weibull [86.7]
Cor_ave - Kernel ranking [RMSE%]
1: ExpPow [2.93] | 2: Logistic [18.93] | 3: LogSec [20.] | 4: twoDt [38.8] | 5:
Weibull [65.33]
Fag_syl - Kernel ranking [RMSE%]
1: twoDt [5.33] | 2: Logistic [7.33] | 3: LogSec [10.] | 4: ExpPow [77.67] | 5:
Weibull [83.33]
Fra_exc - Kernel ranking [RMSE%]
1: Logistic [1.6] | 2: LogSec [3.] | 3: twoDt [7.] | 4: ExpPow [25.2] | 5:
Weibull [63.6]
Pic_abi - Kernel ranking [RMSE%]
1: LogSec [6.8] | 2: Logistic [20.4] | 3: twoDt [28.2] | 4: ExpPow [67.6] | 5:
Weibull [80.8]
Pic_sit - Kernel ranking [RMSE%]
1: LogSec [4.8] | 2: twoDt [5.8] | 3: Logistic [21.6] | 4: ExpPow [34.6] | 5:
Weibull [77.6]
Pin_hal - Kernel ranking [RMSE%]
1: Logistic [14.67] | 2: LogSec [23.8] | 3: twoDt [63.33] | 4: ExpPow [98.67] |
5: Weibull [100.]
Pin_syl - Kernel ranking [RMSE%]
1: LogSec [1.] | 2: Logistic [23.13] | 3: twoDt [44.87] | 4: ExpPow [75.87] | 5:
Weibull [95.87]
Que_coc - Kernel ranking [RMSE%]
1: Logistic [1.2] | 2: twoDt [4.8] | 3: LogSec [5.4] | 4: Weibull [91.2] | 5:
ExpPow [91.6]
Que_ile - Kernel ranking [RMSE%]
```

```
1: LogSec [5.] | 2: twoDt [10.] | 3: Logistic [12.4] | 4: ExpPow [87.2] | 5:
Weibull [92.]
Que_pub - Kernel ranking [RMSE%]
1: Logistic [0.4] | 2: LogSec [18.] | 3: twoDt [37.2] | 4: ExpPow [94.8] | 5:
Weibull [100.]
Que_rob - Kernel ranking [RMSE%]
1: Logistic [4.] | 2: LogSec [9.6] | 3: twoDt [49.2] | 4: ExpPow [93.] | 5:
Weibull [93.]
Til_cor - Kernel ranking [RMSE%]
1: LogSec [3.2] | 2: twoDt [3.8] | 3: ExpPow [37.8] | 4: Logistic [42.2] | 5:
Weibull [82.8]
Ulm_gla - Kernel ranking [RMSE%]
1: twoDt [4.9] | 2: LogSec [25.2] | 3: Logistic [36.3] | 4: ExpPow [76.4] | 5:
Weibull [88.4]
```

9.4.4 Species-specific over/under/good estimations

Good estimations correspond to residuals within 15% from the observed migration speed (maximum value), over- and under-estimations correspond to positive and negative residuals outside of the 15% range, respectively.

Species-specific performance was calculated as residuals (*res*) with respect to the maximum value of observed migration rates, obs_{max} (where obs_{max} corresponds to the maxium value of `Migration rates`, Table 1):

$$res = \frac{100}{obs_{max}} \times (sim - obs_{max})$$

[27]:
```
def estimatesGoodness(df, good_margin=15.):
    target = df.rmse
    par_val = "kernel"

    good_est = df.loc[(target < good_margin) & (target > -good_margin)]
    bad_est = df.loc[(target > good_margin) | (target < -good_margin)]
    over_est = bad_est.loc[bad_est.res_MAX > 0.]
    under_est = bad_est.loc[bad_est.res_MAX < 0.]

    good_sp = good_est.Species.unique()
    for sp in good_sp :
        sp_sub = good_est[good_est['Species'] == sp]
        print('Good estimates: '+sp+' for '+str(sp_sub[par_val].values.tolist()))

    over_sp = over_est.Species.unique()
    for sp in over_sp :
        sp_sub = over_est[over_est['Species'] == sp]
        print('Over-estimations: '+sp+' for '+str(sp_sub[par_val].values.
↪tolist()))
```

```
    under_sp = under_est.Species.unique()
    for sp in under_sp :
        sp_sub = under_est[under_est['Species'] == sp]
        print('Under-estimations: '+sp+' for '+str(sp_sub[par_val].values.
↪tolist()))
```

[28]:
```
print("Observation values are the MAXIMUM values")
estimatesGoodness(rmse_all_df_MAX, good_margin=15.)
```

```
Observation values are the MAXIMUM values
Good estimates: Abi_alb for ['Logistic']
Good estimates: Bet_pen for ['LogSec']
Good estimates: Bet_pub for ['LogSec']
Good estimates: Car_bet for ['LogSec']
Good estimates: Cor_ave for ['ExpPow']
Good estimates: Fag_syl for ['twoDt']
Good estimates: Fra_exc for ['Logistic']
Good estimates: Pic_abi for ['LogSec']
Good estimates: Pic_sit for ['LogSec']
Good estimates: Pin_hal for ['Logistic']
Good estimates: Pin_syl for ['LogSec']
Good estimates: Que_coc for ['Logistic']
Good estimates: Que_ile for ['LogSec']
Good estimates: Que_pub for ['Logistic']
Good estimates: Que_rob for ['Logistic']
Good estimates: Til_cor for ['LogSec']
Good estimates: Ulm_gla for ['twoDt']
```

[29]:
```
print("Observation values are the MAXIMUM values")
estimatesGoodness(rmse_all_df_MAX, good_margin=10.)
```

```
Observation values are the MAXIMUM values
Good estimates: Abi_alb for ['Logistic']
Good estimates: Bet_pen for ['LogSec']
Good estimates: Bet_pub for ['LogSec']
Good estimates: Car_bet for ['LogSec']
Good estimates: Cor_ave for ['ExpPow']
Good estimates: Fag_syl for ['twoDt']
Good estimates: Fra_exc for ['Logistic']
Good estimates: Pic_abi for ['LogSec']
Good estimates: Pic_sit for ['LogSec']
Good estimates: Pin_syl for ['LogSec']
Good estimates: Que_coc for ['Logistic']
Good estimates: Que_ile for ['LogSec']
Good estimates: Que_pub for ['Logistic']
Good estimates: Que_rob for ['Logistic']
Good estimates: Til_cor for ['LogSec']
Good estimates: Ulm_gla for ['twoDt']
```

```
Under-estimations: Pin_hal for ['Logistic']
```

[30]:
```
def RMSE_NegExp_vs_FatTails(best_EVA_df, rmse_all_df):
    rmse_val = "rmse_MAX"
    print('Default NegExp kernel \n--> minimum RMSE:␣
 ↪'+str(round(min(best_EVA_df[rmse_val]),2))+'% for '+
          str(best_EVA_df.loc[best_EVA_df[rmse_val] ==␣
 ↪min(best_EVA_df[rmse_val])].Species.tolist())+
         '\n--> maximum RMSE: '+str(round(max(best_EVA_df[rmse_val]),2))+'% for␣
 ↪'+
          str(best_EVA_df.loc[best_EVA_df[rmse_val] ==␣
 ↪max(best_EVA_df[rmse_val])].Species.tolist()))
    print('Fat-tailed kernels \n--> minimum RMSE: '+str(round(min(rmse_all_df.
 ↪rmse),2))+'% with '+
          str(rmse_all_df.loc[rmse_all_df.rmse == min(rmse_all_df.rmse)].kernel.
 ↪tolist())+' for '+
          str(rmse_all_df.loc[rmse_all_df.rmse == min(rmse_all_df.rmse)].Species.
 ↪tolist())+
         '\n--> maximum RMSE: '+str(round(max(rmse_all_df.rmse),2))+'% with '+
          str(rmse_all_df.loc[rmse_all_df.rmse == max(rmse_all_df.rmse)].kernel.
 ↪tolist())+' for '+
          str(rmse_all_df.loc[rmse_all_df.rmse == max(rmse_all_df.rmse)].Species.
 ↪tolist()))
```

[31]:
```
print("Observation values are the MAXIMUM values")
RMSE_NegExp_vs_FatTails(best_EVA_df, rmse_all_df_MAX)
```

```
Observation values are the MAXIMUM values
Default NegExp kernel
--> minimum RMSE: 1.67% for ['Abi_alb']
--> maximum RMSE: 98.93% for ['Pin_hal']
Fat-tailed kernels
--> minimum RMSE: 0.4% with ['Logistic'] for ['Que_pub']
--> maximum RMSE: 14.67% with ['Logistic'] for ['Pin_hal']
```

9.4.5 Generate Table B3

[32]:
```
## Create dataframe with:
# simulated and observed (MAXIMUM) migration rates
# migration parameters
# life-history traits (dispersal syndromes and kernel shape categories)

def add_lifeHistory(rmse_all_df):
    parCombo_val = "Parameter_combo"
    rmse_val = "rmse_MAX"

    # Add species-specific migration parameters
```

```
    best_all_df = pd.DataFrame()
    best_all_df["Species"] = rmse_all_df.Species
    best_all_df["kernel"] = rmse_all_df.kernel
    best_all_df["shape_par"] = rmse_all_df.shape_par
    best_all_df["sim"] = rmse_all_df.sim
    best_all_df["obs"] = rmse_all_df.obs_MAX
    best_all_df["rmse"] = rmse_all_df.rmse
    best_all_df['FEC_max'] = KA_input.groupby('Species').mean().FEC_max.tolist()
    best_all_df['GERM_p'] = KA_input.groupby('Species').mean().GERM_p.tolist()
    best_all_df['SDD_d'] = KA_input.groupby('Species').mean().SDD_d.tolist()
    best_all_df['LDD_d'] = KA_input.groupby('Species').mean().LDD_d.tolist()

    # Add best default kernel (from EVA)
    EVA_opt_df = best_EVA_df.loc[best_EVA_df[parCombo_val] == 'all_MAX_opt']
    species_best_EVA = EVA_opt_df.Species.to_list()

    for sp in species_best_EVA :
        best_all_df.loc[rmse_all_df['Species'] == sp, 'kernel'] = 'NegExp'
        best_all_df.loc[rmse_all_df['Species'] == sp, 'shape_par'] = 'NA'
        best_all_df.loc[rmse_all_df['Species'] == sp, 'sim'] = float(EVA_opt_df.
↪loc[EVA_opt_df['Species'] == sp].sim)
        best_all_df.loc[rmse_all_df['Species'] == sp, 'rmse'] = float(EVA_opt_df.
↪loc[EVA_opt_df['Species'] == sp, rmse_val])
        best_all_df.loc[rmse_all_df['Species'] == sp, 'LDD_d'] =␣
↪float(EVA_opt_df.loc[EVA_opt_df['Species'] == sp].LDD_d)

    # Add species-specific dispersal syndromes (cf. Table 1)
    best_all_df['dispersal_syndrome'] = ['Mainly-wind', #Abi_alb
                                         'Wind + LDD', #Bet_pen
                                         'Wind + LDD', #Bet_pub
                                         'Wind + LDD', #Car_bet
                                         'Zoochory + LDD', #Cor_ave
                                         'Zoochory + LDD', #Fag_syl
                                         'Wind + LDD', #Fra_exc
                                         'Mainly-wind', #Pic_abi
                                         'Mainly-wind', #Pic_sit
                                         'Mainly-wind', #Pin_hal
                                         'Mainly-wind', #Pin_syl
                                         'Zoochory', #Que_coc
                                         'Zoochory', #Que_ile
                                         'Zoochory', #Que_pub
                                         'Zoochory', #Que_rob
                                         'Wind + LDD', #Til_cor
                                         'Wind + LDD' #Ulm_gla
                                        ]

    # Add species-specific dispersal syndromes (cf. Table 1)
```

```
    best_all_df['dispersal_syndrome2'] = ['Wi', #Abi_alb
                                          'Wi+(Wa)', #Bet_pen
                                          'Wi+(Wa+A)', #Bet_pub
                                          'Wi+(Wa+A)', #Car_bet
                                          'A+(Wa)', #Cor_ave
                                          'A+(Wa)', #Fag_syl
                                          'Wi+(Wa+A)', #Fra_exc
                                          'Wi', #Pic_abi
                                          'Wi', #Pic_sit
                                          'Wi', #Pin_hal
                                          'Wi+(Wa)', #Pin_syl
                                          'A', #Que_coc
                                          'A', #Que_ile
                                          'A', #Que_pub
                                          'A', #Que_rob
                                          'Wi+(Wa)', #Til_cor
                                          'Wi+(Wa)' #Ulm_gla
                                         ]
    return best_all_df
```

[33]:
```
best_all_df_MAX = add_lifeHistory(rmse_all_df_MAX)

# Export to CSV
best_all_df_MAX.to_csv('Table_B3.csv', sep=';', index=False)
```

9.5 Effect of alternative competitor on Migration Speed

Additional simulations with alternative competitors were conducted with optimized parameters, using *Quercus robur* and *sylvestris* as alternative competitors, namely: 1) *Fagus sylvatica*, 2) *Picea abies*, and 3) *Tilia cordata* with 1) *Quercus robur*, 2) *Pinus sylvestris*, and 3) *Quercus robur* or *Pinus sylvestris*, respectively (B. Huntley, 2022, pers. comm., January).

[34]:
```
# Add simulation values of migration rates with alternative competitors:
# Fag_syl vs. Que_rob
# Pic_abi vs. Pin_syl
# Til_cor vs. Que_rob OR Til_cor vs. Pin_syl
comp_lst = [["Fag_syl","Que_rob",258.],["Pic_abi","Pin_syl",441.],
            ["Til_cor","Pin_syl",518.],["Til_cor","Que_rob",448.]]

for comp in comp_lst:
    row = best_all_df_MAX.loc[best_all_df_MAX.Species==comp[0]]
    row = row.copy()
    row.iloc[0,0] = str(comp[0])+"("+str(comp[1])+")"
    row["sim"] = comp[2]
    row["rmse"] = rmse_perc(row.sim,row.obs,row.obs)
    best_all_df_MAX = pd.concat([best_all_df_MAX,row])
best_all_df_MAX = best_all_df_MAX.reset_index()
```

```
best_all_df_MAX.drop('index', axis='columns', inplace=True)
```

[35]:
```
best_all_df_MAX
```

[35]:

	Species	kernel	shape_par	sim	obs	rmse	FEC_max	\
0	Abi_alb	NegExp	NA	305.0	300.0	1.666667	50.0	
1	Bet_pen	LogSec	0.29	778.0	800.0	2.750000	11775.0	
2	Bet_pub	LogSec	0.29	778.0	800.0	2.750000	11775.0	
3	Car_bet	LogSec	0.3625	993.0	1000.0	0.700000	705.0	
4	Cor_ave	ExpPow	0.5	1544.0	1500.0	2.933333	6.0	
5	Fag_syl	twoDt	2.0	284.0	300.0	5.333333	29.0	
6	Fra_exc	Logistic	4.975	508.0	500.0	1.600000	42.0	
7	Pic_abi	NegExp	NA	509.0	500.0	1.800000	163.0	
8	Pic_sit	LogSec	0.5	476.0	500.0	4.800000	50.0	
9	Pin_hal	Logistic	3.75	1280.0	1500.0	14.666667	22.0	
10	Pin_syl	LogSec	0.4	1485.0	1500.0	1.000000	43.0	
11	Que_coc	Logistic	4.1	494.0	500.0	1.200000	5.0	
12	Que_ile	LogSec	0.4	525.0	500.0	5.000000	50.0	
13	Que_pub	Logistic	3.9	498.0	500.0	0.400000	50.0	
14	Que_rob	Logistic	3.5	480.0	500.0	4.000000	50.0	
15	Til_cor	LogSec	0.345	516.0	500.0	3.200000	720.0	
16	Ulm_gla	twoDt	2.2	1049.0	1000.0	4.900000	725.0	
17	Fag_syl(Que_rob)	twoDt	2.0	258.0	300.0	14.000000	29.0	
18	Pic_abi(Pin_syl)	NegExp	NA	441.0	500.0	11.800000	163.0	
19	Til_cor(Pin_syl)	LogSec	0.345	518.0	500.0	3.600000	720.0	
20	Til_cor(Que_rob)	LogSec	0.345	448.0	500.0	10.400000	720.0	

	GERM_p	SDD_d	LDD_d	dispersal_syndrome	dispersal_syndrome2
0	46.0	100.0	710.0	Mainly-wind	Wi
1	19.0	200.0	475.0	Wind + LDD	Wi+(Wa)
2	19.0	200.0	475.0	Wind + LDD	Wi+(Wa+A)
3	80.0	100.0	425.0	Wind + LDD	Wi+(Wa+A)
4	60.0	200.0	1500.0	Zoochory + LDD	A+(Wa)
5	71.0	25.0	200.0	Zoochory + LDD	A+(Wa)
6	60.0	100.0	725.0	Wind + LDD	Wi+(Wa+A)
7	80.0	100.0	785.0	Mainly-wind	Wi
8	75.0	100.0	800.0	Mainly-wind	Wi
9	60.0	100.0	250.0	Mainly-wind	Wi
10	91.0	100.0	250.0	Mainly-wind	Wi+(Wa)
11	70.0	25.0	200.0	Zoochory	A
12	90.0	25.0	200.0	Zoochory	A
13	90.0	25.0	200.0	Zoochory	A
14	95.0	25.0	200.0	Zoochory	A
15	55.0	100.0	374.0	Wind + LDD	Wi+(Wa)
16	65.0	100.0	350.0	Wind + LDD	Wi+(Wa)
17	71.0	25.0	200.0	Zoochory + LDD	A+(Wa)
18	80.0	100.0	785.0	Mainly-wind	Wi

```
19    55.0  100.0   374.0         Wind + LDD             Wi+(Wa)
20    55.0  100.0   374.0         Wind + LDD             Wi+(Wa)
```

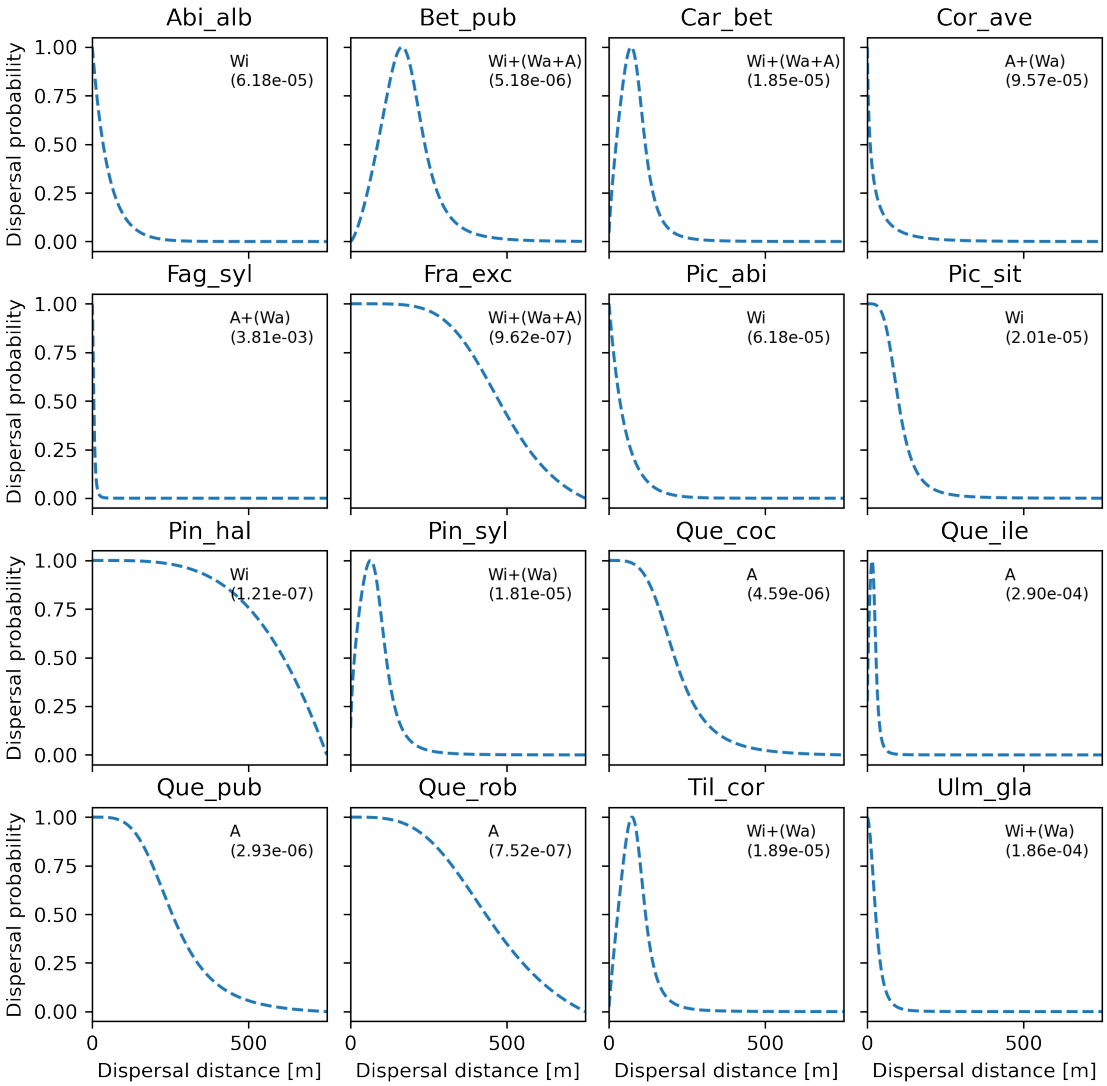
9.6 Best kernel shapes per species

[36]:
```
# Define kernel functions
def neg_exponential(dist, sdd, ldd, short_range_disp_frac) :
    a1 = sdd / 2
    a2 = ldd / 2
    if (a1 <= 0):
        return np.zeros(len(dist))
    dist_prob_sdd = 1 / (2 * np.pi * a1**2) * np.exp(-dist / a1)
    dist_prob_ldd = 1 / (2 * np.pi * a2**2) * np.exp(-dist / a2)
    dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
 ↪short_range_disp_frac)*dist_prob_ldd
    max_prob = max(dist_prob)
    return [scaler_0_1(dist_prob), max_prob]

def exp_power(dist, sdd, ldd, b, short_range_disp_frac) :
    a1 = (gamma(2/b) / gamma(3/b)) * sdd
    a2 = (gamma(2/b) / gamma(3/b)) * ldd
    if (a1 == 0. or b <= 0.):
        return np.zeros(len(dist))
    dist_prob_sdd = b / (2 * np.pi * a1**2 * gamma(2/b)) * np.exp(-(dist**b /␣
 ↪a1**b))
    dist_prob_ldd = b / (2 * np.pi * a2**2 * gamma(2/b)) * np.exp(-(dist**b /␣
 ↪a2**b))
    dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
 ↪short_range_disp_frac)*dist_prob_ldd
    max_prob = max(dist_prob)
    return [scaler_0_1(dist_prob), max_prob]

def twoDt(dist, sdd, ldd, b, short_range_disp_frac) :
    a1 = (gamma(b - 1) / gamma(b - 3/2)) * 2/np.pi * sdd
    a2 = (gamma(b - 1) / gamma(b - 3/2)) * 2/np.pi * ldd
    if (a1 <= 0. or b <= 1.) :
        return np.zeros(len(dist))
    dist_prob_sdd = (b - 1) / (np.pi * a1**2) * (1 + dist**2 / a1**2)**(-b)
    dist_prob_ldd = (b - 1) / (np.pi * a2**2) * (1 + dist**2 / a2**2)**(-b)
    dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
 ↪short_range_disp_frac)*dist_prob_ldd
    max_prob = max(dist_prob)
    return [scaler_0_1(dist_prob), max_prob]

def logistic(dist, sdd, ldd, b, short_range_disp_frac) :
    a1 = (gamma(2/b) * gamma(1 - 2/b) / gamma(3/b) * gamma(1 - 3/b)) * sdd
```

```
    a2 = (gamma(2/b) * gamma(1 - 2/b) / gamma(3/b) * gamma(1 - 3/b)) * ldd
    if (a1 <= 0. or b <= 2.) :
        return np.zeros(len(dist))
    dist_prob_sdd = b / (2 * np.pi * a1**2 * gamma(2/b) * gamma(1 - 2/b)) * (1 +␣
↪dist**b / a1**b)**-1
    dist_prob_ldd = b / (2 * np.pi * a2**2 * gamma(2/b) * gamma(1 - 2/b)) * (1 +␣
↪dist**b / a2**b)**-1
    dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
↪short_range_disp_frac)*dist_prob_ldd
    max_prob = max(dist_prob)
    return [scaler_0_1(dist_prob), max_prob]

def log_sech(dist, sdd, ldd, b, short_range_disp_frac) :
    a1 = sdd
    a2 = ldd
    if (a1 <= 0. or b <= 0.) :
        return np.zeros(len(dist))
    dist_prob_sdd = 1 / (np.pi**2 * b * dist**2) / ((dist / a1)**(1 / b) + (dist␣
↪/ a1)**(-1 / b))
    dist_prob_ldd = 1 / (np.pi**2 * b * dist**2) / ((dist / a2)**(1 / b) + (dist␣
↪/ a2)**(-1 / b))
    dist_prob = short_range_disp_frac*dist_prob_sdd + (1 -␣
↪short_range_disp_frac)*dist_prob_ldd
    max_prob = max(dist_prob)
    return [scaler_0_1(dist_prob), max_prob]

def plot_bestKernels(best_all_df, fig_out="Figure_6.png"):
    f, axes = plt.subplots(4, 4, sharex=True, sharey=True)
    (ax1, ax2, ax3, ax4), (ax5, ax6, ax7, ax8), (ax9, ax10, ax11, ax12), (ax13,␣
↪ax14, ax15, ax16) = axes
    f.set_figheight(7.25*1.25)
    f.set_figwidth(7.25*1.25)
    f.dpi = 300

    # Drop one of the Betula spp. as they are similar
    bestkernels_sp_df = best_all_df.copy()
    bestkernels_sp_df.drop(bestkernels_sp_df[bestkernels_sp_df.Species ==␣
↪'Bet_pen'].index, inplace=True)

    species = bestkernels_sp_df.Species
    distance_range = np.linspace(1.,750.,750)
    short_range_disp_frac = 0.99

    for sp,ax in zip(species,axes.flatten()):
        df_sub = bestkernels_sp_df[bestkernels_sp_df.Species == sp]
```

```
        if df_sub.kernel.item() == 'NegExp':
            x, max_disp = neg_exponential(distance_range, df_sub.SDD_d.item(),␣
↪df_sub.LDD_d.item(),
                                  short_range_disp_frac)
        if df_sub.kernel.item() == 'ExpPow':
            x, max_disp = exp_power(distance_range, df_sub.SDD_d.item(), df_sub.
↪LDD_d.item(),
                                    float(df_sub.shape_par.item()),␣
↪short_range_disp_frac)
        if df_sub.kernel.item() == 'twoDt':
            x, max_disp = twoDt(distance_range, df_sub.SDD_d.item(), df_sub.
↪LDD_d.item(),
                                  float(df_sub.shape_par.item()),␣
↪short_range_disp_frac)
        if df_sub.kernel.item() == 'Logistic':
            x, max_disp = logistic(distance_range, df_sub.SDD_d.item(), df_sub.
↪LDD_d.item(),
                                    float(df_sub.shape_par.item()),␣
↪short_range_disp_frac)
        if df_sub.kernel.item() == 'LogSec':
            x, max_disp = log_sech(distance_range, df_sub.SDD_d.item(), df_sub.
↪LDD_d.item(),
                                    float(df_sub.shape_par.item()),␣
↪short_range_disp_frac)

        ax.plot(distance_range,x,'--')
        ax.text(440, 0.9, df_sub.dispersal_syndrome2.item(), fontsize=8)
        max_disp = "{:.2e}".format(max_disp)
        ax.text(440, 0.8, "("+str(max_disp)+")", fontsize=8)

        ax.grid(False)
        ax.set_xlim(-0.05,distance_range.max())
        ax.set_ylim(-0.05,1.05)
        ax.set_title(sp)

    ax13.set_xlabel('Dispersal distance [m]')
    ax14.set_xlabel('Dispersal distance [m]')
    ax15.set_xlabel('Dispersal distance [m]')
    ax16.set_xlabel('Dispersal distance [m]')
    ax1.set_ylabel('Dispersal probability')
    ax5.set_ylabel('Dispersal probability')
    ax9.set_ylabel('Dispersal probability')
    ax13.set_ylabel('Dispersal probability')
    f.subplots_adjust(wspace=0.1)

    plt.savefig(fig_out, bbox_inches='tight', dpi=300)
```

[37]:
```
plot_bestKernels(best_all_df_MAX, "Figure_6.png")
```

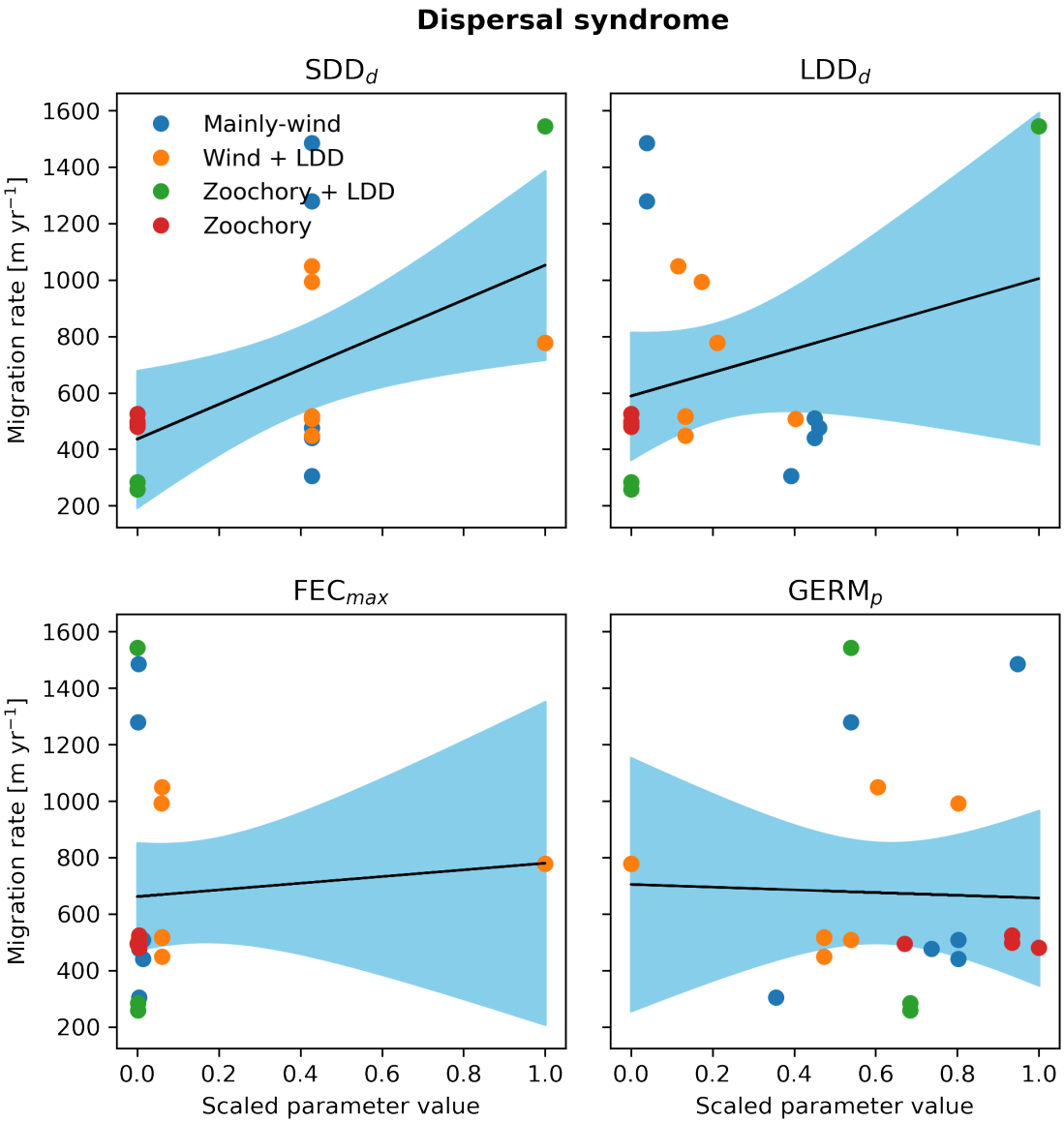


9.7 Migration rate and dispersal syndromes

[38]:
```
# Function to plot migration rate vs. parameter values
# Color points by group
def scatterplot_group(df, group_name='dispersal_syndrome'):
    f, axes = plt.subplots(2, 2, sharex=True, sharey=True)
    (ax1, ax2), (ax3, ax4) = axes
    f.set_figheight(7.25)
    f.set_figwidth(7.25)
    f.dpi = 300
```

```
    headers = ['SDD_d','LDD_d','FEC_max','GERM_p']
    for header,ax in zip(headers,axes.flatten()):

        # Select y, x and scale x
        y = df.sim
        x = df[header]
        x = scaler_0_1(x)
        df[header] = x

        # Select group-color
        group_lab = df[group_name].unique()
        for g_lab in group_lab:
            group_df = df[df[group_name] == g_lab]
            ys = group_df.sim
            xs = group_df[header]
            ax.plot(xs, ys, marker="o", linestyle="", label=g_lab)

        # Calculate statistics
        # ref: https://stackoverflow.com/questions/27164114/
↪show-confidence-limits-and-prediction-limits-in-scatter-plot
        slope, intercept = np.polyfit(x, y, 1)
        y_model = np.polyval([slope, intercept], x)
        x_mean = np.mean(x)
        y_mean = np.mean(y)
        n = x.size #number of samples
        m = 2 #number of parameters
        dof = n - m  #degrees of freedom
        t = stats.t.ppf(0.975, dof) #Students statistic of interval confidence
        residual = y - y_model
        std_error = (np.sum(residual**2) / dof)**.5 #standard deviation of the␣
↪error
        x_line = np.linspace(np.min(x), np.max(x), 100)
        y_line = np.polyval([slope, intercept], x_line)
        ci = t * std_error * (1/n + (x_line - x_mean)**2 / np.sum((x -␣
↪x_mean)**2))**.5

        # Draw regression line and confidence interval
        ax.plot(x, slope*x+intercept, color='black', linewidth=1)
        ax.fill_between(x_line, y_line + ci, y_line - ci, color = 'skyblue')

        # Add headers
        if header == 'SDD_d' :
            ax.set_title('SDD$_d$')
            ax.legend(frameon=False)
        if header == 'LDD_d' :
            ax.set_title('LDD$_d$')
        if header == 'FEC_max' :
```

```
            ax.set_title('FEC$_{max}$')
        if header == 'GERM_p' :
            ax.set_title('GERM$_p$')

    ax1.set_ylabel('Migration rate [m yr$^{-1}$]')
    ax3.set_ylabel('Migration rate [m yr$^{-1}$]')
    ax3.set_xlabel('Scaled parameter value')
    ax4.set_xlabel('Scaled parameter value')

    # Add general title
    if (group_name == 'dispersal_syndrome'):
        f.suptitle('Dispersal syndrome', fontweight='bold')
    if (group_name == 'kernel_shape'):
        f.suptitle('Kernel shape', fontweight='bold')
    f.subplots_adjust(wspace=0.1,top=0.91)

# Groups: dispersal_syndrome
scatterplot_group(df=best_all_df_MAX, group_name='dispersal_syndrome')
plt.savefig('Figure_S5.png', bbox_inches='tight', dpi=300)
```



10 Uncertainty analysis

In order to select the best model structure, we quantified the utility U of each model framework as a synthesis of both sensitivity and error (Snowling and Kramer, 2001):

$$U = \sqrt{2} - \sqrt{(\frac{S}{S_{max}})^2 + (\frac{E}{E_{max}})^2}$$

where the sensitivity S and error E are calculated according to the equations above for II_2 and $RMSE$ across species and parameters, respectively, whereas S_{max} and E_{max} are the maximum sen-

sitivity and error across models, respectively.

Additionally, we calculated an index of model complexity:

$$C = \sum_{j=1}^{N} \sum_{l=1}^{n_j} p_l r_l$$

where N is the number of state variable, n_j is the number of processes implemented for each state variable j, p_l is the number of parameters of each process l, and r_l is the number of equations used to formulate each process.

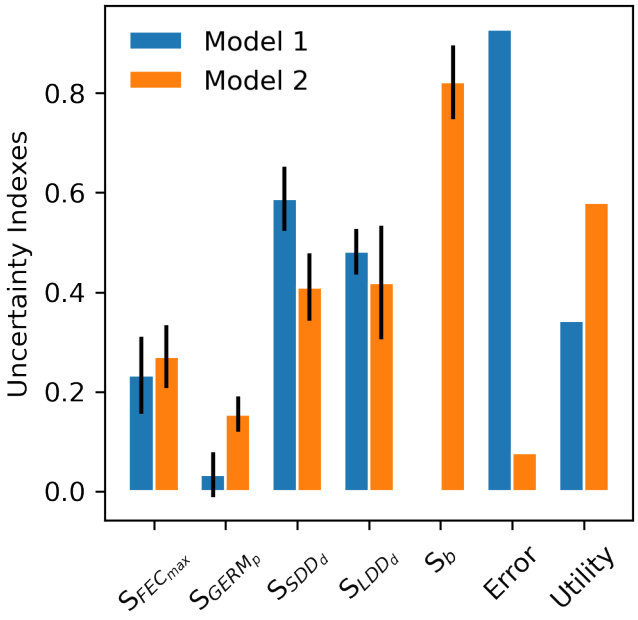
[39]:
```
# Divide datasets based on standard (Betula sp.) or alternative competitors:
# Fag_syl vs. Que_rob
# Pic_abi vs. Pin_syl
# Til_cor vs. Que_rob OR Til_cor vs. Pin_syl
comp_lst = [["Fag_syl","Que_rob",240.],["Pic_abi","Pin_syl",240.],
            ["Til_cor","Pin_syl",240.],["Til_cor","Que_rob",240.]]

best_all_df_altcomp = best_all_df_MAX.copy()
for comp in comp_lst:
    # Drop rows with alternative competitors
    sp_lab = str(comp[0])+"("+str(comp[1])+")"
    index_drop = best_all_df_MAX.loc[best_all_df_MAX.Species==sp_lab]
    if len(index_drop)>0:
        index_drop = index_drop.index.item()
        best_all_df_MAX.drop(index_drop, axis=0, inplace=True)

    # Drop rows with default competitors
    index_drop_altcomp = best_all_df_altcomp.loc[best_all_df_altcomp.
 ↪Species==str(comp[0])]
    if len(index_drop_altcomp)>0:
        index_drop_altcomp = index_drop_altcomp.index.item()
        best_all_df_altcomp.drop(index_drop_altcomp, axis=0, inplace=True)


## Calculate Sensitivities across species for both models
S_model1 = SA_input.groupby('Parameters').agg(['mean',f_ste]).T.loc['II2']
SA_fatkernels_input['II2'] = (SA_fatkernels_input.MigRate_Max -␣
 ↪SA_fatkernels_input.MigRate_Min) / SA_fatkernels_input.MigRate_Max
S_model2 = SA_fatkernels_input.groupby('Parameters').agg(['mean',f_ste]).T.
 ↪loc['II2']

# Create dataframe to plot
S_toPlot = pd.concat([S_model1, S_model2])
S_toPlot.index = ['mean_model1','ste_model1','mean_model2','ste_model2']
S_toPlot.columns = ['S_FEC_max','S_GERM_p','S_LDD_d','S_SDD_d']
```

```
# Calculate II2 for shape parameter (only model 2)
species = KA_input.Species.unique()
II2s = np.zeros(len(species))
pos = 0
for sp in species:
    # Subset for best kernel per species
    sp_sub = KA_input[KA_input['Species'] == sp]
    best_kernel_sp = best_all_df_MAX.loc[best_all_df_MAX['Species'] == sp,␣
 ↪'kernel'].tolist()
    best_kernel_sub = sp_sub[sp_sub['kernel_fun'] == str(best_kernel_sp[0])]

    # Get the minimum and maximum simulated migration speed
    y_min = best_kernel_sub.loc[best_kernel_sub.SIM == min(best_kernel_sub.SIM)].
 ↪SIM.tolist()
    y_min = y_min[0]
    y_max = best_kernel_sub.loc[best_kernel_sub.SIM == max(best_kernel_sub.SIM)].
 ↪SIM.tolist()
    y_max = y_max[0]

    # Calculate II2 per species
    II2 = (y_max - y_min) / y_max
    II2s[pos] = II2
    pos = pos + 1

# Calculate mean and ste II2 across species (shape parameter)
II2_mean_shapepar = np.mean(II2s)
II2_ste_shapepar = II2s.std() / np.sqrt(len(II2s))
S_toPlot['S_shape_par'] = [0,0,II2_mean_shapepar,II2_ste_shapepar] #only model 2

## Calculate mean Sensitivity S across parameters
S_mean_model1 = np.mean(S_toPlot.T.mean_model1)
S_mean_model2 = np.mean(S_toPlot.T.mean_model2)

## Calculate Error E across species (with standard Betula sp. competitor)
E_model1 = rmse_perc(best_EVA_df.sim, best_EVA_df.obs_MAX, best_EVA_df.obs_MAX.
 ↪mean()) / 100
E_model2 = rmse_perc(best_all_df_MAX.sim, best_all_df_MAX.obs, best_all_df_MAX.
 ↪obs.mean()) / 100
S_toPlot['Error'] = [E_model1, 0, E_model2, 0]

## Calculate Error E across species (with alternative competitors)
E_model2_altcomp = rmse_perc(best_all_df_altcomp.sim, best_all_df_altcomp.obs,␣
 ↪best_all_df_altcomp.obs.mean()) / 100

# Find S_max (rounded-up maximum)
S_max = max(S_mean_model1, S_mean_model2)
```

```
S_max = round(S_max,1) + 0.1

# Find E_max (rounded-up maximum)
E_max = max(E_model1, E_model2)
E_max = round(E_max,1) + 0.1

## Calculate Utility U
U_model1 = np.sqrt(2) - np.sqrt((S_mean_model1 / S_max)**2 + (E_model1 /␣
 ↪E_max)**2)
U_model2 = np.sqrt(2) - np.sqrt((S_mean_model2 / S_max)**2 + (E_model2 /␣
 ↪E_max)**2)
S_toPlot['Utility'] = [U_model1, 0, U_model2, 0]
S_toPlot = S_toPlot.T

# Utility for alternative competitors
U_model2_altcomp = np.sqrt(2) - np.sqrt((S_mean_model2 / S_max)**2 +␣
 ↪(E_model2_altcomp / E_max)**2)

# Plot Figure 6
labels =␣
 ↪['S$_{FEC_{max}}$','S$_{GERM_{p}}$','S$_{SDD_{d}}$','S$_{LDD_{d}}$','S$_{b}$','Error','Utilit
x = np.arange(len(labels))
width = 0.35

f, ax = plt.subplots()
f.set_figheight(3.54)
f.set_figwidth(3.54)
f.dpi = 300

rects1 = ax.bar(x - width/2, S_toPlot['mean_model1'], width,␣
 ↪yerr=S_toPlot['ste_model1'], label='Model 1',
                edgecolor="white")
rects2 = ax.bar(x + width/2, S_toPlot['mean_model2'], width,␣
 ↪yerr=S_toPlot['ste_model2'], label='Model 2',
                edgecolor="white")

ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.set_ylabel('Uncertainty Indexes')
ax.legend(frameon=False)
plt.xticks(rotation=45)

# Export figure
plt.savefig('Figure_7.png', bbox_inches='tight', dpi=300)

# Calculate Complexity C
```

```
C_model1 = 5
C_model2 = 6

# Print results
print('Model 1 | Complexity: '+str(round(C_model1,2))+'; Sensitivity:␣
 ↪'+str(round(S_mean_model1,2))+
      '; Error (MAX): '+str(round(E_model1*100,2))+'; Utility (MAX):␣
 ↪'+str(round(U_model1,2))+
     '\nModel 2 | Complexity: '+str(round(C_model2,2))+'; Sensitivity:␣
 ↪'+str(round(S_mean_model2,2))+
      '; Error (MAX): '+str(round(E_model2*100,2))+'; Utility (MAX):␣
 ↪'+str(round(U_model2,2)))

print("\nWith alternative competitors:\nFag_syl vs. Que_rob\nPic_abi vs.␣
 ↪Pin_syl\nTil_cor vs. Pin_syl OR Til_cor vs. Que_rob")
print('Model 2 | Complexity: '+str(round(C_model2,2))+'; Sensitivity:␣
 ↪'+str(round(S_mean_model2,2))+
      '; Error (MAX): '+str(round(E_model2_altcomp*100,2))+
      '; Utility (MAX): '+str(round(U_model2_altcomp,2)))
```

```
Model 1 | Complexity: 5; Sensitivity: 0.27; Error (MAX): 92.81; Utility (MAX):
0.34
Model 2 | Complexity: 6; Sensitivity: 0.42; Error (MAX): 7.7; Utility (MAX):
0.58

With alternative competitors:
Fag_syl vs. Que_rob
Pic_abi vs. Pin_syl
Til_cor vs. Pin_syl OR Til_cor vs. Que_rob
Model 2 | Complexity: 6; Sensitivity: 0.42; Error (MAX): 8.13; Utility (MAX):
0.58
```


Model 1
Model 2
Uncertainty Indexes
0.0
0.2
0.4
0.6
0.8
$S_{FEC_{max}}$
S_{GERM_p}
S_{SDD_d}
S_{LDD_d}
S_b
Error
Utility